



Piwee!

Dossier de projet

Création d'un e-commerce

MOREIRA Stéphane

<https://github.com/S-Moreira06/piwee-laravel>

Sommaire

Introduction

- Contexte général du projet
- Objectifs fonctionnels et techniques
- Contraintes (temps, techniques, sécurité, accessibilité...)

Front-End

- Présentation des outils front utilisés (framework, design system...)
- Maquettes
- Schéma d'enchaînement des écrans
- Extraits de code UI statique (HTML/CSS)
- Extraits de code UI dynamique (JS, React, etc.)
- Adaptation responsive et accessibilité
- Sécurisation côté client

Back-End

- Présentation de l'architecture choisie
- MERISE
- Composants d'accès aux données (SQL, utilisation ORM)
- Extraits de code (accès données)
- Composants métier (logique serveur, POO)
- Sécurisation côté serveur (auth, validation, etc.)

Tests & Déploiement

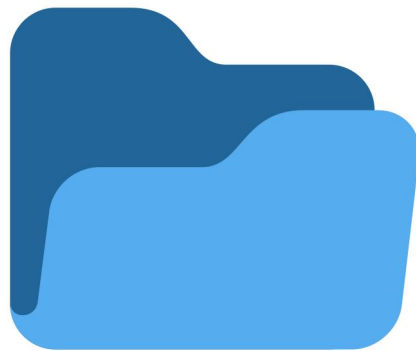
- Jeu d'essai fonctionnel (seeders)
- Stratégie ou procédure de déploiement (locale, cloud, scripts)

Veille technologique & sécurité

- Recherches menées pendant le projet (outils, framework, méthodes...)
- Veille sécurité (faille identifiée, correction appliquée)

Conclusion

- Difficultés rencontrées et résolutions
- Apports personnels (techniques, humains)
 - Conclusion + ouverture (évolutions possibles)



Introduction



Contexte du projet

Piwee est un projet personnel initié en avril 2025, que j'ai conçu en vue de la soutenance du titre professionnel D2WM à l'école LaPlateforme de Cannes. L'objectif était de réaliser un **site e-commerce** complet dédié à la vente de vêtements, afin d'explorer l'ensemble des compétences attendues dans le référentiel.

POURQUOI UN E-COMMERCE?



www.fevad.com

**fédération e-commerce
et vente à distance**

J'ai fait le choix stratégique de m'orienter vers l'e-commerce car ce secteur représente aujourd'hui la majorité des nouveaux sites web créés : selon la FEVAD (La Fédération du e-commerce et de la vente à distance), près de **20 % des sites développés en France** sont à vocation commerciale, et le commerce en ligne poursuit sa progression chaque année. Maîtriser toutes les composantes spécifiques à un site e-commerce (gestion du catalogue, panier, commandes, paiement, espace client, backoffice d'administration) s'avère donc **un atout essentiel pour répondre aux attentes du marché et des employeurs**.

J'ai également souhaité m'inspirer des standards du secteur, en analysant les principaux acteurs français tels que :

- Veepee
- La Redoute
- Zalando
- Showroomprivé
- Cdiscount
- Sarenza

 zalando

Veepee 



SHOW
ROOM
PRIVÉ



Objectifs fonctionnels et techniques identifiés

Objectifs fonctionnels:

- ❑ Gestion complète du **cycle d'achat**
- ❑ Permettre à l'utilisateur de **parcourir le catalogue**, filtrer et rechercher des articles
- ❑ Ajouter, modifier et supprimer des **articles dans le panier**
- ❑ Valider le panier et **passer commande**
- ❑ Suivre l'**état de ses commandes** (en cours, expédiée, annulée)
- ❑ **Authentification** et **gestion du compte** (Inscription, connexion, modification du profil).
- ❑ **Suppression logique** du compte (désactivation sans perte de données historiques).
- ❑ Gestion des **favoris** (Voir , ajouter ou retirer des articles aux favoris)
- ❑ **Synchronisation des favoris** entre différents onglets/navigateurs.
- ❑ **Interface d'administration** (back-office) pour gérer les utilisateurs, commandes, stocks, produits, images, etc.
- ❑ Modification rapide du **statut des commandes**.

Objectifs techniques:

- ❑ Architecture **full-stack** moderne et évolutive, mais accessible pour une **équipe réduite**:
 - **Filament** = back-office
 - **Laravel** = base monolithique
 - **Inertia.js** et **React** = SPA
- ❑ Utilisation de **composants réutilisables** (React, TailwindCSS, DaisyUI) pour accélérer le développement et garantir la cohérence graphique.
- ❑ Gestion efficace de **la session** pour le panier et les favoris.
- ❑ **Optimisation de l'UX** (loaders, transitions, responsive, accessibilité).
- ❑ **Scalabilité** et **maintenance**

MVP (Most Viable Product)

Étant donné que je suis très limité dans le temps pour ce projet et que je suis seul, j'ai décidé d'appliquer le concept de **MVP** à mon processus. Le **Most Viable Product** (ou Produit Minimum Viable) est une version **fonctionnelle** de votre application qui contient uniquement les fonctionnalités essentielles pour répondre aux besoins principaux de vos utilisateurs. Je me suis approprié le concept pour qu'il réponde au **besoin principaux pour ma présentation**, plutôt que pour l'utilisateur.

Ces objectifs garantissent à la fois la couverture des besoins métiers du e-commerce (fonctionnalités attendues par les utilisateurs et les administrateurs) et la robustesse technique nécessaire pour faire évoluer le projet dans la durée.

Le concept de MVP m'a permis de respecter les impératifs de temps imposé par la formation.

Contraintes identifiées

Contraintes techniques

Gestion des stocks en temps réel :

Le stock est synchronisé avec les commandes pour éviter la survente.

Structure modulaire :

Le code doit rester clair, organisé et évolutif pour faciliter l'ajout de nouvelles fonctionnalités (retours, recommandations, etc.).

Compatibilité multi-supports :

Le site doit être responsive et fonctionner sur tous les navigateurs et appareils (ordinateurs, tablettes, mobiles).

Contraintes de sécurité

Protection des données personnelles :

Respect du RGPD, sécurisation des données utilisateurs (chiffrement, accès restreint).

Bonnes pratiques Laravel :

Protection CSRF, filtrage des entrées utilisateur, sécurisation des uploads, gestion stricte des permissions et des Rôles

Contraintes d'accessibilité

Respect des normes d'accessibilité :

À partir de juin 2025, obligation légale d'avoir un site et des services conformes aux standards d'accessibilité (RGAA)

Navigation clavier :

Le site doit être utilisable sans souris, avec des repères clairs pour les technologies d'assistance.

Alternatives textuelles :

Toutes les images porteuses d'information doivent avoir un texte alternatif.

Compatibilité avec les lecteurs d'écran :

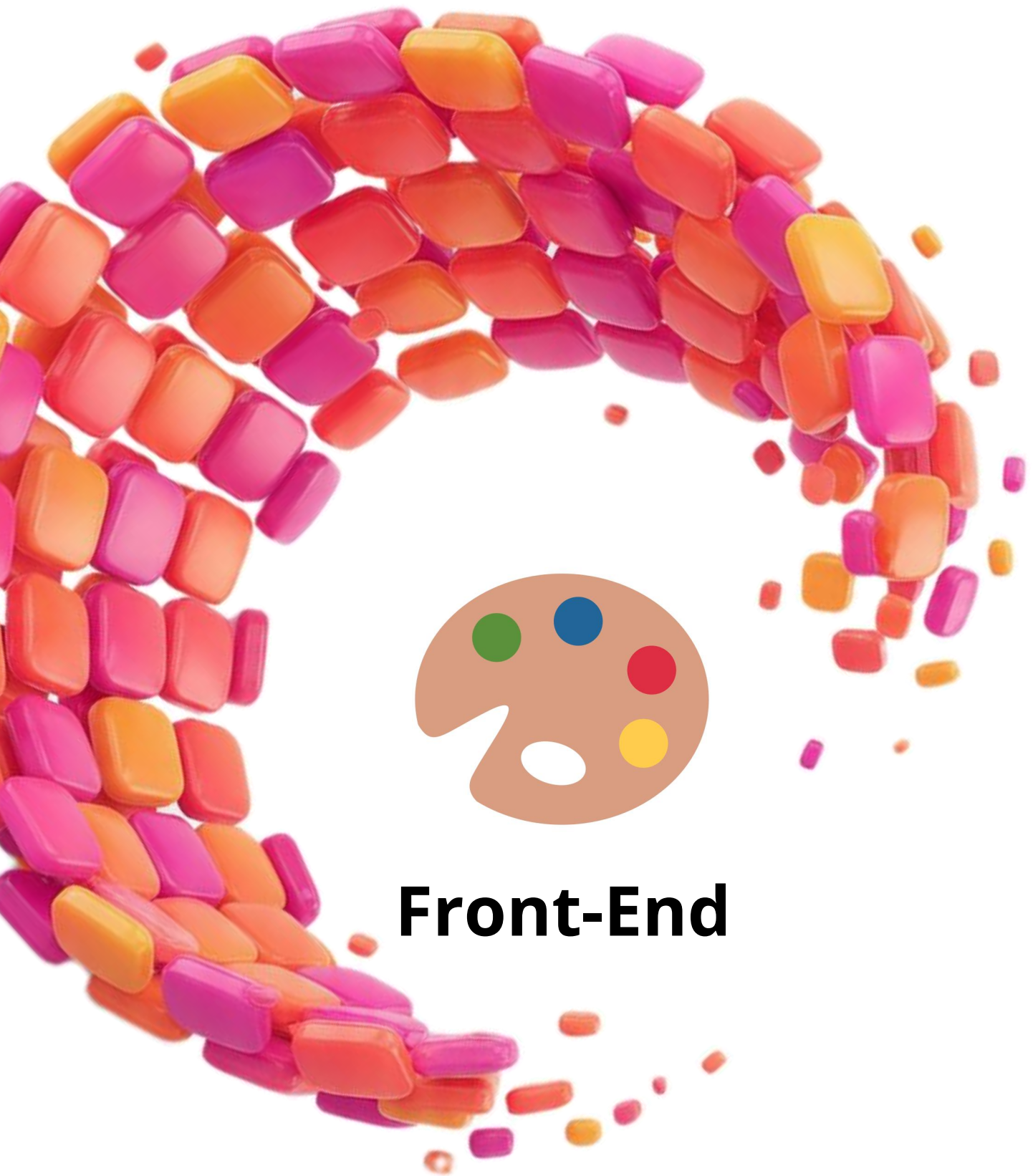
Les formulaires, boutons et liens doivent être correctement étiquetés et structurés.

Accessibilité des processus clés :

Création de compte, validation du panier, paiement, suivi de commande et SAV doivent être accessibles à tous les utilisateurs, y compris en situation de handicap.

Le projet doit répondre à des exigences strictes de performance, de sécurité (protection des données, transactions, accès), et d'accessibilité (conformité légale, expérience inclusive), tout en restant évolutif et facile à maintenir.



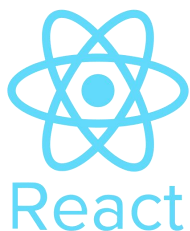


Front-End



Présentation des outils utilisés

Le front-end de Piwee s'appuie sur une sélection d'outils et de frameworks modernes, reconnus pour leur efficacité, leur modularité et leur capacité à offrir une expérience utilisateur fluide et accessible :



React

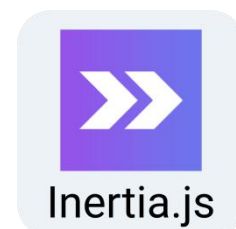
React est une *bibliothèque* JavaScript développée par **Meta**, leader du développement d'interfaces utilisateur dynamiques et réactives. Elle repose sur une **architecture de composants réutilisables** et un **Virtual DOM** qui optimise les performances lors des mises à jour de l'interface.

Rôle dans Piwee : Toute l'interface client (pages, composants interactifs, gestion du panier et des favoris, etc.) est développée avec React, garantissant modularité, réactivité et maintenabilité.

Inertia.js

Inertia.js sert de **pont entre Laravel et React**, permettant de créer une **Single Page Application** (SPA) sans avoir à développer une **API REST** séparée.

Rôle dans Piwee : Inertia gère la navigation et la transmission des données entre le serveur et le front, offrant une expérience fluide sans rechargement complet des pages.



TailwindCSS



TailwindCSS est un **framework CSS utilitaire** qui permet de concevoir des interfaces **modernes, responsives et personnalisées** rapidement, en appliquant des classes **directement dans le code HTML**.

Rôle dans Piwee : Il structure tout le design du site, assure la cohérence visuelle et facilite l'adaptation aux différents supports (desktop, mobile).

