

**Ce document regroupe tous les livrables  
reprise du Livrable 1(vu que le premier livrable était d'une qualité médiocre):**

**1.Introduction:**

Le projet PCOrderApplication est une solution technologique de haute performance, élaborée spécifiquement pour répondre aux exigences de gestion de commandes de composants matériels et logiciels dans une entreprise spécialisée en équipements informatiques. Cette plateforme novatrice se distingue par une organisation intuitive et méthodique, qui permet aux membres de chaque département d'accomplir leurs missions de manière efficace grâce à des rôles précis : Administrator, Assembler, StoreKeeper, et Requester. Ces rôles disposent chacun de fonctionnalités sur mesure qui facilitent et optimisent les processus complexes d'approvisionnement, de gestion des stocks, et de suivi des commandes.

Le but principal de PCOrderApplication est de centraliser l'ensemble de ces opérations dans une plateforme numérique sécurisée, permettant ainsi de réduire les difficultés administratives, de limiter les erreurs humaines, et d'assurer une traçabilité continue des composants, qu'ils soient matériels ou logiciels. L'application se positionne comme un atout stratégique majeur pour l'entreprise, renforçant la productivité, améliorant la coordination entre les équipes et augmentant la réactivité pour mieux répondre aux exigences du marché.

Les Administrators détiennent des autorisations étendues, leur permettant de gérer les utilisateurs, configurer les rôles, et superviser les accès en fonction des responsabilités de chacun. Ils peuvent ainsi créer, modifier ou supprimer des comptes pour les autres utilisateurs – qu'ils soient StoreKeepers, Assemblers, ou Requesters – garantissant une gestion cohérente des accès dans l'organisation. En centralisant le contrôle des profils, ils favorisent une structure hiérarchique claire, renforçant l'organisation et l'efficacité.

Les StoreKeepers jouent un rôle central dans la gestion de l'inventaire. Leur tâche consiste à s'assurer en temps réel que le stock est constamment à jour, en ajoutant, retirant ou consultant les composants disponibles. Le système prend en charge aussi bien les composants matériels (par exemple : boîtiers, cartes mères, modules mémoire, dispositifs de stockage) que les logiciels (tels que navigateurs, suites bureautiques, environnements de développement, et autres outils essentiels). La mise à jour instantanée de l'inventaire assure un accès fiable et précis aux informations, renforçant la transparence et la fluidité des opérations.

Les Requesters interviennent directement dans le processus d'approvisionnement en soumettant des demandes de composants nécessaires à leurs activités. PCOrderApplication leur fournit une interface intuitive et ergonomique, conçue pour simplifier le processus de sélection et de commande en quelques étapes seulement, optimisant ainsi l'expérience utilisateur. Cette fonctionnalité garantit une prise en charge rapide et efficace de chaque demande, pour assurer un approvisionnement optimal adapté aux besoins concrets des équipes sur le terrain.

Les Assemblers, quant à eux, sont des maillons essentiels dans la chaîne de traitement des commandes, avec la responsabilité de vérifier et valider les requêtes soumises par les

Requesters. Ils veillent à ce que chaque commande réponde aux exigences en matière de qualité et de précision. Ils disposent également de la possibilité de rejeter une commande en y ajoutant des commentaires explicatifs, facilitant les corrections nécessaires pour les utilisateurs, ou de finaliser les commandes prêtes pour l'expédition. Cette étape de vérification apporte une garantie supplémentaire de qualité et assure une fluidité jusqu'à la livraison finale.

Pour garantir une sécurité et une confidentialité optimales, chaque utilisateur se connecte à l'application à l'aide d'identifiants uniques et d'un mot de passe sécurisé, ce qui lui donne accès aux fonctionnalités spécifiques à son rôle. Cette méthode d'authentification robuste assure la protection des données sensibles de l'entreprise, et garantit que chaque utilisateur accède uniquement aux fonctionnalités qui lui sont allouées, renforçant l'intégrité et la fiabilité du système.

En conclusion, PCOrderApplication représente une solution complète et innovante pour la gestion de commandes dans le domaine technologique et informatique. En regroupant toutes les fonctionnalités nécessaires à la gestion des stocks, au traitement des commandes, et à la gestion des utilisateurs dans une interface unique et sécurisée, cette application répond aux exigences actuelles de performance, de précision et de sécurité. Elle s'impose comme un pilier central pour l'optimisation des opérations internes, permettant à l'entreprise d'affronter les défis du marché avec souplesse et agilité.

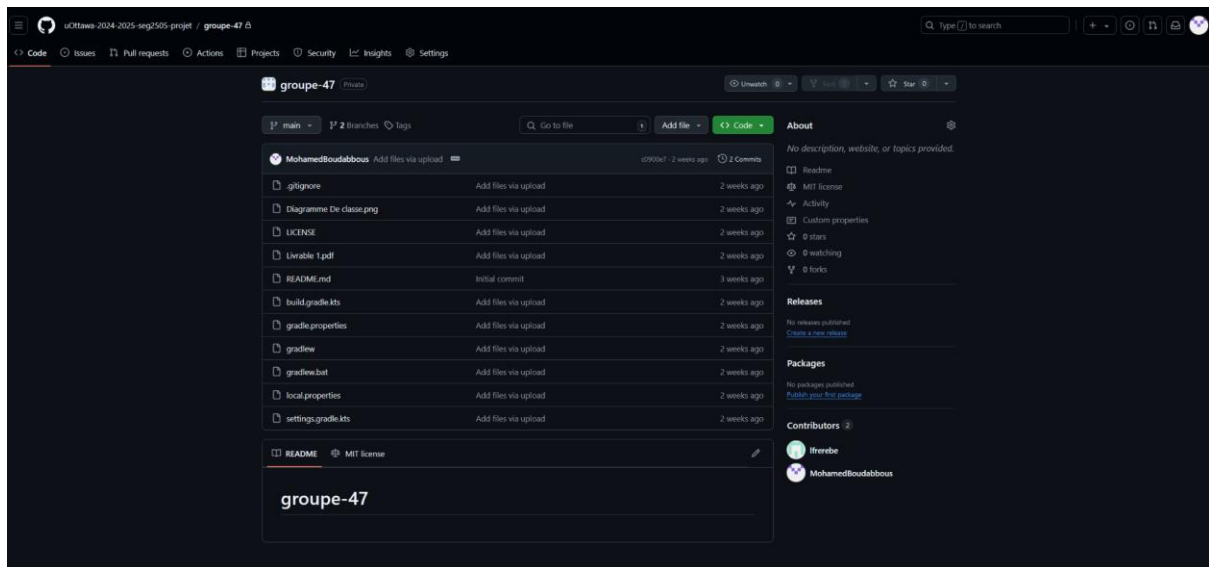
## **2. Création et Configuration de l'Équipe GitHub**

Pour le projet PCOrderApplication, l'équipe GitHub, nommée Groupe 46, a été créée par le professeur, me conférant un accès exclusif en tant que membre unique. Cette configuration me permet de gérer seul le dépôt et l'ensemble des artefacts du projet, en veillant à une organisation rigoureuse et à une documentation complète, accessible exclusivement au professeur pour évaluation.

Capture d'Affectation à l'Équipe GitHub (Équipe 46):

Capture de l'Histoire des Commits:

Capture de la Structure des Dossiers du Dépôt:



Ces captures d'écran attestent de la configuration de Groupe 46 et de l'organisation du dépôt GitHub, soulignant mon rôle de gestionnaire unique et la rigueur dans l'organisation du projet. Avec cette structure bien établie et une gestion méthodique, PCOrderApplication est organisé de manière optimale pour répondre aux exigences du projet en termes de clarté, de performance, et de qualité.-cette capture représente la structure du dossier dans le livrable 1). Celle du livrable 2 est dans la prochaine sous section.

