RAPPORT DE STAGE

Prénom : Enki

Nom : Robert

Filière : BTS SIO 1ère Année

Ecole : Campus Carlo Acutis

Dates du stage : 18 Mai au 26 Juin 2026

Entreprise d'accueil : Zero Limit

Remerciements

Je tiens à remercier l’ensemble du collectif Zero Limit pour m’avoir accueilli durant mon stage, et plus particulièrement Jérôme Fompudie pour sa disponibilité et son aide tout au long de cette période.

Je remercie également mes enseignants pour leur suivi et leurs recommandations, qui m’ont permis de mener à bien ce projet.

Enfin, je remercie toutes les personnes qui ont contribué, de près ou de loin, au bon déroulement de mon stage.

Sommaire

I. Présentation de l'entreprise 4
II. Présentation du projet 5
III. Travail effectué 6
1. Faire une maquette 6
2. Trouver un framework 8
3. Créer une vue « base » 9
4. Créer une fonction « Compte » 10
5. Créer et gérer les produits 13
6. Détails du produits 17
7. Mettre en place une methode de paiement 18
8. Gérer les paniers 23
IV. Bilan du sujet 27
V. Bilian général 29

I. **Présentation de l’entreprise**

Zero Limit est un collectif basé à Tours, créé en juillet 2025 par Jérôme Fompudie et Sullivan Poirier. Depuis avril 2026, Jérôme Fompudie poursuit seul le développement du projet.

Le collectif évolue dans le domaine de la musique électronique, plus précisément dans l’univers de la Hard Music. Son activité principale consiste à organiser des événements festifs tels que des raves.

Zero Limit a pour objectif de développer sa visibilité et de fédérer une communauté autour de ses événements. Dans cette optique, le collectif souhaite diversifier ses activités, notamment à travers la vente de produits dérivés (merchandising) en ligne à partir de l’été 2026.

Le collectif fonctionne de manière indépendante et s’appuie sur différents prestataires pour la communication, l’organisation des événements et la production visuelle.

II. Présentation du projet

Le projet confié durant ce stage consistait à concevoir et développer un site web pour le collectif Zero Limit.

Ce site a pour objectif principal de permettre la vente de produits dérivés (merchandising). Il doit également proposer des fonctionnalités complémentaires, telles que la présentation des événements passés et à venir, ainsi que la publication d'actualités liées au collectif.

Ce projet répond à un besoin de visibilité et de professionnalisation du collectif, en lui permettant de disposer d'une plateforme centralisée pour communiquer et vendre ses produits.

La réalisation du site a été organisée en plusieurs étapes :

- Création d'une maquette visuelle validée par le client
- Choix des technologies utilisées
- Développement du site en PHP
- Mise en place d'un système de paiement sécurisé

III. Travail effectué

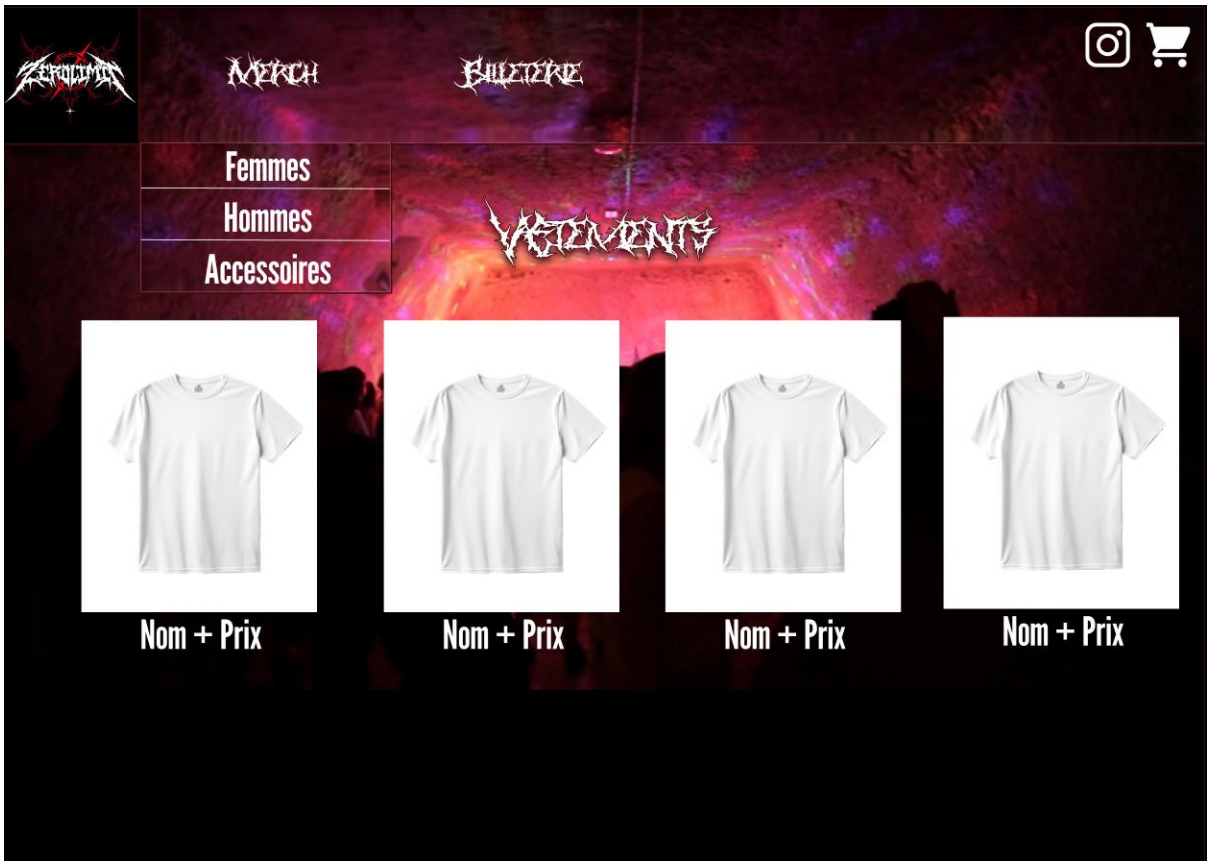
1. Faire une maquette

La première chose que j'ai faite a été de créer une maquette via l'application Figma




MERCH
BILLETERIE
event 1
event 2
event 3
NOM EVENT

Shotgun API


MERCH
BILLETERIE
Femmes
Hommes
Accessoires
VÊTEMENTS
Nom + Prix
Nom + Prix
Nom + Prix
Nom + Prix

Les visuels permettant de donner à moi et à mon client une première idée de l'aspect visuel du site, même s'ils se sont éloignés peu à peu du résultat final par soucis d'épuration.
Voici les idées que je proposais pour le projet :

- Page d'accueil avec des nouvelles concernant le collectif
- Page Merch avec des sous-catégories (Femme/Homme/Accessoires/Etc)
- Page Billetterie avec une API relié à l'application de billeterie Shotgun (actuellement la plus populaire pour ce genre d'évènement)
- Page Réseaux (finalement retirée et remplacé par un logo redirigeant vers le Instagram, étant le seul réseau utilisé par le collectif)

2. Trouver un framework

La première chose que j'ai dû faire pour créer mon projet était de trouver un framework.
Un framework (ou infrastructure logicielle) est un ensemble de composant logiciels agissant comme un squelette pour le projet tout en apportant différents outils afin de faciliter le développement.
Chaque framework étant spécialisé dans des langages et des types de projets différents.