DaisyUI

DaisyUI est une **bibliothèque de composants UI** basée sur **TailwindCSS**, proposant des **éléments prêts à l'emploi** (boutons, formulaires, alertes, etc.) et **facilement personnalisables**.

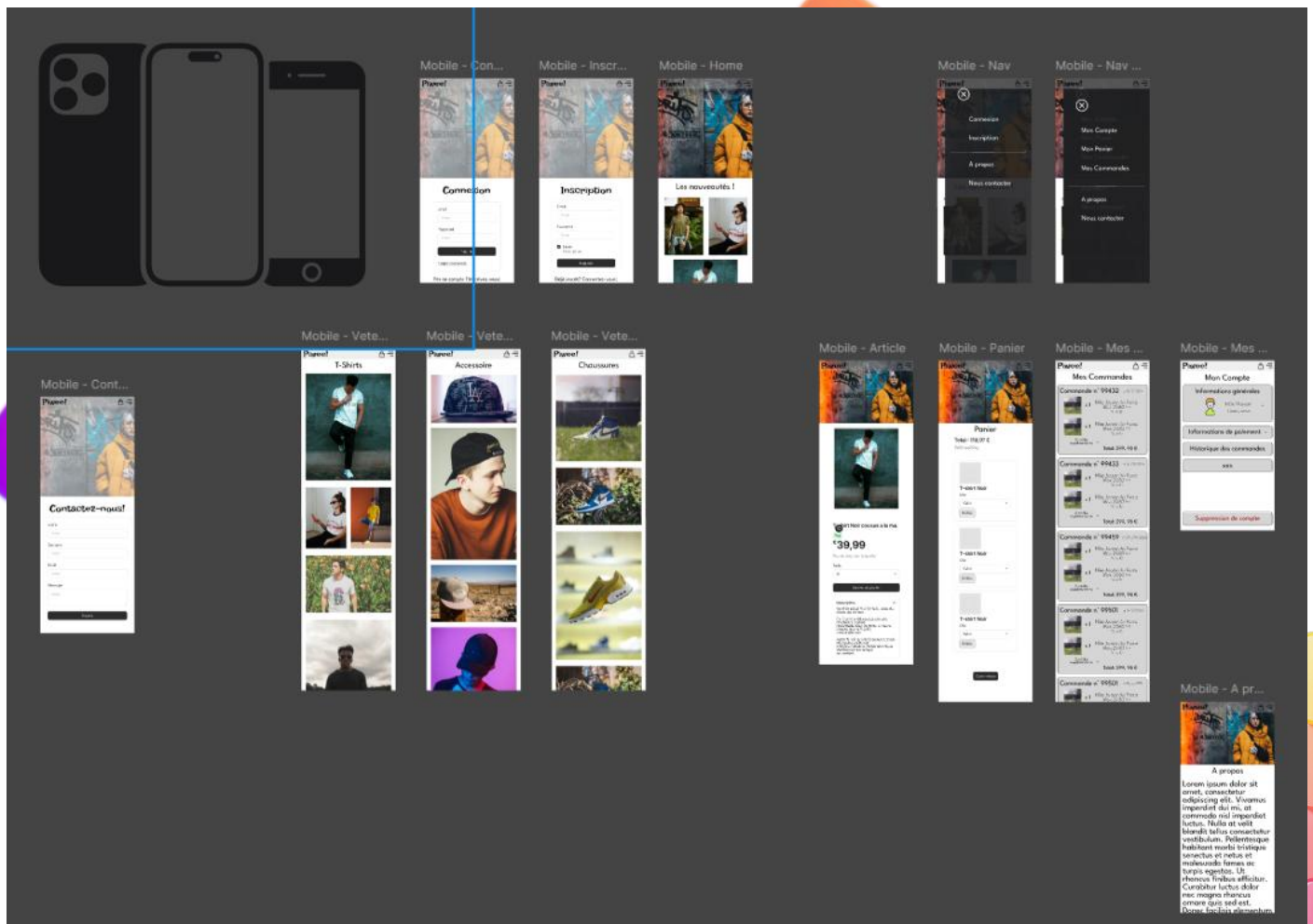
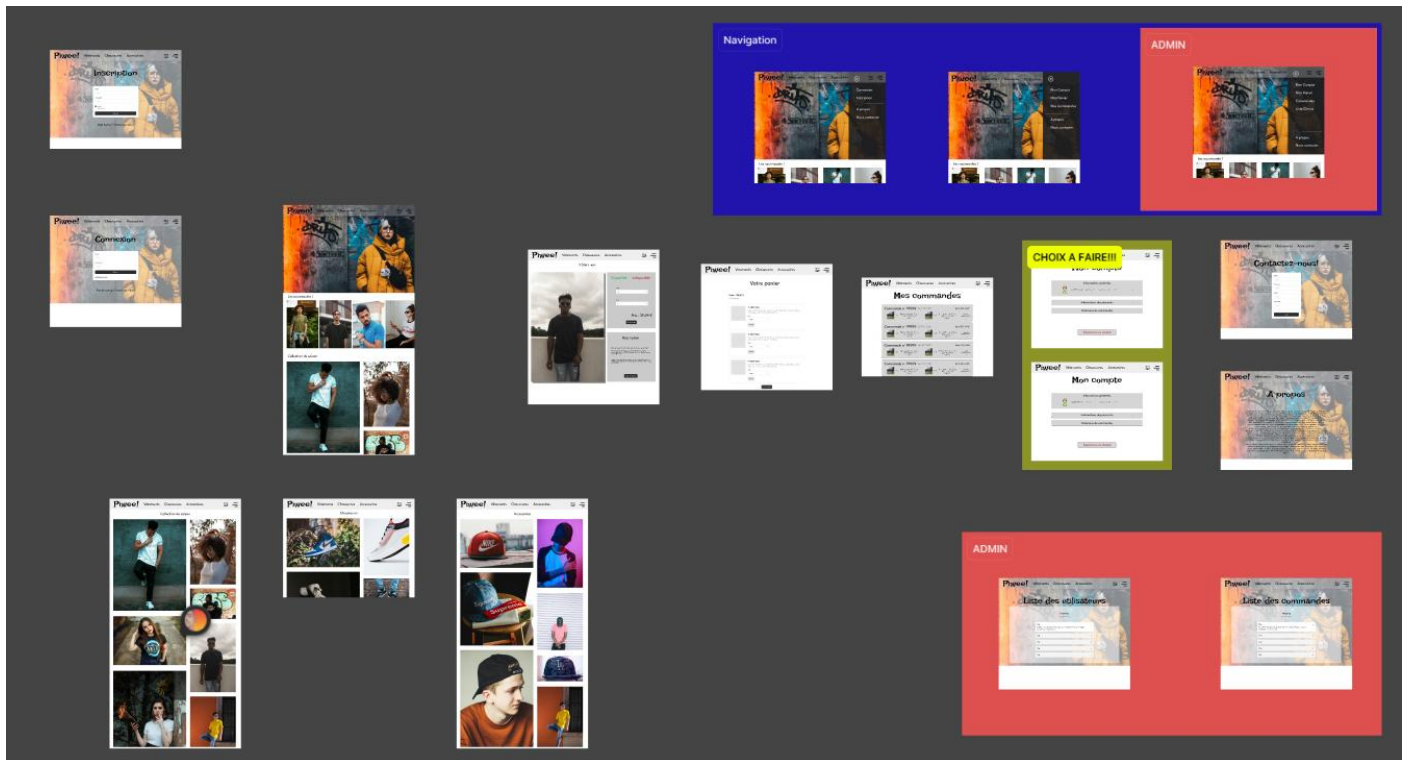
Rôle dans Piwee : Elle accélère le développement de l'interface et garantit une expérience utilisateur cohérente et accessible.



Le choix de React, Inertia.js, TailwindCSS et DaisyUI permet à Piwee de proposer une interface moderne, rapide, accessible et facilement évolutive, en phase avec les meilleures pratiques du développement front-end en 2025



Maquettes





Notre sélection



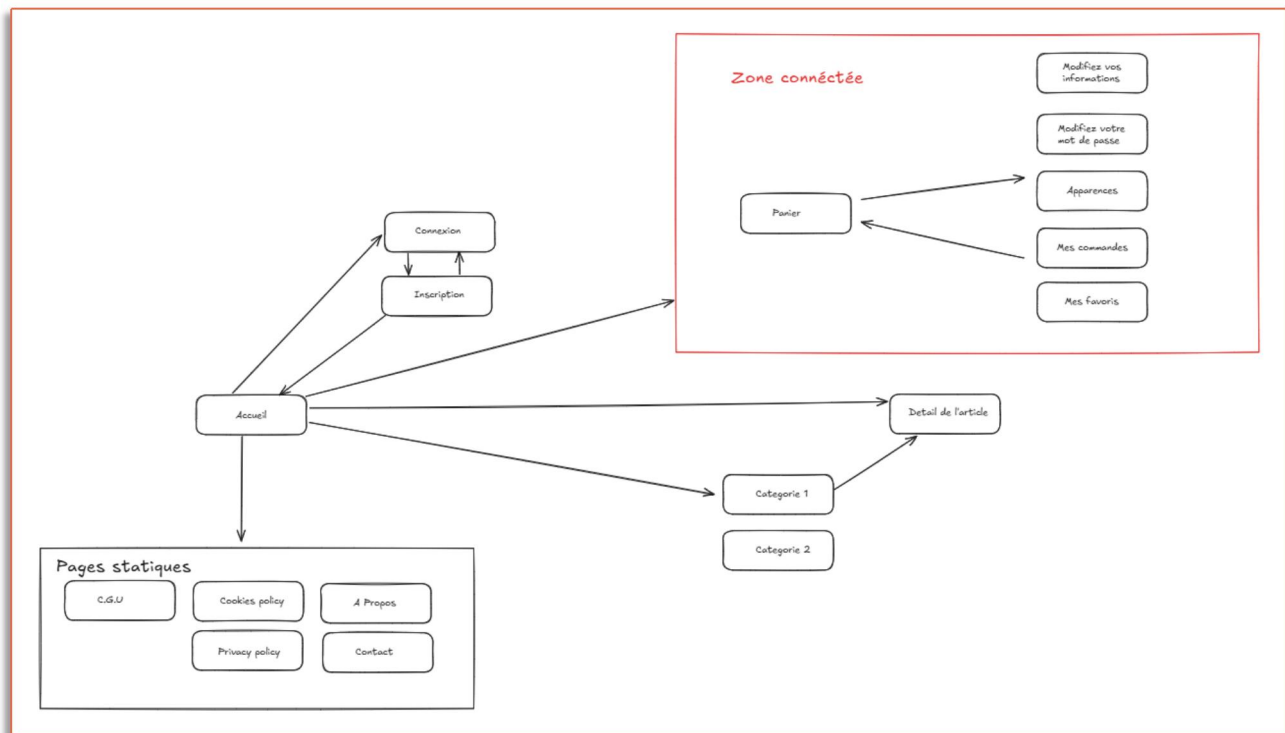
T-shirt



sneakers



Schéma d'enchaînement des écrans



Ce diagramme présente le schéma d'enchaînement des écran, organisée autour de trois zones principales interconnectées.

L'écran d'accueil constitue le **hub central** de l'application, servant de **point de départ** vers toutes les **fonctionnalités principales**.

Depuis cet écran, l'utilisateur peut accéder aux différentes sections de l'application selon ses besoins.

La zone connecté regroupe les **fonctionnalités accessibles aux utilisateurs connectés**.

