# Dossier de conception : Fraicheur Admin App

Mai 2025

## Table des matières

# 1 Présentation du besoin métier

## 1.1 Contexte

L'application "Fraicheur Admin App" s'inscrit dans le cadre d'un projet de plateforme de vente en ligne. Il propose un large catalogue de produits. La gestion quotidienne d'une telle plateforme de vente implique des défis administratifs complexes, notamment la gestion des commandes clients, la mise à jour du catalogue de produits, et la gestion des stocks. Face à ces enjeux, une application mobile d'administration est devenue indispensable. Le projet vise à simplifier les processus administratifs, à améliorer la réactivité des administrateurs et à fournir les outils nécessaires pour optimiser la gestion des stocks et des commandes.

## 1.2 Objectifs du projet

L'application "Fraicheur Admin App" doit permettre aux administrateurs de :

- Suivre l'état des commandes en temps réel, incluant la validation, le marquage comme expédiées et l'annulation si nécessaire.
- Gérer le catalogue de produits, permettant d'ajouter, de modifier ou de supprimer des articles, tout en garantissant que les informations (prix, descriptions, images) sont à jour et correctement affichées.
- Gérer les niveaux de stocks, en recevant des alertes lorsque les stocks sont bas et en ajustant les quantités disponibles en fonction des ventes et des réapprovisionnements.
- Analyser les statistiques de vente et les performances des produits pour optimiser les offres et les stratégies commerciales.
- Assurer la sécurité des données de l'entreprise grâce à un système d'authentification sécurisé, limitant l'accès aux seuls administrateurs autorisés.

## 1.3 Besoins fonctionnels

L'application "Fraicheur Admin App" doit offrir les fonctionnalités principales suivantes :

1. **Tableau de bord administrateur** : Affichage d'une vue d'ensemble des commandes en attente, des niveaux de stock, et des statistiques de vente pertinentes.
2. **Gestion du catalogue de produits** : Permettre l'ajout, la modification et la suppression de produits dans le catalogue.
3. **Gestion des commandes** : Permettre l'affichage, la validation, l'expédition ou la suppression des commandes en attente.
4. **Gestion des stocks** : Afficher les niveaux de stock actuels et permettre les ajustements nécessaires.
5. **Sécurisation des accès** : Implémenter un système d'authentification sécurisé pour garantir l'accès exclusif aux administrateurs.

## 1.4 Bénéfices attendus

L'objectif global est de faciliter la gestion quotidienne de l'administrateur de la plateforme de vente en ligne, tout en améliorant l'efficacité opérationnelle et la réactivité face aux demandes des clients. La mise en œuvre de l'application "Fraicheur Admin App" devrait apporter les bénéfices suivants :

- **Amélioration de l'efficacité** : L'application permettra d'automatiser et de centraliser de nombreuses tâches administratives, ce qui se traduira par une amélioration significative de la productivité de l'équipe administrative.
- **Meilleure gestion des stocks** : En offrant un aperçu en temps réel des niveaux de stock, l'administrateur pourra anticiper les besoins de réapprovisionnement, évitant ainsi les ruptures ou les surstocks.
- **Suivi des performances commerciales** : Grâce à des statistiques et rapports de ventes, l'entreprise pourra mieux comprendre les tendances du marché et ajuster son offre de produits en conséquence.
- **Sécurisation des données** : L'application garantira que seules les personnes autorisées pourront accéder aux informations sensibles, renforçant ainsi la sécurité globale des données de l'entreprise.

# 2 Présentation de l'architecture du projet Fraicheur Admin App

## 2.1 Architecture 4-tiers

L'architecture choisie pour le projet "Fraicheur Admin App" est une architecture 4-tiers (ou 4 couches), conçue pour structurer les applications web et mobiles de manière modulaire, évolutive et maintenable. Cette approche sépare clairement les responsabilités et facilite le développement et la maintenance.

