



Site e-commerce dédié aux bracelets en pierres naturelles artisanales



*Projet réalisé dans le cadre de la présentation au*  
**Titre Professionnel Développeur Web et Web Mobile**

*présenté par*

**Quoc-Nam NGO**

École la Plateforme - 2025

# Sommaire

Introduction.....	4
Liste des compétences couvertes par le projet.....	5
1 - Développer la partie front-end d'une application web ou web mobile sécurisée.....	5
A - Installer et configurer son environnement de travail en fonction du web ou web mobile.....	5
B - Maquetter des interfaces utilisateur web ou web mobile.....	5
C - Réaliser des interfaces statiques web ou web mobile.....	5
D - Développer la partie dynamique des interfaces utilisateur web ou web mobile.....	6
2 - Développer la partie back-end d'une application web ou web mobile sécurisée.....	7
A - Mettre en place une base de donnée relationnelle.....	7
B - Développer les composants d'accès aux données SQL et NoSQL.....	7
C - Développer des composants métier côté serveur.....	8
D - Documenter le déploiement d'une application dynamique.....	8
Résumé du projet.....	9
Cahier des charges.....	9
1 - Objectifs.....	9
2 - User Stories.....	10
A – Cible.....	10
B - Roles.....	10
3 – Arborescence.....	11
4 - MVP (Minimum Viable Product).....	13
5 - Evolutions potentielles.....	13
6 - Wireframes et Maquettes.....	14
Page d'accueil.....	14
Listing de produits.....	16
Fiche produit.....	17
Mon compte.....	18
Récapitulatif de commande.....	19
Spécifications techniques.....	21
1 - Technologies utilisées.....	21
2 - Navigateurs compatibles.....	21
3 - Versionnning.....	22
4 - Création de la base de données.....	23
A – MCD.....	24
B - MLD.....	24
C - Dictionnaire de données.....	24
5 - Back end.....	29
A - Arborescence.....	29
B - Installation.....	31
C - Lancement du serveur.....	33
D - Flux de requêtes.....	36
E - Les routes.....	38
Tableau des routes.....	39
F - Middleware.....	41
checkTokenValid.....	41
checkTokenAdmin.....	42
errorhandler.....	43
G - Controller.....	44
Tableau des controllers.....	45

Hashage des mots de passe (bcrypt).....	46
Génération d'un JWT (jsonwebtoken).....	47
Stockage sécurisé du JWT dans un cookie HttpOnly.....	47
Vérification de l'utilisateur connecté.....	48
Déconnexion (logout).....	48
Paiement Stripe Checkout.....	48
Webhook Stripe (confirmation du paiement).....	49
H - Models.....	51
Tableau des models.....	51
Prévention SQL Injection.....	53
6 - Front end.....	54
A - Arborescence.....	54
B - Installation.....	57
Génération de composants et de service.....	57
Configuration des environnements.....	57
Lancement du projet.....	58
C - Les services.....	58
Structure du service.....	58
Gestion des requêtes HTTP avec fetch.....	59
Exemples de méthodes dans le service.....	59
Observables et partage de données.....	61
D - Les composants.....	62
E - Les routes.....	66
F - Responsive design.....	67
G - Fonctionnalités principales.....	68
Déploiement.....	69
Conclusion.....	70

# Introduction

Depuis plusieurs années, je suis animé par une passion pour l'informatique et le développement web, malgré un parcours professionnel initial éloigné de ce domaine. Anciennement responsable des achats, j'ai acquis des compétences solides en gestion, organisation et relationnel, qui m'accompagnent aujourd'hui dans ma reconversion professionnelle.

Dans le cadre de ma reconversion, j'ai choisi de suivre la formation Développeur Web et Web Mobile (DWWM) à l'école la plateforme, afin de développer mes compétences techniques et de m'immerger pleinement dans l'univers du développement web. Le format télé-présentiel de la formation m'a permis de suivre les cours à distance tout en bénéficiant d'un accompagnement pédagogique structuré et de ressources professionnelles de qualité.

Pour clôturer cette formation, j'ai réalisé un projet pratique appelé "WrapVintage", qui consiste à concevoir et développer un site web fonctionnel. J'ai choisi de travailler sur un site e-commerce dédié à la vente de bracelets en pierres naturelles, un projet à la fois technique et créatif.

Contrairement à certains projets en équipe, j'ai mené ce projet de manière autonome, ce qui m'a permis de gérer l'ensemble des étapes : conception, développement, intégration et mise en ligne de la plateforme.

Ce projet m'a permis de consolider mes compétences techniques, de pratiquer les bonnes pratiques du développement web et de comprendre l'ensemble du processus nécessaire à la création d'un site e-commerce complet et fonctionnel. Il reflète également ma capacité à mener un projet de A à Z et à appliquer de manière concrète les connaissances acquises au cours de ma formation.

# Liste des compétences couvertes par le projet

## 1 - Développer la partie front-end d'une application web ou web mobile sécurisée

### A - Installer et configurer son environnement de travail en fonction du web ou web mobile

J'ai mis en place un environnement complet sur mon PC sous Debian 12, en utilisant Visual Studio Code comme éditeur principal. Le backend est développé en Node.js avec Express, le frontend en Angular, et la base de données MariaDB est exécutée via Docker. Le code est versionné sur GitHub, ce qui me permet de suivre l'évolution du projet et de travailler de manière structurée. Les maquettes sont réalisées sur le site Figma et prévisualisées directement dans Firefox, tandis que les user stories sont rédigées avec LibreOffice afin de cadrer les fonctionnalités et les parcours utilisateurs.

### B - Maquetter des interfaces utilisateur web ou web mobile

Avant de commencer le développement, j'ai rédigé les user stories pour définir clairement les parcours des différents profils d'utilisateurs : visiteur, utilisateur authentifié et administrateur. J'ai ensuite conçu les wireframes sous figma des pages essentielles — accueil, listing produits (homme, femme, nouveautés), fiche produit, panier, compte et commande — afin de valider la structure, la hiérarchie visuelle et l'emplacement des zones d'action sans me concentrer sur le style graphique.

Une fois cette étape validée, j'ai réalisé les maquettes en appliquant la charte graphique, avec un choix de couleurs naturelles, des contrastes lisibles, une typographie adaptée et une iconographie cohérente. J'ai d'abord créé les versions desktop, puis adapté l'interface en responsive pour mobile et tablette, en simplifiant la navigation.

### C - Réaliser des interfaces statiques web ou web mobile

Avant d'ajouter la logique dynamique, j'ai construit les interfaces sous forme de vues statiques Angular en suivant une structure modulaire. Le dossier layout regroupe les éléments communs comme le header, le footer, le menu burger, les modals de connexion et d'inscription, ainsi que la bannière de cookies pour garantir une cohérence visuelle. Les pages fonctionnelles sont isolées dans features, tandis que les composants réutilisables, comme la carte produit, sont centralisés dans shared.

Les maquettes Figma ont été transposées en HTML et CSS dans les templates sans logique métier, avec des grilles produits et des blocs visuels créés grâce à Flexbox et CSS Grid. Les textes et images ont été insérés en dur depuis le dossier public. Les composants ont été initialement créés avec un code TypeScript minimal pour séparer présentation et logique, puis enrichis après validation de l'interface.

Une responsivité de base a été mise en place via des ajustements CSS, et les images ont été optimisées avec des vignettes pour l'aperçu et des versions HD pour le détail. La cohérence visuelle a été assurée par la réutilisation des composants, chaque composant disposant de son propre fichier CSS pour encapsuler le style et éviter les interférences.

## D - Développer la partie dynamique des interfaces utilisateur web ou web mobile

Après la phase statique, j'ai rendu l'application interactif en connectant les composants Angular à l'API backend. Les données des listes de produits et des fiches détaillées sont récupérées via un service central, ApiService, qui gère les requêtes, les erreurs et l'envoi automatique des cookies de session. L'état côté front, comme le panier ou l'utilisateur connecté, est maintenu dans des propriétés internes de composants ou de services pour refléter instantanément les actions de l'utilisateur.

L'authentification fonctionne via un formulaire de connexion qui envoie les identifiants à l'API ; le backend pose un cookie HttpOnly et les requêtes suivantes sont effectuées avec credentials include. L'interface adapte dynamiquement l'affichage selon la présence de l'utilisateur. La navigation est gérée par le routing Angular, avec accès conditionnel à certaines pages et URL paramétrées pour les fiches produits.

Les interactions utilisateur comprennent l'ajout ou la suppression d'articles dans le panier, l'ouverture et la fermeture des modals, et la recherche de produits avec mise à jour dynamique de la grille.

Côté sécurité, aucun token n'est stocké dans le localStorage, tout repose sur des cookies sécurisés et chaque réponse API est vérifiée pour adapter l'interface avec redirection ou affichage d'erreurs. Enfin, pour garantir performance et propreté, les requêtes serveurs sont centralisés, le composant carte produit est réutilisé sur toutes les listes.

## 2 - Développer la partie back-end d'une application web ou web mobile sécurisée

### A - Mettre en place une base de donnée relationnelle

Pour la base de données relationnelle, j'ai commencé par modéliser le schéma avec drawdb.vercel.app, en définissant les entités principales comme les utilisateurs, les produits, les

catégories, la table de liaison produit-catégorie, le panier et les commandes, avec éventuellement les lignes de commande. Les relations entre les tables ont été validées, en utilisant des associations un-à-plusieurs et plusieurs-à-plusieurs, avec des clés primaires auto-incrémentées et des types simples.

J'ai choisi MariaDB pour sa légèreté et sa compatibilité MySQL, et j'ai conteneurisé le tout avec Docker Compose, en créant un service mariadb associé à un volume persistant et en utilisant un fichier .env pour centraliser l'hôte, l'utilisateur, le mot de passe et le nom de la base. La connexion à la base est centralisée dans un fichier database.js et injectée dans les modèles, avec des requêtes paramétrées pour prévenir toute injection SQL.

Les principales tables gèrent les utilisateurs, les produits, les catégories avec leur table de liaison N-N, le panier, les commandes et éventuellement les messages de contact pour la traçabilité. Les modèles exploitent les joins pour récupérer les produits par catégorie, ainsi que des filtres pour les éléments archivés ou les nouveautés, et construisent dynamiquement les URLs des images.

Pour la persistance et la maintenance, un volume Docker permet de conserver les données même lors des rebuilds, et pour la sécurité, les credentials sont stockés dans les variables d'environnement, tandis que les images ne sont jamais stockées en base mais uniquement sur le filesystem.

## B - Développer les composants d'accès aux données SQL et NoSQL

L'accès aux données se fait via une base relationnelle MariaDB, structurée autour des entités principales : produits, catégories, utilisateurs, panier, commandes et la liaison produit-catégorie. Chaque table possède un modèle dédié dans le dossier models, qui encapsule toutes les requêtes SQL et évite que les contrôleurs manipulent directement la base.

La connexion à la base est centralisée dans database.js et injectée dans chaque modèle, avec des requêtes paramétrées pour prévenir toute injection SQL. Les contrôleurs se concentrent sur la validation et l'orchestration, tandis que les modèles gèrent l'accès aux tables, les opérations CRUD, les filtres et les jointures.

Chaque modèle est organisé de manière logique : product.model.js s'occupe des listes par catégorie, des nouveautés, des images et de l'archivage ; category.model.js gère les catégories ; user.model.js prend en charge la création des comptes, la sécurisation des mots de passe et le soft delete ; cart.model.js gère les articles et le total du panier ; order.model.js crée les commandes et leurs lignes associées ; contact.model.js stocke les messages. Les URLs des images sont construites au niveau des modèles pour enrichir les données, et le traitement des limites ou du soft delete se fait via des champs spécifiques comme is\_archived ou deleted\_at.

Le serveur Node.js est lancé depuis server.js, qui utilise Express pour exposer les endpoints et fait appel aux modèles via database.js pour interagir avec la base de manière sécurisée et structurée.

## C - Développer des composants métier côté serveur

J'ai développé les composants métier côté serveur en suivant une architecture en couches bien structurée. Les routes déclarent les différents endpoints, les contrôleurs orchestrent les cas d'usage,

valident les entrées et appellent les modèles, tandis que les modèles encapsulent tout l'accès SQL sans inclure de logique HTTP. Des middlewares gèrent l'authentification via JWT, les rôles utilisateurs, les erreurs et l'exposition des médias.

Côté authentification, l'inscription sécurise les mots de passe avec un hash et vérifie l'unicité, la connexion valide les credentials et émet un JWT dans un cookie HttpOnly, et l'endpoint /auth/me permet de maintenir la session. Pour les produits et catégories, les contrôleurs gèrent le listing par genre, les nouveautés, l'archivage logique et l'enrichissement des données avec les URLs d'images, tout en filtrant selon l'état des produits. Le panier et les commandes sont traités de manière cohérente : ajout ou retrait d'articles, création de commandes à partir du panier, en préparation d'une intégration du paiement avec Stripe. Le paiement lui-même est pris en charge via la génération de sessions Checkout et la gestion de webhooks pour confirmer les transactions.