## Étapes de Configuration et Gestion :

En premier lieu, le professeur a configuré l'équipe sous l'identifiant Équipe 46 et m'a attribué le rôle de membre unique, garantissant un accès exclusif aux ressources du dépôt. Cette configuration me permet de travailler en autonomie sur le projet, en m'assurant que tous les éléments sont correctement structurés et sécurisés pour répondre aux besoins du cours. En tant que seul gestionnaire de l'équipe, j'ai configuré le dépôt GitHub pour une utilisation structurée et méthodique, ajustant les paramètres d'accès et organisant les permissions de manière à faciliter l'organisation des artefacts. Ce paramétrage garantit un environnement de travail clair et ordonné, permettant une navigation et un accès aux ressources rapide et efficace. Ensuite, j'ai structuré les artefacts en plusieurs sections distinctes, chacune dédiée à une partie spécifique du projet. Cela inclut des dossiers pour le projet Android Studio, la documentation technique, et les ressources de support. Cette organisation assure que chaque composant est facilement accessible et documenté, offrant une visibilité claire sur l'évolution et l'implémentation du projet, essentielle pour un examen précis par le professeur.

## 3. Version du Modèle Conceptuel:

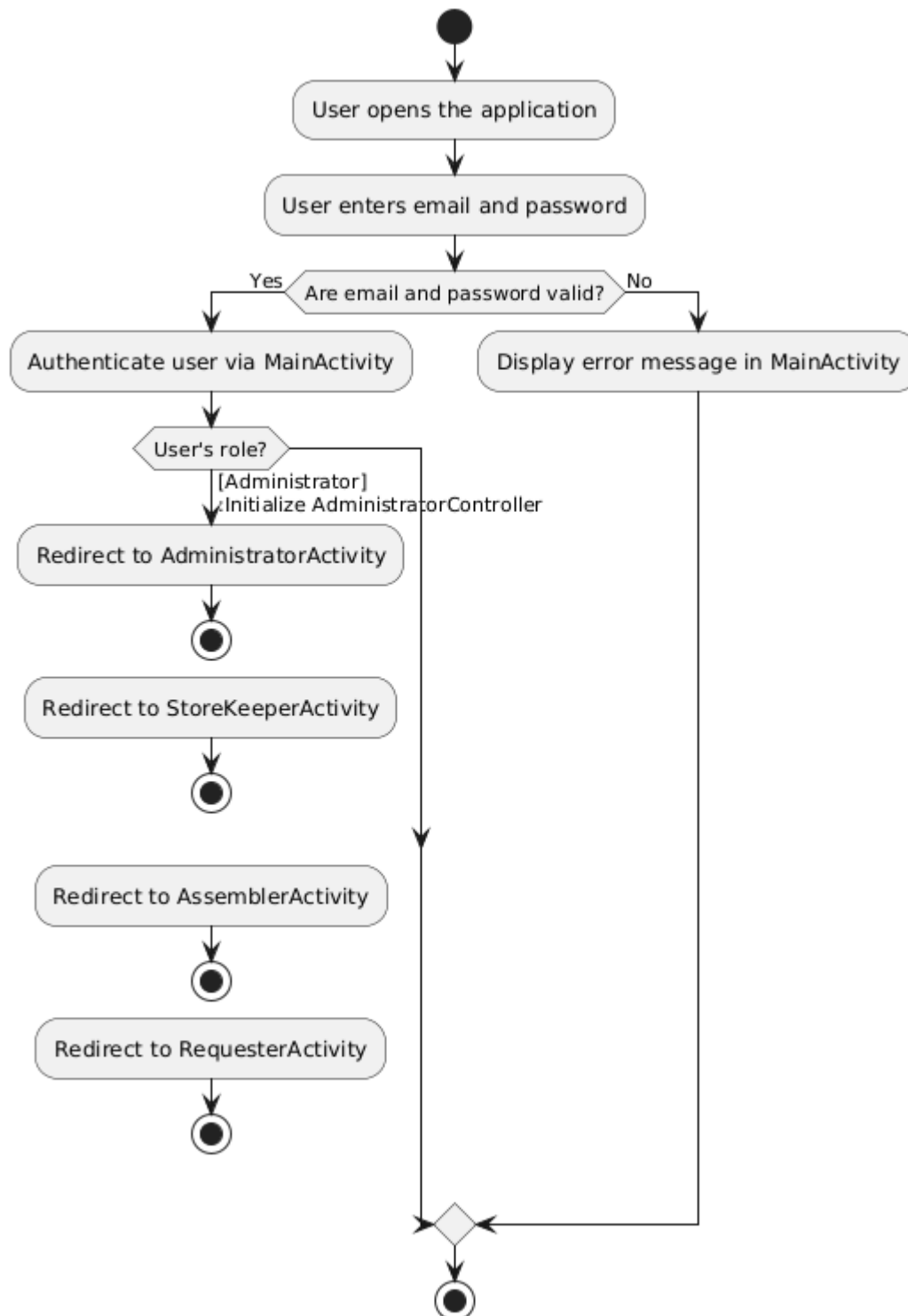
pour cette partie l'outil en ligne PlantUML a été utilisé pour générer les différents diagramme.

**Diagramme de Classe: voir la photo PNG dans le dossier Livrable et diagramme, c'est modifi é à chaque livrable, pour le code source voir Annexe**

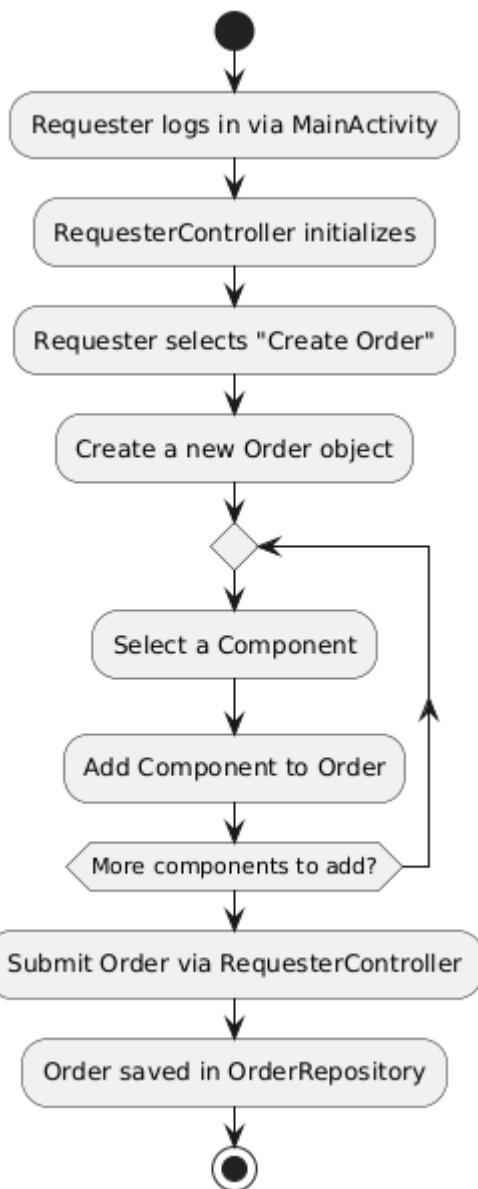
### Diagrammes d'activité

Les codes UML pour cette sous section ont été rédigé en anglais pour une meilleur lisibilité parceque le code initial est en anglais