(Gestion de compte, favoris, panier)

Les page statique sont un espace **informatif accessible à tous**

Extraits de code statique

Via **React**, j'ai d'abord créé une constante section afin pour que mon code soit *scalable* et *maintenable* rapidement et facilement.

```
const sections = [
  {
    title: "1. Notre histoire",
    content: (
      <p>
        Piwee! a été créé pour offrir une expérience de shopping unique a
      </p>
    ),
  },
  {
    title: "2. Notre mission",
    content: (
      <p>
```

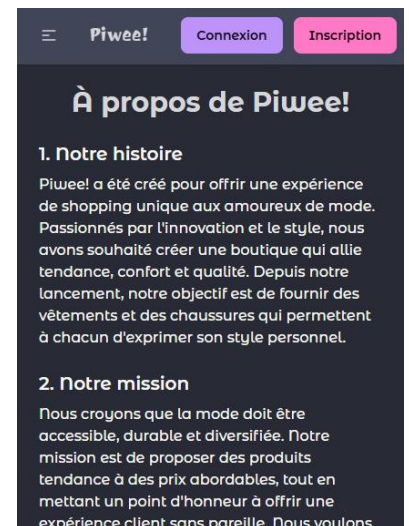
J'ai ensuite mis en place une *boucle* pour afficher celui-ci:

```
<div className="">
  {sections.map((section) => (
    <div>
      <h2 className="">{section.title}</h2>
      {section.content}
    </div>
  ))}
</div>
```


Et enfin : mise en page avec **TailwindCSS** et **DaisyUI** , responsive puis petites animations avec **motion**:

```
export default function About() {
  return (
    <Layout className="min-h-screen">
      <Head title={"A propos"} />
      <motion.section
        className="max-w-4xl mx-auto p-6 bg-base-100 shadow-lg my-10"
        initial={{ opacity: 0, y: 30 }}
        animate={{ opacity: 1, y: 0 }}
        transition={{ duration: 0.6, ease: "easeOut" }}
      >
        <h1 className="text-3xl font-bold mb-6 text-center">À propos de Piwee!</h1>

        <div className="space-y-6 text-base-content">
          {sections.map((section, index) => (
            <motion.div
              key={index}
              initial={{ opacity: 0, y: 20 }}
              animate={{ opacity: 1, y: 0 }}
              transition={{ duration: 0.5, delay: 0.2 + index * 0.1 }}
            >
              <h2 className="text-xl font-semibold mb-2">{section.title}</h2>
              {section.content}
            </motion.div>
          ))}
        </div>
      </motion.section>
    </Layout>
  );
}
```



Extraits de code dynamique

Initialisation de l'état dynamique avec useState

```
const { item } = usePage().props;
const [selectedSize, setSelectedSize] = useState(item.sizes[0]?.size || "");
const [quantity, setQuantity] = useState(1);
const selectedStock = item.sizes.find(s => s.size === selectedSize)?.stock || 0;
const isQuantityTooHigh = quantity > selectedStock;
```

On **instancie** deux **états locaux** pour gérer la taille sélectionnée **selectedSize** et la quantité **quantity**.

L'init de **selectedSize** propose **par défaut** la première taille disponible (robuste pour UX).

selectedStock calcule **dynamiquement** le stock dispo pour la taille choisie, via un simple **find()** dans le tableau des tailles.

isQuantityTooHigh sert à afficher un **message d'erreur** en cas de dépassement du stock (validation UX immédiate).

Sélection dynamique des tailles : mapping, animation et accessibilité

```
88 {
89   {item.sizes.map(({ size }) => (
90     <motion.p
91       key={size}
92       whileHover={{ scale: 1.05 }}
93       className={`badge px-2.5 place-self-center cursor-pointer ${
94         selectedSize === size ? "badge-primary" : "badge-outline"
95       }}
96       variants={{
97         hidden: { opacity: 0, y: 10 },
98         visible: { opacity: 1, y: 0 },
99       }}
100       onClick={() => setSelectedSize(size)}
101     >
102       {size}
103     </motion.p>
104   )}
105 }
```

On **affiche dynamiquement** les tailles disponibles pour le produit, sous forme de **badges interactifs**.

On utilise les animation de **Framer Motion** (motion.p, variante visible/hidden et effet au survol) pour améliorer l'UX.

Le visuel du badge reflète la sélection de l'utilisateur via la classe CSS qui résulte de la condition de **selectedSize**.

Gestion incrémentale/décémentale de la quantité

```
109 <motion.button  
110   whileHover={{ scale: 1.1 }}  
111   className="btn btn-primary"  
112   onClick={() => setQuantity(q => Math.max(1, q - 1))}  
113 >  
114   -  
115 </motion.button>  
116 <p className="place-self-center">{quantity}</p>  
117 <motion.button  
118   whileHover={{ scale: 1.1 }}  
119   className="btn btn-primary"  
120   onClick={() => setQuantity(q => q + 1)}  
121 >  
122   +  
123 </motion.button>
```

L'utilisateur ajuste la quantité via deux boutons (+/-), la **décrémentation ne descend jamais en dessous de 1** (sécurité UX).

Animation sur les boutons pour **inviter l'action**.

Affichage du nombre sélectionné au centre, mise à jour instantanée.

Système d'alerte et validation du stock

```
{isQuantityTooHigh && (  
  <div className="text-error text-center mb-2">  
    La quantité demandée dépasse le stock disponible ({selectedStock}).  
  </div>  
)}
```

Dès que l'utilisateur essaie de dépasser le stock, un **message s'affiche en temps réel**, évitant de mauvaises surprises au checkout.

Validation **immédiate** côté front sans attendre une **réponse serveur**.

Message contextuel, impact direct sur la satisfaction et la confiance utilisateur.



Adaptation responsive et accessibilité



Respect des règles d'accessibilité



Pour garantir une expérience accessible à tous les utilisateurs, y compris les personnes en situation de handicap, le projet Piwee intègre les bonnes pratiques suivantes :

Utilisation des attributs ARIA

ARIA (Accessible Rich Internet Applications) est un ensemble de rôles, propriétés et états qui complètent le HTML pour rendre les contenus web dynamiques accessibles aux technologies d'assistance (lecteurs d'écran, etc.).

Les **rôles ARIA** permettent de définir la nature des éléments (ex: `role="navigation"` pour une barre de navigation, `role="main"` pour le contenu principal, `role="form"` pour un formulaire), **facilitant la compréhension et la navigation** pour les utilisateurs de **lecteurs d'écran**.

Les **propriétés ARIA** indiquent l'état des éléments interactifs (ex: `aria-expanded`, `aria-checked`) et permettent de gérer les **mise à jour dynamiques** du contenu.

Contraste des couleurs selon WebAIM



Le contraste entre le texte (ou éléments interactifs) et leur arrière-plan est essentiel pour garantir la lisibilité, notamment pour les personnes avec une déficience visuelle ou daltoniennes.

Le projet utilise l'outil **WebAIM Color Contrast Checker** pour mesurer et valider les contrastes de couleurs.

Les exigences minimales sont :

- **Ratio de contraste de 4,5:1** pour le texte normal (taille standard).

- **Ratio de contraste de 3:1** pour le texte **large** (≥ 18 pt ou 14pt en gras) et les **éléments graphiques interactifs** (boutons, bordures de formulaire).

Ces seuils correspondent aux standards **WCAG 2.0 niveau AA**, garantissant une bonne **accessibilité** pour la majorité des utilisateurs.

*L'intégration d'ARIA permet de **structurer sémantiquement** les interfaces et de rendre les contenus dynamiques **accessibles**, tandis que la vérification rigoureuse des **contrastes** avec WebAIM assure une lisibilité optimale, répondant aux exigences légales et aux **bonnes pratiques d'accessibilité web**.*





L'adaptabilité de l'interface à tous les supports (ordinateurs, tablettes, mobiles) est un enjeu majeur pour l'expérience utilisateur sur Piwee. Deux leviers principaux sont mis en œuvre : l'utilisation de composants DaisyUI et la gestion fine des breakpoints avec TailwindCSS.

Composants DaisyUI

Les composants fournis par **DaisyUI** (boutons, cartes, barres de navigation, etc.) sont conçus pour être **adaptatifs par défaut**. Ils s'appuient sur la grille et les classes utilitaires de **TailwindCSS**, ce qui garantit une compatibilité optimale sur **tous les écrans**.

DaisyUI permet d'écrire beaucoup moins de classes CSS : il suffit d'utiliser des classes comme `btn`, `card`, `navbar`, etc., qui sont déjà **optimisées** pour le **responsive**. Cela rend le code **plus lisible**, **plus rapide** à écrire et **plus facile** à maintenir.

Chaque composant peut être modifié ou enrichi avec les **utilitaires Tailwind** pour répondre à des besoins spécifiques, tout en restant **responsive**.

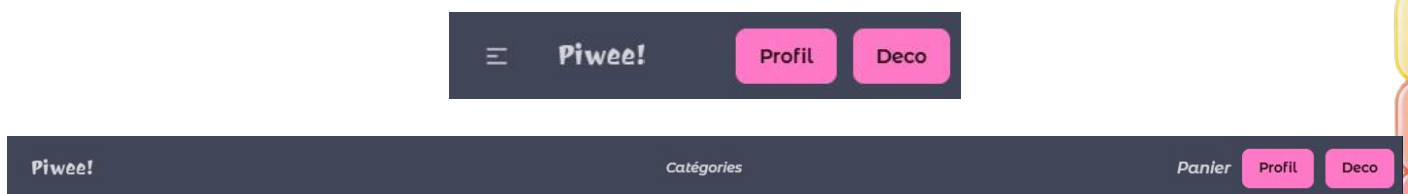
Breakpoints TailwindCSS

TailwindCSS propose par défaut une série de **breakpoints** (`sm`, `md`, `lg`, `xl`, `2xl`) correspondant à différentes tailles d'écran.

Par exemple :

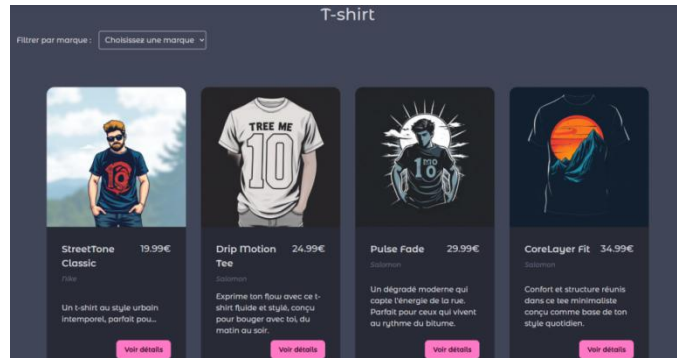
`sm : ≥ 640px / md : ≥ 768px / lg : ≥ 1024px`

Les barres de navigation **DaisyUI** utilisent les **breakpoints** pour masquer ou afficher certains éléments selon la **taille de l'écran** (ex : menu hamburger sur mobile, liens étendus sur desktop).



Les grilles d'articles s'ajustent **automatiquement** : *plusieurs colonnes* sur desktop, *une seule colonne* sur mobile, grâce aux classes **Tailwind**.





L'association de DaisyUI et TailwindCSS permet de garantir une interface responsive, moderne et cohérente, tout en facilitant le développement et la maintenance grâce à des composants prêts à l'emploi et une gestion fine des breakpoints

Sécurisation front-end

Dans Piwee, la sécurisation côté front-end vise à garantir la fiabilité des interactions utilisateur, à limiter les risques d'abus et à protéger les données sensibles lors des échanges avec le back-end.

Validation côté client

Validation immédiate des formulaires :

Chaque formulaire (inscription, connexion, ajout au panier, etc.) est validé **côté client** à l'aide de **React** et des fonctionnalités HTML5. Cela permet de vérifier le format des emails, la présence des champs obligatoires, la force des mots de passe, etc., et d'offrir un **retour instantané à l'utilisateur**.

Limites :

Cette validation améliore l'expérience utilisateur mais n'est pas suffisante pour la sécurité : toutes les données sont systématiquement revalidées côté serveur, car le front-end peut être contourné ou modifié par un utilisateur malveillant.

Authentification côté client

Gestion des sessions et des tokens :

Après une authentification réussie, le front-end gère les **tokens d'accès** transmis par le **back-end**. Ces tokens sont stockés de façon sécurisée des **cookies** `HttpOnly` pour limiter les risques d'exploitation par des scripts malveillants.

```
Set-Cookie XSRF-
TOKEN=eyJpdil6ljl1UmxsbjEwWmNwUW5xK3BHbTNwUUE9PSIsInZhbHVlIj
oiN3QrZnMxWTdrbytObSt6eUYrUdTamVxVHJYcHd4a29SSzBIUnlrTi96VVZ
pa1hISzVaSTR5Yng4SIZpcXdsc1BSRTQ3MWI6ZUZZK2Z5VHd2eGtnU3ZzW
DVRVXVNeDJmckJCXYdXbUpSOFNiZENyY04vanloMEVLckNTTzhITcILCjY
WMiOiJiZTdIMjQwYjFjNzJlYzBhYzM5NWMzMtY1ODJjNzc1ZDZmMzkxMD
M0MdBmMGMyZjJhYTRmZDQ2ZjYwYTU4YTJlIiwidGFnljoiln0%3D;
expires=Thu, 25 Sep 2025 13:49:23 GMT; Max-Age=7200; path=/;
samesite=lax

Set-Cookie piwee_session=eyJpdil6lkljU3BEEdTdDdW1zbStiRGNSK1ltb0E9PSIsInZhbHVl
ljoilOUiReWJDukQ4cGFNZNk4Q245UE1rWGY0OGZPMWJKV0NLRWQ0L0M
5VIFnUHFfTG85aHdaQ0ZUc2REbzRSb1I5WTh0dWN3K2ZhYUpDTEhueVB6
UFFCZzBCV0IDTXRTQ3dPZGNOUXYxNXh0eXZxbE9MVUgwR0N0MHdKSng
4ZHM2cTAiLCJtYWMiOiJhZjNiODg0MWM3ZWMyYTI3Y2lyNDE5YjlyYjY4OD
M1YTU3NjNkMTZjOWJhZTlyMWwQ2N2Y5OTE1N2QwNWQ0MTgzliwidGFnlj
oiln0%3D; expires=Thu, 25 Sep 2025 13:49:23 GMT; Max-Age=7200;
path=/; httponly; samesite=lax
```

Navigation sécurisée :

L'accès aux pages sensibles (profil, commandes, favoris) est **conditionné** à la présence d'une **session** ou d'un **token** valide. Les routes protégées côté front sont doublées d'une vérification côté serveur, car un utilisateur peut toujours manipuler le code du front pour tenter d'accéder à des pages non autorisées.