Les utilisateurs disposent d'un CRUD restreint côté admin, avec soft delete pour préserver les données et protection des informations sensibles. Les contacts et messages sont enregistrés en base pour assurer la traçabilité. Les règles métier transverses incluent des contrôles simples des entrées, la protection des rôles administrateurs via middleware, la séparation stricte entre contrôleurs et SQL, et la construction dynamique des URLs d'images au niveau des modèles.

## D - Documenter le déploiement d'une application dynamique

Pour le déploiement, j'ai choisi un VPS chez Hostinger, ce qui me permet d'héberger l'ensemble de l'application de manière autonome. J'utilise Docker Compose pour lancer les différents services : une base MariaDB avec volume persistant, un backend Node.js exposé uniquement en interne, et un frontend Angular compilé puis servi par Nginx (fichiers statiques + fallback SPA). En "edge", Caddy gère le HTTPS avec Let's Encrypt et fait le routage entre le backend et le frontend. Enfin, grâce à DuckDNS, j'associe mon domaine wrapvintage.duckdns.org à l'IP publique du VPS, ce qui rend l'application accessible et sécurisée depuis Internet.

# Résumé du projet

Ce projet e-commerce autour des bracelets en pierres naturelles est né de ma reconversion professionnelle, passant du métier de responsable achats au développement web.

L'idée était de créer une application qui combine ma passion pour les minéraux et la mise en pratique des compétences techniques que j'ai acquises : Angular pour le front, Node/Express pour l'API, une base de données relationnelle, l'authentification, la gestion du panier et des commandes jusqu'au paiement.

Le site permet de présenter un catalogue organisé (hommes, femmes, nouveautés), de gérer un compte utilisateur et de suivre un parcours complet d'achat.

Choisir un thème qui me tient à cœur a renforcé ma motivation tout au long du projet, tout en me donnant un terrain concret pour montrer ma capacité à structurer le code, séparer les différentes couches, containeriser l'application et gérer son déploiement. Ce projet illustre donc à la fois mes compétences techniques et ma transition vers un rôle de développeur.

## Cahier des charges

### 1 - Objectifs

L'objectif de ce site e-commerce est de proposer une expérience d'achat simple et agréable autour des bracelets en pierres naturelles.

Il vise à présenter un catalogue structuré, permettant aux visiteurs de découvrir facilement les produits selon les catégories homme, femme ou nouveautés. Le site doit également offrir une gestion sécurisée des comptes utilisateurs, un panier fonctionnel et un suivi complet des commandes, jusqu'au paiement.

Au-delà de l'aspect commercial, le projet a pour objectif de mettre en œuvre et de démontrer mes compétences techniques : structuration du code, modularité, intégration front-end et back-end, gestion de la base de données et déploiement de l'application de manière professionnelle et sécurisée.

### 2 - User Stories

#### A – Cible

Le public visé par WrapVintage est constitué d'hommes et de femmes âgés de 20 à 45 ans, sensibles aux bijoux et accessoires de mode en matières naturelles. Ces clients recherchent des bracelets à la fois esthétiques, de qualité, et porteurs de signification. La marque s'adresse à une clientèle moderne, active et soucieuse de son style, mais également attentive aux valeurs associées aux pierres naturelles (authenticité, énergie, bien-être).

WrapVintage cible aussi bien des acheteurs individuels à la recherche d'un bijou personnel que des consommateurs en quête d'un cadeau original et intemporel.

## B - Rôles

Les fonctionnalités de l'application WrapVintage sont organisées autour des différents types d'utilisateurs.

### Visiteur non connecté

- En tant que visiteur, je souhaite parcourir les bracelets en pierres naturelles par catégorie (homme/femme) afin de découvrir les produits disponibles.
- En tant que visiteur, je souhaite cliquer sur un bracelet pour voir sa fiche détaillée (nom, description, prix, image HD) afin de prendre ma décision d'achat.
- En tant que visiteur, je souhaite accéder à une section "nouveautés" afin de voir les derniers bracelets ajoutés au catalogue.
- En tant que visiteur, je souhaite utiliser une barre de recherche afin de trouver rapidement un bracelet spécifique par nom.
- En tant qu'utilisateur connecté, je souhaite ajouter ou retirer des bracelets dans mon panier et voir le montant total mis à jour afin de gérer mes achats.
- En tant que visiteur, je souhaite m'inscrire avec email/mot de passe afin d'accéder aux fonctionnalités membres.
- En tant que visiteur, je souhaite envoyer un message via un formulaire de contact afin d'obtenir des informations supplémentaires sur les produits.
- En tant que visiteur, je souhaite consulter les conditions de livraison, de paiement et de vente afin de comprendre les modalités avant d'acheter.
- En tant que visiteur, je souhaite accéder aux réseaux sociaux (Facebook, Instagram, Pinterest) afin de suivre l'actualité de la boutique et découvrir d'autres contenus.

### Utilisateur connecté

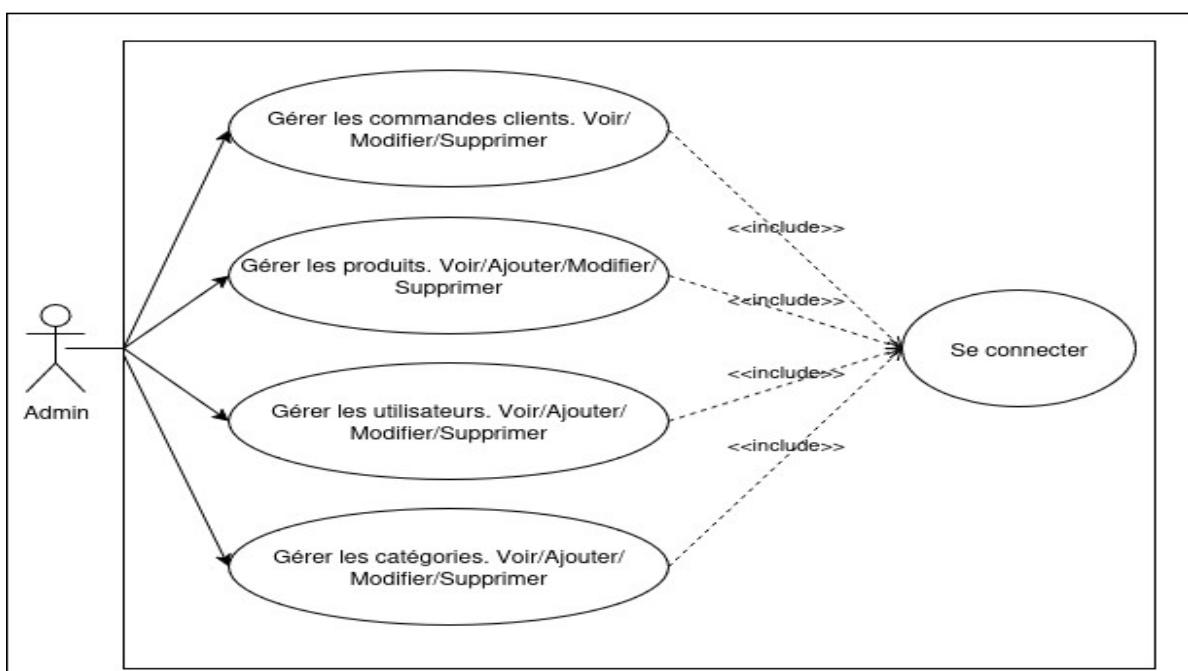
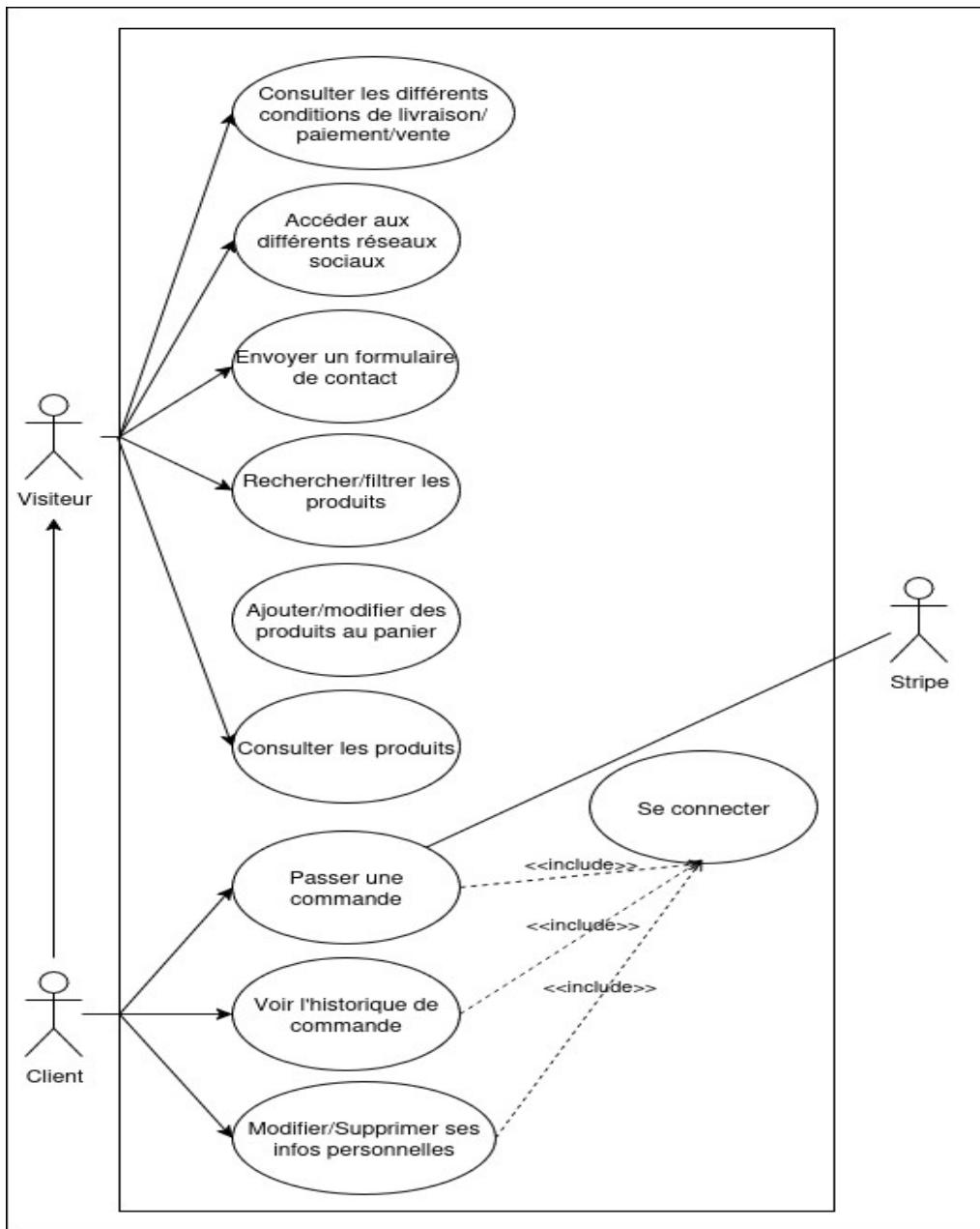
- En tant qu'utilisateur connecté, je dispose également de toutes les fonctionnalités accessibles aux visiteurs non connectés.
- En tant qu'utilisateur inscrit, je souhaite me connecter avec mes identifiants afin d'accéder à mon espace personnel.

- En tant qu'utilisateur connecté, je souhaite valider mon panier et payer via Stripe afin de recevoir confirmation de ma commande.
- En tant qu'utilisateur connecté, je souhaite consulter l'historique de mes achats afin de voir le détail des produits, montants et dates.
- En tant qu'utilisateur connecté, je souhaite modifier mes informations personnelles (nom, email, adresse) afin de tenir mon profil à jour.
- En tant qu'utilisateur connecté, je souhaite supprimer définitivement mon compte afin d'effacer mes données personnelles, conformément à la réglementation RGPD.

## **Administrateur**

- En tant qu'admin, je souhaite créer, modifier et archiver des bracelets avec leurs informations (nom, prix, description, images, stock) afin de gérer le catalogue.
- En tant qu'admin, je souhaite créer et gérer les catégories de produits afin de structurer le catalogue.
- En tant qu'admin, je souhaite consulter la liste des commandes passées afin de suivre les ventes.
- En tant qu'admin, je souhaite voir la liste des comptes utilisateurs et pouvoir les désactiver si nécessaire afin de gérer les utilisateurs.

## **3 – Arborescence**



## 4 - MVP (Minimum Viable Product)

Le MVP du site correspond à la première version pleinement fonctionnelle, concentrée sur l'essentiel pour offrir une valeur immédiate aux utilisateurs.

L'objectif principal est de valider à la fois l'intérêt des visiteurs, en leur permettant de naviguer, d'ajouter des produits au panier et de passer commande, et la faisabilité technique, en testant la communication entre l'API, la base de données et le paiement Stripe.