Voici l'explication des différentes couches :

### 2.1.1 Client (Tier 1 - Interface utilisateur)

Le Client représente l'interface avec laquelle l'utilisateur interagit. Dans le cas de "Fraicheur Admin App", il s'agit d'une application mobile développée avec **Flutter**. Cette couche est responsable de :

- L'affichage des données à l'utilisateur sous une forme compréhensible et interactive (par exemple, le catalogue de produits, les commandes, le tableau de bord).
- La capture des interactions de l'utilisateur (clics, saisies, etc.) et leur traduction en requêtes.
- L'envoi de requêtes HTTP (GET, POST, PUT, DELETE) vers l'API REST pour récupérer ou envoyer des données.
- La réception et l'interprétation des réponses HTTP de l'API, souvent au format JSON.

### 2.1.2 API REST (Tier 2 - Couche intermédiaire / Service Web)

L'API REST sert d'interface entre le client (application Flutter) et le serveur (logique métier et base de données). Elle est développée en **PHP** et est hébergée sur **WampServer**. Ses responsabilités incluent :

- La réception des requêtes HTTP du client.
- L'interprétation de ces requêtes et la transmission des appels aux fonctions appropriées sur le serveur.
- La gestion de la communication avec la base de données via le serveur (par exemple, récupération des produits, authentification des administrateurs).
- La renvoi des réponses au client dans un format exploitable, principalement JSON. Les exemples incluent `auth.php` pour l'authentification et `products.php` pour la récupération du catalogue.

### 2.1.3 Serveur (Tier 3 - Logique métier)

Cette couche contient la logique métier de l'application et est également gérée par l'environnement **PHP/Apache de WampServer**. Elle est responsable de :

- Le traitement des requêtes complexes qui nécessitent l'exécution de règles de gestion (par exemple, calcul des prix avec réduction, validation des données d'entrée, gestion des stocks).
- L'interaction avec la base de données pour lire, écrire, mettre à jour ou supprimer des informations (CRUD - Create, Read, Update, Delete).

- Le retour des résultats de ces opérations sous forme de réponses HTTP standardisées à l'API.

### 2.1.4 Base de données (Tier 4 - Stockage des données)

Cette couche est dédiée au stockage et à la gestion des données persistantes de l'application. Pour ce projet, la base de données est **MySQL**, gérée via **phpMyAdmin**. Elle contient toutes les informations essentielles telles que :

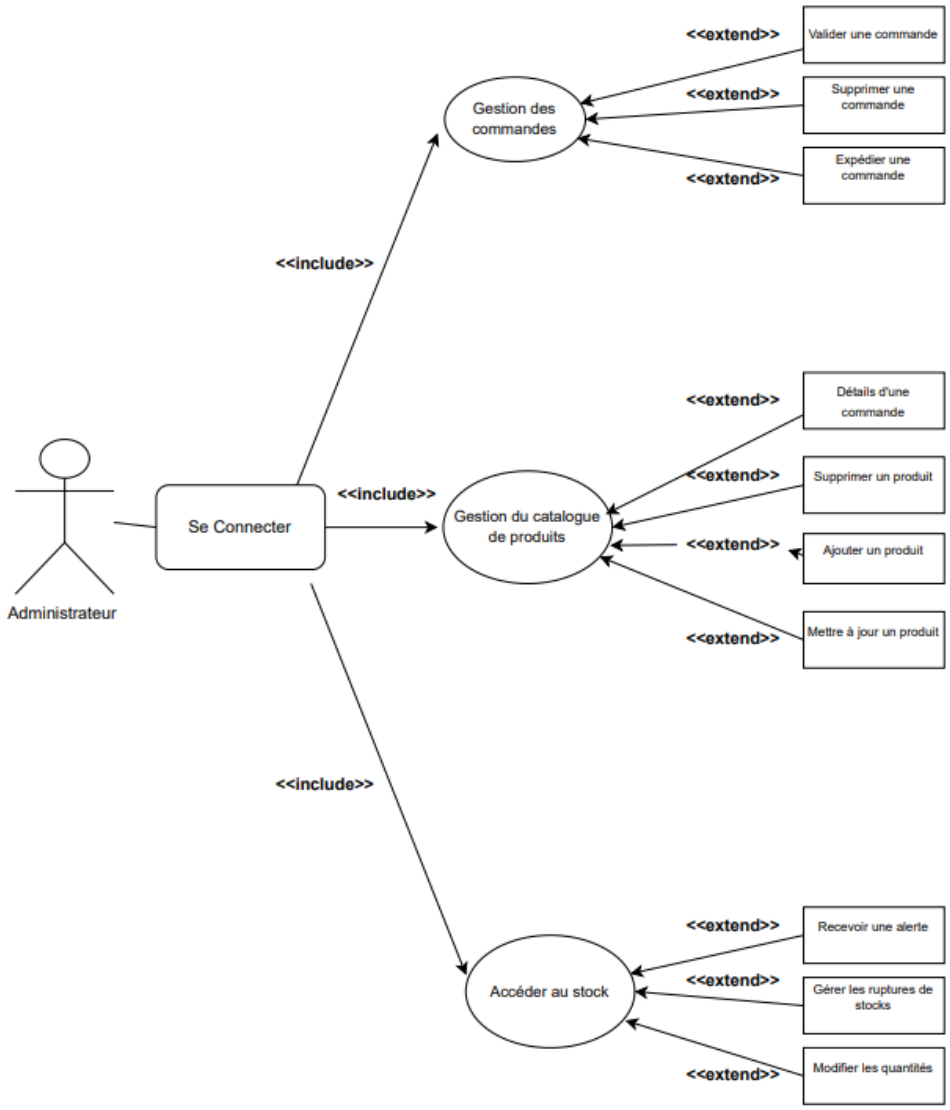
- Les informations sur les produits (désignation, description, prix, réductions, URLs d'images).
- Les données des utilisateurs (administrateurs, clients), y compris les informations d'authentification.
- Les informations sur les commandes (détails, statut, paiement, etc.).
- Les données de stock pour chaque produit.
- Les données relatives aux catégories ou types de produits.