#### 1-pour la connexion d'un utilisateur:



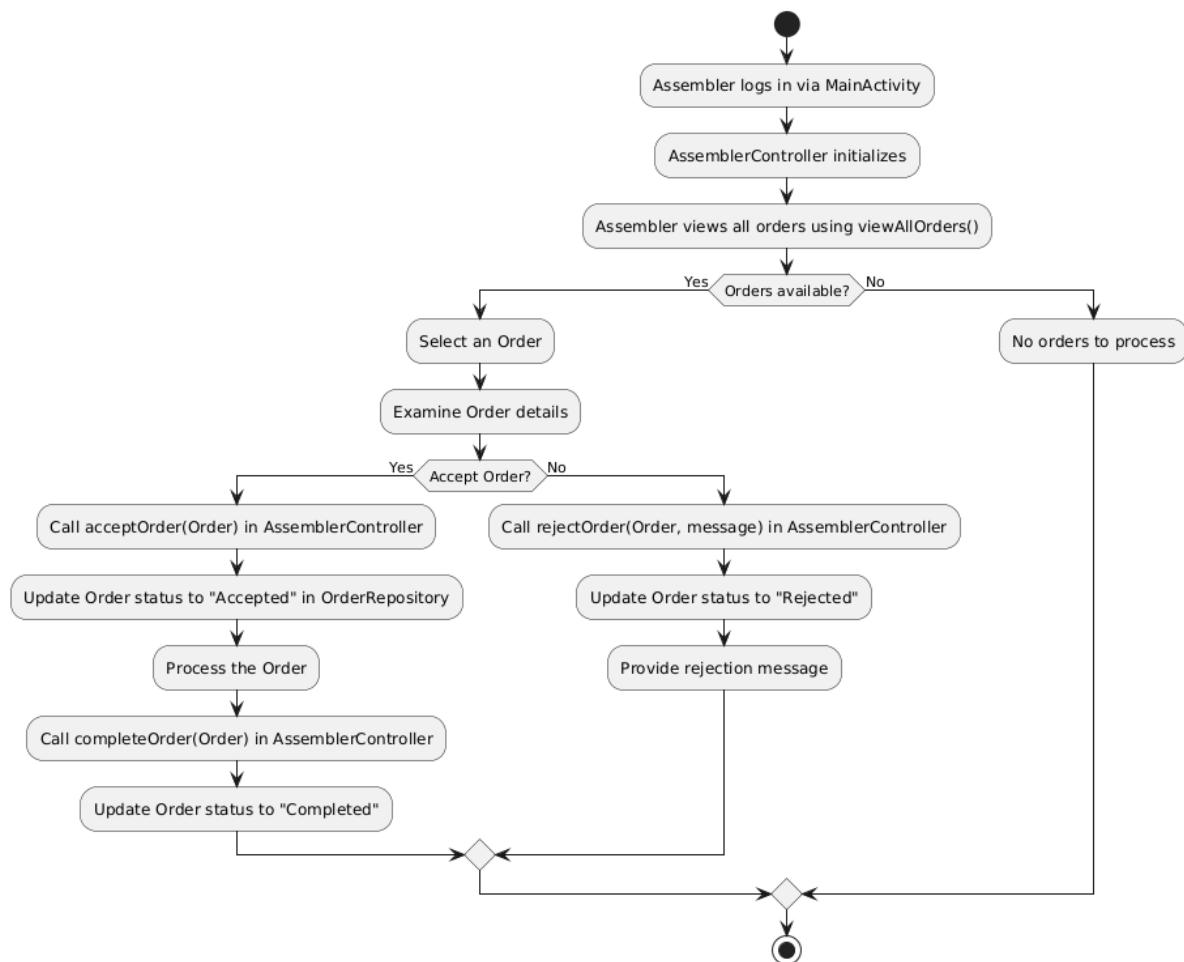
#### -pour la création d'une commande par un Requester:



---

**description**

**-pour le traitement d'une commande par un assembleur :**



**Diagramme d'activité pour la gestion des composants par le StoreKeeper**  
**Diagramme de Séquence**

description

### **Diagramme d'État:**

#### **description**

#### **5.Architecture utilisée dans ce projet:**

L'architecture utilisée dans ce projet est L'architecture MVC (Modèle-Vue-Contrôleur). Cette dernière a été adoptée pour les avantages suivants:

#### **5.1Avantages:**

Elle apporte des bénéfices significatifs dans des projets structurés comme PCOrderApplication, en proposant une organisation modulaire et bien définie qui facilite le développement, la maintenance, et l'évolution du code. En séparant les responsabilités en trois composants distincts – le Modèle, la Vue, et le Contrôleur – cette architecture aide à structurer le code de manière claire. Le Modèle gère les données et la logique métier, la Vue se charge de l'interface utilisateur, et le Contrôleur coordonne l'interaction entre les deux, facilitant une organisation fluide des processus et rendant le code plus facile à lire et à modifier.

Grâce à cette modularité, il devient possible d'apporter des changements spécifiques à chaque composant. Par exemple, des mises à jour dans l'interface utilisateur peuvent être

réalisées dans la vue sans modifier la logique métier dans le Modèle. Cela rend les modifications moins risquées et permet de développer de nouvelles fonctionnalités de façon plus efficace, ce qui est particulièrement utile pour les projets avec des mises à jour régulières. De plus, chaque composant étant indépendant, la réutilisabilité du code est considérablement améliorée : des portions de code bien isolées peuvent être reprises pour d'autres projets ou sections, contribuant à un gain de temps pour l'équipe de développement.

Un autre avantage essentiel du modèle MVC est la facilité de test. La logique métier étant isolée dans le Modèle, il devient possible de tester chaque fonctionnalité individuellement, facilitant les tests unitaires et permettant une détection rapide et précise des erreurs. Cette séparation entre la logique métier et l'interface utilisateur contribue également à rendre les tests d'intégration plus robustes, car chaque composant peut être évalué dans un contexte d'utilisation réaliste.

Enfin, l'architecture MVC permet une collaboration efficace, car les développeurs peuvent se spécialiser sur des composants spécifiques : certains peuvent travailler sur la Vue pour affiner l'interface utilisateur, tandis que d'autres se concentrent sur le Modèle ou l'intégration de données. Ce découpage du travail facilite une coordination harmonieuse, chaque membre pouvant se concentrer sur son domaine sans interférer avec les autres. En somme, MVC apporte une structure méthodique, améliore la clarté et la gestion du code, assure une meilleure testabilité, et permet une collaboration plus fluide, tout en offrant une architecture qui répond aux besoins modernes en matière de développement et de performance.

## **5.2 Inconvénients:**

Bien que l'architecture MVC offre des avantages notables, elle comporte aussi des inconvénients qui peuvent affecter le développement dans certains contextes. D'abord, la séparation stricte des composants peut introduire une complexité accrue dans la structure du code, rendant parfois l'architecture plus complexe et difficile à suivre pour des projets de petite ou moyenne envergure. La maintenance de trois composants distincts – le Modèle, la Vue et le Contrôleur – peut sembler excessive dans des cas où une modularité aussi rigoureuse n'est pas nécessaire, augmentant ainsi la quantité de code à gérer. Cette séparation exige aussi un effort supplémentaire en termes de coordination, car chaque fonctionnalité doit être distribuée entre les composants, allongeant ainsi le temps de développement.