Les framework les plus utilisés dans le développement PHP étant Laravel et Symfony. J'ai choisi Laravel car fonctionnant en architecture MVC (modèle-vue-contrôleur), architecture avec laquelle j'étais déjà à l'aise.
Outils de Laravel :

- Artisan :

Artisan est une interface en ligne de commande permettant d'interagir avec le projet pour mettre en route le projet, ajouter des modèles, des contrôleurs, effectuer des migrations ou récupérer des information comme les différents routes actives.

Les commandes se font sur un invité de commande avec le préfixe « php artisan ».

- Eloquent :

Eloquent est une ORM (Object-Relationnal Mapping), c'est-à-dire une couche logicielle qui fait le lien entre la base de donnée et le code. Il permet d'interagir avec la base de donnée sans avoir à écrire de SQL et donc de manière beaucoup plus fluide.

- Blade :

Blade est un moteur de template, c'est-à-dire qu'il permet de créer des template de code réutilisable et adaptable, ce qui permet notamment de récupérer une base pour les différentes pages du site sans avoir à copier-coller le code à chaque fois.
Il donne également accès à des commandes en PHP simplifié (comme @if ou @foreach).

3. Créer une vue « base »

J'utilise Blade pour créer une vue avec la barre de navigation et le fond d'écran que je pourrais utiliser comme template pour base pour le reste du site. J'utilise pour ça la fonction @extends('base') au début de chaque autre vue pour reprendre le code de la vue 'base' puis la fonction @yield qui me premet de créer des section que je peux modifier sur d'autre page avec la fonction @yield. Par exemple, je crée un @yield('title') qui me servira a mettre le titres de mes pages avec @section('title',

‘Nom de la page’). Je fais également un @yield(‘content’) qui me servira à mettre le contenu de ma page.

J’utilise Tailwind CSS, qui me permet de faire du CSS simplifié directement dans les attributs de mes éléments HTML.

Je crée une barre de navigation avec un bouton connexion, le logo (qui redirige vers le menu) et un bouton panier.



J’ai simplifié par rapport à la maquette pour coller plus à la demande initiale sans rajouter des fonctionnalités qui n’auraient de toute façon pas été exploitées par le client.

4. Créer une fonction « Compte »

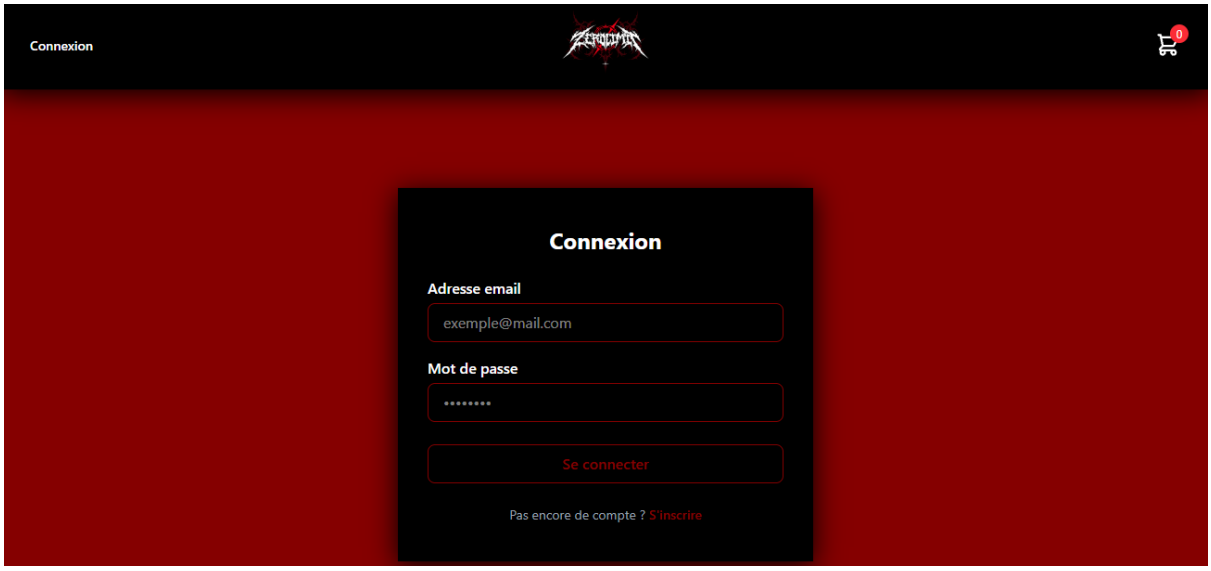
Créer une fonction de compte est la première étape car elle permet de mettre en place des administrateurs qui auront accès à une interface différence, pour ajouter et modifier des produits. J’ai donc créé une vue 'login.blade.php' et 'register.blade.php', regroupées dans un dossier 'account'. Ces deux vues contenant un formulaire pour la connexion et l'inscription de l'utilisateur. Laravel prenant déjà en charge la gestion de utilisateur via la classe Auth, je rajoute simplement l’attribut ‘role’ dans le model User existant et je crée un AccountController qui me permet d’y intégrer les fonction me permettant :

- D’afficher le formulaire de connexion
- De récupérer les informations du formulaire pour se connecter
- D’afficher le formulaire d’inscription