Protection contre les attaques courantes

Protection XSS (Cross-Site Scripting):

React **échappe automatiquement** toutes les valeurs dans les expressions JSX, protégeant contre les attaques XSS:

```
const userInput = "<script>alert('XSS')</script>";  
return <div>{userInput}</div>; // Rendu sécurisé : &lt;script&gt;
```

Protection CSRF (Cross-Site Request Forgery):

Inertia.js s'intègre parfaitement avec la protection **CSRF** de Laravel. Le hook `useForm` gère automatiquement les tokens CSRF

Bonnes pratiques spécifiques à Piwee

Affichage conditionnel :

Les éléments réservés aux utilisateurs connectés (ex : historique des commandes, gestion du profil) ne sont affichés que si l'utilisateur est authentifié côté front, mais surtout, l'accès est toujours vérifié côté **back-end**.

Gestion des erreurs :

Les messages d'erreur sont clairs pour l'utilisateur mais ne révèlent jamais d'informations sensibles ou techniques, pour éviter de donner des indices à un attaquant potentiel.

Séparation stricte front/back :

Avec l'architecture Inertia.js et React, **le front et le back sont découplés** : toutes les actions sensibles (gestion du panier, commandes, favoris, etc.) nécessitent une validation serveur, limitant ainsi les risques liés à la manipulation du code côté client.

Piwee combine validation côté client, gestion sécurisée des sessions et des tokens, et contrôle strict côté serveur pour garantir la sécurité des données et des parcours utilisateurs. La sécurisation front-end est pensée comme une première barrière UX, mais la confiance et la protection reposent toujours sur le back-end.





Back-End

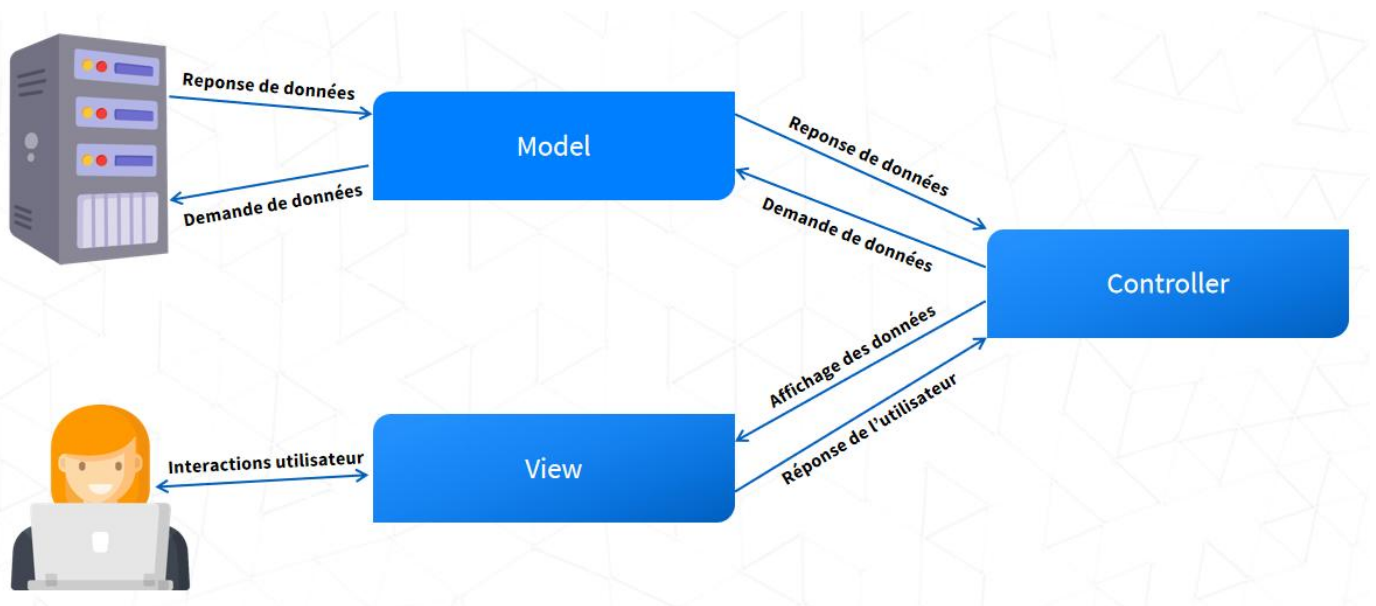


Présentation de l'architecture choisie

Le back end de Piwee repose sur une architecture moderne, robuste et évolutive, conçue pour répondre aux exigences d'un site e-commerce en 2025.

Framework principal : Laravel

Laravel structure toute la logique serveur selon le modèle **MVC (Modèle-Vue-Contrôleur)**, assurant une séparation claire entre la gestion des données (Models), la logique métier (Controllers) et la présentation (Views).



Ce choix garantit un code propre, maintenable, facile à tester et à faire évoluer.

Dossiers organisés par domaine : chaque fonctionnalité (utilisateurs, commandes, panier, produits, favoris...) dispose de ses propres **modèles, contrôleurs, polices et requêtes de validation**, facilitant la maintenance et la scalabilité.

Gestion des données et de la base

Migrations et seeders : la structure de la base de données est **versionnée et automatisée** grâce aux migrations **Laravel**, ce qui permet de collaborer efficacement en équipe et de faire évoluer le schéma sans perte de données.



Relations entre tables : la base est normalisée, avec des clés étrangères pour lier les utilisateurs, commandes, articles, favoris, etc.

```
Schema::create('items', function (Blueprint $table) {
    $table->id();
    $table->foreignId('brand_id')->constrained('brands')->cascadeOnDelete();
    $table->foreignId('category_id')->constrained('categories')->cascadeOnDelete();
    $table->string('name');
    $table->string('slug');
    $table->string('description');
    $table->unsignedInteger('price');
    $table->boolean('isDeleted');
    $table->timestamps();
});
```

Sécurité et accès

Laravel gère l'authentification des utilisateurs (clients et administrateurs) via son système natif, avec possibilité d'étendre vers des solutions API (Sanctum, Passport) pour les besoins futurs.

Chaque action sensible (ex : accès à une commande) est protégée par :

-des policies:

```
class FavoritePolicy
{
    /** ...
    public function viewAny(User $user): bool
    {
        return true; // Un utilisateur peut voir ses propres favoris
    }

    /** ...
    public function create(User $user): bool
    {
        return true; // Tout utilisateur connecté peut créer des favoris
    }
}
```

Exemple de policies pour protéger les favoris

-des middleware :

Le middleware `HandleInertiaRequests` partage les données d'authentification dans toutes les pages via la prop `auth.user`.

Backoffice et outils d'administration

Le backoffice est propulsé par **Filament**, qui permet de gérer facilement les contenus, commandes, utilisateurs et stocks via une **interface dédiée**, sans impacter la logique métier principale.

API et communication avec le front

Inertia.js agit comme un pont entre **Laravel** et **React**, permettant d'exposer les données du back end directement aux composants front, sans créer une **API REST** séparée. Cela simplifie la synchronisation des états et accélère le développement.

Le projet utilise une architecture full-stack moderne combinant un backend Laravel MVC avec un frontend SPA React, reliés par Inertia.js qui élimine le besoin d'APIs REST traditionnelles.



Dictionnaire de données (non technique)

Nom de la données	Format	Longueur	Contraintes
Nom	Alphabétique	30	Obligatoire
Prénom	Alphabétique	30	Obligatoire
Adresse E-mail	Alphanumérique	50	Obligatoire, unique
Date d'anniversaire	Date	-	Obligatoire
Genre	Alphabétique	10	Obligatoire
Adresse	Alphanumérique		Obligatoire
Code postal	Alphanumérique	6	Obligatoire
Ville	Alphabétique	30	Obligatoire
Mot de passe	Alphanumérique	>6	Obligatoire
Document : Client			

Nom de la données	Format	Longueur	Contraintes
Nom de l'article	Alphanumérique	30	Obligatoire
Marque	Alphanumérique	30	Obligatoire
Type de vêtements	Alphabétique	30	Obligatoire, unique
Prix	Numérique	-	Obligatoire
Document : Article			

Nom de la données	Format	Longueur	Contraintes
Référence unique	Alphanumérique	6	Obligatoire, unique
Frais de livraisons	Numérique	-	Obligatoire
Taxe	Numérique	-	Obligatoire, unique
Numero de facture	Alphanumérique	10	Unique
Statut de livraison	Alphabétique	10	Obligatoire
Document : Commande			