Cette version inclut un catalogue produits avec pages de listing et détails, une segmentation simple par catégorie (homme, femme, nouveautés), un système d'inscription et de connexion sécurisé via JWT en cookie, ainsi qu'un panier fonctionnel permettant d'ajouter, retirer des articles et calculer le total. Côté administration, il est possible de créer, modifier et archiver des produits, tandis que le serveur API et la base relationnelle sont conteneurisés pour faciliter le déploiement et la maintenance.

L'ensemble constitue une base stable qui permet aux utilisateurs de parcourir le catalogue, gérer leur compte et effectuer un paiement, tout en offrant aux administrateurs les outils essentiels pour gérer les produits. Les fonctionnalités avancées, telles que la recherche complexe, la gestion fine des stocks, les avis clients ou le suivi détaillé des commandes, sont laissées pour les itérations futures, le MVP servant de socle solide pour tester le concept et orienter les développements suivants.

## 5 - Evolutions potentielles

Afin d'accompagner la croissance de WrapVintage et d'enrichir progressivement l'expérience client, plusieurs évolutions sont envisageables à moyen terme, tout en préservant la simplicité du cœur de la plateforme.

### Catalogue & expérience produit

L'intégration d'un suivi de stock en temps réel, avec alertes de rupture et de retour en disponibilité, permettrait d'améliorer la transparence pour nos clients. Les fiches produits pourraient également être enrichies avec des informations plus détaillées telles que la composition énergétique des pierres, leur origine ou encore des conseils d'entretien.

### Personnalisation utilisateur

Pour renforcer l'engagement et la fidélisation, nous pourrions ajouter des recommandations personnalisées (produits similaires, "vus récemment"), la gestion de codes promotionnels et cartes cadeaux, ainsi que la possibilité pour l'utilisateur d'enregistrer plusieurs adresses ou préférences de

livraison. Les statuts de commande (en préparation, expédiée, livrée) accompagnés de notifications email viendraient compléter cette expérience.

## Fonctionnalités communautaires et marketing

Un module d'abonnement à la newsletter, connecté à un service externe, permettrait de diffuser les nouveautés et promotions. L'ajout d'avis clients modérés et d'un système de notation produits renforcerait la confiance et la crédibilité de la boutique.

## Internationalisation et accessibilité

La plateforme pourrait être proposée en plusieurs langues (FR/EN) avec une conversion de devises (EUR/USD), tout en intégrant un paramétrage des taxes et de la TVA selon les pays. Enfin, pour améliorer le confort d'utilisation, un mode sombre viendrait enrichir l'expérience utilisateur.

Ces évolutions représentent des axes de développement progressifs et cohérents, pensés pour offrir une expérience complète et moderne, tout en s'adaptant aux attentes de nos clients.

# 6 - Wireframes et Maquettes

## Page d'accueil

La page d'accueil s'ouvre sur un header clair et fonctionnel, où le logo occupe une place centrale. Sur le côté gauche, un bouton de menu permet d'accéder facilement aux univers Homme, Femme et Nouveautés. À droite, trois icônes assurent une navigation intuitive : la première pour la connexion au compte client, la seconde, une loupe, permet d'ouvrir une barre de recherche cachée qui glisse à l'écran lorsqu'on appuie dessus, facilitant ainsi l'exploration des produits. La troisième icône donne accès au panier, lui aussi masqué par défaut et affiché en slide au moment de l'interaction.

Juste en dessous, la section principale capte l'attention grâce à une grande image immersive qui reflète pleinement l'univers des bracelets en pierres naturelles. Cette mise en avant visuelle est accompagnée d'un slogan ou d'une phrase de présentation qui exprime l'identité de la marque et son univers.

Pour enrichir l'atmosphère et valoriser l'authenticité, des visuels complémentaires viennent illustrer la qualité des matériaux, l'ambiance unique et le savoir-faire de l'atelier.

Enfin, le footer rassemble les informations essentielles : détails sur la livraison et les paiements, conditions générales de vente et liens pratiques vers la page de contact. Des icônes redirigeant vers les réseaux sociaux comme Instagram, Facebook et Pinterest prolongent la relation avec la marque.

et renforcent sa présence auprès de la communauté. Un court texte précise également les droits d'auteur ainsi que la conformité au RGPD.

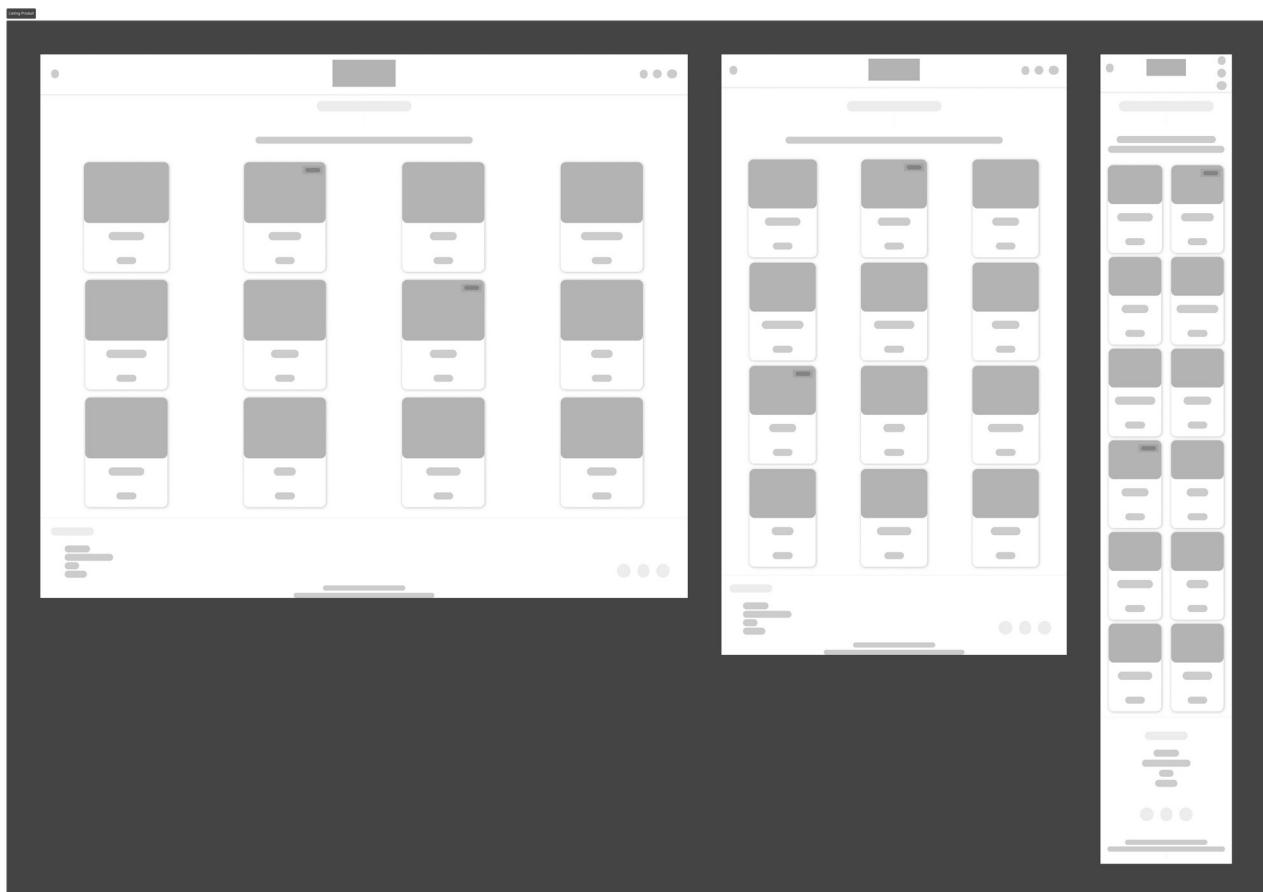
## Côté gauche de l'image sur écran d'ordinateur, milieu sur tablette et côté droit sur écran de smartphone.

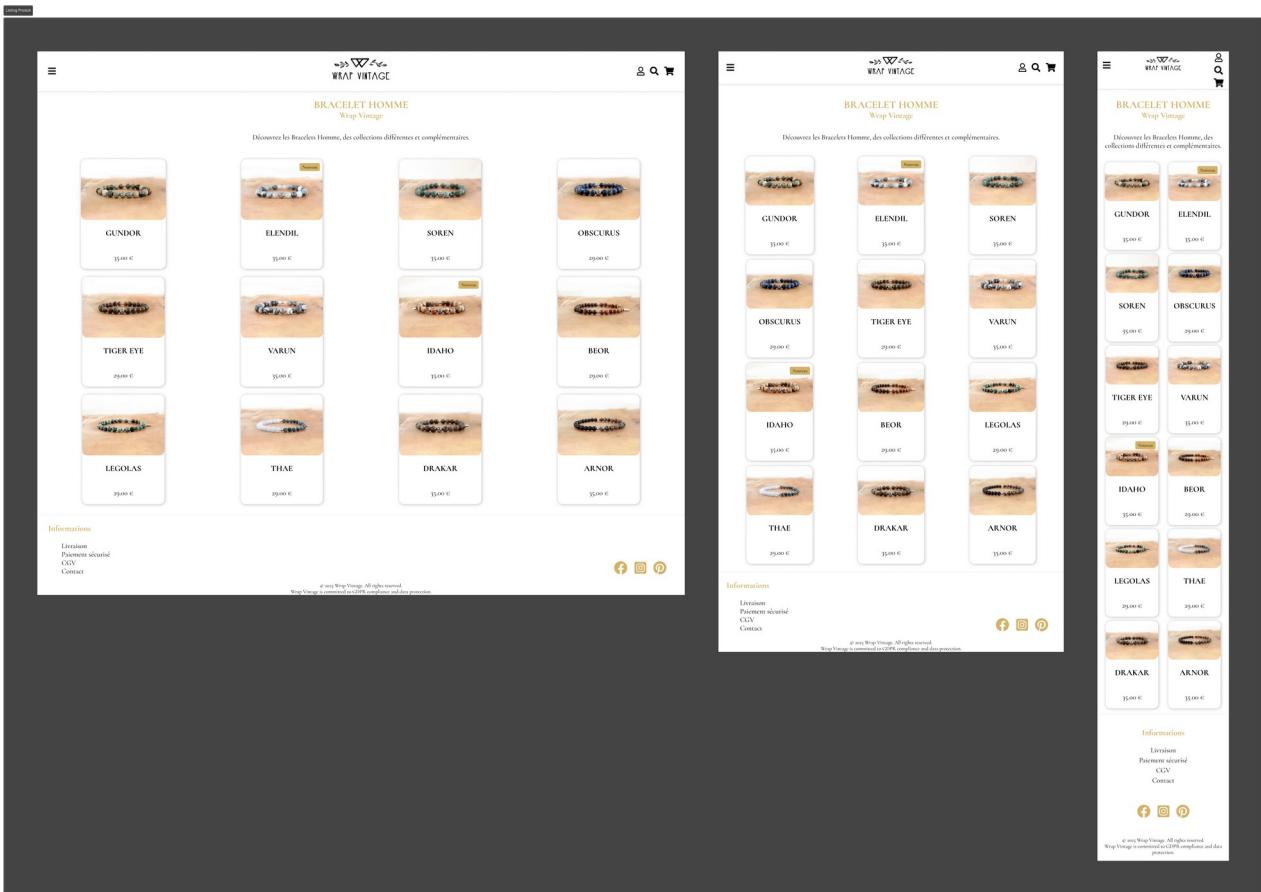


The image shows three screenshots of the Wrap Vintage website, demonstrating its responsive design across desktop, tablet, and smartphone platforms. The desktop version (left) features a navigation bar with links like 'Accueil', 'Nos créations', 'Bracelet Homme', and 'Bracelet Femme'. It includes a large product image of multiple bracelets, several descriptive text blocks with icons (e.g., 'Un bijou créateur Made in France', 'Des pierres semi-précieuses', 'Des bracelets vegan'), and a 'Wrap Vintage, une marque engagée' section with a turtle image. The tablet version (middle) has a similar structure but with a more compact layout. The smartphone version (right) is a vertical scroll-based layout, featuring a sidebar on the left with the same navigation links and a main content area on the right that displays the same information in a mobile-friendly format.

## Listing de produits

En haut de la page, un bloc texte centré introduit la collection avec le titre « Wrap Vintage », accompagné d'un court paragraphe de présentation qui met en valeur l'esprit et l'originalité de la gamme. Juste en dessous, le regard se pose sur une grille de produits qui occupe toute la largeur de l'écran. L'organisation s'adapte selon le support : quatre colonnes sur ordinateur pour une présentation généreuse, trois colonnes sur tablette et deux sur mobile afin de conserver un affichage clair et harmonieux. Chaque produit est présenté dans une carte élégante, parfaitement centrée dans sa cellule, pour mettre en avant les détails et faciliter la navigation.