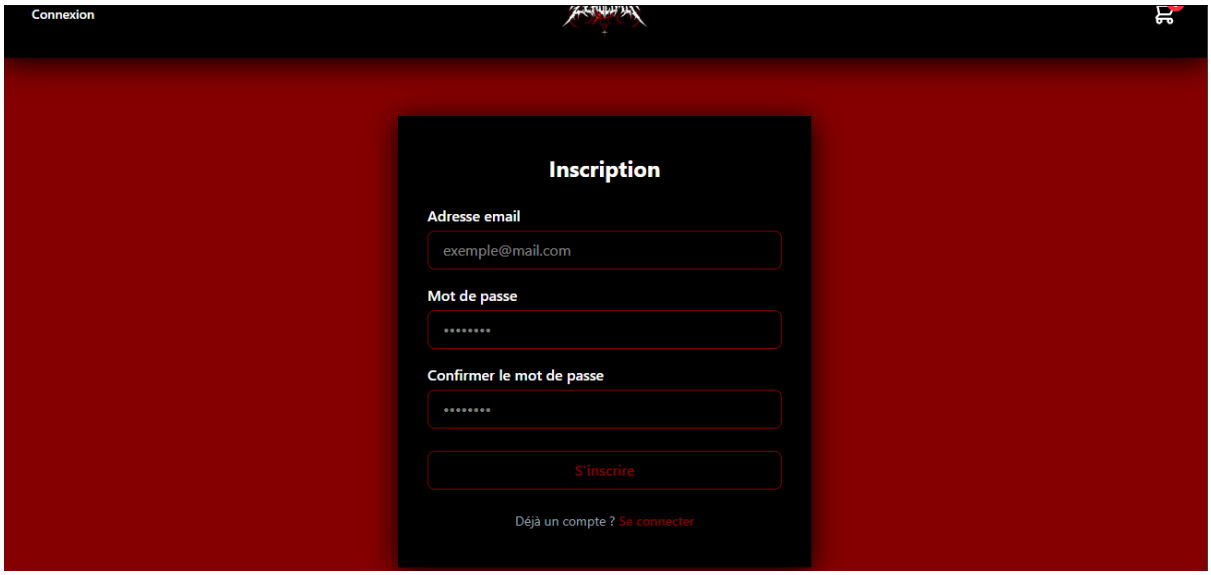
- De récupérer et de vérifier que les informations d'inscription soient pour créer et connecter l'utilisateur
- De se déconnecter

Les méthodes importantes pour cela sont :

- La classe Request qui permet de récupérer les données envoyés dans la requête
- Validate() qui permet de vérifier la conformité des données envoyés via des paramètres qu'on applique directement pour chaque donnée
- Auth :attempt() qui connecte l'utilisateur sur la session avec les données récupérées
- User :create qui crée un objet utilisateur et l'ajoute à la base de données
- Auth :login() qui permet de connecter un utilisateur directement avec un objet utilisateur
- Auth :logout() pour déconnecter l'utilisateur de la session
- bcrypt() pour hasher le mot de passe dans la base de données



Et la vue Inscription dans le même style :



Si il y a une erreur elle est affichée directement pour l'utilisateur :

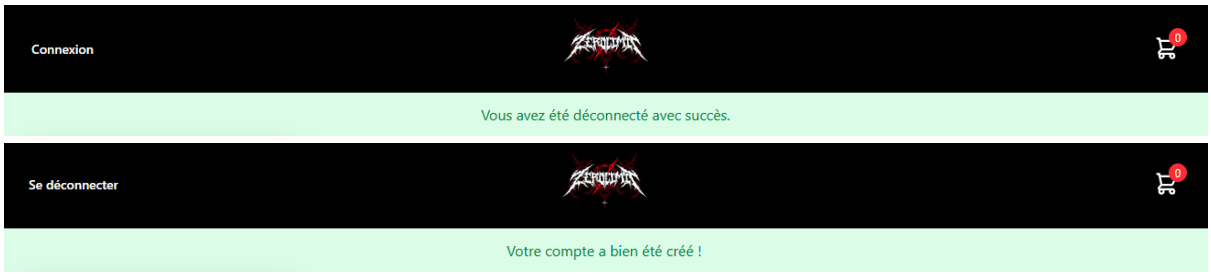


L'inscription possède les contraintes suivantes :

- Email qui n'est pas déjà utilisé
- Le mot de passe doit faire entre 8 et 72 caractères et posséder au moins une minuscule, une majuscule, un chiffre et un caractère spécial
- La confirmation de mot de passe doit correspondre

J'utilise 'session('success')' et 'session('error)' dans ma vue base qui me permet d'afficher un message à l'utilisateur de manière

unique, ici elle me permet de confirmer à l'utilisateur la déconnexion et la création d'un compte :

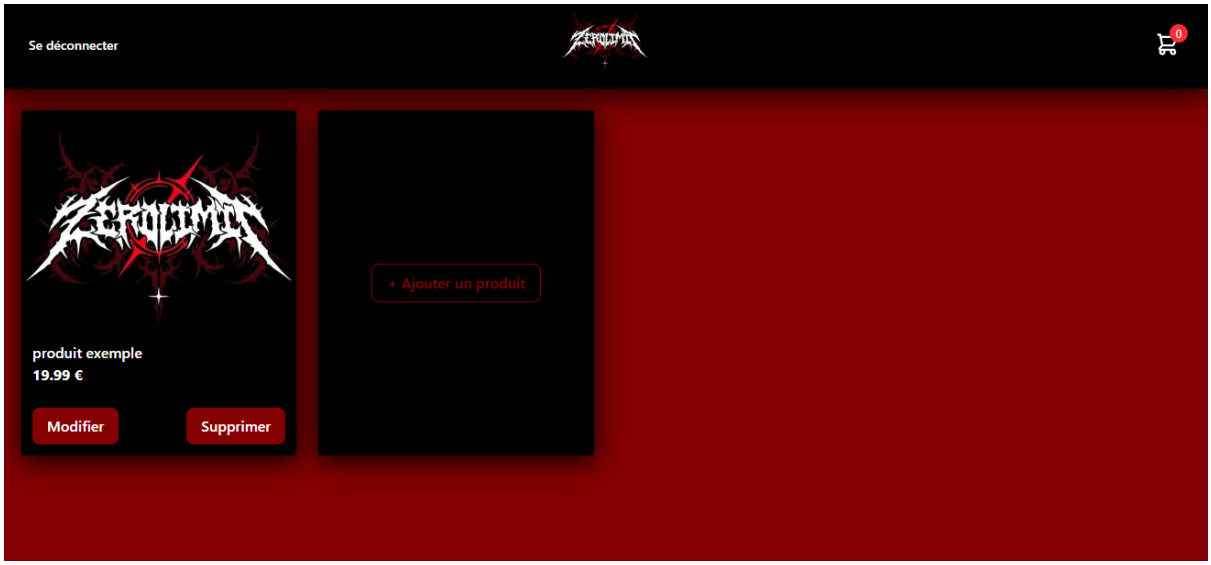


A partir de ça il suffit de cacher les options Administrateur en vérifiant si l'utilisateur connecté possède le rôle 'admin'.