De plus, les dépendances entre le Contrôleur et les autres composants peuvent devenir compliquées, surtout lorsque le Contrôleur se trouve surchargé par la coordination de la logique métier et de l'interface utilisateur. Cela peut rendre le code plus difficile à maintenir et à déboguer au fur et à mesure que l'application évolue. Sur le plan des tests, même si le Modèle est relativement facile à tester de façon indépendante, la Vue et le Contrôleur posent des défis particuliers. La Vue étant étroitement liée à l'interface utilisateur, elle nécessite souvent des outils spécifiques pour être testée efficacement, tandis que le Contrôleur, en raison de sa fonction de coordination, peut compliquer la testabilité de l'application.

Enfin, pour les développeurs moins expérimentés, la compréhension et la mise en œuvre de l'architecture MVC peuvent s'avérer déroutantes et nécessiter une courbe d'apprentissage plus importante. A cause de cela, la mise en place de cette architecture m'a demandé un temps de travail considérable. En effet, cette architecture impose une certaine discipline dans la séparation des responsabilités, et ce besoin de structuration rigoureuse peut ralentir le projet, surtout au départ. En somme, bien que MVC offre structure et modularité, elle peut introduire une complexité excessive, des dépendances lourdes et une augmentation du temps de développement et de test, ce qui en limite parfois la pertinence pour certains projets.

### **5.3 mise en pratique dans l'application PCorderApplication:**

J'ai organisé le projet en trois packages principaux – controller, model, et view – chacun ayant un rôle spécifique pour assurer la séparation des préoccupations, la maintenabilité, et l'évolutivité de l'application. Cette organisation permet une séparation claire des responsabilités, rendant l'application plus modulaire, facile à maintenir, et propice aux évolutions futures. Le package `model` gère la logique métier et la gestion des données, tandis que le package `view` se concentre sur l'interface utilisateur, facilitant les interactions entre l'application et ses utilisateurs. Voici une description enrichie de cette structure. Voici une description détaillée de la manière dont j'ai implémenté cette architecture.

#### **5.3.1. Controller:**

le package controller joue un rôle central en assurant la médiation entre la logique métier et l'interface utilisateur, chaque contrôleur étant dédié aux besoins d'un rôle spécifique dans l'application. Cette approche modulaire permet de maintenir un code bien organisé et de gérer les responsabilités de chaque rôle de manière isolée. Voici comment j'ai structuré les différents contrôleurs pour optimiser la gestion des fonctionnalités :

-AdministratorController : Ce contrôleur est spécialement conçu pour prendre en charge les opérations d'administration. En regroupant les fonctionnalités critiques de gestion des utilisateurs – telles que l'ajout, la suppression, et la modification de profils – dans cette classe, j'assure une séparation rigoureuse des droits d'accès. Ce contrôleur est essentiel pour garantir que seules les actions autorisées sont exécutées par les administrateurs, en renforçant la sécurité et le contrôle d'accès de l'application. De plus, en centralisant ces responsabilités, j'ai simplifié la maintenance des fonctionnalités d'administration, permettant de facilement adapter ou étendre les droits et permissions si nécessaire.

-AssemblerController : Ce contrôleur gère les tâches propres aux assembleurs, offrant une organisation dédiée pour la gestion du traitement des commandes.

L'AssemblerController permet aux assembleurs de valider des commandes, de marquer l'avancement des étapes d'assemblage, et de signaler la préparation des commandes pour la livraison. En structurant ces actions dans une classe indépendante, j'assure une gestion efficace du processus d'assemblage, facilitant les ajouts et modifications des fonctionnalités liées aux assembleurs sans impact sur les autres rôles.

-RequesterController : Conçu pour les utilisateurs qui soumettent des demandes, le RequesterController centralise les fonctionnalités nécessaires pour les demandeurs



(requesters) dans l'application. Ces utilisateurs peuvent passer de nouvelles commandes, suivre le statut de celles-ci, et même annuler des demandes en cours si besoin. En ayant un contrôleur dédié pour ce rôle, j'ai pu organiser le flux de travail des demandeurs de manière structurée et intuitive, assurant que toutes les actions spécifiques aux demandes soient gérées de manière isolée. Cela améliore la lisibilité du code et facilite les modifications futures pour le rôle des demandeurs.

-StoreKeeperController : Ce contrôleur est destiné aux magasiniers (StoreKeepers), regroupant toutes les actions liées à la gestion de l'inventaire. Grâce au StoreKeeperController, les magasiniers ont la possibilité d'ajouter de nouveaux composants, de mettre à jour les quantités, de supprimer des éléments du stock, et de vérifier les niveaux d'inventaire en temps réel. En structurant les fonctionnalités d'inventaire dans une classe dédiée, j'ai pu optimiser l'efficacité de la gestion des stocks et garantir une navigation fluide et organisée pour les magasiniers. Ce contrôleur permet une évolutivité accrue en cas de modifications de la gestion d'inventaire, tout en limitant l'impact sur les autres rôles.

Grâce à cette structure, chaque rôle dispose d'un accès limité aux fonctionnalités qui lui sont spécifiques, ce qui renforce l'organisation et la flexibilité du projet.

### **5.3.2.Model:**

Dans mon projet, le package model constitue le noyau central de l'application, regroupant l'ensemble des composants relatifs aux données et à la logique métier. Pour assurer une organisation optimale et modulaire, j'ai segmenté ce package en différents sous-packages, chacun traitant un aspect spécifique du modèle.

-database : Ce sous-package contient AppDatabase, qui sert de point d'accès principal pour la connexion et la configuration de la base de données. Cette classe facilite l'intégration de SQLite au sein de l'application, garantissant une gestion cohérente des opérations de stockage et de récupération des données. DatabaseSQLite joue également un rôle essentiel en centralisant les opérations SQL telles que l'insertion, la mise à jour, et la suppression. En regroupant toutes les interactions de ce type, j'ai optimisé l'efficacité et la sécurité des opérations sur les données, assurant un accès fiable et performant aux informations stockées.

-tools : Dans ce sous-package, j'ai inclus des classes comme AccessLocal, UserRepository, et OrderRepository, qui sont cruciales pour la gestion des données. AccessLocal facilite le stockage et la récupération d'informations en local, sans besoin de connexion à un serveur distant. UserRepository se concentre sur la gestion des utilisateurs, tandis que OrderRepository est spécialement conçu pour gérer les commandes. En attribuant à chaque type de données son propre repository, j'ai structuré l'accès aux données de manière organisée et sécurisée, permettant des modifications futures ou des ajouts de fonctionnalités sans complexifier l'architecture.

-entity : Ce sous-package regroupe les entités principales du projet, telles que Component, OrderRepository, et UserInfo. La classe Component représente les divers éléments de l'inventaire, qu'il s'agisse de matériel ou de logiciel. OrderRepository est dédié aux opérations de gestion des commandes, et UserInfo sert à stocker des informations essentielles sur les utilisateurs. Cette structure d'entités offre une base de données d'objets bien définie et permet une gestion plus fluide des informations, simplifiant ainsi la manipulation des données pour chaque entité.

-interfaces : Ce sous-package contient les interfaces Authenticable et Manageable, qui enrichissent le modèle en ajoutant des fonctionnalités spécifiques. Authenticable définit les méthodes d'authentification de base pour les utilisateurs, tandis que Manageable regroupe des méthodes génériques de gestion pour certaines entités. Grâce à ces interfaces, j'ai pu maintenir une certaine flexibilité et cohérence dans le code, facilitant l'extension des fonctionnalités tout en respectant des comportements standardisés au sein du modèle.