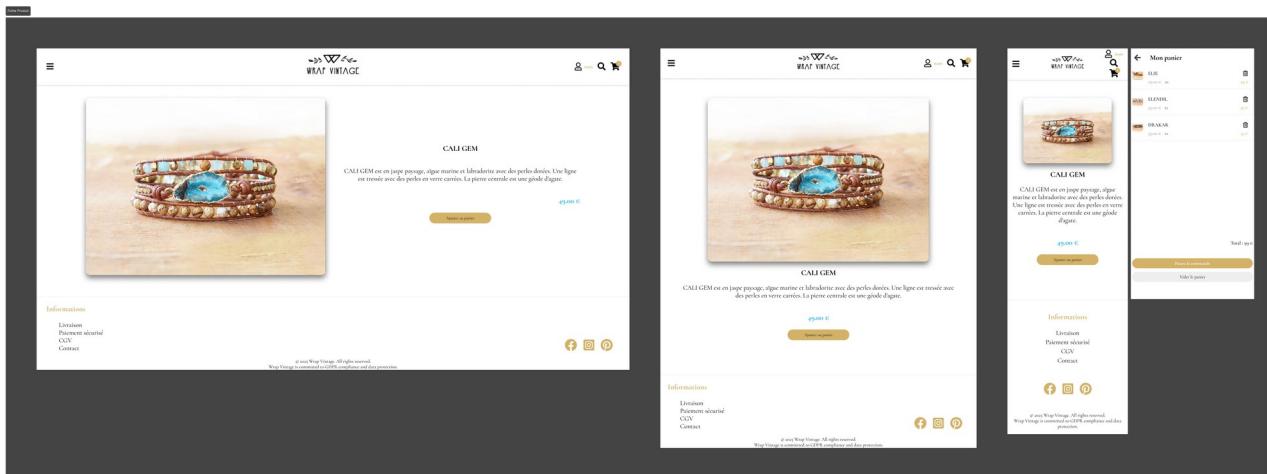




## Fiche produit

La page produit s'organise autour d'une mise en page épurée qui met le bracelet au premier plan. Sur ordinateur, la présentation se fait en deux colonnes : à gauche, une grande image du produit occupe toute la largeur de son espace, avec des bords arrondis et une légère ombre pour lui donner du relief. À droite, les informations essentielles sont regroupées de façon structurée : le nom du bracelet, une description détaillée et le prix, discrètement aligné à droite. Juste en dessous, un bouton large et arrondi, marqué d'un « Ajouter au panier », invite à l'action.

Sur tablette et mobile, la mise en page se réorganise pour plus de confort. L'image du bracelet se recentre et s'affiche en occupant environ 80 % de la largeur de l'écran. Le prix, le texte descriptif et le bouton d'achat sont également recentrés, formant un bloc harmonieux et lisible. Le style reste volontairement minimaliste, avec des couleurs d'accent qui mettent en avant le prix et les messages de confirmation, renforçant ainsi la clarté et l'efficacité de l'expérience d'achat.



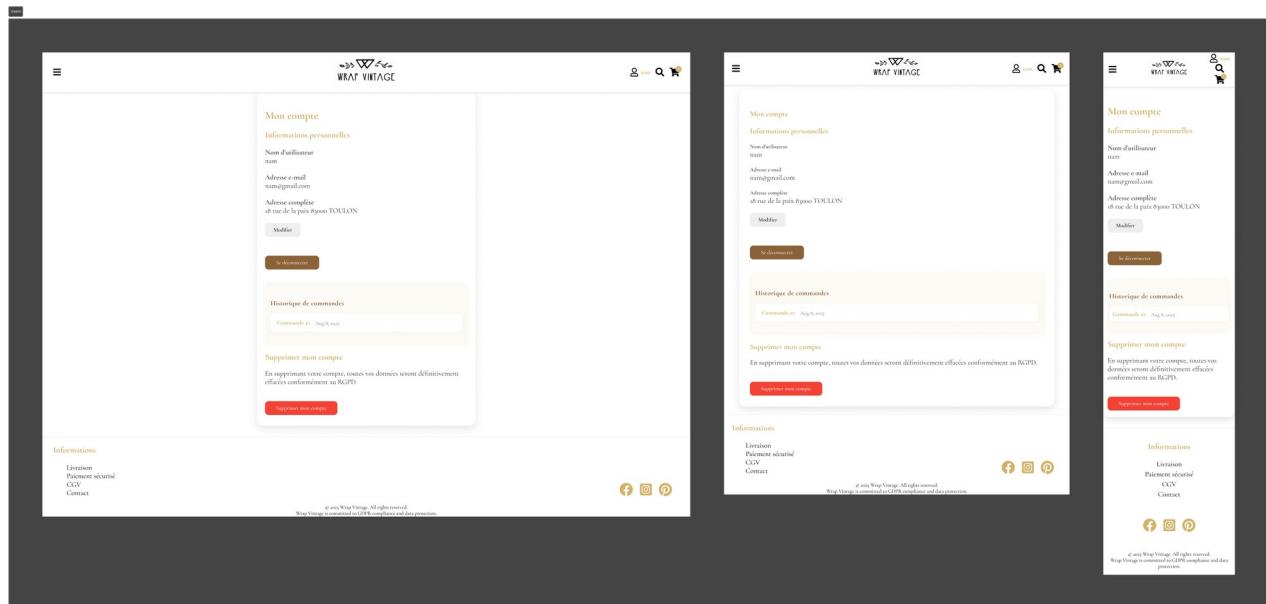
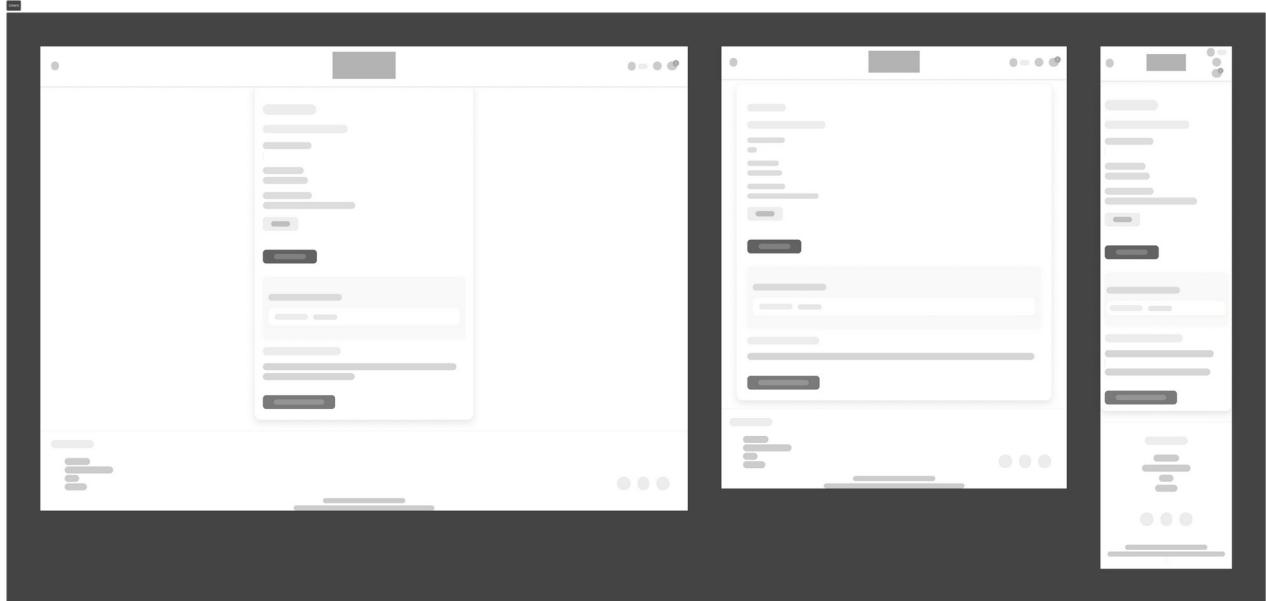
## Mon compte

La page Mon compte s'ouvre sur un en-tête clair avec le titre principal. Au centre, une carte au style épuré rassemble toutes les informations personnelles de l'utilisateur. Selon le mode d'affichage, les données comme le nom d'utilisateur, l'adresse e-mail ou l'adresse postale apparaissent en lecture seule ou sous forme de champs éditables. Lorsqu'on passe en mode édition, une zone dédiée permet également de modifier le mot de passe.

Les actions principales sont proposées sous forme de boutons adaptés à l'état du formulaire : *Modifier*, *Enregistrer* ou *Annuler*. Un bouton de déconnexion, volontairement séparé, offre un accès direct et simple à la sortie du compte.

En complément, un bloc historique de commandes présente la liste des achats réalisés. Enfin, un espace est réservé à la suppression de compte : il contient une brève explication suivie d'un bouton dédié, permettant à l'utilisateur de gérer librement ses données.

La mise en page adopte une présentation centrée et soignée, avec une carte unique dotée de légères ombres. Le design reste entièrement responsive : sur mobile, les marges se réduisent pour maintenir confort et lisibilité.



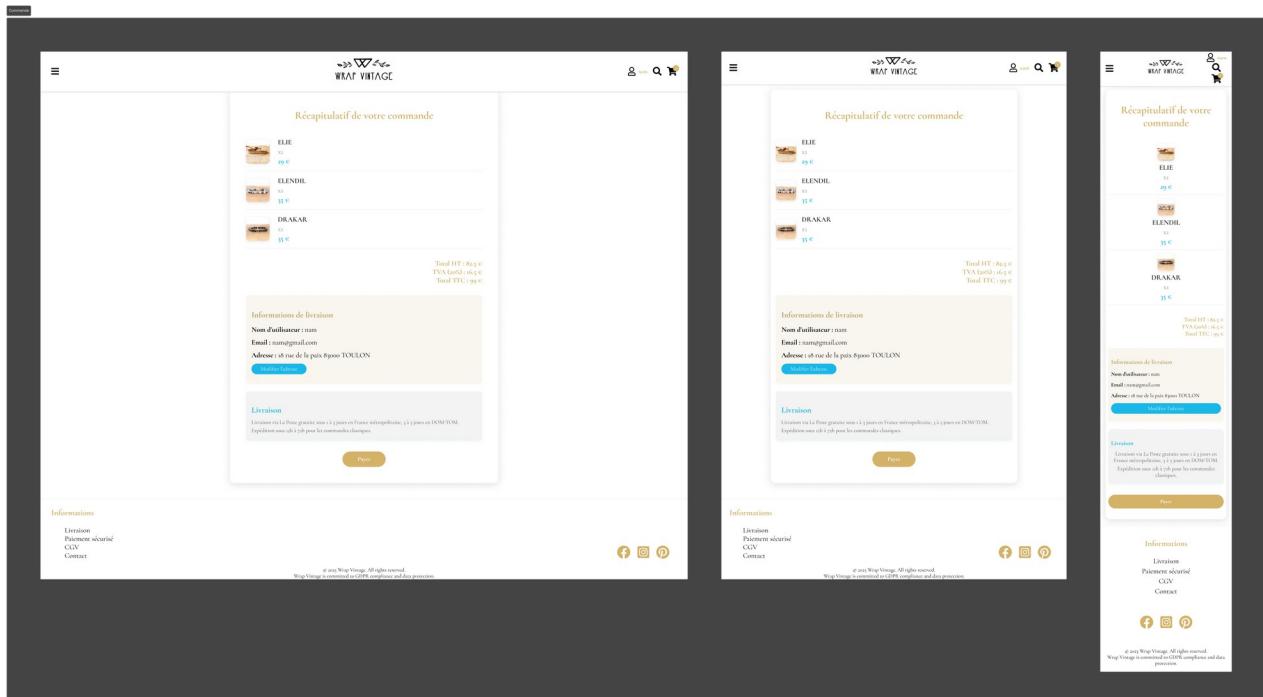
## Récapitulatif de commande

La page récapitulatif de commande s'affiche dans un container central en forme de carte, avec un titre placé en en-tête pour situer immédiatement le contexte. La partie principale présente la liste des articles choisis : chaque ligne associe une vignette visuelle à gauche et, à droite, un bloc d'informations détaillant le nom du produit, la quantité et le sous-total correspondant.

À côté, un bloc récapitulatif met en avant les montants globaux de la commande : total hors taxes, montant de la TVA et total TTC, présentés de façon claire et hiérarchisée. Un autre espace regroupe les informations utilisateur, comprenant le nom, l'adresse e-mail et l'adresse postale. Si certaines données sont manquantes, des champs de saisie ou d'édition apparaissent pour les compléter avant le paiement.

Un bloc spécifique est dédié à la livraison, avec un texte explicatif permettant de rappeler les conditions ou options. Enfin, en bas de page, un bouton large et centré « Payer » conclut le parcours, incite à finaliser la commande et redirige directement vers la plateforme de paiement sécurisée Stripe..