Le serveur y accède directement pour récupérer ou enregistrer les informations, et les données récupérées sont ensuite renvoyées au serveur, qui les formate et les transmet au client via l'API.

# 3 Modélisation UML

La modélisation UML (Unified Modeling Language) est une étape cruciale dans la conception de systèmes logiciels. Elle permet de visualiser, spécifier, construire et documenter les artefacts d'un système. Pour le projet "Fraicheur Admin App", le diagramme de cas d'utilisation est particulièrement pertinent pour décrire les interactions fonctionnelles entre les acteurs (ici, l'administrateur) et le système.

Le diagramme de cas d'utilisation pour "Fraicheur Admin App" met en évidence les fonctionnalités clés offertes à l'administrateur, telles que définies dans les besoins fonctionnels (Section 1.3). Il représente les objectifs principaux de l'utilisateur par rapport au système.



# 4 Modélisation de la base de données

La base de données est le cœur de l'application "Fraicheur Admin App", assurant la persistance de toutes les informations essentielles concernant les produits, les catégories, les utilisateurs (administrateurs et clients), les commandes et les éléments de commande. La base de données est implémentée avec **MySQL** et est gérée via **phpMyAdmin**.

La modélisation de la base de données a été réalisée en respectant les principes de normalisation pour garantir l'intégrité, la cohérence et l'efficacité des données.

Représentation des tables :

- **ADMIN** :
    - `idAdmin` (Clé Primaire)
    - `nom`
    - `prenom`
    - `email`
    - `mdp` (mot de passe haché)
    - `image`
    - `tel`
    - `dateCreation`
    - `adresse`
    - `ville`
    - `codePostal`
    - `pays`
- **CLIENT** :
    - `idClient` (Clé Primaire)
    - `nom`
    - `prenom`
    - `email`
    - `mdp` (mot de passe haché)
    - `tel`
    - `adresse`
    - `ville`
    - `codePostal`
    - `pays`
- **PRODUIT** :
    - `idProduit` (Clé Primaire)
    - `designation`
    - `description`
    - `prix`