Schéma conceptuel de la BDD: MCD

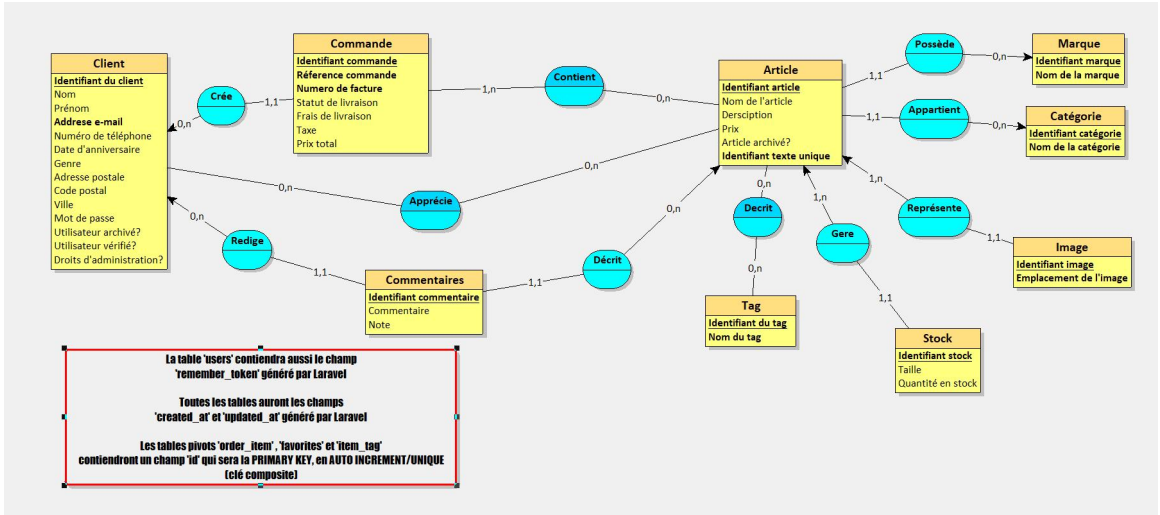


Schéma logique de la BDD: MLD

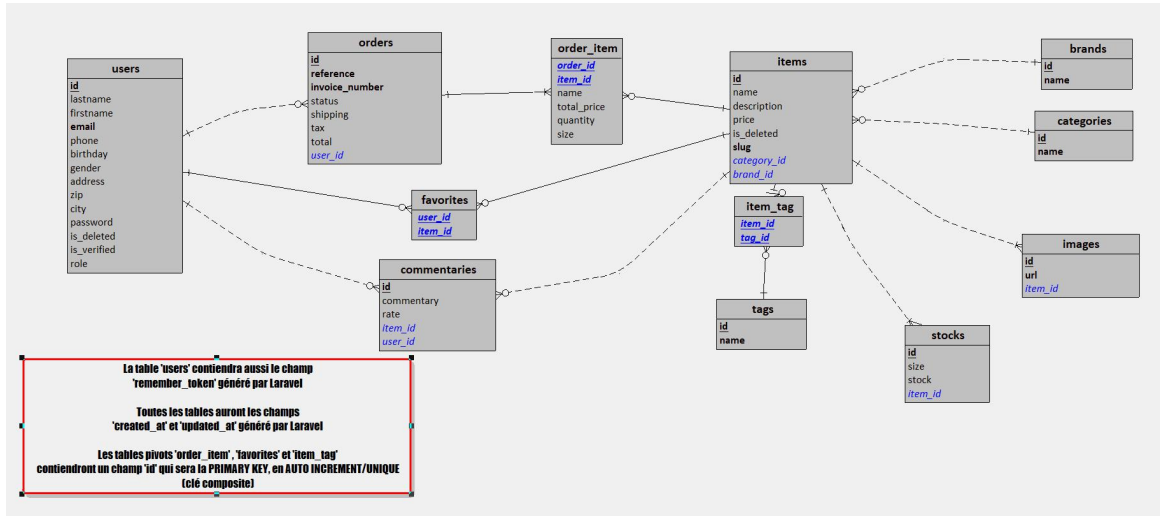
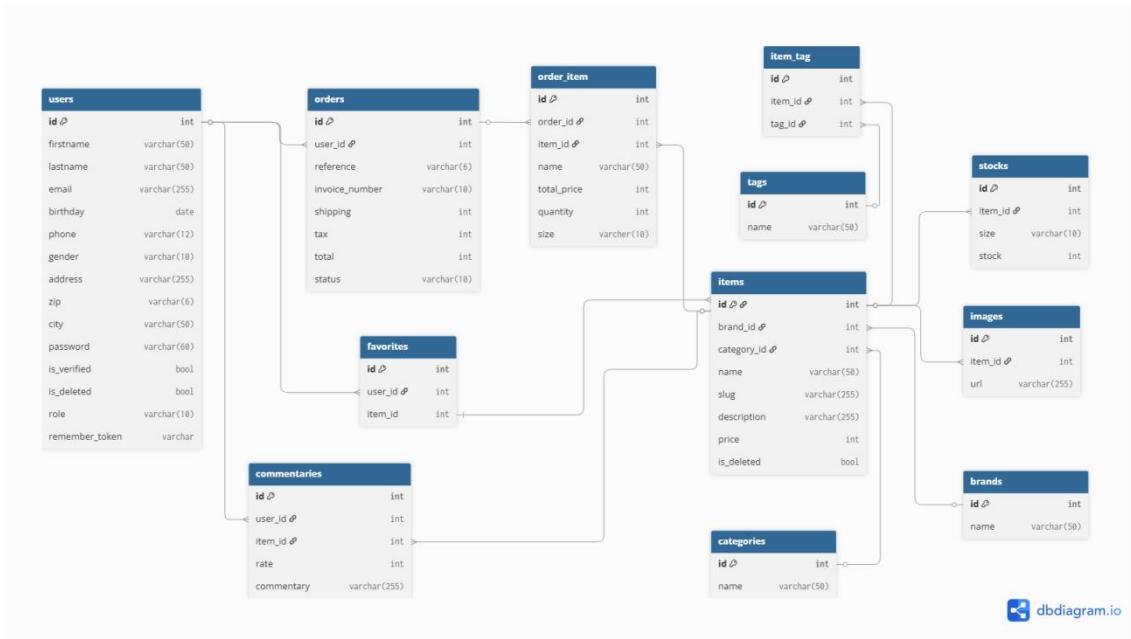


Schéma physique de la BDD: MPD



Language de définition de données:

CREATE TABLE users(

```
id INT,
    lastname VARCHAR(50) NOT NULL,
    firstname VARCHAR(50) NOT NULL,
    email VARCHAR(255) NOT NULL,
    birthday DATE NOT NULL,
    gender VARCHAR(10) NOT NULL,
    address VARCHAR(255) NOT NULL,
    zip VARCHAR(6) NOT NULL,
    city VARCHAR(50) NOT NULL,
    password VARCHAR(60) NOT NULL,
    is_deleted LOGICAL NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(email)
```

);

CREATE TABLE orders(

```
id INT order,
    reference VARCHAR(6) NOT NULL,
    invoice_number VARCHAR(10),
    status VARCHAR(10) NOT NULL,
    shipping DECIMAL(15,2) NOT NULL,
    tax DECIMAL(15,2) NOT NULL,
    total DECIMAL(15,2) NOT NULL,
    user_id INT NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(reference),
    UNIQUE(invoice_number),
    FOREIGN KEY(user_id) REFERENCES users(id)
```

);

CREATE TABLE brands(

```
id INT,
    name VARCHAR(30) NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
```

);

CREATE TABLE categories(

```
id INT,
    name VARCHAR(30) NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
```

);

CREATE TABLE tags(

```
id INT,
    Name VARCHAR(30) NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
```

);

CREATE TABLE items(

```
id INT item,
    name VARCHAR(30) NOT NULL,
    description VARCHAR(255) NOT NULL,
    price DECIMAL(15,2) NOT NULL,
    is_deleted LOGICAL NOT NULL,
    slug VARCHAR(255) NOT NULL,
    category_id INT NOT NULL,
    brand_id INT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY(category_id) REFERENCES
categories(id),
    FOREIGN KEY(brand_id) REFERENCES brands(id)
```

);

CREATE TABLE commentaries(

```
id INT,
    commentary VARCHAR(255) NOT NULL,
    rate INT NOT NULL,
    item_id INT NOT NULL,
    user_id INT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY(item_id) REFERENCES items(id),
    FOREIGN KEY(user_id) REFERENCES users(id)
```

);

CREATE TABLE images(

```
id INT,
    url VARCHAR(255) NOT NULL,
    item_id INT NOT NULL,
```



```
PRIMARY KEY(id),  
UNIQUE(url),  
FOREIGN KEY(item_id) REFERENCES items(id)  
);
```

```
CREATE TABLE stocks(  
id INT,  
size VARCHAR(10) NOT NULL,  
stock INT NOT NULL,  
item_id INT NOT NULL,  
PRIMARY KEY(id),  
FOREIGN KEY(item_id) REFERENCES items(id)  
);
```

```
CREATE TABLE order_item(  
id INT,  
order_id INT,  
item_id INT,  
name varchar(50),  
total_price INT,  
quantity INT,  
size varchar(10),  
PRIMARY KEY(order_id, item_id),  
FOREIGN KEY(order_id) REFERENCES orders(id),  
FOREIGN KEY(item_id) REFERENCES items(id)  
);
```

```
CREATE TABLE item_tag(  
item_id INT,  
tag_id INT,  
PRIMARY KEY(item_id, tag_id),  
FOREIGN KEY(item_id) REFERENCES items(id),  
FOREIGN KEY(tag_id) REFERENCES tags(id)  
);
```

```
CREATE TABLE favorites(  
user_id INT,  
item_id INT,  
PRIMARY KEY(user_id, item_id),  
FOREIGN KEY(user_id) REFERENCES users(id),  
FOREIGN KEY(item_id) REFERENCES items(id)  
);
```



Dictionnaire de données technique

Nom de la données	Format	Longueur	Contraintes
lastname	Varchar	50	Not null
firstname	varchar	50	Not null
email	varchar	255	Not null, unique
birthday	Date	-	Not null
gender	varchar	10	Not null
address	varchar	255	Not null
zip	varchar	6	Not null
city	varchar	50	Not null
password	varchar	60	Not null
is_deleted	boolean		Not null
is_verified	boolean		Not null
role	varchar	10	Not null
Document : users			

Clé primaire: (id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
user_id	int	-	Not null
reference	varchar	6	Not null, unique
invoice_number	varchar	10	Not null, unique
shipping	int	-	Not null
tax	int	-	Not null
total	int	-	Not null
status	varchar	10	Not null
Document : orders			

Clé primaire: (id) // Clé(s) secondaire(s): (user_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
name	varchar	50	Not null, unique
brand_id	int	-	Not null
category_id	int	-	Not null
name	varchar	50	Not null, unique
slug	varchar	255	Not null, unique
price	int	-	Not null
is_deleted	boolean	-	Not null
Document : items			

Clé primaire: (id) // Clé(s) secondaire(s): (brand_id, category_id)



Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
user_id	int	-	Not null
item_id	int	-	Not null
rate	int	-	Not null
commentary	varchar	10	Not null
Document : commentaries			

Clé primaire: (id) // Clé(s) secondaire(s): (user_id, item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
user_id	int	-	Not null
item_id	int	-	Not null
Document : favorites			

Clé primaire: (id) // Clé(s) secondaire(s): (user_id, item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
order_id	int	-	Not null
item_id	int	-	Not null
name	varchar	50	Not null
total_price	int	-	Not null
quantity	int	-	Not null
size	varchar	10	Not null
Document : order_item			

Clé primaire: (id) // Clé(s) secondaire(s): (order_id, item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
tag_id	int	-	Not null
item_id	int	-	Not null
Document : item_tag			

Clé primaire: (id) // Clé(s) secondaire(s): (tag_id, item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
name	varchar	50	Not null, unique
Document : tags			

Clé primaire: (id)



Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
item_id	int	-	Not null
stock	int	-	Not null
size	varchar	10	Not null
Document : stocks			

Clé primaire: (id) // Clé(s) secondaire(s): (item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
item_id	int	-	Not null
url	varchar	255	Not null
Document : images			

Clé primaire: (id) // Clé(s) secondaire(s): (item_id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
name	varchar	50	Not null, unique
Document : brands			

Clé primaire: (id)

Nom de la données	Format	Longueur	Contraintes
id	int	-	Not null, unique
name	varchar	50	Not null, unique
Document : categories			

Clé primaire: (id)



Composants d'accès aux données (SQL, ORM Eloquent, Filament)

Dans Piwee, l'accès et la gestion des données reposent sur une combinaison d'outils puissants : l'ORM Eloquent de Laravel, le Query Builder et l'interface d'administration Filament, qui exploite ces couches pour offrir une expérience de gestion avancée.

Eloquent ORM

Description : Eloquent est l'ORM natif de Laravel. Il permet de **manipuler les tables SQL** sous forme **d'objets PHP** (modèles), simplifiant les opérations CRUD (Create, Read, Update, Delete) et la gestion des relations (ex : utilisateurs, commandes, favoris).

Rôle dans Piwee :

Chaque entité métier (User, Order, Item, Favorite, etc.) possède son **modèle Eloquent**.

Les **relations** (ex : un utilisateur a plusieurs commandes) sont déclarées dans les modèles, facilitant **les jointures et les requêtes complexes**. Les opérations sur les données (création de commande, ajout au panier, gestion des favoris) s'effectuent principalement via Eloquent.

Exemple :

php

```
$orders = Order::where('user_id', $userId)->with('items')->get();
```

Query Builder Laravel

Description : Le Query Builder permet d'écrire des requêtes SQL complexes en PHP, avec une syntaxe fluide et sécurisée.

Rôle dans Piwee :

Idéal pour les statistiques, les agrégats ou les requêtes multi-tables. Offre une protection contre les injections SQL.

Exemple

```
$stats = DB::table('orders')
    ->selectRaw('count(*) as total, status')
    ->groupBy('status')
    ->get();
```



```

public function registerPost(Request $request): RedirectResponse
{
    $request->validate([
        'firstname' => 'required|string|max:50',
        'lastname' => 'required|string|max:50',
        'birthday' => 'required|date|before:today',
        'gender' => 'required|string|in:homme,femme,autre',
        'address' => 'required|string|max:255',
        'zip' => 'required|string|max:50',
        'city' => 'required|string|max:50',
        'phone' => 'required|string|max:20',
        'email' => 'required|string|lowercase|email|max:255|unique:'.User::class,
        'password' => ['required', 'confirmed', Rules\Password::defaults()],
    ]);

    $user = User::create([
        'firstname' => $request->firstname,
        'lastname' => $request->lastname,
        'birthday' => $request->birthday,
        'gender' => $request->gender,
        'address' => $request->address,
        'zip' => $request->zip,
        'city' => $request->city,
        'phone' => $request->phone,
        'email' => $request->email,
        'password' => Hash::make($request->password),
        'verified' => false, // Valeur par défaut
        'role' => 'user', // Valeur par défaut
        'is_deleted' => false, // Valeur par défaut
    ]);
    event(new Registered($user));
    Auth::login($user);
    return redirect()->route('auth.login')->with('success', 'Votre compte a bien été créé');
}

```

Filament : interface d'administration et gestion des données

Description : Filament est un **framework d'interface d'administration** pour Laravel, basé sur **Livewire**, **Alpine.js** et **Tailwind CSS**. Il permet de créer rapidement des panneaux d'administration puissants, ergonomiques et personnalisables, sans écrire de JavaScript.

Rôle dans Piwee :

Filament s'appuie sur les **modèles Eloquent** pour **générer automatiquement des interfaces de gestion** (CRUD) : gestion des utilisateurs, des commandes, des stocks, des produits, etc.

Les administrateurs peuvent visualiser, filtrer, trier et éditer les données grâce à des tables interactives, des formulaires dynamiques et des dashboards personnalisés. Les ressources Filament sont **directement liées aux modèles Eloquent**, ce qui garantit la cohérence des accès et la sécurité des opérations (contrôle des permissions, validation des données).

Avantages :

- Productivité : accélère la création d'interfaces d'administration, **réduit le temps de développement**.
- Sécurité : s'appuie sur les politiques Laravel pour **contrôler l'accès aux données sensibles**.
- Extensibilité : possibilité d'ajouter des plugins, de personnaliser les composants, de créer des dashboards sur-mesure.