-orders : Ce sous-package inclut les classes Order et OrderStatus, qui modélisent les commandes dans l'application. Order définit les caractéristiques de base et le comportement d'une commande, tandis que OrderStatus permet de suivre et de gérer l'évolution de chaque commande. En structurant les données de cette manière, j'ai assuré une traçabilité détaillée et facilité la gestion de chaque commande tout au long de son cycle de vie, offrant une vue d'ensemble structurée des transactions.

-users : Enfin, dans le sous-package users, j'ai organisé les différents rôles utilisateurs de l'application, incluant Administrator, Assembler, Requester, StoreKeeper, et User. Chaque rôle est implémenté comme une classe distincte, permettant d'attribuer des attributs et des permissions spécifiques en fonction des besoins de chaque utilisateur. Par exemple, Administrator dispose de privilèges étendus pour gérer les utilisateurs, tandis que StoreKeeper se concentre sur les opérations d'inventaire. Cette organisation facilite la gestion des autorisations et permet de structurer les actions en fonction des profils, optimisant ainsi la sécurité et l'efficacité des processus.

### **5.3.3. View:**

Le package view joue un rôle essentiel en se consacrant à l'affichage visuel de l'application et à la conception de l'interface utilisateur. Ce package est conçu pour fournir une expérience de navigation intuitive, permettant aux utilisateurs d'interagir efficacement avec le système tout en accédant aux informations de manière claire et structurée. J'ai divisé ce package en sous-packages et classes spécifiques pour répondre aux différents besoins d'affichage et d'interaction de chaque rôle utilisateur.

-adapter : Le sous-package adapter comprend OrderAdapter, une classe clé dans la gestion de la présentation des commandes. OrderAdapter organise l'affichage des commandes sous forme de liste ou de tableau, garantissant que chaque commande est présentée de façon ordonnée et facile à lire. En utilisant cet adaptateur, j'ai optimisé la lisibilité et la clarté des informations de commande, facilitant l'interaction des utilisateurs.

avec les données. Ce composant assure une présentation homogène et professionnelle des commandes, permettant aux utilisateurs de visualiser les détails de chaque commande rapidement et de manière intuitive. L'adaptateur sert également de lien entre les données et l'affichage, simplifiant l'intégration des données de commande dans l'interface utilisateur.

-activities : Le package view comprend plusieurs classes d'activités spécifiques à chaque type d'utilisateur, notamment AdministratorActivity, AssemblerActivity, MainActivity, RequesterActivity, et StoreKeeperActivity. Chaque activité est soigneusement conçue pour fournir une interface adaptée aux besoins et aux responsabilités du rôle de l'utilisateur. Par exemple, AdministratorActivity offre aux administrateurs des outils de gestion des utilisateurs, tandis que StoreKeeperActivity présente une vue centrée sur la gestion de l'inventaire, permettant aux magasiniers de contrôler et d'organiser les composants disponibles en stock. En connectant chaque activité aux contrôleurs respectifs, j'ai veillé à ce que chaque utilisateur puisse accéder aux fonctionnalités spécifiques à son rôle de manière fluide et intuitive, assurant une expérience utilisateur homogène.

Chaque activité dans le package view est développée pour améliorer la clarté de l'interface, en présentant uniquement les options et informations pertinentes pour le rôle de l'utilisateur, ce qui minimise la confusion et renforce l'efficacité opérationnelle. Cette conception orientée utilisateur permet à chaque rôle d'interagir avec l'application selon ses besoins spécifiques, en créant une séparation visuelle qui améliore la productivité. En intégrant ces éléments d'affichage, j'ai rendu l'application plus accueillante et fonctionnelle, garantissant une expérience utilisateur optimisée à chaque étape de navigation et d'interaction

## **6. La Gestion de la base de donnée:**

### **4. Développement et Soumission du Projet Android Studio:**

#### **Capture du Projet dans Android Studio**

### **5. Développement et Soumission du Projet Android Studio:**

#### **6.Fonctionnalités de Connexion et Rôles des Utilisateurs**

#### **7. Création d'Utilisateurs par l'Administrateur**

#### **8.Validation des Champs Utilisateur et Messages d'Erreur**

#### **9. (Optionnel) Gestion de la Base de Données**

#### **10.Conclusion:**

## **Livrable2**

### **1.Introduction**

Dans ce second livrable, l'objectif est de concevoir et d'intégrer toutes les fonctionnalités dédiées au rôle de StoreKeeper dans l'application PCOrderApplication. Ce rôle, essentiel dans le cadre de la gestion de l'inventaire, garantit une organisation précise et une

supervision efficace des stocks de composants matériels et logiciels pour l'entreprise. En fournissant une interface structurée et des fonctionnalités avancées, le rôle de StoreKeeper répond aux besoins de suivi, de mise à jour et de consultation des ressources disponibles, couvrant aussi bien les équipements physiques (comme les boîtiers, cartes mères, dispositifs de mémoire) que les logiciels (navigateurs, suites bureautiques, et environnements de développement).

Les StoreKeepers sont ainsi dotés d'un accès complet aux opérations d'inventaire, avec des responsabilités spécifiques concernant la gestion de chaque composant stocké. Chacun de ces éléments doit être enregistré avec des attributs fondamentaux : un type (matériel ou logiciel), un sous-type (boîtier, carte mère, etc.), un titre descriptif unique, une quantité modifiable, et des métadonnées de traçabilité comprenant les dates et heures de création et de modification, non modifiables pour garantir l'intégrité des données. Ce rôle inclut également la capacité d'ajouter, modifier, supprimer et visualiser chaque composant dans l'inventaire. Pour une vue d'ensemble optimisée, les composants sont affichés dans une interface tabulaire, excluant les champs de détails comme les commentaires et les dates de traçabilité pour une lecture rapide et efficace de l'inventaire en cours.

Ce livrable introduit aussi des fonctionnalités supplémentaires afin de simplifier la gestion de l'inventaire et des utilisateurs, avec une fonction d'initialisation de données permettant de charger des éléments de stock ou des profils de Requesters directement à partir de fichiers externes, en veillant à éviter les duplications. De plus, le rôle d'Administrator bénéficie de nouvelles fonctionnalités pour gérer la base de données de manière avancée et sécurisée, incluant une option pour vider complètement la base de données (supprimant tous les éléments de stock et les utilisateurs Requesters), réinitialiser la base en chargeant une liste préexistante de Requesters et de composants ou réinitialiser uniquement le stock de composants pour remettre à jour l'inventaire.

Dans cette version étendue de PCOrderApplication, l'intégration d'une base de données devient obligatoire afin d'assurer la persistance des données, un élément crucial pour la gestion efficace de l'inventaire et des utilisateurs en temps réel. Cette base de données garantira la fiabilité et la cohérence des informations stockées, assurant ainsi une gestion sécurisée et conforme aux besoins de l'entreprise.

Le dépôt GitHub de l'équipe sera mis à jour avec l'ensemble du projet Android Studio, incluant l'APK de démonstration, le fichier README détaillant les informations essentielles du projet, la version actualisée du modèle conceptuel en formats PDF et source, et un tag « deliverable-2 » associé au commit final correspondant à ce livrable. Cette mise à jour garantit que toutes les parties prenantes peuvent accéder facilement aux éléments requis pour examiner et valider les fonctionnalités livrées.

