#### ****1. Objectif de l'Application****

Le projet "AgoraBo" est une application web de type back-office conçue pour l'administration d'un catalogue de jeux vidéo. Son objectif principal est de fournir une interface permettant aux administrateurs de réaliser l'ensemble des opérations CRUD (Create, Read, Update, Delete - Créer, Lire, Modifier, Supprimer) sur les entités clés du catalogue.

Les entités gérées par l'application sont :

* **Jeux** : gestion des fiches produits (référence, nom, prix, etc.).
* **Genres** : gestion des catégories (Action, Aventure, RPG, etc.).
* **Plateformes** : gestion des supports (PC, PlayStation, Xbox, etc.).
* **Marques** : gestion des éditeurs et constructeurs (Sony, Nintendo, etc.).
* **Classifications PEGI** : gestion des classifications d'âge (PEGI 3, PEGI 18, etc.).

#### ****2. Architecture Technique et Structure du Projet****

Le projet repose sur l'architecture **Modèle-Vue-Contrôleur (MVC)**, un patron de conception reconnu qui garantit une organisation claire et une maintenance aisée du code en séparant la logique métier, la présentation des données et la gestion des actions utilisateur.

La structure des fichiers du projet reflète fidèlement ce modèle :

**a) Le Modèle (M) : La couche d'accès aux données**

* **Fichier principal** : class.PdoJeux.inc.php
* **Rôle** : Cette classe centralise et isole toute interaction avec la base de données (jeux.sql). Elle contient l'ensemble des requêtes SQL (SELECT, INSERT, UPDATE, DELETE), encapsulées dans des méthodes spécifiques (ex: getLesJeux()). Le reste de l'application (notamment les contrôleurs) interroge le Modèle via ces méthodes, sans jamais écrire de code SQL directement, assurant ainsi la sécurité et la cohérence des accès aux données.

**b) La Vue (V) : La couche de présentation**

* **Fichiers** : vue/\*.html.twig
* **Rôle** : La Vue est exclusivement responsable de l'affichage des informations. Constituée de gabarits (templates) utilisant le moteur **Twig**, elle contient le code HTML et des instructions simples pour afficher les données fournies par le contrôleur (ex: {{ variable }}, {% for item in items %}). Ce choix technique marque une évolution par rapport à d'anciennes vues en PHP (v\_lesGenres.php), permettant une séparation stricte entre la logique d'affichage et la logique applicative.

**c) Le Contrôleur (C) : Le chef d'orchestre**

* **Fichiers** : controlleur/c\_\*.php
* **Rôle** : Le Contrôleur agit comme un intermédiaire entre le Modèle et la Vue. Ses responsabilités sont les suivantes :
  1. **Intercepter** les requêtes de l'utilisateur (ex: un clic sur un bouton "Supprimer").
  2. **Solliciter** le Modèle pour effectuer les actions requises (ex: $db->supprimerPegi($id)).
  3. **Récupérer** les données nécessaires auprès du Modèle (ex: $tbPegis = $db->getLesPegis()).
  4. **Transmettre** ces données à la Vue appropriée pour qu'elle génère le rendu final.

**d) Point d'entrée et Routage**

* **Fichier** : index.php
* **Rôle** : Ce fichier unique sert de **contrôleur frontal** (Front Controller). Toutes les requêtes HTTP convergent vers lui. Son instruction switch($\_GET['uc']) analyse l'URL pour déterminer quelle action est demandée et quel contrôleur doit être chargé pour traiter la requête. Il est le point de départ de toute exécution.

**e) Autres répertoires notables**

* \_config.inc.php : Fichier de configuration centralisant les informations de connexion à la base de données.
* vendor/ : Répertoire géré par Composer, contenant les dépendances externes du projet, notamment la bibliothèque Twig.
* web/ : Dossier public contenant les ressources statiques (CSS, JavaScript, images) directement accessibles par le navigateur.

#### ****3. Justification de l'Adoption de l'Architecture MVC****

Le choix du modèle MVC n'est pas anodin ; il répond à un besoin fondamental de structuration pour surmonter les limites du "code spaghetti", où les logiques PHP, HTML et SQL sont entremêlées dans un même fichier, rendant toute modification complexe et risquée.

Le principe fondamental du MVC est la **séparation des préoccupations** (Separation of Concerns). Chaque composant a une responsabilité unique et bien définie. Pour ce projet, les avantages concrets sont :

* **Maintenance facilitée** : Pour modifier l'apparence de la liste des PEGI, il suffit d'éditer le fichier v\_lesPegis.html.twig. La logique métier et l'accès aux données demeurent inchangés, réduisant le risque de régressions.
* **Collaboration optimisée** : Un développeur front-end peut travailler sur les vues (dossier vue/) pendant qu'un développeur back-end se concentre sur le modèle et les contrôleurs, sans interférence.
* **Réutilisabilité du code** : Une méthode définie dans le Modèle, comme getLesGenres(), peut être appelée par plusieurs contrôleurs pour différents besoins, évitant ainsi la duplication de code.
* **Évolutivité de l'application** : Grâce à cette architecture découplée, il serait aisé de faire évoluer le projet. Par exemple, pour créer une API REST renvoyant des données en JSON, il suffirait de créer de nouvelles vues (ou une nouvelle couche de présentation) qui réutiliseraient le Modèle existant sans aucune modification de ce dernier. Le cœur logique de l'application reste stable et indépendant de la manière dont les données sont présentées.