L'ensemble est conçu pour s'adapter à tous les écrans : sur mobile, les lignes produits passent en affichage empilé avec l'image au-dessus des informations, les marges et espacements se réduisent, et le bouton de paiement s'étend en pleine largeur pour une utilisation plus confortable.



# Spécifications techniques

## 1 - Technologies utilisées

### Front-end :

- Angular 19
- TypeScript, HTML, CSS

### Back-end :

- Node.js
- Express
- Fetch API côté front
- JSON Web Tokens (authentification via cookie HttpOnly)
- bcrypt pour le hash des mots de passe

### Base de données :

- MariaDB (SQL), gestion des données via le bash
- Modélisation préalable avec DrawDB

### Paiement :

- Stripe (Checkout et webhook)

### Infrastructure / Déploiement :

- VPS Hostinger pour l'hébergement
- Docker
- Docker Compose
- DuckDNS pour la gestion du nom de domaine dynamique
- Caddy (reverse proxy + HTTPS automatique via Let's Encrypt)
- Nginx (serveur statique pour le frontend Angular, fallback SPA)
- Gestion des variables d'environnement (.env)

### Gestion & Outils :

- Git / GitHub
- Figma pour les wireframes et maquettes

### Divers :

- Service de fichiers statiques pour les images produits et vignettes

## 2 - Navigateurs compatibles

En France, les navigateurs les plus utilisés en 2025 se répartissent ainsi :

**Desktop :**

- **Google Chrome** : 57,6 %
- **Microsoft Edge** : 15,5 %
- **Firefox** : 13,9 %
- **Safari** : 9 %
- **Opera** : 3,3 %

**Mobile :**

- **Google Chrome** : 74,8 %
- **Safari** : 17,6 %
- **Samsung Internet** : 3,9 %
- **Firefox** : 0,87 %
- **Opera** : 0,79 %

Ces chiffres montrent que la majorité des internautes français naviguent principalement via Chrome, suivi de Safari sur mobile et d'Edge ou Firefox sur desktop, ce qui est un point clé à considérer pour l'optimisation et les tests d'un site web. J'ai donc testé le site principalement sur Chrome et Firefox.

Source: [Blog du modérateur](#)

## 3 - Versionning

J'ai utilisé Git avec GitHub afin de gérer l'évolution du code de manière organisée et sécurisée. L'ensemble du projet est hébergé dans le repository : wrapvintage, ce qui permet de suivre chaque modification, de revenir à des versions précédentes si nécessaire et de collaborer facilement en cas de développement futur. Cette gestion de versions assure un suivi clair de l'historique du projet et facilite la maintenance ainsi que les mises à jour.

La totalité du code se trouve ici : [WrapVintage](#)

## 4 - Crédation de la base de données

J'ai créé la base de données du projet en utilisant DrawDB sur drawdb.vercel.app, en élaborant préalablement le schéma MCD pour structurer efficacement les relations entre les données.

Ma base de données est composée de plusieurs entités reliées entre elles. Les utilisateurs (user) possèdent des informations personnelles comme un nom d'utilisateur, un email, un mot de passe, une adresse et des attributs de gestion (statut admin, dates de création et suppression). Chaque utilisateur est lié à un seul panier (cart), qui peut contenir plusieurs produits grâce à la table d'association productCart.

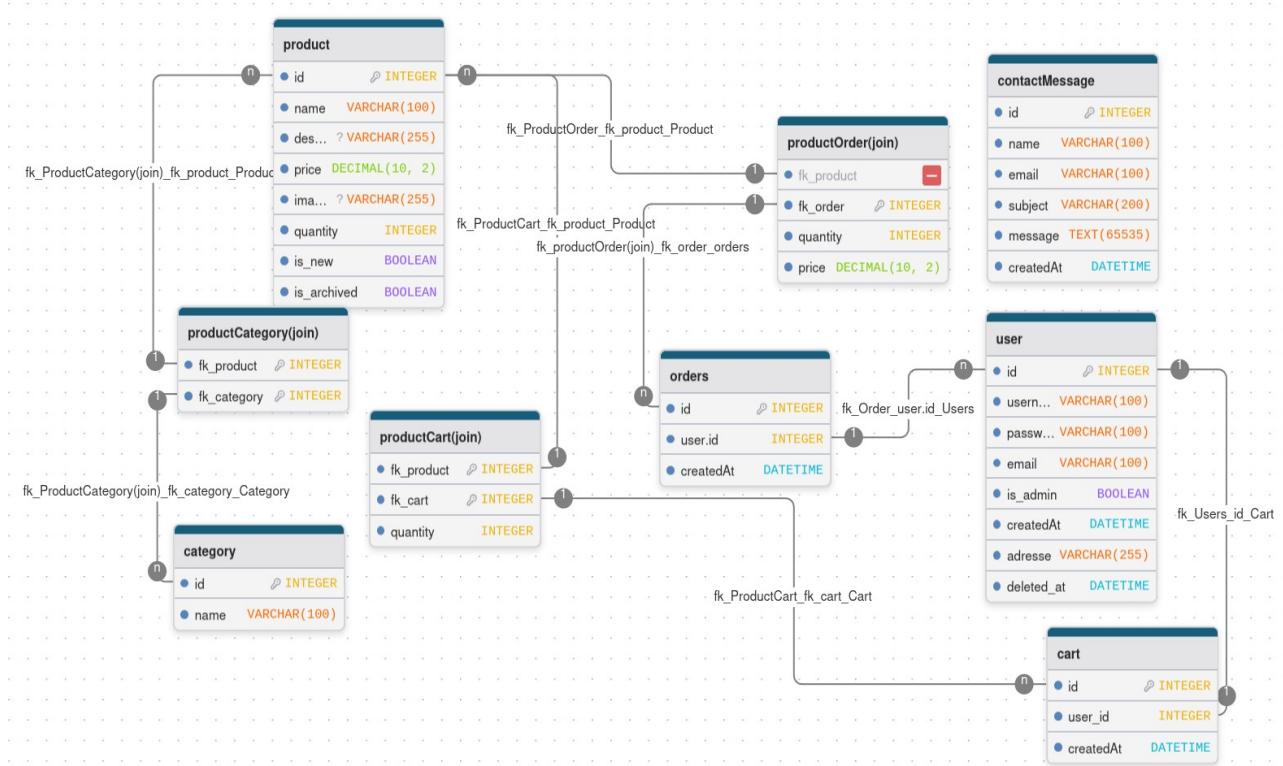
Les produits (product) sont décrits par leur nom, une description, un prix, une image, une quantité disponible ainsi que des indicateurs d'état (nouveau, archivé). Ils peuvent appartenir à plusieurs catégories (category) via la table d'association productCategory.

Les commandes (orders) permettent de relier un utilisateur à une ou plusieurs lignes de commande via productOrder, qui enregistre pour chaque produit la quantité commandée et le prix fixé au moment de l'achat.

Enfin, les messages de contact (contactMessage) sont stockés indépendamment, avec les informations de l'expéditeur et le contenu du message.

En résumé, un utilisateur possède un seul panier et peut passer plusieurs commandes. Les produits peuvent appartenir à plusieurs catégories, être ajoutés à un panier ou à une commande, tandis que les messages de contact restent isolés du reste du modèle.

## A – MCD



## B - MLD

**user** (id, username [unique], password, email [unique], adresse, is\_admin, createdAt, deleted\_at)

**product** (id, name, description, price, imageURL, quantity, is\_new, is\_archived)

**category** (id, name)

**productCategory** (fk\_product [PK, FK → product.id], fk\_category [PK, FK → category.id])

**cart** (id, user\_id [FK → user.id, UNIQUE], createdAt)

**productCart** (fk\_product [PK, FK → product.id], fk\_cart [PK, FK → cart.id], quantity)

**orders** (id, user\_id [FK → user.id], createdAt)

**productOrder** (fk\_product [PK, FK → product.id], fk\_order [PK, FK → orders.id], quantity, price)

**contactMessage** (id, name, email, subject, message, createdAt)

## C - Dictionnaire de données

**Table : user**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique de l'utilisateur
username	VARCHAR(100)	UNIQUE, NULL possible	Nom d'utilisateur
password	VARCHAR(100)	NULL possible	Mot de passe chiffré
email	VARCHAR(100)	UNIQUE, NULL possible	Adresse email
adresse	VARCHAR(255)	NULL possible	Adresse postale
is_admin	BOOLEAN	DEFAULT FALSE	Statut administrateur
createdAt	DATETIME	DEFAULT CURRENT_TIMESTAMP	Date de création du compte
deleted_at	DATETIME	NULL	Date de suppression logique (soft delete)

**Table : product**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique du produit
name	VARCHAR(100)	NOT NULL	Nom du produit
description	VARCHAR(255)	NULL	Description du produit
price	DECIMAL(10,2)	NOT NULL	Prix unitaire
imageURL	VARCHAR(255)	NULL	Lien vers l'image du produit
quantity	INT	NOT NULL	Quantité disponible
is_new	BOOLEAN	DEFAULT FALSE	Produit marqué comme "nouveau"
is_archived	BOOLEAN	DEFAULT FALSE	Produit archivé ou non

**Table : category**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique de la catégorie
name	VARCHAR(100)	NOT NULL	Nom de la catégorie

**Table : productCategory**

Attribut	Type	Contraintes	Description
fk_product	INT	PK, FK → product(id)	Référence au produit
fk_category	INT	PK, FK → category(id), ON DELETE CASCADE	Référence à la catégorie

**Table : cart**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique du panier
user_id	INT	UNIQUE, NOT NULL, FK → user(id), ON DELETE CASCADE	Un utilisateur ne peut avoir qu'un seul panier
createdAt	DATETIME	DEFAULT CURRENT_TIMESTAMP	Date de création du panier

**Table : productCart**

Attribut	Type	Contraintes	Description
fk_product	INT	PK, FK → product(id)	Produit ajouté au panier
fk_cart	INT	PK, FK → cart(id), ON DELETE CASCADE	Panier concerné
quantity	INT	NOT NULL	Quantité du produit dans le panier

**Table : orders**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique de la commande
user_id	INT	FK → user(id)	Référence à l'utilisateur ayant passé la commande
createdAt	DATETIME	DEFAULT CURRENT_TIMESTAMP	Date de la commande

**Table : productOrder**

Attribut	Type	Contraintes	Description
fk_product	INT	PK, FK → product(id)	Produit inclus dans la commande
fk_order	INT	PK, FK → orders(id)	Commande associée
quantity	INT	NOT NULL	Quantité commandée
price	DECIMAL(10,2)	NOT NULL	Prix du produit au moment de la commande

**Table : contactMessage**

Attribut	Type	Contraintes	Description
id	INT	PK, AUTO_INCREMENT	Identifiant unique du message
name	VARCHAR(100)	NULL	Nom de l'expéditeur
email	VARCHAR(100)	NULL	Adresse email de l'expéditeur

Attribut	Type	Contraintes	Description
subject	VARCHAR(200)	NULL	Sujet du message
message	TEXT	NULL	Contenu du message
createdAt	DATETIME	DEFAULT NOW()	Date d'envoi du message

J'ai choisi de mettre en place la base de données directement à travers le code et le terminal, sans utiliser d'interface graphique comme Adminer ou PhpMyAdmin. La création des tables se fait dans le fichier database.mjs : ce fichier contient toutes les instructions SQL nécessaires (CREATE TABLE IF NOT EXISTS, clés primaires, clés étrangères, contraintes, etc.).

```
backend > src > JS database.mjs > [e] product
1 import mariadb from "mariadb";
2 import { DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME } from "./config.js";
3
4 // Je me connecte au serveur MariaDB avec les infos du fichier .env
5 export const connection = await mariadb.createConnection({
6   host: DB_HOST,
7   port: DB_PORT ? parseInt(DB_PORT) : 3306,
8   user: DB_USER,
9   password: DB_PASSWORD,
10  database: DB_NAME // Nom de la base de données
11 );
12
13 const product = await connection.execute(``CREATE TABLE IF NOT EXISTS product (
14   id INT AUTO_INCREMENT PRIMARY KEY,
15   name VARCHAR(100) NOT NULL,
16   description VARCHAR(255),
17   price DECIMAL(10, 2) NOT NULL,
18   imageURL VARCHAR(255),
19   quantity INT NOT NULL,
20   is_new BOOLEAN DEFAULT FALSE,
21   is_archived BOOLEAN DEFAULT FALSE
22 )
23 ``);
24 if (product.warningStatus === 0) console.log("Table product created");
25 const user = await connection.execute(`
```

Les informations sensibles de connexion (hôte, port, utilisateur, mot de passe, nom de la base) ne sont pas écrites en dur dans le code, elles sont stockées dans un fichier .env afin d'être sécurisées et facilement modifiables si besoin.