    - `reduction` (pourcentage de réduction)
    - `imageUrl` (chemin vers l'image du produit)
    - `idCategorie` (Clé Étrangère vers CATEGORIE)
    - `idAdmin` (Clé Étrangère vers ADMIN)
    - `quantiteStock`
- **CATEGORIE** :
    - `idCategorie` (Clé Primaire)
    - `nomCategorie`
- **COMMANDE** :
    - `idCommande` (Clé Primaire)
    - `dateCommande`
    - `statutCommande` (ex: En attente, Validée, Expédiée, Annulée)
    - `montantTotal`
    - `idClient` (Clé Étrangère vers CLIENT)
    - `idAdmin` (Clé Étrangère vers ADMIN, pour la validation/gestion)
- **ELEMENT_COMMANDE** :
    - `idElementCommande` (Clé Primaire)
    - `idCommande` (Clé Étrangère vers COMMANDE)
    - `idProduit` (Clé Étrangère vers PRODUIT)
    - `quantite`
    - `prixUnitaire`
- **LIVRAISON** :
    - `idLivraison` (Clé Primaire)
    - `dateLivraison`
    - `statutLivraison`
    - `adresseLivraison`
    - `idCommande` (Clé Étrangère vers COMMANDE)
- **PANIER** :
    - `idPanier` (Clé Primaire)
    - `idClient` (Clé Étrangère vers CLIENT)
    - `dateCreation`
- **ELEMENT_PANIER** :
    - `idElementPanier` (Clé Primaire)
    - `idPanier` (Clé Étrangère vers PANIER)
    - `idProduit` (Clé Étrangère vers PRODUIT)
    - `quantite`

**Relations entre les entités :**

- Un `ADMIN` peut gérer plusieurs `PRODUITs` et `COMMANDEs`.

- Un CLIENT peut passer plusieurs COMMANDEs et posséder un PANIER.
- Une CATEGORIE peut regrouper plusieurs PRODUITs.
- Une COMMANDE contient plusieurs ELEMENT_COMMANDEs.
- Une COMMANDE est associée à une LIVRAISON.
- Un PANIER contient plusieurs ELEMENT_PANIERs.
- Un ELEMENT_COMMANDE et un ELEMENT_PANIER sont liés à un PRODUIT.

# 5 Plan de sauvegarde de la base de données

## 5.1 Introduction

La continuité des opérations pour l'application "Fraicheur Admin App" dépend directement de la disponibilité et de l'intégrité de ses données. La base de données MySQL, contenant toutes les informations essentielles sur les produits, les commandes, les clients et les administrateurs, représente un actif critique. Ce plan de sauvegarde vise à définir une stratégie robuste pour protéger ces données contre les pertes accidentelles, les pannes matérielles, les erreurs logicielles ou les cyberattaques, assurant ainsi la résilience du système et la récupération en cas de sinistre.

## 5.2 Objectifs du plan de sauvegarde

Les objectifs principaux de ce plan de sauvegarde sont :

- **Assurer la récupération des données :** Permettre une restauration complète et rapide de la base de données à un état stable en cas de perte de données ou de corruption.
- **Minimiser les pertes de données :** Réduire au maximum la quantité de données perdues entre la dernière sauvegarde et l'incident (Objectif de Point de Récupération - RPO).
- **Réduire les temps d'arrêt :** Minimiser l'indisponibilité de l'application après un incident (Objectif de Temps de Récupération - RTO).
- **Maintenir l'intégrité des données :** S'assurer que les données restaurées sont exactes et cohérentes avec l'état souhaité.
- **Conformité :** Répondre aux éventuelles exigences réglementaires ou commerciales en matière de rétention et de protection des données.

## 5.3 Types de données à sauvegarder

### 5.3.1 Base de données

La base de données MySQL est la cible principale de ce plan de sauvegarde. Elle contient toutes les tables définies dans la section 4 (ADMIN, CLIENT, PRODUIT, CATEGORIE, COMMANDE, ELEMENT_COMMANDE, LIVRAISON, PANIER, ELEMENT_PANIER).

## 5.4 Fréquence des sauvegardes

La fréquence des sauvegardes est définie pour équilibrer la protection des données et les ressources nécessaires à la sauvegarde (espace disque, temps d'exécution).

### 5.4.1 Sauvegarde complète

- **Fréquence :** Une fois par semaine.
- **Moment :** Hors des heures de forte activité (par exemple, chaque dimanche à 02h00 du matin).
- **Contenu :** Copie intégrale de toute la base de données.
- **Outil :** Utilisation de `mysqldump` pour exporter la base de données en un fichier `.sql`.