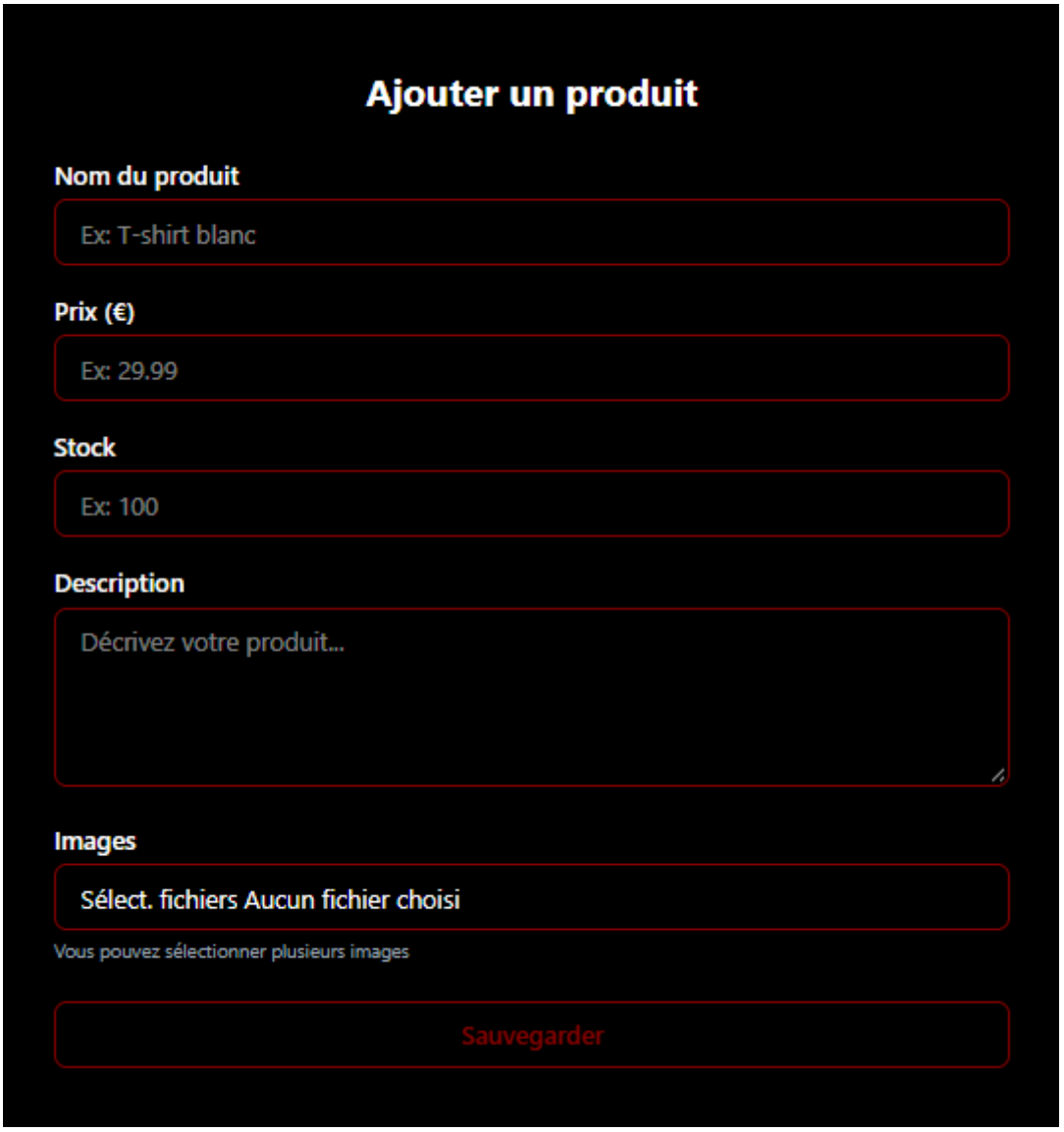
5. Créer et gérer les produits

Je commence par faire un ProductController pour la gestion des produits et un model Product avec : nom, prix, stock et description. Pour les images je fais un model ProductImage avec l'id du produit (clé étrangère) et path (le chemin pour accéder à l'image). Les deux sont liées grâce aux fonctions hasMany/belongsTo, ce qui me permet d'avoir plusieurs images pour un produit.

Je crée aussi un AdminMiddleware, un middleware ayant pour but d'exécuter du code lors d'une requête juste avant l'exécution de la requête, cela me permet de faire une vérification de si l'utilisateur est bien authentifié et administrateur pour exécuter les requêtes visant à la création, modification et suppression des articles.
Dans la vue 'home' j'ajoute un bouton 'ajouter un produit' visible uniquement par les administrateur et redirigeant vers un formulaire 'create'.



J’utilise la balise ‘multiple accept="image/*"’ sur l’input Images pour que l’utilisateur puisse y uploader plusieurs images.



Le bouton sauvegarder exécutant la requête ‘store’ qui va dans l’ordre :

- Vérifier la validité des données entrées
- Créer un produit avec ces données qui sera ajouté dans la BDD
- Pour chaque image, utiliser store('products', 'public') pour stocker le fichier et rentrer le chemin dans une variable ‘$path’ puis créer un objet avec l’identifiant du produit et le chemin récupéré.

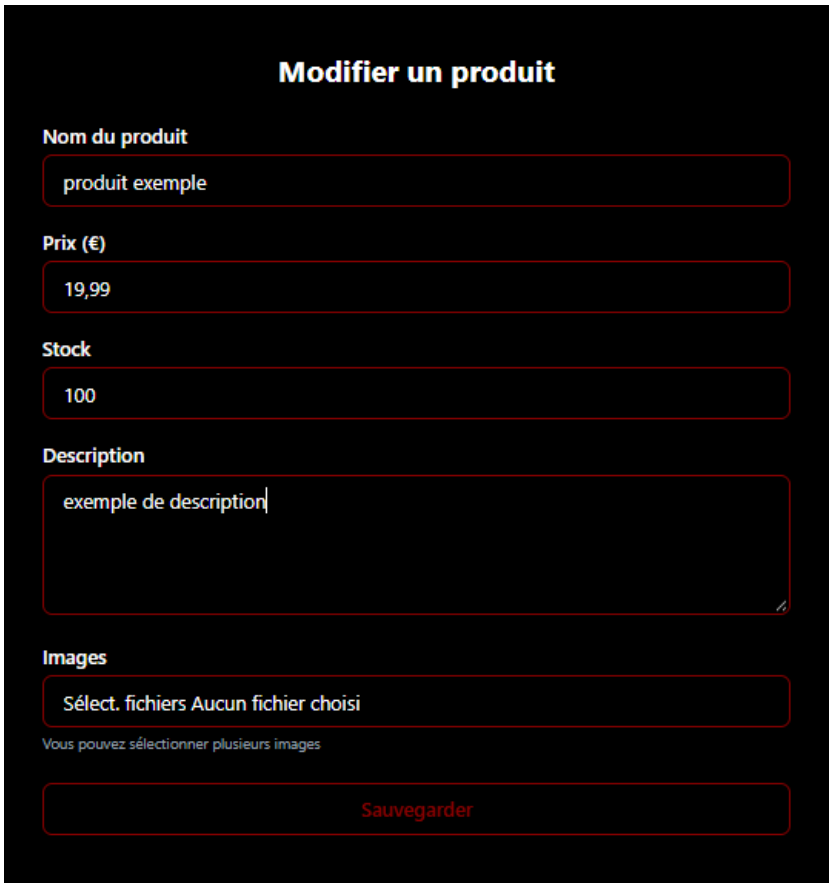
Je souhaite afficher mes articles sur ma page d’accueil pour ça je fais simplement un @foreach($products as $product).