Avec cette intégration, PCOrderApplication franchit une nouvelle étape vers la mise en place d'un outil de gestion de stock robuste et polyvalent, en parfaite adéquation avec les exigences actuelles de performance, de sécurité, et de flexibilité dans le secteur technologique

## **2. Création et Configuration de l'Équipe GitHub:**

Voir la même partie du livrable 1 ci-dessus.

### 3. Version du Modèle Conceptuel

### 4. Développement et Soumission du Projet Android Studio:

#### Structure du Projet Android:

Tag « deliverable-2 »

### 4. Fonctionnalités du Rôle « StoreKeeper »

#### Liste Tabulaire des Composants

#### Interface de Gestion de Stock

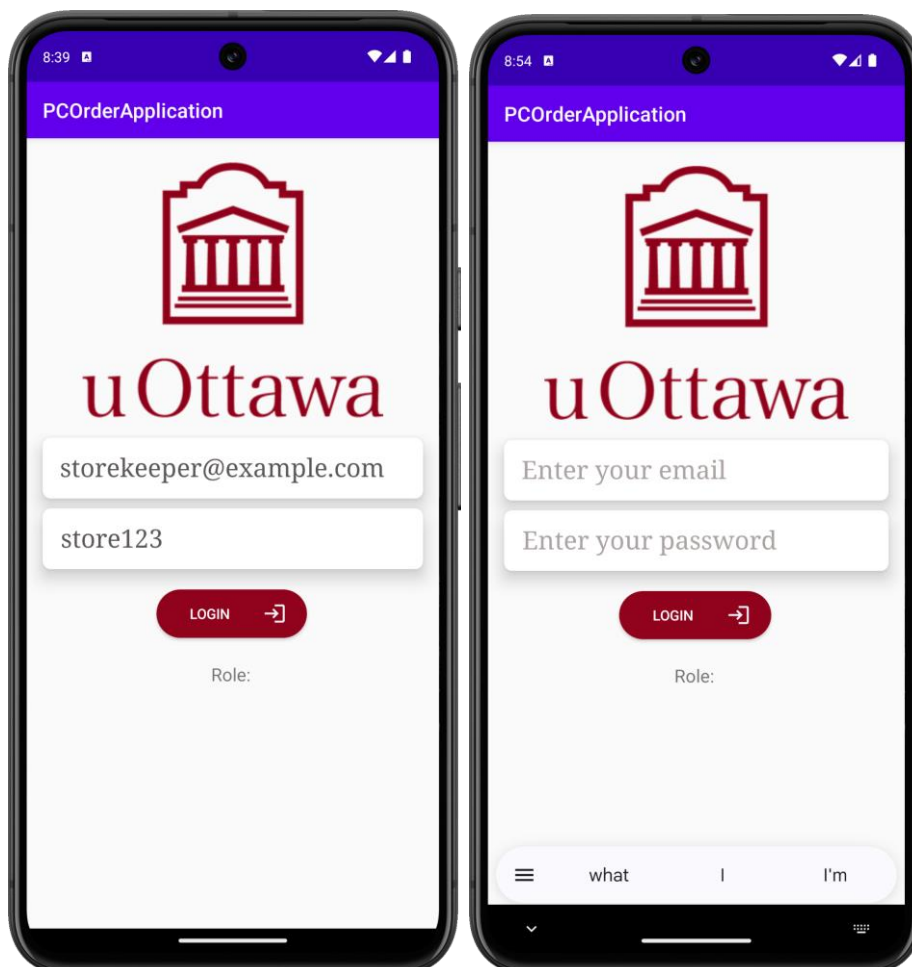
### 5.5. Fonctionnalités Administratives : Suppression et Importation de Données

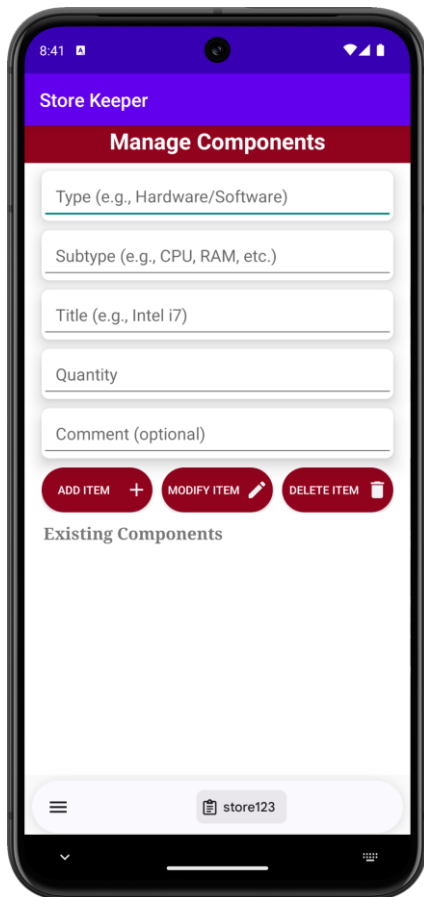
### 6. Validation des Champs et Messages d'Erreur

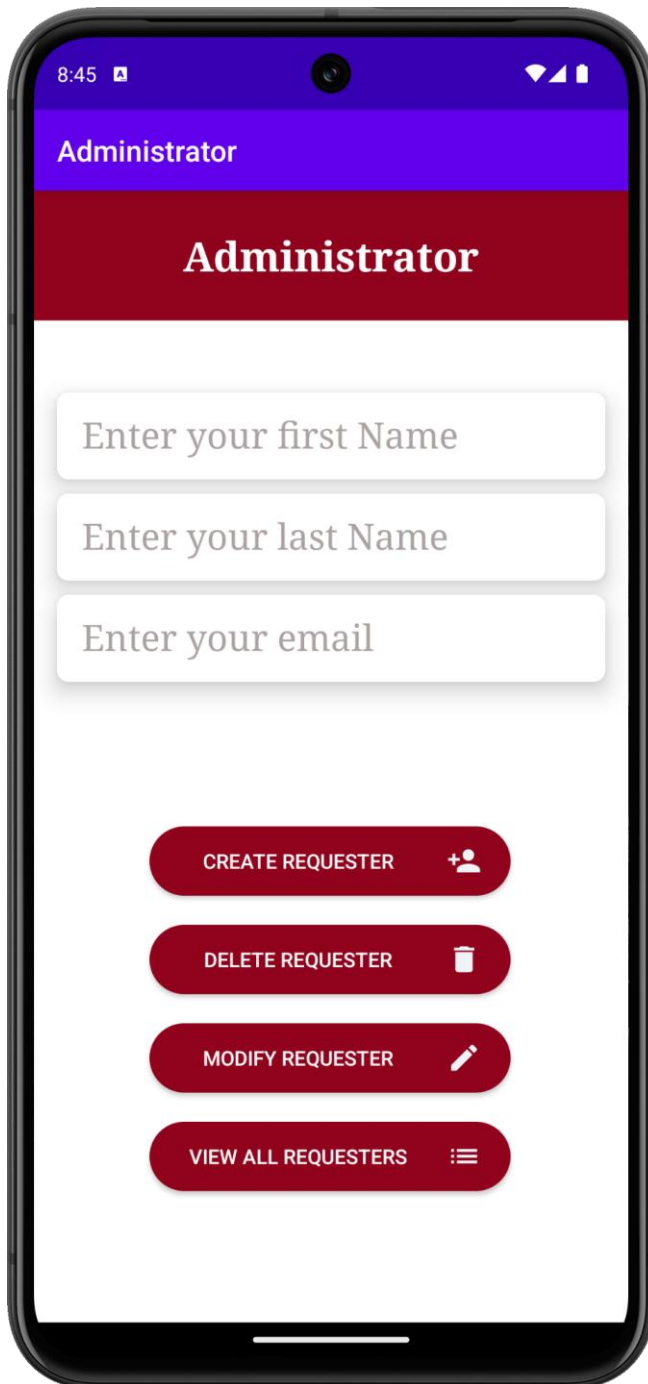
### 7.7. scénario d'utilisation:

#### 1.1. Page d'accueil

la première image représente juste un exemple enregistré par défaut pour permettre

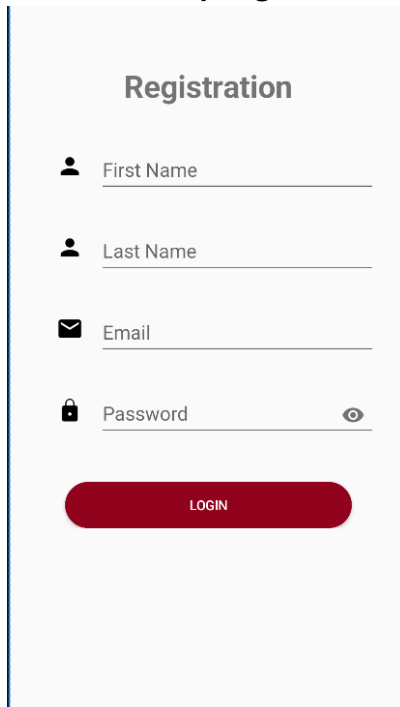






### Livrable 3 : Requester :

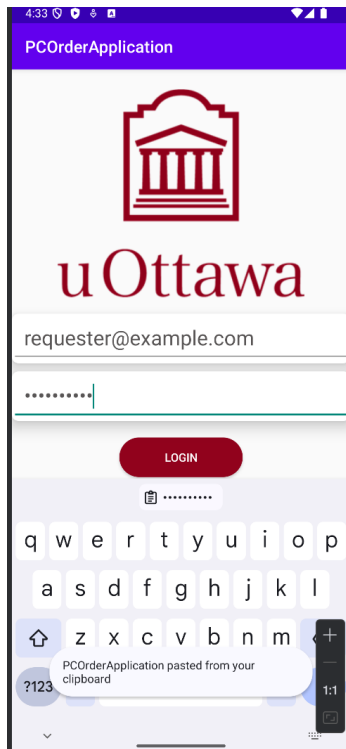
En premier, une nouvelle Page d'inscription a été ajoutée où l'utilisateur peut s'authentifier par gmail et mot de passe (cet utilisateur ne peut être qu'un requester)

A registration form with a light gray background and a dark blue border. At the top, the word "Registration" is centered in a bold, dark gray font. Below it, there are four input fields, each preceded by a small icon: a person icon for "First Name", another person icon for "Last Name", an envelope icon for "Email", and a padlock icon for "Password". The "Password" field has a small eye icon to its right for toggling visibility. At the bottom of the form, there is a red, rounded rectangular button with the word "LOGIN" in white, uppercase letters.

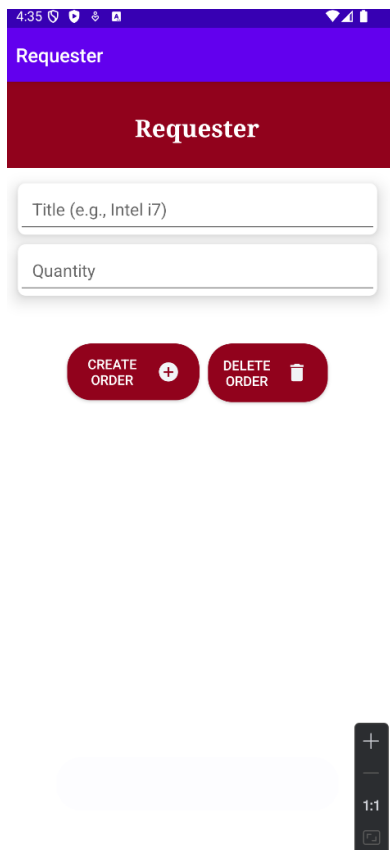
En conséquence une classe LoginController a été implémenté (dans le package controller)

Le role Requester a été complété aussi ainsi que RequesterActivity et RequesterController





**Vu que pour le moment l'Inscription est impossible le code mainActivity comporte un requester fait par défaut : le code mainActivity comporte un requester fait par défaut**



Ici le requester rentre un article qui existe dans la base données (Storekeeper dans page dispose de tous les articles du magasin et peut les modifier).

Pour le diagramme de classe c'est principalement les classes Requester, RequesterActivity, RequesterController et LoginController qui ont été modifié, voici le nouveau diagramme de classe

Les autres diagrammes quant à eux restent les meme

Prière de voir les nouveauPNG dna le doissier (Livrable 3 et Diagrammes et copie de l'APK)

L'APK se trouve dans son emplacement d'origine, mais je l'ai copié pour qu'il soit accessible

## Annexe1: Code UML pour diagramme de classe

@startuml

' Interfaces

```
interface Authenticable {  
    + login()  
    + logout()  
}
```

```
interface Manageable {  
    + create()  
    + modify()  
    + delete()  
}
```

' Classe abstraite User

```
abstract class User {  
    - String firstName  
    - String lastName  
    - String email  
    - String password  
    - LocalDateTime creationDateTime  
    - LocalDateTime modificationDateTime  
    - boolean isLoggedIn  
    - int id  
    + login(String email, String password): boolean  
    + logout()  
    + getFirstName(): String  
    + getLastName(): String  
    + getEmail(): String  
    + getPassword(): String  
    + getId(): int  
    + isLoggedIn(): boolean  
    + setFirstName(String)  
    + setLastName(String)  
    + setEmail(String)  
    + setPassword(String)  
    + setId(int)  
}
```

User ..|> Authenticable

' Classes dérivées de User

```
class Administrator {  
    + createRequester(String firstName, String lastName, String email, String password)
```

```

    + modifyRequester(String newFirstName, String newLastName, String newEmail)
    + deleteRequester(String firstName, String lastName, String email)
}

```

```

class StoreKeeper {
    + addComponent(Component component)
    + removeComponent(Component component)
    + viewStock(): List<Component>
}

```

```

class Assembler {
    + acceptOrder(Order order)
    + rejectOrder(Order order, String message)
    + completeOrder(Order order)
    + viewAllOrders(): List<Order>
    + addOrder(Order order)
}

```

```

class Requester {
    - List<Order> orders
    + createOrder(List<Component> components)
    + viewOwnOrders(): List<Order>
    + deleteOrder(int orderId)
}

```

```

User <|-- Administrator
User <|-- StoreKeeper
User <|-- Assembler
User <|-- Requester

```

' Updated Controllers

```

class LoginController {
    - AccessLocal accessLocal
    - UserRepository userRepository
    - UserInfo currentUser
    + login(String email, String password): boolean
    + logout()
    + isUserLoggedIn(): boolean
    + getCurrentUser(): UserInfo
}

```

```

class MainController {
    - static MainController instance
    - Context appContext
    - LoginController loginController
    + getInstance(Context context): MainController
    + loginUser(String email, String password): boolean
    + logoutUser()
}