Piwee combine la puissance d'Eloquent pour la manipulation des données, la flexibilité du Query Builder et du SQL natif pour les besoins avancés, et l'efficacité de Filament pour l'administration et la gestion visuelle des données. Cette architecture garantit à la fois la robustesse, la sécurité et la rapidité de gestion, tout en offrant une expérience utilisateur optimale pour les administrateurs comme pour les développeurs



Extraits de code (accès données)

Eloquent

```
$item = Item::with(['category', 'brand', 'images', 'stocks'])->findOrFail($id);
```

Ici, on veut récupérer grâce au modèle «Item» et un id, les **données de l'article** avec (grâce aux relations) **la catégorie, la marque, les images et les détails du stock lié à l'article en question**. La méthode `findOrFail()` permet de rediriger **une erreur 404 si aucune correspondance n'est trouvée** (alors que la méthode `find()` retournera «null» en réponse.

Filament

```
public static function form(Form $form): Form
{
    return $form
        ->schema([
            Forms\Components\TextInput::make('name')->required(),
        ]);
}
```

Ici, ce morceau de code permet de créer une marque, par exemple. Il définit le **formulaire** d'ajout ou de modification d'une marque dans le backoffice : le champ saisi par l'administrateur sera automatiquement lié à la colonne name de la table brand.

Filament va **gérer automatiquement** :

- l'**affichage** du formulaire dans l'interface d'administration,
- la **validation** du champ (ici, le nom est requis grâce à `->required()`),
- la **liaison** entre la saisie de l'administrateur et la base de données (insertion ou mise à jour de la marque),
- et le **retour utilisateur** (messages de succès ou d'erreur, rechargement du listing...).

L'administrateur n'a donc pas besoin d'écrire de code supplémentaire pour la gestion du formulaire ou la persistance des données : Filament prend en charge toute la logique technique sous-jacente, ce qui accélère le développement et fiabilise la gestion des entités.



Composants métier (logique serveur, POO),

Client

Dans Piwee, la gestion du panier est centralisée dans le `CartController`, qui encapsule toute la logique métier liée au panier côté serveur. Ce composant permet à l'utilisateur de manipuler son panier de façon sécurisée et cohérente tout au long de son parcours d'achat.

Rôle métier du panier

- Permettre à l'utilisateur d'ajouter, retirer ou modifier la quantité d'articles avant achat.
- Conserver l'état du panier pendant toute la navigation (session).
- Garantir la cohérence métier : contrôle des stocks, gestion des variantes (taille, couleur), calcul des totaux.

1. Affichage du panier

La méthode `index()` récupère le contenu du panier stocké en session, identifie tous les produits uniques, puis charge depuis la base les informations détaillées de chaque article (nom, marque, catégorie, images, stocks, etc.).

Les données sont ensuite formatées et transmises à la vue via Inertia, permettant un affichage riche et actualisé du panier.

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Models\Item;
6 use Illuminate\Http\Request;
7 use Inertia\Inertia;
8
9 class CartController extends Controller
10 {
11     public function index()
12     {
13         $cart = session('cart', []);
14         $ids = collect($cart)->pluck('id')->unique()->all();
15         $items = Item::with(['brand', 'category', 'images', 'stocks'])
16             ->whereIn('id', $ids)
17             ->get()
18             ->map(function ($item) {
19                 return [
20                     'id' => $item->id,
21                     'name' => $item->name,
22                     'brand' => $item->brand ? ['name' => $item->brand->name] : null,
23                     'category' => $item->category ? ['id' => $item->category->id, 'name' => $item->category->name] : null,
24                     'price' => $item->price,
25                     'description' => $item->description,
26                     'image' => $item->images->first()
27                         ? asset('storage/' . $item->images->first()->url)
28                         : '/placeholder.jpg',
29                     'stocks' => $item->stocks->map(fn($s) => [
30                         'size' => $s->size,
31                         'stock' => $s->stock,
32                     ]),
33                 ];
34             });
35         return Inertia::render('cart', [
36             'cart' => $cart,
37             'items' => $items,
38         ]);
39     }
40 }
```

Cette logique garantit que l'utilisateur dispose toujours d'un aperçu fiable et enrichi de son panier, avec contrôle des stocks et des variantes.

2. Ajout d'un article au panier



La méthode `addToCart()` gère l'ajout d'un produit dans le panier :

Elle sécurise la quantité (minimum 1), récupère la taille choisie, puis ajoute autant d'entrées que la quantité demandée dans le panier stocké en session.

À chaque ajout, la session est mise à jour et l'utilisateur est redirigé avec un message de confirmation.

```
41 public function addToCart(Request $request, $id)
42 {
43     $quantity = max(1, (int) $request->input('quantity', 1));
44     $size = $request->input('size', null);
45     $cart = session()->get('cart', []);
46     for ($i = 0; $i < $quantity; $i++) {
47         $cart[] = [
48             'id' => (int) $id,
49             'size' => $size,
50         ];
51     }
52     session()->put('cart', $cart);
53     return redirect()->route('cart.index')->with('success', 'Produit ajouté');
54 }
55
```

Cette méthode garantit l'intégrité des données et la cohérence métier (quantité minimale, gestion des variantes).

3. Suppression d'un article du panier

La méthode `removeFromCart()` permet de retirer toutes les occurrences d'un produit (pour une taille donnée) du panier :

Elle filtre le panier pour supprimer les entrées correspondant à l'identifiant et à la taille spécifiés, puis réindexe le tableau et met à jour la session.

```
56 public function removeFromCart(Request $request, $id)
57 {
58     $size = $request->input('size');
59     $cart = session()->get('cart', []);
60     $cart = array_filter($cart, function ($entry) use ($id, $size) {
61         return !(
62             isset($entry['id'], $entry['size']) &&
63             $entry['id'] == $id &&
64             $entry['size'] == $size
65         );
66     });
67     $cart = array_values($cart);
68     session(['cart' => $cart]);
69     return redirect()->back();
70 }
```

Cette logique permet une gestion fine du panier, en tenant compte des variantes de produit.

4. Vider le panier

La méthode `clearCart()` supprime tout simplement la clé `cart` de la session, vidant ainsi complètement le panier de l'utilisateur.

```
74 public function clearCart()
75 {
76     session()->forget('cart');
77     return redirect()->back()->with('success', 'Panier vidé');
78 }
79
```



```

public function increment(Request $request)
{
    $id = $request->input('id');
    $size = $request->input('size');
    $cart = session()->get('cart', []);
    $cart[] = ['id' => (int)$id, 'size' => $size];
    session()->put('cart', $cart);
    return redirect()->route('cart.index');
}

public function decrement(Request $request)
{
    $id = $request->input('id');
    $size = $request->input('size');
    $cart = session()->get('cart', []);
    $index = array_search(['id' => (int)$id, 'size' => $size], $cart);
    if ($index !== false) {
        unset($cart[$index]);
        $cart = array_values($cart);
    }
    session()->put('cart', $cart);
    return redirect()->route('cart.index');
}

```

Résumé et intérêt de l'approche:

Centralisation de la logique métier : Toutes les opérations sur le panier sont regroupées dans un seul contrôleur, ce qui facilite la maintenance et l'évolution.

Utilisation de la session : Le panier reste persistant et sécurisé pendant toute la navigation, sans nécessiter de création de compte immédiate.

Sécurité et robustesse : Les entrées sont validées, les quantités contrôlées, et chaque action est suivie d'une mise à jour de la session.

POO et bonnes pratiques Laravel : Utilisation des modèles Eloquent, des collections, et des helpers Laravel pour manipuler efficacement les données.

En résumé :

Le panier de Piwee est un composant métier structuré en POO, isolant la logique serveur dans une classe dédiée, accessible via une interface claire, et utilisé par le controller pour toutes les opérations d'ajout, suppression, affichage et validation. Cette organisation garantit la maintenabilité, la sécurité et l'évolutivité du système.

A NOTER: Filament utilise Livewire pour son interface d'administration, tandis qu'Inertia.js fonctionne avec React pour le front-end. Les deux stacks sont donc indépendantes et incompatibles en termes de composants d'interface utilisateur

Composant métier backoffice : gestion des articles (ItemResource) et ses liaisons

Dans Piwee, la gestion des articles (produits du catalogue) côté administration est centralisée dans le backoffice grâce à Filament. Ce composant métier permet aux administrateurs de créer, modifier, organiser et superviser l'ensemble des articles, ainsi que leurs liaisons avec les marques, catégories, images et stocks.

1. Rôle métier du composant

- Permettre la création et la modification rapide des articles via un formulaire ergonomique.
- Gérer les liaisons essentielles : chaque article est relié à une marque, une catégorie, des images et des stocks (par taille).
- Superviser le catalogue : visualiser, filtrer, éditer et supprimer les articles selon les besoins.
- Assurer la gestion du cycle de vie de l'article (activation, suppression logique, réactivation).

2. Organisation POO et logique serveur

a) Modèle Eloquent Item

Le modèle Eloquent Item définit les relations avec les autres entités :

```

public function brand(): BelongsTo
{
    return $this->belongsTo(Brand::class);
}
public function category(): BelongsTo
{
    return $this->belongsTo(Category::class);
}
public function images(): HasMany
{
    return $this->hasMany(Image::class);
}
public function stocks()
{
    return $this->hasMany(Stock::class);
}
/**
 * Obtenir tous les favoris pour cet article.
 */
public function favorites()
{
    return $this->hasMany(Favorite::class);
}
/**
 * Obtenir tous les utilisateurs qui ont mis cet article en favori.
 */
public function favoritedBy()
{
    return $this->belongsToMany(User::class, 'favorites');
}

```

Ce code définit les relations Eloquent du modèle Item :

- Un article appartient à une marque et une catégorie,
- possède plusieurs images et stocks,
- peut avoir plusieurs favoris,
- permet d'obtenir tous les utilisateurs l'ayant ajouté à leurs favoris via une relation many-to-many.

b) Ressource Filament : centralisation de la logique métier

La ressource ItemResource regroupe toute la logique d'administration des articles :

Formulaire de création/édition

Tableau de bord (listing, tris, filtres)

Actions personnalisées (suppression logique, réactivation)

Gestion des relations (images, stocks)

2. Exemples concrets de gestion métier

a) Création/édition d'un article avec liaisons

Dans le formulaire Filament, les champs `brand_id` et `category_id` utilisent la méthode `->relationship` pour lier l'article à une marque ou une catégorie.

On peut même créer une nouvelle marque ou catégorie à la volée :

```

public static function table(Table $table): Table

```



```
Forms\Components\Select::make('brand_id')
->relationship('brand', 'name')
->searchable()
->preload()
->createOptionForm([
    Forms\Components\TextInput::make('name')
        ->label('Nom de la marque')
        ->required()
        ->maxLength(50),
])
->required(),
```

*L'administrateur peut choisir une marque existante ou en créer une nouvelle directement depuis le formulaire d'article.
Même logique pour la catégorie.*

b) Affichage des stocks disponibles par taille

Dans la table Filament, la colonne personnalisée tailles_disponibles affiche dynamiquement les tailles et les quantités en stock :

```
Tables\Columns\TextColumn::make('tailles_disponibles')
->label('Tailles disponibles')
->html()
->getStateUsing(function ($record) {
    return $record->stocks
        ->map(function ($stock) {
            $size = $stock->size;
            $count = $stock->stock;
            return "<span
                class='inline-block px-2 py-1 bg-gray-200 text-gray-600 rounded text-xs font-semibold mr-1 mb-1'
                title='Stock disponible : $count'
            >$size</span>";
        })
        ->implode(' ');
}),
```

Pour chaque article, on parcourt les stocks associés et on affiche chaque taille disponible avec un badge, le tout en HTML pour une meilleure lisibilité. Un tooltip affiche la quantité restante pour chaque taille.

c) Suppression logique et réactivation d'un article

Deux actions personnalisées sont proposées dans le formulaire :

Marquer comme supprimé (sans effacer la donnée en base)

Réactiver (rendre l'article à nouveau disponible)

```
Forms\Components\Actions::make([
    FormAction::make('marquerCommeSupprime')
        ->label('Marquer comme supprimé')
        ->color('danger')
        ->icon('heroicon-o-trash')
        ->visible(fn ($record) => $record && !$record->isDeleted)
        ->requiresConfirmation()
        ->action(function ($get, $record, $set) {
            $record->isDeleted = true;
            $record->save();
            $set('isDeleted', true);