Toute la base de données est exécutée dans un conteneur grâce à Docker Compose. Cela me permet d'avoir une configuration stable, portable et facile à relancer en cas de problème. Je n'ai pas eu besoin d'installer MariaDB directement sur ma machine, tout tourne dans le conteneur.

```
backend > 🐳 docker-compose.yml > ...
    ▷ Run All Services
1   services:
    |> Run Service
2     mariadb:
3       image: mariadb:latest
4       restart: always
5       environment:
6         MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
7         MYSQL_DATABASE: ${DB_NAME}
8         MYSQL_USER: ${DB_USER}
9         MYSQL_PASSWORD: ${DB_PASSWORD}
10      ports:
11        - "${DB_PORT}:3306"
12      volumes:
13        - mariadb_data:/var/lib/mysql
14
```

Pour gérer la base de données et faire mes tests, j'ai volontairement choisi d'utiliser uniquement le terminal. Je me connecte à la base avec la commande suivante :

```
mariadb --host=192.168.1.xxx --port=3306 --user=xxxx --password=xxxx
```

Une fois connecté, j'arrive sur le MariaDB monitor et je peux taper manuellement mes requêtes SQL (SELECT, INSERT INTO, UPDATE, DELETE, ...).

J'ai fait ce choix d'utiliser le terminal plutôt qu'un outil visuel comme Adminer parce que je voulais apprendre et pratiquer les commandes SQL à la main, comprendre leur fonctionnement et m'entraîner à écrire directement les requêtes. C'est plus formateur et ça me donne une vraie maîtrise de la base de données.

```
namoux@NamZenbook:~$ mariadb --host=192.168.1.107 --port=3306 --user=root --password=xxxx
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 27
Server version: 11.4.5-MariaDB Alpine Linux

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use shop;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

## 5 - Back end

Pour le développement du backend, j'ai choisi Node.js avec Express. Ce choix m'a permis de bénéficier d'un environnement léger, rapide et très flexible pour créer des API REST, tout en profitant de l'écosystème riche de npm pour intégrer facilement des middlewares et des services comme l'authentification JWT ou la gestion des paiements avec Stripe. L'architecture adoptée suit un style MVC léger (Models, Controllers, Routes) avec des middlewares pour les services transverses, ce qui facilite la lisibilité, la maintenance et la séparation des responsabilités dans le code.

### A - Arborescence

```
backend/
├── .env
├── Dockerfile
├── package.json
└── public/
    ├── vignettes/
    │   └── [images produits]
    └── produits/
        └── [images produits]
└── src/
    ├── server.js
    ├── config.js
    ├── database.mjs
    └── controllers/
        ├── auth.controller.js
        ├── cart.controller.js
        ├── category.controller.js
        ├── contact.controller.js
        └── order.controller.js
```

```

|   |   └── product.controller.js
|   |   └── stripe.controller.js
|   |   └── user.controller.js
|   └── models/
|       ├── cart.model.js
|       ├── category.model.js
|       ├── contact.model.js
|       ├── order.model.js
|       ├── product.model.js
|       └── user.model.js
|   └── middlewares/
|       ├── auth.middleware.js
|       └── error.middleware.js
└── routes/
    ├── auth.routes.js
    ├── cart.routes.js
    ├── category.routes.js
    ├── contact.routes.js
    ├── order.routes.js
    ├── product.routes.js
    └── stripe.routes.js

```

Il est organisé de façon logique pour séparer clairement les responsabilités et faciliter la maintenance.

- .env : contiennent les variables sensibles comme les identifiants de la base de données ou les clés JWT. Elles ne sont pas commitées pour la sécurité.
- Dockerfile : définit l'image Node.js du backend pour Docker.
- package.json : gère les dépendances Node et les scripts du projet.
- public/ : contient les fichiers statiques accessibles depuis le frontend.
  - vignettes/ : petites images des produits, utilisées par l'interface.
  - produits/ : images détaillées des produits.

- src/server.js : point d'entrée du backend, configure Express, les middlewares globaux, et monte toutes les routes.
- src/config.js : lit et expose les variables d'environnement pour l'application.
- src/database.mjs : crée la connexion à MariaDB et initialise les tables si nécessaire.
- src/controllers/ : contiennent la logique métier pour chaque domaine (auth, produits, catégories, panier, commandes, paiement Stripe, utilisateurs, contact). Les controllers reçoivent les requêtes HTTP, appellent les modèles et renvoient les réponses formatées.
- src/models/ : gèrent l'accès aux données et les requêtes SQL. Ils isolent la base de données du reste de l'application.
- src/middlewares/ :
  - auth.middleware.js : vérifie les JWT et les rôles pour sécuriser les routes.
  - error.middleware.js : gère centralement les erreurs pour renvoyer des réponses cohérentes.
- src/routes/ : définissent les endpoints pour chaque domaine et les relient aux controllers correspondants.

L'architecture suit un style MVC léger : les routes dirigent les requêtes vers les controllers, qui appellent les modèles pour interagir avec la base de données. Les middlewares gèrent les aspects transverses comme l'authentification ou la gestion des erreurs. Les fichiers statiques sont servis depuis le dossier public. Cette organisation rend le code lisible, modulable et maintenable.

## B - Installation

Il est nécessaire d'abord d'installer Node.js et npm sur mon PC avec la commande suivante sous Debian : sudo apt install nodejs npm

Ensuite, j'ai créé le dossier backend/ et initialisé un nouveau projet avec la commande : npm init -y

J'ai installé les dépendances petit à petit avec npm install pour chaque package dont j'avais besoin, par exemple :

npm install express

npm install mariadb

npm install cors

etc

À chaque installation, les packages se sont automatiquement ajoutés dans package.json, ce qui m'a permis de garder une trace de toutes les dépendances.

J'ai également mis "type": "module" dans package.json pour indiquer à Node que tous les fichiers .js du projet doivent être traités comme des modules ES (ESM), ce qui me permet d'utiliser import / export au lieu de require / module.exports.

Pour le développement, j'ai installé nodemon en devDependencies pour que le serveur se relance automatiquement à chaque modification du code : npm install --save-dev nodemon

Puis j'ai ajouté dans le script du package.json :

```
"scripts": {  
  "start": "node src/server.js",  
  "dev": "nodemon src/server.js",  
  "test": "echo \\\"Error: no test specified\\\" && exit 1"  
}
```

Pour lancer le serveur en développement, j'utilise : npm run dev

Pour le lancer en “production” de manière simple : npm start

```

backend > {} package.json > {} scripts > dev
 1  {
 2    "dependencies": {
 3      "bcrypt": "^6.0.0",
 4      "cookie-parser": "^1.4.7",
 5      "cors": "^2.8.5",
 6      "dotenv": "^16.5.0",
 7      "express": "^5.1.0",
 8      "jsonwebtoken": "^9.0.2",
 9      "mariadb": "^3.4.2",
10      "stripe": "^18.4.0"
11    },
12    "name": "serverexpress",
13    "version": "1.0.0",
14    "description": "Full Shop en express",
15    "main": "server.js",
16    "type": "module",
17    ▷Debug
18    "scripts": [
19      "test": "echo \"Error: no test specified\" && exit 1",
20      "start": "node src/server.js",
21      "dev": "nodemon src/server.js"
22    ],
23    "keywords": [],
24    "author": "",
25    "license": "ISC",
26    "devDependencies": {
27      "nodemon": "^3.1.10"
28    }
  
```

## C - Lancement du serveur

Pour gérer la base de données, je crée une connexion unique dans database.js via le package mariadb et les informations du fichier .env vu plus haut.

Cette constante connection contient toute la configuration et la connexion à MariaDB, et je l'importe ensuite dans server.js pour l'utiliser dans tous les modules et routes qui ont besoin d'accéder à la base. Cela centralise la gestion de la base de données et évite de créer plusieurs connexions inutiles.

Dans server.js, j'importe les modules nécessaires :

- express : framework pour créer le serveur HTTP et gérer les routes.
- cors : middleware pour autoriser les requêtes cross-origin depuis le frontend Angular.
- cookie-parser : pour lire les cookies HTTP, notamment le JWT.
- middlewares, routes et connection : pour gérer la logique métier, les endpoints et l'accès aux données.

Je crée ensuite une instance d'Express : const app = express();

Et je configure les middlewares globaux :

- express.json() pour parser le corps des requêtes en JSON.
- cors() avec l'URL du frontend pour autoriser les requêtes et les cookies.

Par défaut, les navigateurs bloquent les requêtes entre origines différentes pour des raisons de sécurité. C'est là que le middleware cors d'Express intervient.

L'option origin permet de préciser quelle adresse est autorisée à accéder à mon API.

L'option credentials: true autorise l'envoi de cookies ou d'en-têtes d'authentification (exemple : sessions, JWT stockés dans des cookies).

- cookieParser() pour gérer les cookies.

Il est souvent nécessaire de lire et gérer les cookies envoyés par le navigateur. Par défaut, Express ne sait pas interpréter automatiquement les cookies d'une requête HTTP, ils apparaissent juste comme une chaîne brute dans l'en-tête.

Tous les cookies envoyés par le client sont automatiquement parsés et accessibles dans req.cookies.

- express.static('public') pour servir les fichiers statiques comme les images produits ou vignettes.

```
const app = express();

app.use(express.json());
app.use(cors({
    origin: `${CLIENT_URL}`, // frontend Angular
    credentials: true           // autorise les cookies
}));
app.use(cookieParser());
app.use(express.static('public')); // Dossier pour les fichiers statiques
```

Le baseUrl correspond à l'URL complète du serveur backend, combinant :

- HOST : l'adresse du serveur (par exemple http://localhost ou une IP).
- PORT : le port sur lequel le serveur écoute (ex. 3000).

```
const baseUrl = `${HOST}:${PORT}`;
```

Cette constante est pratique pour :

1. Créer des liens complets dans les réponses API, par exemple pour renvoyer l'URL complète d'une image ou d'une ressource accessible depuis le frontend.
2. Paramétriser certains services comme les routes Stripe ou les redirections, où il faut fournir une URL complète et accessible.

Ainsi, au lieu d'écrire à plusieurs endroits l'host et le port en dur, on utilise baseUrl. Cela rend le code plus maintenable et adaptable si on change le port ou si le backend passe sur un autre serveur.

Ensuite, je définis les routes principales du backend :

```
// Routes
app.use("/products", productRoutes(connection, baseUrl));
app.use("/users", userRoutes(connection));
app.use("/categories", categoryRoutes(connection));
app.use("/auth", authRoutes(connection, baseUrl));
app.use("/cart", cartRoutes(connection, baseUrl));
app.use("/api/stripe", stripeRoutes(connection, baseUrl));
app.use('/orders', orderRoutes(connection));
app.use('/api/contact', contactRoutes(connection));
```

Chaque route reçoit la connexion et, si nécessaire, le baseUrl pour les redirections ou liens. Les routes suivent un flux classique :

1. Route HTTP reçue par Express.
2. Middleware pour l'authentification et la validation.
3. Controller qui gère la logique métier.
4. Model qui interagit avec la base de données.
5. Retour des données enrichies au controller.
6. Réponse HTTP envoyée au client.

Enfin, je gère les erreurs et les pages 404 :

```

// Handle 404 as default route
app.use((req, res) => {
  console.log(`X 404 - ${req.method} ${req.originalUrl}`);
  res.status(404).json({ msg: "Page Not Found" });
});

// Gestion des erreurs
app.use(errorHandler);

```

Et je lance le serveur sur le port défini dans .env

```

app.listen(PORT, () => {
  console.log(`Server listening on ${baseUrl}`);
});

```

Ainsi, tout est prêt pour recevoir et traiter les requêtes du frontend, avec un accès centralisé à la base de données et des routes organisées de manière logique.

Le fichier config.js a pour rôle de centraliser toutes les variables d'environnement et certaines configurations globales utilisées dans le backend.

Chargement du fichier .env :

```
import dotenv from "dotenv";
dotenv.config();
```

Cette ligne permet à Node de lire automatiquement les variables définies dans .env et de les rendre accessibles via process.env. Cela permet de ne pas mettre d'informations sensibles en dur dans le code, comme les identifiants de base de données ou les clés secrètes.

Export des variables :