### 5.4.2 Sauvegarde incrémentielle

- **Fréquence :** Quotidienne (du lundi au samedi).
- **Moment :** Chaque jour à 02h00 du matin, après la sauvegarde complète du dimanche.
- **Contenu :** Enregistre uniquement les changements intervenus depuis la *dernière* sauvegarde (complète ou incrémentielle).
- **Outil :** Nécessite une configuration spécifique de MySQL (par exemple, activation du binaire logging ou des outils comme Percona XtraBackup si le volume de données le justifie). Pour une configuration simple avec `mysqldump`, on pourrait se contenter de sauvegardes complètes quotidiennes pour des bases de petite taille si les incrémentielles sont complexes à mettre en place.

### 5.4.3 Sauvegarde différentielle

- **Fréquence :** Non prévue dans ce plan initial pour simplifier, l'incrémentielle étant privilégiée après la complète pour l'efficience. Si la complexité des restaurations incrémentielles devient un problème, une stratégie différentielle pourrait être envisagée (sauvegarde des changements depuis la *dernière sauvegarde complète*).

## 5.5 Outils de sauvegarde

### 5.5.1 Base de données

- **`mysqldump` :** C'est l'outil en ligne de commande standard pour MySQL. Il permet d'exporter une base de données ou un ensemble de bases de données sous forme de fichiers SQL.
    - Exemple de commande pour une sauvegarde complète :
      Bash

      ```
      mysqldump -u [utilisateur] -p[mot_de_passe] [nom_base_de_donnees] > [chemin_sauvegarde]/[nom_base_de_donnees]_complete_$(date +%Y%m%d%H%M%S).sql
      ```

- **phpMyAdmin :** Pour les sauvegardes manuelles ou ad-hoc, l'interface d'exportation de phpMyAdmin peut être utilisée pour exporter la base de données (section "Exporter").

## 5.6 Lieux de stockage des sauvegardes

Pour garantir la résilience et éviter la perte de données due à une défaillance physique du système principal, les sauvegardes seront stockées à plusieurs emplacements.

### 5.6.1 Stockage local

- Les sauvegardes complètes et incrémentielles seront initialement stockées sur un disque dur local distinct de celui où est installée la base de données principale. Cela permet une récupération rapide en cas de corruption logicielle ou d'erreur humaine.
- Chemin suggéré : `D:\Sauvegardes_FraicheurApp\` (ou un chemin similaire hors du dossier de WampServer).

#### 5.6.2 Serveurs GitHub (ou autre service de versioning/cloud)

- Les fichiers de sauvegarde de la base de données (notamment les `.sql`) seront poussés vers un dépôt privé sur GitHub ou un autre service de stockage cloud sécurisé (ex: Google Drive) après chaque sauvegarde complète et potentiellement les incrémentielles si leur taille est gérable. Cela offre une protection contre les sinistres locaux (incendie, vol, défaillance matérielle majeure du serveur).
- **Attention :** Ne jamais stocker de mots de passe ou de données sensibles directement dans des dépôts publics. Toujours utiliser des dépôts privés et des pratiques de sécurité appropriées.

### 5.7 Rétention des sauvegardes

La politique de rétention définit combien de temps les sauvegardes doivent être conservées.

#### 5.7.1 Sauvegardes complètes

- Conserver les 4 dernières sauvegardes complètes (environ un mois d'historique).
- Conserver la première sauvegarde complète de chaque trimestre pendant 1 an.

### 5.8 Gestion des incidents

En cas de perte de données ou de corruption de la base de données :

1. **Identifier la cause racine :** Déterminer la nature et l'étendue de l'incident.
2. **Arrêter l'application :** Mettre l'application "Fraicheur Admin App" hors ligne pour éviter d'aggraver la situation.
3. **Localiser la dernière sauvegarde valide :** Identifier la sauvegarde la plus récente et non corrompue (complète + incrémentielles si utilisées).
4. **Restaurer la base de données :** Utiliser la commande `mysql` pour importer le fichier SQL de sauvegarde.
   Bash
   ```
   mysql -u [utilisateur] -p[mot_de_passe] [nom_base_de_donnees] < [chemin_sauvegarde]/[nom_fichier_sauvegarde].sql
   ```
5. **Vérifier l'intégrité :** Tester la base de données restaurée pour s'assurer de sa cohérence et de sa fonctionnalité.
6. **Redémarrer l'application :** Remettre l'application en ligne une fois la restauration validée.
7. **Documenter l'incident :** Enregistrer l'incident, sa résolution et les leçons apprises pour améliorer le plan de sauvegarde.