            Notification::make()
                ->title('L\'item a été marqué comme supprimé.')
                ->success()
                ->send();
        }),
    FormAction::make('reactiver')
        ->label('Réactiver')
        ->color('success')
        ->icon('heroicon-o-arrow-path')
        ->visible(fn ($record) => $record && $record->isDeleted)
        ->requiresConfirmation()
        ->action(function ($get, $record, $set) {
            $record->isDeleted = false;
            $record->save();
            $set('isDeleted', false);

            Notification::make()
                ->title('L\'item a été réactivé.')
                ->success()
                ->send();
        }),
])
->visible(fn ($record) => $record != null),
```

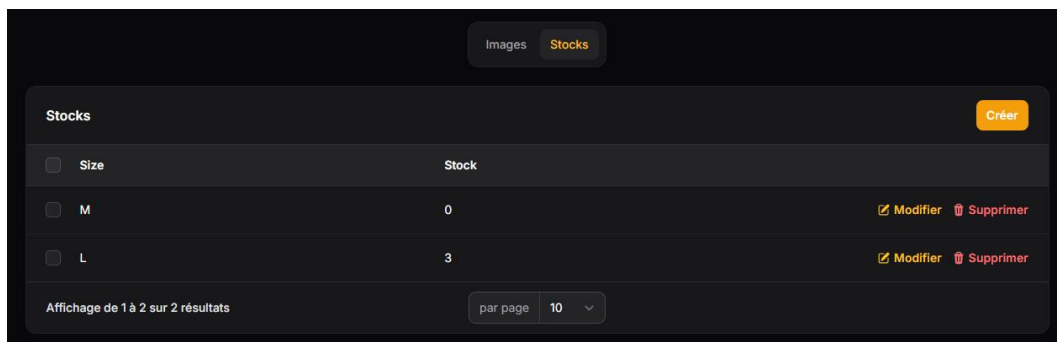
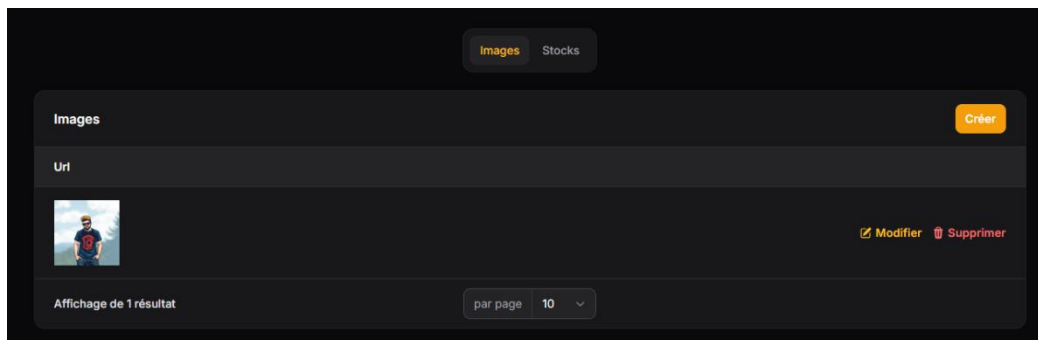


Ces actions évitent la suppression physique des articles, ce qui permet de conserver l'historique et de réactiver un produit si besoin. L'utilisateur reçoit une notification de confirmation à chaque action.

d) Gestion des relations images et stocks

Les relations sont gérées via des RelationManagers Filament, qui permettent d'ajouter, modifier ou supprimer les images et les stocks directement depuis la fiche article.

```
public static function getRelations(): array
{
    return [
        RelationManagers\ImagesRelationManager::class,
        RelationManagers\StocksRelationManager::class,
    ];
}
```



L'administrateur peut gérer toutes les images d'un article (ajout, suppression, modification de l'ordre).

Il peut aussi gérer les stocks (ajouter une nouvelle taille, ajuster la quantité, etc.) sans quitter la fiche produit.

4. Avantages de cette approche

Centralisation et cohérence : Toute la gestion des articles et de leurs liaisons est réunie dans une ressource unique, ce qui garantit la cohérence métier et la facilité de maintenance.

Productivité : Filament automatise la génération des formulaires, des tables et des actions, tout en permettant une personnalisation avancée.

Sécurité et traçabilité : Les suppressions sont logiques, les notifications informent l'administrateur, et les relations sont gérées de façon transparente.

Extensibilité : Il est facile d'ajouter de nouvelles fonctionnalités (tags, commentaires, promotions...) ou de modifier la structure sans remettre en cause l'ensemble du système.

En résumé



Le composant métier "gestion des articles" de Piwee, construit avec Filament, s'appuie sur la programmation orientée objet, l'ORM Eloquent et une interface d'administration moderne pour offrir une gestion complète, sécurisée et évolutive du catalogue produits et de toutes ses liaisons (marques, catégories, images, stocks). L'administrateur bénéficie d'une expérience fluide et puissante, tout en garantissant la qualité et la cohérence des données du site.

Sécurisation côté serveur (auth, validation, etc.)

La sécurité côté serveur dans Piwee est pensée pour répondre aux exigences concrètes d'un site e-commerce moderne, en s'appuyant sur les outils natifs de Laravel et sur des mesures adaptées à la gestion de comptes, de commandes et de données sensibles.

1. Authentification et gestion des accès

Authentification Laravel :

Piwee utilise le système d'authentification intégré de Laravel : inscription, connexion, réinitialisation de mot de passe, validation d'email. Les mots de passe sont stockés hachés (bcrypt/argon2), et la vérification de l'email est obligatoire pour activer un compte.



Gestion des rôles et permissions :

Les accès aux fonctionnalités sensibles (commandes, backoffice, profils) sont contrôlés par des politiques Laravel :

Un utilisateur ne peut accéder qu'à ses propres commandes (OrderPolicy).

Les routes d'administration sont protégées par des middlewares vérifiant le rôle administrateur.

Suppression logique des comptes :

La suppression d'un compte utilisateur ne supprime pas les données mais active le flag `is_deleted`, assurant la traçabilité et la conformité RGPD.

2. Validation et filtrage des données

FormRequest centralisés

Toutes les entrées utilisateur (création de compte, profil, panier, favoris...) sont validées via des classes `FormRequest` dédiées. Cela inclut :

- Format des emails, unicité, longueur des champs
- Validation des quantités et des stocks lors de l'ajout au panier
- Contrôle des statuts lors de la modification d'une commande

Échappement des sorties

Toutes les données affichées dans les vues (Blade, Inertia, Filament) sont échappées par défaut, empêchant l'exécution de scripts malveillants (protection XSS).

3. Protection contre les attaques courantes

CSRF (Cross-Site Request Forgery) :

Laravel intègre par défaut la protection CSRF via un token unique pour chaque session utilisateur, inclus dans tous les formulaires et vérifié côté serveur.

SQL Injection :

L'utilisation d'Eloquent ORM et du Query Builder protège contre les injections SQL en utilisant des requêtes préparées et des bindings automatiques.

Sécurisation des fichiers uploadés :

Les images et fichiers uploadés (ex : photos d'articles) sont stockés dans un dossier protégé, avec vérification du type et de la taille côté serveur.

4. Sécurité des API et du backoffice

Backoffice Filament sécurisé

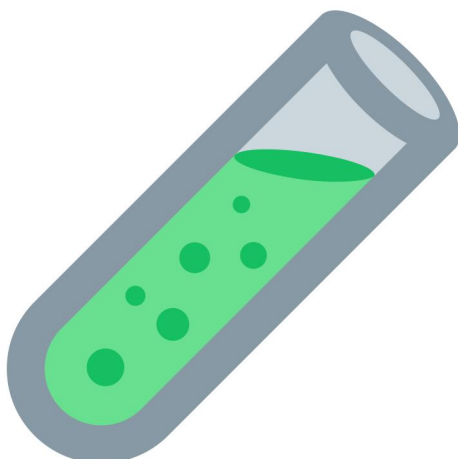
L'accès au backoffice est restreint aux administrateurs authentifiés. Les actions sensibles (modification des stocks, suppression d'articles) sont protégées par des confirmations et des politiques.

Sécurisation des routes API

Les routes API (favoris, commandes, etc.) sont protégées par des middlewares d'authentification et de vérification des droits. Aucune action critique n'est accessible sans contrôle serveur.







TESTS ET DEPLOIEMENT

Jeu d'essai fonctionnel

Pour garantir la qualité et la stabilité de l'application Piwee, des jeux d'essai fonctionnels sont mis en place grâce aux seeders Laravel. Les seeders permettent de pré-remplir la base de données avec des données de test cohérentes, facilitant ainsi le développement, les tests et la démonstration des fonctionnalités.

Qu'est-ce qu'un seeder ?

Un seeder est une classe Laravel qui insère automatiquement des données dans la base, selon une structure définie. Cela permet :

- de simuler un environnement réel (marques, produits, utilisateurs, commandes...),
- de tester les fonctionnalités sans avoir à saisir manuellement les données,
- de réinitialiser rapidement la base lors des phases de développement.



Voici un exemple de seeder pour la table `brands` (Marques):

```
1 <?php
2
3 namespace Database\Seeders;
4
5 use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6 use App\Models\Brand;
7 use Illuminate\Database\Seeder;
8
9 class BrandSeeder extends Seeder
10 {
11     public function run(): void
12     {
13         $brands = [
14             ['id' => 1, 'name' => 'Nike'],
15             ['id' => 2, 'name' => 'Adidas'],
16             ['id' => 3, 'name' => 'Puma'],
17             ['id' => 4, 'name' => 'Reebok'],
18             ['id' => 5, 'name' => 'Under Armour'],
19             ['id' => 6, 'name' => 'New Balance'],
20             ['id' => 7, 'name' => 'Asics'],
21             ['id' => 8, 'name' => 'Converse'],
22             ['id' => 9, 'name' => 'Vans'],
23             ['id' => 10, 'name' => 'Skechers'],
24             ['id' => 11, 'name' => 'Fila'],
25             ['id' => 12, 'name' => 'Kappa'],
26             ['id' => 13, 'name' => 'Lacoste'],
27             ['id' => 14, 'name' => 'Le Coq Sportif'],
28             ['id' => 15, 'name' => 'Diadora'],
29             ['id' => 16, 'name' => 'K-Swiss'],
30             ['id' => 17, 'name' => 'Onitsuka Tiger'],
31             ['id' => 18, 'name' => 'Hoka One One'],
32             ['id' => 19, 'name' => 'Salomon'],
33             ['id' => 20, 'name' => 'Mey'],
34         ];
35
36         foreach ($brands as $brand) {
37             Brand::updateOrCreate(['id' => $brand['id']], $brand);
38         }
39     }
40 }
```

Explications :

Le tableau `$brands` contient une liste de marques avec leur identifiant et leur nom.

La boucle `foreach` parcourt chaque marque et utilise `updateOrCreate` pour insérer la marque si elle n'existe pas, ou la mettre à jour si elle existe déjà.

Ce mécanisme garantit l'unicité des marques et évite les doublons lors de plusieurs exécutions du seeder.

Utilité dans le cycle de développement

Tests automatisés : Les seeders assurent que chaque développeur ou testeur travaille sur une base identique, ce qui rend les tests reproductibles et fiables.

Déploiement : Lors du déploiement sur un nouvel environnement (préproduction, production), les seeders peuvent être utilisés pour initialiser les données de référence.

Démonstration : Ils facilitent la présentation de l'application avec des données réalistes, sans risque de manipuler des données sensibles.

En résumé

Les seeders sont essentiels pour automatiser la création de jeux d'essai fonctionnels, accélérer les tests, fiabiliser le développement et garantir la cohérence des données sur tous les environnements. Ils font partie intégrante de la stratégie de tests et de déploiement de Piwee.



Stratégie de déploiement en local

Pour Piwee, le déploiement local est simplifié grâce à l'utilisation du système intégré de gestion de base de données SQLite fourni par Laravel. Cette approche permet à chaque développeur de travailler rapidement, sans configuration complexe ni dépendance à des outils externes.

1. Initialisation du projet en local

Clonage du dépôt

Récupérez le code source du projet via Git.

Installation des dépendances

-Backend :

```
bash
composer install
```

-Frontend :

```
bash
npm install
npm run dev
```

2. Configuration de l'environnement

2. Configuration de l'environnement

- Fichier d'environnement
 - Copiez le fichier d'exemple:

```
bash
cp .env.example .env
```
 - Dans `.env`, configurez la base de données pour utiliser SQLite:

```
text
DB_CONNECTION=sqlite
DB_DATABASE=database/database.sqlite
```
 - Créez le fichier de base de données vide:

```
bash
touch database/database.sqlite
```

3. Migration et seeders

Création de la structure de la base

```
bash
php artisan migrate
```

Remplissage avec des données de test



```
bash
php artisan db:seed
```

Les seeders injectent des jeux d'essai fonctionnels (marques, articles, utilisateurs...) pour permettre de tester toutes les fonctionnalités de Piwee.

4. Lancement de l'application

Démarrage du serveur et du front en une commande

Utilisez :