Chaque produit est présenté avec la première image correspondant à l’article, le titre et le prix.
J’utilise un hover sur le CSS de l’image qui va zoomer légèrement sur la photo et la griser quand l’utilisateur passera sa souris dessus pour lui comprendre qu’elle est cliquable. Ce qui lui permettra à celui-ci d’accéder aux détails de l’article par la suite.

L’administrateur à également accès aux boutons modifier et supprimer.

Le bouton modifier donne accès à un formulaire similaire à celui pour créer un article.

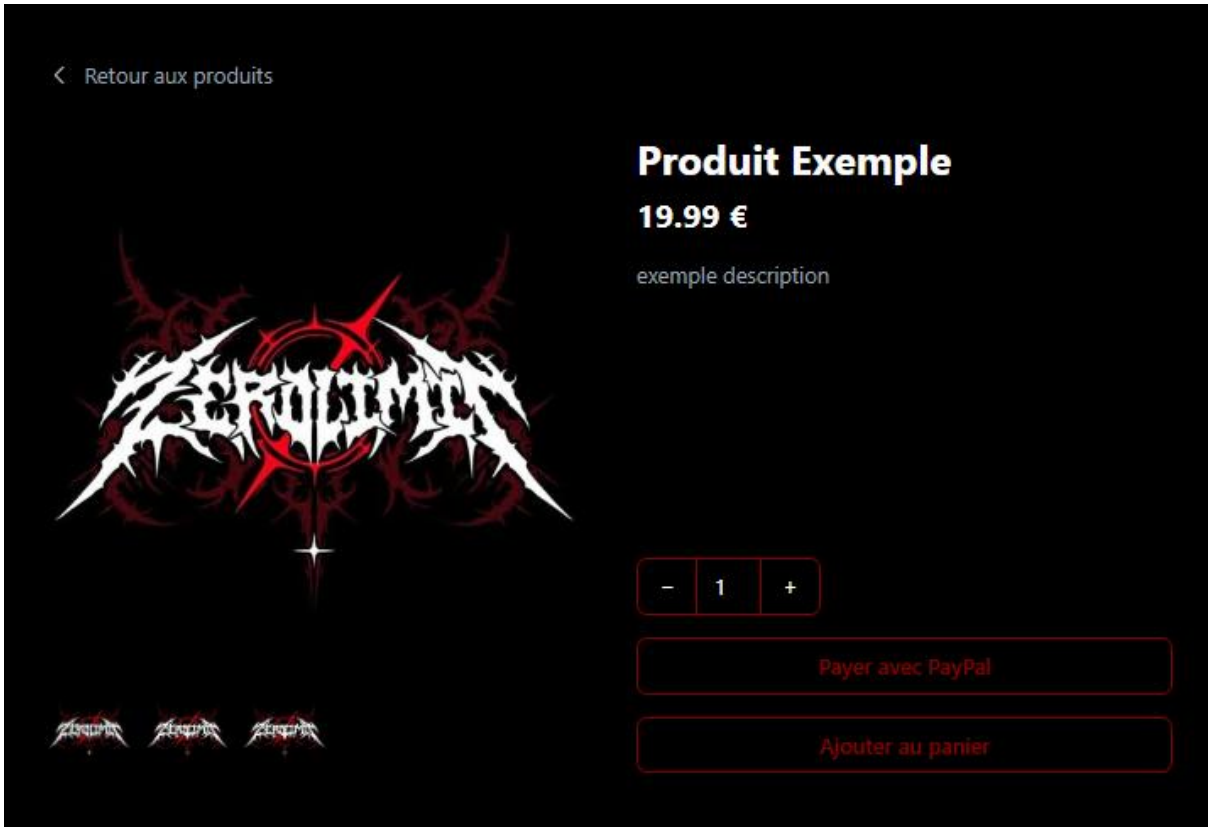


Je récupère les données de l’objet selectionné et les mets par défaut pour que l’administrateur n’ai pas à les réécrire.
Le bouton sauvegarder execute la requête ‘update’, similaire à la requête ‘store’ mais qui va récupérer l’objet Product et utiliser la fonction update() pour modifier les données dans la BDD.
Si l’administrateur à ajouté des images, on supprime toute les images existantes via Storage::disk('public')->delete($image->path) et on recrée des nouveaux objets avec les nouvelles images. Ce n’est pas la méthode la plus optimisée mais elle permet d’éviter les doublons.

Enfin le bouton supprimer execute la requête ‘delete’ qui va supprimer les images stockées correspondant au produits avec la même methode que vue précédement et supprimer l’objet de la BDD avec la methode delete().

6. Détails du produits

Pour faire la partie détail du produit je rajoute une requête ‘show’ dans mon ProductController qui m’ouvre une vue ‘show’.



Cette vu possède un bouton retour, les informations sur le produit, un bouton pour selectionner une quantité, un bouton pour payer et un bouton pour ajouter au panier.

Pour cette partie j'ai dû utiliser un peu de javascript, notamment pour la selection d'image : je donne à mon image principale un attribut **id= ‘‘main-image’’** et sur les autre un attribut **onclick="document.getElementById('main-image').src = this.src"**, ce qui me permet en cliquant sur les petites images de modifier la source de l'image principale.

Egalement pour que mes boutons « + » et « - » puissent modifier directement la valeur de la quantité, je crée une fonction javascript **changeQty(delta)** qui ajoute ou retire le delta à la quantité selectionnée et qu'elle reste entre 0 et le maximum en stock. Je rajoute un attribut **onclick= ‘‘ChangeQty(1)’’** au bouton « + » et un **onclick= ‘‘ChangeQty(-1)’’** au bouton « - ».