---

# 6 Tests Unitaires - API

Les tests unitaires de l'API sont essentiels pour garantir la fiabilité et le bon fonctionnement des différents points d'accès (endpoints) développés en PHP. Ces tests visent à valider le comportement de chaque fonction ou module de l'API de manière isolée, en s'assurant qu'ils renvoient les réponses attendues pour des entrées données. Pour les API REST, cela implique de vérifier les codes de statut HTTP, la structure des réponses JSON et la logique métier associée à chaque opération.

Les tests sont réalisés en interagissant directement avec les endpoints de l'API via des requêtes HTTP (GET, POST). Les résultats sont observés pour s'assurer qu'ils correspondent aux spécifications fonctionnelles.

## 6.1 Lancement du serveur

**Objectif :** Vérifier que le serveur web et la base de données sont opérationnels et que l'API est accessible.

- **Procédure :**
    1. Démarrer WampServer. S'assurer que l'icône Wamp est verte, indiquant que tous les services (Apache, MySQL) sont démarrés correctement.
    2. Accéder à l'URL de base de l'API dans un navigateur (par exemple, `http://localhost/fraicheur/api/`).
- **Résultat attendu :**
    1. Le serveur répond sans erreur (pas de 404 Not Found, pas d'erreurs internes du serveur).
    2. Accès réussi aux endpoints spécifiques comme `http://localhost/fraicheur/api/auth.php` et `http://localhost/fraicheur/api/products.php`, qui devraient renvoyer des réponses JSON ou des messages d'erreur de validation (e.g., `{"status":false,"message":"Veuillez fournir l'email et le mot de passe."}`).

## 6.2 Validation d'une commande

**Objectif :** Vérifier la capacité de l'API à valider une commande existante et à en modifier le statut.

- **Endpoint :** `POST /fraicheur/api/validate_order.php` (ou similaire, selon l'implémentation de votre API)
- **Données d'entrée (Body - JSON) :**
  JSON

  ```
  {
    "idCommande": [ID_DE_LA_COMMANDE_A_VALIDER],
    "idAdmin": [ID_DE_L_ADMINISTRATEUR_VALIDANT]
  }
  ```

- **Procédure :**
    1. Sélectionner une commande en attente dans la base de données.

2. Envoyer une requête POST à l'endpoint de validation avec l'ID de la commande et l'ID de l'administrateur.

- **Résultat attendu :**
  1. Code de statut HTTP : `200 OK`.
  2. Réponse JSON : `{"success": true, "message": "Commande validée avec succès."}`.
  3. Vérification dans la base de données : Le statut de la commande correspondante (`statutCommande` dans la table `COMMANDE`) doit être mis à jour à "Validée".

## 6.3 Affichage des produits

**Objectif :** Vérifier que l'API renvoie correctement la liste complète des produits disponibles dans le catalogue.

- **Endpoint :** `GET /fraicheur/api/products.php`
- **Procédure :**
  1. Envoyer une requête GET à l'endpoint des produits.
- **Résultat attendu :**
  1. Code de statut HTTP : `200 OK`.
  2. Réponse JSON : Un tableau d'objets JSON, chaque objet représentant un produit avec les champs `id`, `designation`, `description`, `prix`, `reduction_pourcentage`, `image_url`, `idType`, et `full_image_url`.
  3. Vérification des données : La liste des produits doit correspondre aux données présentes dans la table `PRODUIT` de la base de données.