```
bash
composer run dev
```

Cette commande va lancer à la fois le serveur Laravel (`php artisan serve`) et le serveur de développement front (`npm run dev`), généralement dans deux terminaux ou processus parallèles.

Cela permet de compiler les assets front (JS, CSS) à la volée, de bénéficier du hot-reload, et d'accéder à l'application sur `http://localhost:8000` (backend) et/ou le port Vite (frontend).

5. Avantages de cette approche

Simplicité : Un seul fichier `.sqlite` à gérer, et une seule commande pour lancer toute la stack de développement.

Productivité : Pas besoin de lancer plusieurs terminaux ou de gérer des services externes : tout est intégré à Laravel et Node.js.

Portabilité : Chaque membre de l'équipe peut démarrer rapidement, sur n'importe quel OS, sans configuration complexe.

6. Procédure type pour un nouveau développeur

Cloner le dépôt et installer les dépendances.

Copier le fichier `.env` et configurer SQLite.

Créer le fichier `database.sqlite` si il n'est pas créé au lancement des serveurs ou des migrations.

Lancer les migrations et les seeders.

Démarrer le serveur Laravel intégré.

Accéder à l'application et commencer à développer ou tester.

En résumé

L'utilisation de SQLite et de la commande personnalisée `composer run dev` permet à chaque développeur de lancer l'environnement local complet de Piwee en une seule étape, avec gestion automatique du backend et du frontend, pour un développement fluide et sans friction





Veille technologique & sécurité



Recherches menées pendant le projet

Tout au long du développement de Piwee, plusieurs axes de recherche ont été explorés afin d'assurer la robustesse, la modernité et la qualité de la solution, tant sur le plan technique que fonctionnel.

1. Comparaison des architectures front/back pour SPA e-commerce

L'objectif étant de trouver une architecture moderne et adaptée à un développeur seul afin de correspondre à la réalité du marché et à mes délais

Laravel

Base de projet monolithique, c'est-à-dire que le back end et le front end sont regroupés dans une seule et même application. Ce choix présente plusieurs avantages : simplicité de déploiement, maintenance facilitée, cohérence des accès et de la sécurité, et rapidité de développement pour un développeur seul.

Source:

- *Sambeau (formateur) .*
- *<https://www.startechup.com/fr/blog/monolithic-vs-microservices-architecture/>.*
- *<https://aws.amazon.com/fr/compare/the-difference-between-monolithic-and-microservices-architecture/#:~:text=L'architecture%20monolithique%20limite%20la,métier%20dans%20les%20applications%20existantes.>*

Justification du choix d'Inertia.js et de React

Après analyse, la solution Inertia.js/React a été retenue pour plusieurs raisons :

Inertia.js permet de bénéficier de la puissance de Laravel côté back end tout en profitant de l'expérience utilisateur moderne offerte par React côté front end, sans avoir à développer une API REST séparée.

Ce choix simplifie la synchronisation des données, la gestion des routes et la transmission des états entre le serveur et l'interface utilisateur.

Préférence personnelle pour React et Js eu lieu de Ts : sa modularité, sa communauté active et la facilité de création de composants réutilisables ont pesé dans la décision. J'ai aussi lors de mes recherches trouvé plusieurs méthodes pour convertir la structure Ts en Js.

Sources:

- *https://dev.to/mohammad_naim_443ffb5d105/inertiajs-vs-restful-api-choosing-the-right-approach-for-your-laravel-application-4khj.*
- *<https://medium.com/@yazidkhaldi/convert-laravel-react-starter-kit-from-typescript-to-javascript-1691d38e5d3d>*

En résumé

Le projet utilise une architecture full-stack moderne combinant un backend et l'approche monolithique de Laravel (MVC) avec un frontend SPA React, reliés par Inertia.js qui élimine le besoin d'APIs REST traditionnelles.

2. Recherche sur la synchronisation des favoris entre onglets



Un des défis UX identifiés était la synchronisation des favoris en temps réel entre plusieurs onglets du navigateur.

Nous avons mené des recherches sur :

L'utilisation de l'événement `storage` du navigateur pour détecter les changements dans `localStorage`

La création d'un hook React personnalisé pour gérer l'état des favoris et notifier les autres onglets lors des modifications

Les stratégies de notification et de gestion des conflits d'état

Cette recherche a permis d'implémenter une synchronisation fiable des favoris, améliorant l'expérience utilisateur multi-onglets.

Sources:

- <https://dev.to/mattlewandowski93/persistence-pays-off-react-components-with-local-storage-sync-2bfk>.
- <https://dev.to/vikirobles/creating-favourites-with-local-storage-and-use-ref-in-react-1c3d>.
- <https://stackoverflow.com/questions/76474923/react-sync-with-the-localstorage>.

3. Étude des bonnes pratiques d'accessibilité et de contraste

Pour anticiper la réglementation de juin 2025 et garantir un site inclusif, nous avons approfondi :

L'intégration des attributs ARIA pour les composants dynamiques React

Les règles de contraste WebAIM/WCAG pour les couleurs, testées systématiquement avec le Color Contrast Checker

La navigation clavier et la compatibilité avec les lecteurs d'écran, en adaptant les composants DaisyUI/Tailwind

Cela a permis de bâtir une interface accessible et conforme aux standards actuels.

Sources:

- <https://inside.lanecce.edu/atc/software/t/13295>
- <https://johnwilsondesign.co.uk/colour-contrast-checker/>
- https://accessate.net/r2056/webaim_color_contrast_checker



Veille sécurité (faille identifiée, correction appliquée)

La sécurité a été une préoccupation constante tout au long du développement de Piwee. Voici un exemple concret de veille sécurité menée sur le projet, illustrant la détection d'une faille et la mise en place d'une correction adaptée.

Exemple 1

1. Faille identifiée : Injection de données non filtrées (XSS)

Lors d'un audit du code, une faille de type Cross-Site Scripting (XSS) a été repérée sur le backoffice : Dans le formulaire d'ajout/modification d'un article, le champ description acceptait du texte libre. Or, si un administrateur malveillant ou peu attentif saisisait du code HTML ou JavaScript, ce contenu pouvait être affiché sans filtrage dans la liste des articles ou sur la fiche produit.

Risque :

Exécution de scripts malveillants dans le navigateur d'un autre utilisateur (vol de session, redirection, phishing...)

Dégradation de l'intégrité de l'interface d'administration

2. Correction appliquée

Pour corriger cette faille, plusieurs mesures ont été mises en place :

Échappement systématique des sorties

Toutes les valeurs affichées dans les vues Filament (tableau, formulaire, etc.) utilisent désormais l'échappement par défaut `{{ $description }}` au lieu de `{!! $description !!}`, empêchant l'exécution de scripts injectés.

Validation renforcée côté serveur

Le `FormRequest` associé au modèle vérifie que la description ne contient pas de balises interdites ou de scripts suspects :

```
$request->validate([
    'description' => [
        'required',
        'string',
        'max:1000',
        'not_regex:/<script\b[^>]*>(.*?)</script>/i',
    ],
]);
```

3. Résultat et bonnes pratiques adoptées

Suppression immédiate du risque XSS sur tous les champs texte affichés dans l'admin.

Généralisation de la validation sur tous les formulaires sensibles (utilisateurs, articles, commentaires...).



Exemple 2

1. Faille identifiée : accès non restreint aux commandes d'autres utilisateurs

Lors des tests, il a été constaté que l'interface pouvait afficher les commandes sans vérifier que l'utilisateur connecté était bien le propriétaire. Cela exposait les données personnelles et les informations sensibles des commandes d'autres clients.

Risque:

- Fuite de données personnelles
- Atteinte à la confidentialité et à la confiance des utilisateurs

2. Correction appliquée

Mise en place d'une policy Laravel (`OrderPolicy`) qui vérifie systématiquement que l'utilisateur authentifié est bien le propriétaire de la commande avant d'autoriser l'accès ou l'affichage.

Contrôle côté controller : avant de retourner les données, on vérifie que `$order->user_id === auth()->id()`, sinon la requête est bloquée avec une erreur 403.

Filtrage dans les requêtes : les listes de commandes récupèrent uniquement celles liées à l'utilisateur connecté (`where('user_id', auth()->id())`).

3. Résultat et bonnes pratiques adoptées

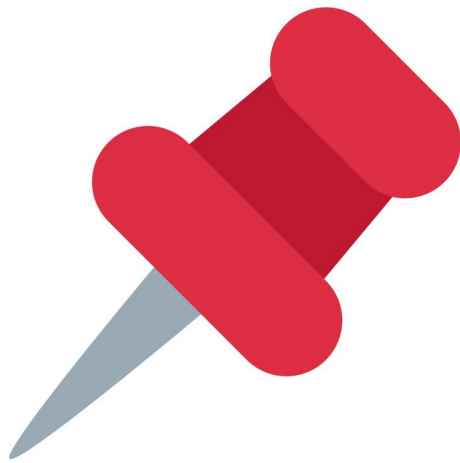
Sécurisation stricte de l'accès aux commandes

Respect de la confidentialité des données utilisateurs

Adoption d'une politique de contrôle d'accès systématique sur toutes les ressources sensibles

Cette correction garantit que chaque utilisateur ne peut consulter que ses propres commandes, protégeant ainsi les données personnelles et renforçant la confiance dans la plateforme.





CONCLUSION



Difficultés rencontrées et résolutions

Gestion de la synchronisation des favoris entre onglets

L'un des défis majeurs a été d'assurer une synchronisation fiable des favoris lorsque l'utilisateur ouvrait plusieurs onglets ou navigateurs. Après plusieurs essais, la solution retenue combine l'écoute de l'événement storage du navigateur et un hook React personnalisé, permettant une mise à jour instantanée de l'état des favoris sur tous les onglets ouverts.

Sécurisation des données et prévention des failles XSS

Lors des phases de développement du backoffice, une faille XSS a été identifiée sur les champs texte libres (ex : description d'article). La correction a consisté à renforcer la validation côté serveur et à systématiser l'échappement des sorties dans les vues Filament, éliminant tout risque d'exécution de scripts malveillants.

Accessibilité et conformité réglementaire

Adapter l'interface pour répondre aux normes d'accessibilité (contraste, navigation clavier, ARIA) a nécessité des recherches approfondies et des tests réguliers. L'intégration d'outils comme le Color Contrast Checker et l'adaptation des composants DaisyUI/Tailwind ont permis d'atteindre les standards attendus.

Simplicité du déploiement local pour tous

Pour éviter les problèmes de configuration multi-OS, le choix de SQLite et de la commande unique composer run dev a permis à chaque membre de l'équipe de démarrer rapidement, sans dépendances externes complexes.

Apports personnels (techniques, humains)

Montée en compétences sur Laravel, React et Inertia.js

Le projet m'a permis de consolider mes connaissances sur l'écosystème Laravel et d'approfondir l'intégration de React via Inertia.js, en mettant en place une architecture monolithique moderne et efficace.

Maîtrise des outils d'administration modernes

L'utilisation de Filament pour le backoffice m'a appris à concevoir des interfaces d'administration robustes, ergonomiques et sécurisées, tout en automatisant la gestion des entités métier. Cet outil peut aussi être utilisé pour créer des sites complets. Un vrai plus pour mon CV!

Développement d'une démarche de veille et de sécurité

J'ai mis en place une routine de revue de code et de veille sur les failles de sécurité, les outils et possibilités adaptable à mon environnement de travail.



Travail d'équipe et communication

Même sur un projet personnel, la rédaction de documentation claire, la structuration du code et la préparation à l'accueil de nouveaux contributeurs ont été essentiels pour garantir la pérennité et la maintenabilité du projet.

Conclusion & ouverture (évolutions possibles)

Ce projet m'a permis de mener à bien un site e-commerce complet, moderne et sécurisé, en combinant les meilleures pratiques du développement web actuel : architecture monolithique Laravel, front dynamique React, administration avancée avec Filament, et respect des exigences de sécurité et d'accessibilité.

Ouvertures et évolutions possibles :

- Ajout du paiement en ligne (Stripe, PayPal) pour finaliser le parcours client.
- Gestion avancée des retours et annulations pour une expérience utilisateur complète.
- Mise en place d'un dashboard statistique pour le suivi des ventes et l'aide à la décision.
- Recommandations personnalisées basées sur les favoris et les historiques d'achat.
- Amélioration continue de l'accessibilité en anticipation des évolutions réglementaires.
- Déploiement sur des environnements cloud et automatisation des tests via CI/CD.

Ce projet constitue une base solide, évolutive et conforme aux attentes du e-commerce moderne, tout en ouvrant la voie à de nombreuses extensions fonctionnelles et techniques pour l'avenir.

Drivee!