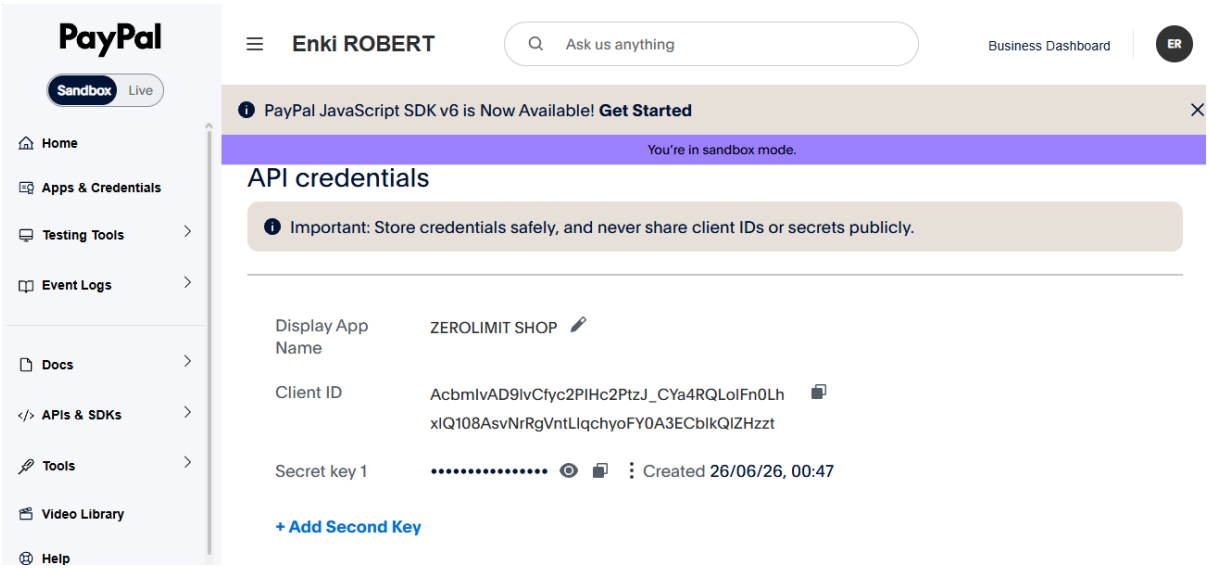
Il ne me reste plus qu'à mettre en place la methode de paiement et la gestion des panier.

7. Mettre en place une methode de paiement

Pour le système de paiement j'utilise PayPal qui a l'avantage d'avoir une API accessible et facile à mettre en place.

J'installe le package srmklive/paypal via une ligne de commande pour intégrer l'API à mon projet.

Je crée un compte PayPal Developer et une application Sandbox qui me permettra de simuler des paiements pendant le développement. J'obtiens alors un ID et un Clé secrette que je met dans le fichier .env de mon projet (fichier de configuration).



Une fois les identifiants récupérés, je crée un OrderController qui va gérer tout ce qui concerne le paiement et les commandes. Il contient trois fonctions :

- **checkout** qui prépare le paiement et envoie l'utilisateur sur PayPal
- **success** qui est appelée quand le paiement a réussi
- **cancel** qui est appelée quand l'utilisateur annule le paiement

La fonction checkout

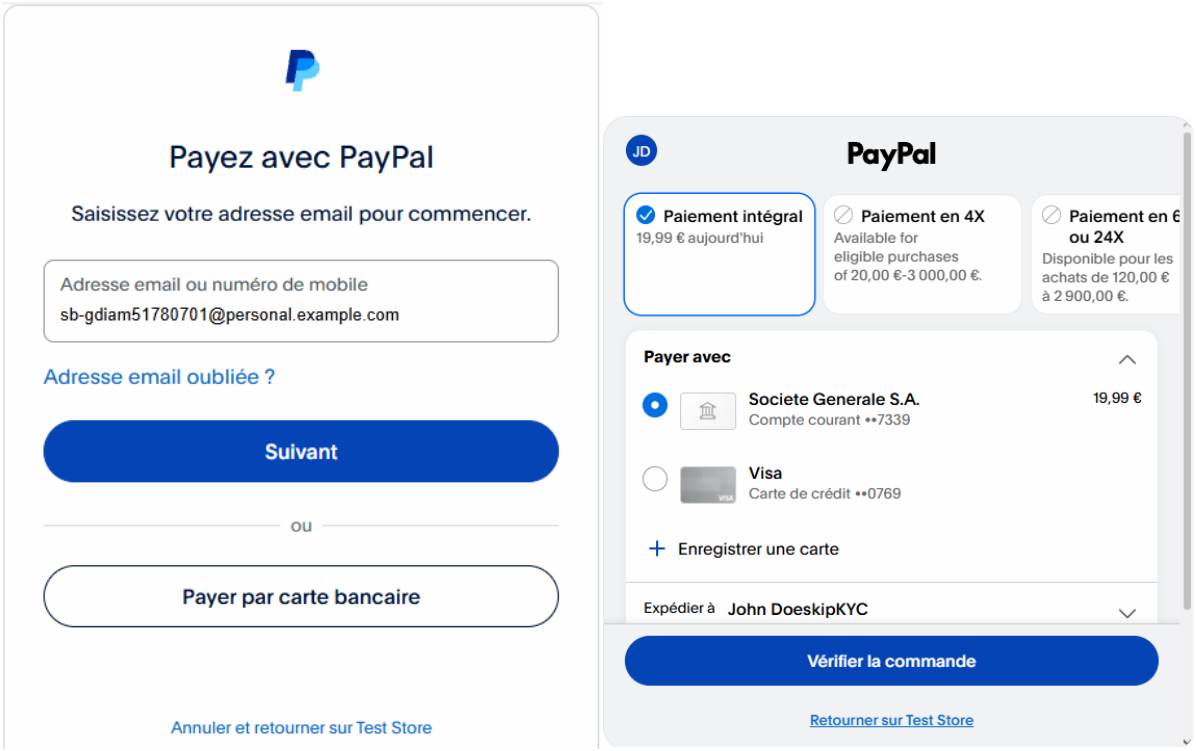
La première chose que fait cette fonction est de se connecter à PayPal. Pour ça j'utilise la classe PayPalClient fournie par le package, je lui donne mes identifiants (stockés dans le fichier .env) et je récupère un « access token ». Un access token est une sorte de clé temporaire que PayPal me donne pour prouver que mon site a le droit d'utiliser son API. Toutes les requêtes suivantes devront contenir cette clé.
Ensuite je calcule le montant total à payer. Il y a deux cas possibles, que je différencie grâce à un champ caché « type » dans le formulaire :

- Si le type est « cart », l'utilisateur paye son panier entier. Je récupère tous les articles de son panier et j'additionne le prix de chaque produit multiplié par sa quantité.
- Sinon, l'utilisateur a cliqué sur « Payer » directement depuis la page d'un produit. Je récupère le produit et la quantité choisie et je multiplie les deux.

Une fois le total calculé, j'utilise la fonction **createOrder()** du package pour créer une commande chez PayPal. Je lui envoie :

- L'intention « CAPTURE », qui veut dire que l'argent sera prélevé immédiatement à la validation (et non pas simplement réservé)
- Le montant et la devise (EUR)
- Une « return_url » : l'adresse vers laquelle PayPal renverra l'utilisateur si le paiement est validé
- Une « cancel_url » : l'adresse vers laquelle PayPal renverra l'utilisateur s'il annule

PayPal me répond avec un ensemble de liens, parmi lesquels se trouve un lien « approve ». C'est le lien vers la page PayPal sur laquelle l'utilisateur va se connecter et confirmer son paiement. Je redirige donc l'utilisateur vers ce lien. Si pour une raison ou une autre PayPal ne me renvoie pas ce lien, je renvoie l'utilisateur en arrière avec un message d'erreur.