```

```
+ navigateToRoleActivity()  
}
```

```
class RequesterController {  
    - static RequesterController instance  
    - static Requester requester  
    - static AccessLocal accessLocal  
    + getInstance(Context, String, String, String, String): RequesterController  
    + createOrder(List<Component> components): boolean  
    + deleteOrder(int orderId)  
    + viewOwnOrders(): List<Order>  
    + loginRequester()  
    + logoutRequester()  
    + requestComponent(String title): List<Component>  
}
```

' Classes de modèle

```
class Component {  
    - String type  
    - String subtype  
    - String title  
    - int quantity  
    - String comment  
    - String creation_Date  
    - String modification_date  
    + getters et setters  
}
```

```
class Order {  
    - int id  
    - OrderStatus status  
    - String message  
    - LocalDateTime creationDateTime  
    - LocalDateTime modificationDateTime  
    - Requester requester  
    - List<Component> components  
    + addComponent(Component component)  
    + removeComponent(Component component)  
    + updateStatus(String newStatus)  
    + getId(): int  
    + getStatus(): OrderStatus  
    + getMessage(): String  
    + setMessage(String)  
    + getCreationDateTime(): LocalDateTime  
    + getUserId(): int  
}
```

```
class OrderStatus {
```

```

- String status
- LocalDateTime updatedAt
+ setStatus(String newStatus)
+ validateTransition(String newStatus): boolean
+ getStatus(): String
+ getUpdatedAt(): LocalDateTime
}

```

```

class UserInfo {
- int id
- String firstName
- String lastName
- String email
- String password
- String role
- LocalDateTime createdAt
- LocalDateTime modifiedAt
+ getters et setters
}

```

' Repositories

```

class UserRepository {
- DatabaseSQLite dbHelper
+ insertUser(UserInfo user)
+ findUserByEmail(String email): UserInfo
+ updateUser(UserInfo user)
+ getAllUsers(): List<UserInfo>
}

```

```

class OrderRepository {
- DatabaseSQLite dbHelper
+ insertOrder(Order order)
+ updateOrder(Order order)
+ deleteOrder(Order order)
+ getAllOrders(): List<Order>
}

```

```

class ComponentRepository {
- List<Component> componentDatabase
+ updateComponent(Component component)
+ deleteComponent(Component component)
+ getAllComponents(): List<Component>
}

```

' Classes de base de données

```

class DatabaseSQLite {
- String DATABASE_NAME
- int DATABASE_VERSION
}

```

```

+ onCreate(SQLiteDatabase db)
+ onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
+ getBd(): SQLiteDatabase
}

```

```

class AccessLocal {
- SQLiteDatabase bd
- DatabaseSQLite databaseSQLite
+ addComponent(Component component): long
+ findComponentByTitle(String title): Component
+ getAllComponents(): List<Component>
+ upload(): ArrayList<String>
}

```

```

class AppDatabase {
- UserRepository userRepository
- OrderRepository orderRepository
- ComponentRepository componentRepository
- DatabaseSQLite dbHelper
+ getInstance(Context context): AppDatabase
+ getDbHelper(): DatabaseSQLite
+ getUserRepository(): UserRepository
+ getOrderRepository(): OrderRepository
+ getComponentRepository(): ComponentRepository
}

```

' Updated Views

```

class MainActivity {
- ActivityMainBinding binding
+ onCreate(Bundle savedInstanceState)
+ authenticateUser(String email, String password)
+ handleUserRole(String role)
+ showErrorMessage()
}

```

```

class RequesterActivity {
- Requester requester
- TextView ordersListTextView
+ onCreate(Bundle savedInstanceState)
+ createOrder()
+ displayOrders()
+ deleteOrder()
}

```

' Relations entre les classes avec multiplicités

UserRepository --> "1" DatabaseSQLite

OrderRepository --> "1" DatabaseSQLite

ComponentRepository --> "1" AccessLocal

AccessLocal --> "1" DatabaseSQLite

AppDatabase --> "1" UserRepository

AppDatabase --> "1" OrderRepository

AppDatabase --> "1" ComponentRepository

UserRepository --> "0..\*" UserInfo

OrderRepository --> "0..\*" Order

ComponentRepository --> "0..\*" Component

Order "1" --> "1" Requester

Requester "1" --> "0..\*" Order

Order "1" \*-- "1" OrderStatus

Order "1" --> "0..\*" Component

Component "1" --> "0..\*" Order

MainController --> LoginController

LoginController --> UserRepository

RequesterController --> "1" Requester

RequesterActivity --> RequesterController

@enduml

@enduml

## **Annexe2: Diagrammes d'activité:**

### **1-pour la connexion d'un utilisateur:**

@startuml

start

:User opens the application;

:User enters email and password;

if (Are email and password valid?) then (Yes)

  :Authenticate user via MainController;

  if (User's role?) then

    -> [Administrator]

    :Initialize LoginController;

    :Redirect to AdministratorActivity via MainController;

    stop

    -> [StoreKeeper]

    :Initialize LoginController;

    :Redirect to StoreKeeperActivity via MainController;

    stop

    -> [Assembler]

    :Initialize LoginController;

    :Redirect to AssemblerActivity via MainController;

    stop

    -> [Requester]

    :Initialize LoginController;



```

        :Redirect to RequesterActivity via MainController;
    stop
endif
else (No)
    :Display error message in MainActivity;
endif
stop
@enduml
-pour la création d'une commande par un Requester:

```

```

@startuml
start
:Requester logs in via MainController;
:RequesterController initializes;
:Requester selects "Create Order" in RequesterActivity;
:Create a new Order object;
repeat
    :Select a Component;
    :Add Component to Order;
repeat while (More components to add?)
:Submit Order via RequesterController;
:Order saved in OrderRepository;
stop
@enduml
-pour le traitement d'une commande par un assembleur :
@startuml
start
:Assembler logs in via MainController;
:AssemblerController initializes;
:Assembler views all orders using viewAllOrders() in AssemblerActivity;
if (Orders available?) then (Yes)
    :Select an Order;
    :Examine Order details;
    if (Accept Order?) then (Yes)
        :Call acceptOrder(Order) in AssemblerController;
        :Update Order status to "Accepted" in OrderRepository;
        :Process the Order;
        :Call completeOrder(Order) in AssemblerController;
        :Update Order status to "Completed" in OrderRepository;
    else (No)
        :Call rejectOrder(Order, message) in AssemblerController;
        :Update Order status to "Rejected" in OrderRepository;
        :Provide rejection message;
    endif
endif
else (No)
    :No orders to process;
endif

```

```

stop
@enduml-Diagramme d'activité pour la gestion des composants par le Storekeeper:
@startuml
start
:Magasinier se connecte;
repeat
    :Choisir une action;
    if (Action == "Ajouter un composant") then
        :Saisir les détails du composant;
        :Ajouter le composant à la base de données;
    elseif (Action == "Supprimer un composant") then
        :Sélectionner le composant à supprimer;
        :Retirer le composant de la base de données;
    elseif (Action == "Consulter le stock") then
        :Afficher la liste des composants;
    endif
repeat while (Autres actions?)
stop
@enduml

```

**-Diagramme d'activité pour la création d'un demandeur par l'administrateur :**

```

@startuml
start
:Administrateur se connecte;
:Choisir "Créer un demandeur";
:Entrer les informations du demandeur;
if (Informations valides?) then (Oui)
    :Créer le compte du demandeur;
    :Notifier le succès de la création;
else (Non)
    :Afficher les erreurs;
endif
stop
@enduml

```