```
export const PORT = process.env.PORT;
export const HOST = process.env.HOST;
```

## D - Flux de requêtes

Le flux d'une requête dans mon backend se déroule de manière très structurée pour séparer clairement les responsabilités et faciliter la maintenance du code.

## 1. Client :

Tout commence avec le client, le frontend Angular. Il envoie une requête HTTP (GET, POST, PUT, DELETE) vers le serveur pour récupérer des données, créer un nouvel utilisateur, ajouter un produit au panier, etc.

## 2. Route (routes/) :

La requête arrive sur un endpoint défini dans les fichiers de routes (auth.routes.js, product.routes.js, etc.).

Les routes ne contiennent pas de logique métier : elles font juste le lien entre l'URL appelée et le contrôleur correspondant.

C'est aussi ici qu'on peut appliquer des middlewares spécifiques à certaines routes.

## 3. Middleware (middlewares/) :

Les middlewares sont des fonctions transverses qui interviennent avant le contrôleur.

auth.middleware.js vérifie le JWT pour s'assurer que l'utilisateur est bien authentifié.

## 4. Contrôleur (controllers/) :

Si la requête passe les middlewares, elle arrive au contrôleur correspondant.

Le contrôleur contient la logique métier : il décide quoi faire avec la requête, quelles vérifications supplémentaires effectuer, et quelles données récupérer ou modifier.

Il appelle les modèles pour accéder à la base de données.

## 5. Modèle (models/) :

Les modèles exécutent les requêtes SQL sur la base de données MariaDB via la connexion centralisée (connection).

Ils sont responsables uniquement de l'accès aux données : SELECT, INSERT, UPDATE, DELETE.

Une fois les données récupérées ou modifiées, elles sont renvoyées au contrôleur.

## 6. Contrôleur (suite) :

Le contrôleur peut ensuite enrichir ou transformer les données reçues du modèle avant de les formater en JSON ou autre format attendu par le client.

## 7. Client :

Enfin, la réponse formatée est renvoyée à l'utilisateur via la route initiale.

Le client peut alors afficher les informations, mettre à jour l'interface ou gérer les erreurs reçues.

Chaque requête suit un chemin clair :

Client (navigateur / frontend)



Route



Middleware

(authentification, validation...)



Controller

(logique métier)



Model

(accès à la base de données)



Base de données



Model (transforme / enrichit les données)



Controller (formatage final de la réponse)



Route



Client (réponse JSON / message)

En résumé :

1. Le client envoie la requête → elle passe par Route → Middleware → Controller → Model  
→ Base de données.
2. La base de données renvoie les données au Modèle.
3. Ensuite, les données remontent au contrôleur, qui peut les transformer ou les enrichir.
4. Le contrôleur renvoie la réponse à la route, qui la transmet finalement au client.

## E - Les routes

Les routes définissent les points d'entrée HTTP pour le client. Chaque route est associée à un controller qui contient la logique métier, et certaines sont protégées par des middlewares comme checkTokenValid ou checkTokenAdmin pour gérer l'authentification et les droits d'accès.

Chaque route est définie dans un fichier dédié par type de ressource : produits, utilisateurs, catégories, panier, commandes, Stripe, contact, et authentication. Les routes utilisent les models pour interagir avec la base de données.

## Exemple : les routes des produits (product.routes.js)

```
/**  
 * Définit les routes pour la gestion des produits  
 * @param {object} connection - Instance de connexion à la base de données  
 * @param {string} baseUrl - URL de base pour les images  
 * @returns {import('express').Router} - Router Express configuré  
 */  
const productRoutes = (connection, baseUrl) => [  
    const router = express.Router();  
    const productModel = new ProductModel(connection, baseUrl);  
  
    router.get('/all', getAllProducts(productModel));  
    router.get('/news', getAllNewProducts(productModel));  
    router.get('/archived', checkTokenAdmin(connection), getAllArchivedProducts(productModel));  
    router.get('/hommes', getProductHomme(productModel));  
    router.get('/femmes', getProductFemme(productModel));  
    router.get('/search/:query', getSearchProduct(productModel));  
    router.get('/:id', getProductById(productModel));  
    router.post('/add', checkTokenAdmin(connection), addProduct(productModel));  
    router.put('/update/:id', checkTokenAdmin(connection), updateProduct(productModel));  
    router.delete('/delete/:id', checkTokenAdmin(connection), deleteProduct(productModel));  
  
    return router;  
};  
  
export default productRoutes;
```



Chaque route appelle une fonction du controller qui elle-même utilise le model pour accéder aux données dans la base. Les middlewares comme checkTokenAdmin ou checkTokenValid peuvent intercepter la requête pour vérifier les droits avant d'atteindre le controller.

## Tableau des routes

Ressource	Route	Méthode	Middleware	Action
Auth	/auth/login	POST	-	Connexion utilisateur
Auth	/auth/signup	POST	-	Création de compte
Auth	/auth/me	GET	-	Récupérer l'utilisateur courant
Auth	/auth/logout	POST	-	Déconnexion
Utilisateurs	/users/all	GET	checkTokenAdmin	Récupérer tous les utilisateurs actifs
Utilisateurs	/users/archived	GET	checkTokenAdmin	Récupérer les utilisateurs archivés
Utilisateurs	/users/:id	GET	checkTokenAdmin	Récupérer un utilisateur par ID
Utilisateurs	/users/update/:id	PUT	checkTokenValid	Modifier un

Ressource	Route	Méthode	Middleware	Action
				utilisateur
Utilisateurs	/users/delete/:id	DELETE	checkTokenValid	Supprimer un utilisateur
Produits	/products/all	GET	-	Récupérer tous les produits
Produits	/products/news	GET	-	Récupérer les nouveautés
Produits	/products/archived	GET	checkTokenAdmin	Produits archivés (admin)
Produits	/products/hommes	GET	-	Produits homme
Produits	/products/femmes	GET	-	Produits femme
Produits	/products/search/:query	GET	-	Recherche par mot clé
Produits	/products/:id	GET	-	Produit par ID
Produits	/products/add	POST	checkTokenAdmin	Ajouter un produit
Produits	/products/update/:id	PUT	checkTokenAdmin	Modifier un produit
Produits	/products/delete/:id	DELETE	checkTokenAdmin	Supprimer un produit
Catégories	/categories/all	GET	checkTokenAdmin	Récupérer toutes les catégories
Catégories	/categories/:id	GET	checkTokenAdmin	Récupérer une catégorie par ID
Catégories	/categories/add	POST	checkTokenAdmin	Ajouter une catégorie
Catégories	/categories/update/:id	PUT	checkTokenAdmin	Modifier une catégorie
Catégories	/categories/delete/:id	DELETE	checkTokenAdmin	Supprimer une catégorie
Panier	/cart/me	GET	checkTokenValid	Récupérer le panier de l'utilisateur
Panier	/cart/add	POST	checkTokenValid	Ajouter un produit au panier
Panier	/cart/remove	DELETE	checkTokenValid	Retirer un produit du panier
Panier	/cart/clear	DELETE	checkTokenValid	Vider le panier
Commandes	/orders/last/:userId	GET	checkTokenValid	Récupérer la dernière commande

Ressource	Route	Méthode	Middleware	Action
Commandes	/orders/all/:userId	GET	checkTokenValid	Récupérer toutes les commandes d'un utilisateur
Commandes	/orders/:orderId	GET	checkTokenValid	Récupérer une commande par ID
Stripe	/api/stripe/create-checkout-session	POST	-	Créer une session de paiement
Stripe	/api/stripe/webhook	POST	-	Webhook Stripe (body raw)
Contact	/api/contact/	POST	-	Envoyer un message de contact

## F - Middleware

J'utilise les middlewares car c'est un concept central dans Express. Un middleware, c'est tout simplement une fonction qui s'exécute entre la requête du client et la réponse du serveur. Il agit comme une étape intermédiaire, un "intercepteur" qui peut analyser, modifier ou bloquer la requête avant qu'elle n'arrive au controller.

Un middleware reçoit toujours les paramètres (req, res, next) :

- req contient les informations de la requête (headers, body, cookies, etc.)
- res permet d'envoyer une réponse au client
- next sert à passer la main au middleware suivant ou au controller final

En résumé, un middleware me permet de factoriser du code et de sécuriser et fiabiliser mon API en contrôlant le flux des requêtes avant qu'elles n'arrivent à la logique métier.

J'utilise deux middlewares principaux pour sécuriser les routes : checkTokenValid et checkTokenAdmin, j'utilise également un middleware global de gestion des erreurs pour centraliser le traitement des exceptions : errorHandler.

### checkTokenValid

Ce middleware sert à vérifier que l'utilisateur est bien authentifié.

```

const token = req.cookies.token;
const decodeToken = jwt.verify(token, SECRETKEY);

// Ajoute l'id et le rôle dans req.user
req.user = {
    id: decodeToken.id,
    is_admin: decodeToken.is_admin
};

console.log("Token authorized");
next();

```

- Il récupère le token JWT depuis les cookies de la requête.
- Il le décodifie grâce à notre clé secrète SECRETKEY.
- Il extrait l'ID de l'utilisateur et son rôle (is\_admin) et les stocke dans req.user.
- Si le token est absent ou invalide : return res.status(401).json({ msg: "Wrong token" });
 le middleware renvoie une réponse 401 Unauthorized.

## checkTokenAdmin

Ce middleware est une extension de checkTokenValid pour les routes qui nécessitent des droits d'administration.

```

const token = req.cookies.token;

const decodeToken = jwt.verify(token, SECRETKEY);

const IdUser = decodeToken.id;

const user = await connection.execute('SELECT * FROM user WHERE id = ?', [IdUser]);

if (user[0].is_admin === 1) {
    console.log("Token Admin valid");
    next();
} else {
    console.log("Access denied, not Admin");
    return res.status(403).json({ msg: "Access denied, not Admin" });
}

```

- Il récupère et décode également le token.
- Il identifie l'utilisateur dans la base de données pour vérifier s'il est admin (is\_admin === 1).
- Si oui, la requête continue vers le controller.

- Sinon, le middleware renvoie un 403 Forbidden.
- En cas de token invalide, on renvoie un 401 Unauthorized.

Ces deux middlewares permettent de protéger les routes sensibles, en s'assurant que seuls les utilisateurs authentifiés, et éventuellement les admins, peuvent y accéder. Cela sécurise notre API et empêche des utilisateurs non autorisés de modifier des données ou d'accéder à des informations confidentielles.

## errorhandler

```
/** 
 * Middleware de gestion des erreurs Express
 * @param {Error} err - L'erreur capturée
 * @param {import('express').Request} req - Requête Express
 * @param {import('express').Response} res - Réponse Express
 * @param {Function} next - Fonction next middleware
 * @returns {void}
 */
export const errorHandler = (err, req, res, next) => {
  console.error("🔥 Error:", err.message);
  return res.status(500).json({ error: err.message || 'Internal server error' });
};
```

Ce middleware prend quatre paramètres : l'erreur capturée err, la requête req, la réponse res et la fonction next.

- Il loggue l'erreur dans la console pour aider au débogage.
- Il renvoie une réponse HTTP 500 avec le message de l'erreur, ou un message générique si l'erreur n'en contient pas.

Dans un controller, lorsqu'une opération peut générer une erreur, on utilise try/catch et on passe l'erreur au middleware via next(error) :

```
} catch (error) {
  console.log("Error fetching men's products ");
  next(error); // L'erreur est transmise à errorHandler
}
```

Ici, si la récupération des produits échoue, l'erreur est capturée dans le catch.

- En appelant next(error), elle est transmise automatiquement au middleware errorHandler.

- Cela permet de centraliser le traitement des erreurs et d'éviter de répéter les mêmes réponses d'erreur dans chaque controller.

Ce système rend l'application plus robuste et maintenable, car toutes les erreurs sont gérées de manière uniforme, avec un message clair côté serveur et une réponse structurée côté client.

## G - Controller

Les controllers représentent la partie “logique métier” de mon application. Concrètement, quand une requête arrive sur une route, c'est le controller associé qui va exécuter le traitement nécessaire :

- récupérer des données depuis le model
- appliquer une logique (filtrer, valider, calculer, etc.)
- et renvoyer une réponse formatée au client (JSON, message d'erreur, etc.).

Les controllers permettent donc de séparer clairement la logique applicative des autres couches :

- Routes → définissent les URL et redirigent vers le bon controller
- Controllers → contiennent la logique métier
- Models → gèrent la communication avec la base de données

```
/** 
 * Récupère les produits de la catégorie Homme
 * @param {ProductModel} productModel - Modèle produit
 * @returns {Function} - Middleware Express
 */
export const getProductHomme = (productModel) => async (req, res, next) => {
  try {
    console.log("Client gets all men's products");

    const limit = parseInt(req.query.limit);

    const products = await productModel.getProductHomme(limit);

    console.log("All men's products was sent to client");

    return res.status(200).json(products);