La fonction success

Quand l'utilisateur valide son paiement, PayPal le renvoie sur mon site avec un « token » dans l'adresse. Ce token est l'identifiant de la commande chez PayPal. Je me reconnecte à l'API de la même manière que précédemment puis j'utilise la fonction capturePaymentOrder() avec ce token pour demander à PayPal de prélever réellement l'argent.

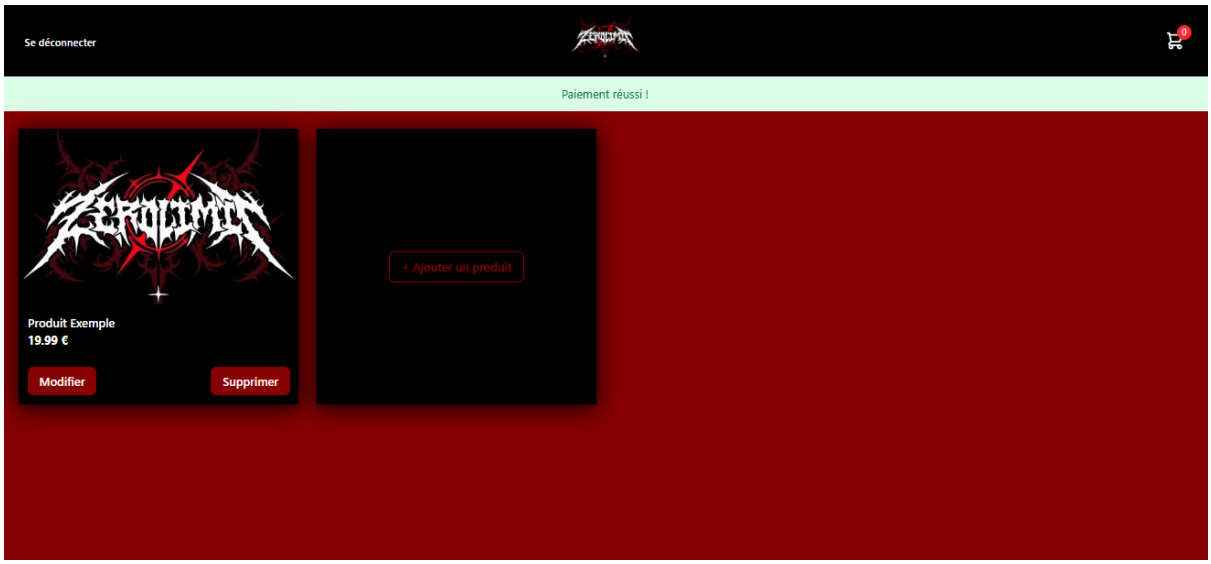
PayPal me renvoie alors toutes les informations sur la transaction. Je vérifie d'abord que le statut est bien « COMPLETED ». Si ce n'est pas le cas, je renvoie l'utilisateur à l'accueil avec un message d'erreur.

Si le paiement est bien passé, je crée un model Order dans ma base de données pour garder une trace de la commande. Je récupère dans la réponse de PayPal :

- Le nom, le prénom et l'email de l'acheteur
- Son adresse de livraison (rue, ville, code postal, pays)
- Le montant réellement payé

- L'identifiant de la commande et l'identifiant du prélèvement chez PayPal, qui permettront de retrouver la transaction en cas de problème

L'avantage de cette méthode est que je n'ai pas besoin de faire remplir un formulaire d'adresse sur mon site : PayPal me transmet directement l'adresse enregistrée sur le compte de l'acheteur.
Une fois la commande enregistrée, je vide le panier de l'utilisateur et je le redirige vers l'accueil avec un message « Paiement réussi ! » affiché grâce au session('success') de ma vue base.



La fonction cancel
Si l'utilisateur annule sur PayPal, il est renvoyé vers son panier avec le message « Paiement annulé. ».

Pour finir, j'ajoute trois routes dans mon fichier web.php : une route POST /payment pour lancer le paiement, et deux routes GET /payment/success et /payment/cancel qui correspondent aux adresses de retour données à PayPal.

8. Gérer les paniers

Le panier est une fonctionnalité un peu particulière car il doit fonctionner dans deux situations différentes : quand l'utilisateur est connecté et quand il ne l'est pas. Je voulais qu'un visiteur puisse ajouter des articles sans être obligé de créer un compte.

Le stockage du panier
Pour les utilisateurs connectés, je crée un model Cart avec : l'identifiant de l'utilisateur, l'identifiant du produit et la quantité. Chaque ligne de la table correspond à un article dans le panier d'un utilisateur. Comme pour les images, je relie le model Cart aux models User et Product avec **belongsTo**.
Pour les visiteurs non connectés, je stocke le panier dans la session. La session est un espace de stockage temporaire propre à chaque visiteur, qui est conservé par le serveur tant que celui-ci garde son navigateur ouvert (ou pendant une durée limitée). Le panier y est stocké sous forme d'une liste associant chaque identifiant de produit à sa quantité. Rien n'est écrit dans la base de données, ce qui est suffisant pour un panier temporaire.

Le CartController
Je crée ensuite un CartController avec trois fonctions, qui à chaque fois vérifient avec **Auth::check()** si l'utilisateur est connecté pour savoir où aller chercher le panier :

- **add** : ajoute un produit au panier avec la quantité choisie (1 par défaut). Si le produit est déjà dans le panier, j'ajoute la quantité à celle existante au lieu de créer un doublon. Je