## 6.4 Affichage des commandes

**Objectif :** Vérifier que l'API renvoie correctement la liste des commandes.

- **Endpoint :** `GET /fraicheur/api/orders.php` (ou similaire)
- **Procédure :**
  1. Envoyer une requête GET à l'endpoint des commandes.
- **Résultat attendu :**
  1. Code de statut HTTP : `200 OK`.
  2. Réponse JSON : Un tableau d'objets JSON, chaque objet représentant une commande avec ses détails (ID, date, statut, montant, etc.).
  3. Vérification des données : La liste et les détails des commandes doivent correspondre aux données dans la table `COMMANDE` et `ELEMENT_COMMANDE` de la base de données.

## 6.5 Expédition d'une commande

**Objectif :** Vérifier la capacité de l'API à marquer une commande comme expédiée.

- **Endpoint :** `POST /fraicheur/api/ship_order.php` (ou similaire)
- **Données d'entrée (Body - JSON) :**
  JSON

```
{
  "idCommande": [ID_DE_LA_COMMANDE_A_EXPEDIER],
  "idAdmin": [ID_DE_L_ADMINISTRATEUR_EXPEDIANT]
}
```

- **Procédure :**
    1. Sélectionner une commande validée dans la base de données.
    2. Envoyer une requête POST à l'endpoint d'expédition avec l'ID de la commande et l'ID de l'administrateur.
- **Résultat attendu :**
    1. Code de statut HTTP : `200 OK`.
    2. Réponse JSON : `{"success": true, "message": "Commande expédiée avec succès."}`.
    3. Vérification dans la base de données : Le statut de la commande correspondante (`statutCommande` dans la table `COMMANDE`) doit être mis à jour à "Expédiée" et/ou une entrée dans la table `LIVRAISON` doit être créée ou mise à jour.

## 6.6 Gestion des erreurs - Expédition

**Objectif :** Vérifier que l'API gère correctement les scénarios d'erreur lors de la tentative d'expédition d'une commande invalide ou non existante.

- **Endpoint :** `POST /fraicheur/api/ship_order.php`
- **Procédure :**
    1. Tenter d'expédier une commande avec un `idCommande` qui n'existe pas dans la base de données.
    2. Tenter d'expédier une commande déjà expédiée ou annulée (si la logique métier le permet).
    3. Tenter d'expédier une commande sans fournir l'ID de l'administrateur ou l'ID de la commande.
- **Résultat attendu :**
    1. Code de statut HTTP : `400 Bad Request` ou `404 Not Found` ou `500 Internal Server Error` (selon l'implémentation de l'erreur dans l'API).
    2. Réponse JSON : `{"success": false, "message": "Erreur: Commande non trouvée ou statut invalide."}` ou un message d'erreur approprié (e.g., "Veuillez fournir l'ID de commande.").

## 6.7 Création d'un produit

**Objectif :** Vérifier la capacité de l'API à ajouter un nouveau produit au catalogue.

- **Endpoint :** `POST /fraicheur/api/create_product.php` (ou similaire)
- **Données d'entrée (Body - JSON) :**
  JSON

```
{
  "designation": "Nom du Nouveau Produit",
  "description": "Description détaillée du nouveau produit.",
```

```
  "prix": 25.50,
  "reduction_pourcentage": 5.0,
  "imageUrl": "nouvelle_image.jpg",
  "idType": 1,
  "idAdmin": 1
}
```

- **Procédure :**
    1. Envoyer une requête POST à l'endpoint de création de produit avec les données du nouveau produit.
- **Résultat attendu :**
    1. Code de statut HTTP : `201 Created` ou `200 OK`.
    2. Réponse JSON : `{"success": true, "message": "Produit créé avec succès.", "idProduit": [NOUVEL_ID]}`.
    3. Vérification dans la base de données : Un nouveau produit doit être inséré dans la table `PRODUIT` avec les données fournies.

---

# 7 Maquette

Cette section présentera les maquettes de l'interface utilisateur de l'application "Fraicheur Admin App", illustrant la conception visuelle et l'expérience utilisateur des principales pages de l'application (écran de connexion, catalogue de produits, détails de produit, gestion des commandes, tableau de bord, etc.). Ces maquettes permettent de visualiser l'agencement des éléments, la navigation et l'esthétique générale avant le développement complet de l'interface.