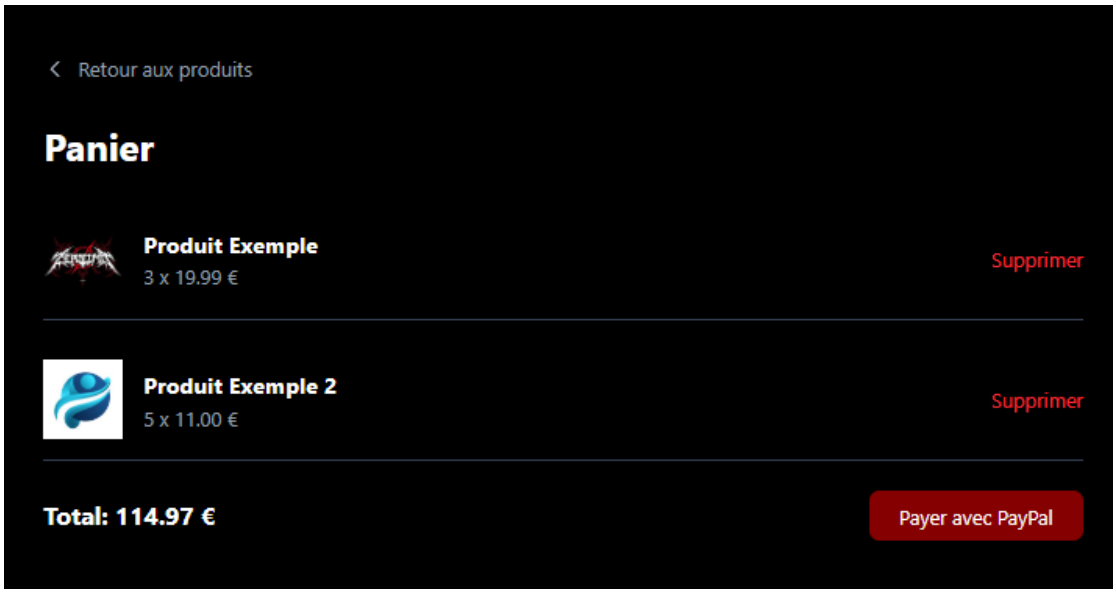
renvoie ensuite l'utilisateur sur la page avec le message « Ajouté au panier ».

- **remove** : supprime un produit du panier.
- **show** : affiche la page du panier. Pour les utilisateurs connectés je récupère directement les lignes de la table Cart avec leur produit associé. Pour les visiteurs, je parcours la session et je vais chercher chaque produit dans la base de données pour reconstruire une liste au même format. Je calcule ensuite le total en additionnant le prix de chaque produit multiplié par sa quantité.

J'ajoute les routes correspondantes dans web.php avec le préfixe **/cart**.

La vue du panier

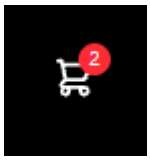
La vue reprend la même mise en forme que le reste du site. Chaque article est affiché avec sa première image, son nom, sa quantité et son prix unitaire, ainsi qu'un bouton « Supprimer ». En bas de page j'affiche le total et un bouton « Payer avec PayPal » qui envoie le formulaire avec le type « cart » vu dans la partie précédente. Si le panier est vide, j'affiche simplement un message.



Le compteur dans la barre de navigation

Je voulais afficher le nombre d'articles dans le panier sur le bouton de la barre de navigation. Le problème est que cette barre est présente sur toutes les pages, et je ne voulais pas avoir à calculer ce nombre dans chaque contrôleur.

Pour ça j'utilise un « View composer » dans le fichier AppServiceProvider. C'est une fonction qui s'exécute automatiquement avant l'affichage de n'importe quelle vue (grâce au '*'), et qui me permet de leur transmettre une variable commune. Ici je compte le nombre de produits dans le panier (dans la base de données ou dans la session selon le cas) et je transmets le résultat dans une variable **cartCount**, que je peux ensuite afficher dans ma vue base.



Fusionner le panier à la connexion

Il restait un cas à gérer : un visiteur ajoute des articles dans son panier sans être connecté, puis se connecte. Son panier de session serait perdu puisque le site irait chercher le panier dans la base de données.

Pour résoudre ça j'utilise le système d'événements de Laravel. Un événement est un signal envoyé par le framework quand quelque chose se produit, ici l'événement Login qui est envoyé à chaque connexion d'un utilisateur. Je crée un « **listener** » **MergeCartOnLogin**, c'est-à-dire une classe qui attend cet événement et exécute du code quand il se produit. Je l'enregistre dans AppServiceProvider.

Quand un utilisateur se connecte, ce listener récupère le panier de la session et, pour chaque produit, l'ajoute dans la table Cart

de l'utilisateur (en additionnant les quantités si le produit y est déjà). Puis il vide le panier de la session. De cette manière, l'utilisateur retrouve bien ses articles après s'être connecté.

IV. Bilan du sujet

Le projet qui m'a été confié était de créer le site web du collectif Zero Limit. Il devait surtout permettre de vendre les produits dérivés du collectif, et aussi de présenter ses événements et son actualité.

Dès le début, la demande de mon maître de stage était assez peu précise. Il n'y avait pas de vrai cahier des charges, seulement une idée générale de ce que le site devait permettre. Le projet du collectif était aussi encore récent et peu développé : Zero Limit n'avait pas encore commencé à vendre ses produits et n'avait pas de catalogue fixe ni d'organisation de vente définie. Il était donc difficile de faire un site très poussé, puisque beaucoup de besoins n'étaient pas encore connus.

J'ai donc dû faire moi-même certains choix. J'ai commencé par une maquette sur Figma pour donner une forme concrète à l'idée de départ et la faire valider. Ensuite j'ai gardé les fonctionnalités indispensables et j'ai simplifié ou retiré celles qui n'étaient pas utiles pour l'instant, comme la billetterie ou les catégories de produits.

À la fin du stage, le site permet :

- de créer un compte et de se connecter, avec un rôle administrateur pour le collectif ;
- d'ajouter, modifier et supprimer des produits avec leurs images ;
- d'afficher le détail de chaque produit ;
- d'ajouter des produits dans un panier, connecté ou non ;

- de payer en ligne avec PayPal et d'enregistrer les commandes.

Même s'il reste simple, le site répond à la demande de départ : le collectif peut présenter et vendre ses produits en ligne. Il pourra aussi évoluer quand le projet sera plus avancé. Il faudra par exemple enregistrer le détail des produits de chaque commande, gérer le stock automatiquement, ajouter la billetterie et une partie actualités, puis passer PayPal du mode test au mode réel avant la mise en ligne.

V. Bilian général

Ce stage m'a permis de travailler pour la première fois sur un vrai projet, du début à la fin. J'ai dû passer d'une idée à un site qui fonctionne, en passant par la maquette, le choix des outils, le développement et les tests.

Sur le plan technique, j'ai beaucoup progressé avec Laravel, surtout sur l'architecture MVC, la base de données avec Eloquent et les vues avec Blade. J'ai aussi découvert des choses que je n'avais jamais faites, comme utiliser une API externe pour le paiement, gérer une session ou utiliser les événements de Laravel.

Le manque de précision dans la demande a été la principale difficulté, mais il m'a aussi beaucoup appris. J'ai dû poser des questions, proposer des solutions et faire des choix sans attendre que tout soit décidé à ma place. J'ai compris que l'analyse du besoin est une étape très importante, et que sur un prochain projet il vaudrait mieux écrire un cahier des charges avec le client avant de commencer à coder.

Pour finir, ce stage m'a conforté dans mon choix de m'orienter vers le développement. Il m'a donné une première expérience concrète que je pourrai réutiliser pendant ma deuxième année de BTS SIO.