  } catch (error) {
    console.log("Error fetching men's products ");
    next(error); // L'erreur est transmise à errorHandler
  }
};
```

Ici, le controller :

1. utilise la méthode du model (productModel.getAllProducts()) pour interroger la BDD
2. renvoie la réponse en JSON au client (res.json(products))
3. en cas d'erreur, délègue la gestion à errorHandler via next(error)

Pourquoi les controllers sont importants ?

- Ils centralisent la logique métier, ce qui rend le code plus lisible et maintenable.
- Ils permettent de réutiliser facilement les models (exemple : productModel est injecté dans plusieurs fonctions).
- Ils isolent les responsabilités : les routes s'occupent juste de la navigation, les controllers du traitement, et les models de la data.

## Tableau des controllers

Fichier	Fonction (controller)	Description (logique métier)
auth.controller.js	register	Inscription d'un nouvel utilisateur (hash du mot de passe, insertion en BDD, retour d'un token).
	login	Authentifie un utilisateur, vérifie le mot de passe, renvoie un JWT.
	logout	Supprime le token du cookie pour déconnecter l'utilisateur.
user.controller.js	checkAuth	Vérifie si l'utilisateur est encore connecté et renvoie ses infos.
	getAllUsers	Récupère tous les utilisateurs depuis la BDD (admin uniquement).
	getUserById	Récupère un utilisateur précis par son ID.
	updateUser	Met à jour les infos d'un utilisateur (profil, email, etc.).
	deleteUser	Supprime un utilisateur de la BDD.
product.controller.js	getAllProducts	Retourne la liste complète des produits.
	getProductById	Récupère un produit précis grâce à son ID.
	createProduct	Ajoute un nouveau produit (admin).
	updateProduct	Met à jour un produit existant.
category.controller.js	deleteProduct	Supprime un produit (admin).
	getAllCategories	Retourne toutes les catégories

Fichier	Fonction (controller)	Description (logique métier)
		disponibles.
	getCategoryById	Récupère une catégorie spécifique.
	createCategory	Crée une nouvelle catégorie (admin).
	updateCategory	Modifie une catégorie existante.
	deleteCategory	Supprime une catégorie (admin).
cart.controller.js	getCartByUser	Récupère le panier d'un utilisateur connecté.
	addToCart	Ajoute un produit au panier.
	updateCartItem	Modifie la quantité d'un produit dans le panier.
	removeFromCart	Supprime un produit du panier.
	clearCart	Vide complètement le panier.
order.controller.js	createOrder	Crée une commande à partir du panier (paiement validé).
	getOrdersByUser	Retourne toutes les commandes passées par un utilisateur.
	getOrderById	Récupère une commande précise.
	getAllOrders	Liste toutes les commandes (admin).
stripe.controller.js	createCheckoutSession	Crée une session de paiement Stripe.
	handleWebhook	Gère les événements Stripe (paiement validé, échec, etc.).
contact.controller.js	sendMessage	Reçoit le message d'un utilisateur et l'envoie par email / stocke en BDD.

Parmi ces controllers, je vais vous parler de l'authentification et paiement.

Notre API e-commerce gère à la fois l'authentification sécurisée et le paiement en ligne. Voici comment chaque brique fonctionne.

## Hashage des mots de passe (bcrypt)

Lors de l'inscription, on ne stocke jamais un mot de passe en clair. On utilise bcrypt pour générer un hash sécurisé avant de l'enregistrer en base

```
const hashPassword = await bcrypt.hash(password, 10);

// Crée l'utilisateur et récupère son id
const userId = await userModel.createUser({ username, hashPassword, email, is_admin });
```

Résultat : même si la base de données fuit, les mots de passe sont illisibles.

## Génération d'un JWT (jsonwebtoken)

Au login, on vérifie le mot de passe via bcrypt.compare. Si c'est valide, on crée un JWT (JSON Web Token) qui contient les infos utiles de l'utilisateur (id, username).

```
const token = jwt.sign(
  {
    username: user.username,
    id: user.id,
  },
  SECRETKEY,
  { expiresIn: '1h' }
);
```

Ce token est une **preuve d'identité** : il sera envoyé avec chaque requête pour authentifier l'utilisateur.

## Stockage sécurisé du JWT dans un cookie HttpOnly

Plutôt que de renvoyer le token brut au client (risque XSS), on l'envoie dans un cookie HttpOnly. Cela le rend inaccessible en JavaScript côté navigateur.

```
// Envoie le token en cookie sécurisé
res.cookie('token', token, {
  httpOnly: true,          // pas accessible en JS client
  secure: false,           // en local!
  sameSite: 'Strict',     // protection CSRF
  // secure: true,          // △ nécessite HTTPS (en prod) nécessaire si SameSite=None ...
  maxAge: 3600000          // 1 heure en ms
});
```

Le cookie sera automatiquement envoyé par le navigateur avec chaque requête → plus besoin de gérer le token manuellement côté frontend.

## Vérification de l'utilisateur connecté

Pour accéder aux ressources protégées, l'API récupère le token depuis les cookies et le vérifie avec la clé secrète.

```
// Récupère le token JWT depuis les cookies
const token = req.cookies.token;

// Si aucun token, l'utilisateur n'est pas authentifié
if (!token) {
    console.log("Non authentifié");
    return res.status(401).json({ error: "Non authentifié" });
}

// Vérifie et décode le token avec la clé secrète
const decoded = jwt.verify(token, SECRETKEY);

// Récupère l'utilisateur correspondant à l'id du token
const users = await userModel.getUserById(decoded.id);
```

Si le token est invalide ou expiré → 401 Unauthorized.  
Sinon, on retourne les infos de l'utilisateur.

## Déconnexion (logout)

La déconnexion supprime simplement le cookie du navigateur :

```
// Supprime le cookie 'token' côté client
res.clearCookie('token', {
    httpOnly: true, // Le cookie n'est pas accessible en JS côté client
    sameSite: 'Lax', // Protection CSRF, ou 'None' si sous-domaines/HTTPS
    secure: NODE_ENV === 'production' // Cookie envoyé uniquement en HTTPS
});
```

## Paiement Stripe Checkout

Pour gérer le paiement, on utilise Stripe Checkout. On transforme le panier du client en line\_items adaptés à Stripe :

```
// Transforme chaque produit du panier en un objet Stripe "line_item"
const line_items = cart.map(item => ({
  price_data: {
    currency: 'eur', // Devise utilisée
    product_data: {
      name: item.name, // Nom du produit
      metadata: {
        // On stocke l'id du produit de notre base dans le metadata Stripe pour le retrouver facilement dans le webhook
        product_id: item.product_id.toString()
      }
    },
    unit_amount: Math.round(item.price * 100), // Convertit le prix en euros (€) en centimes (Stripe attend un prix en centimes,
  },
  quantity: item.quantity, // Quantité commandée
}));
```

Puis on crée une session de paiement Stripe :

```
// Crée une session Stripe Checkout avec les infos du panier et de l'utilisateur
const session = await stripe.checkout.sessions.create({
  payment_method_types: ['card'], // Méthodes de paiement acceptées
  mode: 'payment', // Mode de paiement unique
  line_items, // Produits à payer
  customer_email: user.email, // Email du client pour Stripe
  success_url: `${CLIENT_URL}/success`, // URL de redirection après paiement réussi
  cancel_url: `${CLIENT_URL}/cancel`, // URL de redirection si paiement annulé
});
```

Le frontend reçoit sessionId et redirige l'utilisateur vers la page sécurisée de paiement Stripe.

## Webhook Stripe (confirmation du paiement)

Une fois le paiement validé, Stripe envoie un webhook à notre serveur. On intercepte l'événement checkout.session.completed pour créer une commande et vider le panier.

```

// Vérifie si l'événement Stripe reçu correspond à un paiement réussi (checkout.session.completed)
if (event.type === 'checkout.session.completed') {
    // Récupère l'objet session Stripe associé à l'événement
    const session = event.data.object;

    // Récupère les line_items via l'API Stripe
    const lineItems = await stripe.checkout.sessions.listLineItems(session.id);

    // Récupère l'email du client
    const email = session.customer_email;

    // Utilise le UserModel pour retrouver l'utilisateur
    const users = await userModel.getUserByEmail(email);
    const user = Array.isArray(users) ? users[0] : users;
    if (!user || !user.id) {
        console.log("Utilisateur non trouvé pour l'email :", email);
        return res.status(200).send();
    }

    const items = await Promise.all(lineItems.data.map(async item => {
        // Pour chaque produit du panier Stripe, on récupère l'objet produit Stripe associé
        // Cela permet d'accéder au metadata où on a stocké l'id du produit de notre base
        const product = await stripe.products.retrieve(item.price.product);

        return {
            // On récupère l'id du produit de notre base depuis le metadata Stripe
            // (stocké lors de la création de la session)
            product_id: parseInt(product.metadata.product_id, 10),
            // Nom du produit (récupéré depuis Stripe)
            name: product.name,
            // Prix total de la ligne (Stripe le donne en centimes, on le convertit en euros)
            price: item.amount_total / 100,
            // Quantité commandée pour ce produit
            quantity: item.quantity
        };
    }));
}

console.log("user.id utilisé pour createOrder :", user.id);

// Crée la commande dans la base de donnée via orderModel
const orderId = await orderModel.createOrder(user.id, items);

console.log("Commande créée via Stripe webhook, id :", orderId);

// Vide le panier de l'utilisateur via CartModel
const cartData = await cartModel.getCart(user.id);

```

Cela garantit que la commande est créée uniquement après un vrai paiement validé par Stripe.

## Résumé

- Sécurité Auth : mot de passe hashé avec bcrypt + token JWT en cookie HttpOnly.
- Session utilisateur : lecture du token pour vérifier l'authentification.
- Déconnexion : suppression du cookie.
- Paiement Stripe : création d'une session Checkout, redirection sécurisée.
- Webhook Stripe : confirmation du paiement, création de commande, vidage du panier.

Chaque controller est protégé par des blocs try/catch afin de gérer les erreurs et de les transmettre au middleware prévu à cet effet.

Un point important : les controllers ne doivent pas contenir trop de logique métier. Leur rôle est surtout d'orchestrer les appels aux modèles et de formater la réponse, mais la vraie logique (calculs, accès à la base, règles métier) doit rester dans les modèles.

## H - Models

les Models représentent la couche qui s'occupe de la gestion des données. Ils servent à interagir avec la base de données MariaDB : lire, créer, modifier ou supprimer des informations. Chaque model correspond à une entité du projet : utilisateurs, produits, paniers, catégories, commandes ou messages de contact.

Le rôle principal des Models est de centraliser l'accès aux données et de garantir que toutes les requêtes SQL soient organisées et sécurisées. Cela permet aux controllers de se concentrer sur la logique métier et les réponses HTTP, sans se soucier des détails de la base de données.

### Tableau des models

Model	Méthodes principales	Description
<b>OrderModel</b>	createOrder(userId, items)	Crée une commande et enregistre les produits associés.
	getLastOrder(userId)	Récupère la dernière commande d'un utilisateur.
	getAllOrders(userId)	Liste toutes les commandes d'un utilisateur.
	getOrderById(orderId)	Récupère une commande spécifique avec ses produits.
<b>ProductModel</b>	getAllProductsActive(limit)	Liste des produits actifs (non supprimés).
	addProduct(newProducts)	Ajoute un ou plusieurs produits en BDD.
	getProductById(id)	Récupère un produit par son id.
	getAllArchivedProducts(limit)	Liste des produits archivés.
	getAllNewProducts(limit)	Produits marqués comme "nouveaux".
	getProductHomme(limit)	Produits Homme.
	getProductFemme(limit)	Produits Femme.
	updateProduct(id, product, editProduct)	Met à jour un produit.
	deleteProduct(id)	Archive (soft delete) un produit.
	getSearchProduct(query)	Recherche de produits par nom.
	addImageUrl(products)	Ajoute URL vignette aux produits.

<b>Model</b>	<b>Méthodes principales</b>	<b>Description</b>
<b>UserModel</b>	addImageUrlHD(products)	Ajoute URL HD aux produits.
	getAllUsersActive(limit)	Liste des utilisateurs actifs.
	getAllUsersArchived(limit)	Liste des utilisateurs archivés.
	getUserById(id)	Récupère un utilisateur par id.
	createUser(data)	Crée un nouvel utilisateur.
	getUserForLogin(username)	Récupère un utilisateur pour le login.
	updateUser(id, user, editUser)	Met à jour un utilisateur.
	deleteUser(id)	Archive un utilisateur.
<b>CartModel</b>	getUserByEmail(email)	Récupère un utilisateur par email.
	getOrCreateCart(userId)	Récupère ou crée un panier pour l'utilisateur.
	addToCart(cartId, productId, quantity)	Ajoute un produit au panier.
	getCart(userId)	Récupère le contenu du panier.
	removeFromCart(cartId, productId)	Retire un produit du panier.
<b>CategoryModel</b>	clearCart(cartId)	Vide entièrement le panier.
	addCategory(newCategory)	Ajoute une ou plusieurs catégories.
	getAllCategories(limit)	Liste toutes les catégories.
	getCategoryById(id)	Récupère une catégorie par id.
	updateCategory(id, category, editCategory)	Met à jour une catégorie.
<b>ContactModel</b>	deleteCategory(id)	Supprime une catégorie.
	saveMessage(data)	Sauvegarde un message de contact (nom, email, sujet, message).
	getAllMessages()	Récupère tous les messages envoyés via le formulaire.

## Les différents types de requêtes utilisés

Type de requête	Mots-clés / Paramètres	Usage / Explication	Exemple
<b>SELECT</b>	WHERE, JOIN, LIMIT, ORDER BY, AS	Récupérer des données, filtrer ou trier	SELECT p.name AS productName, pc.quantity FROM productCart pc JOIN product p ON pc.fk_product = p.id WHERE pc.fk_cart = ? LIMIT 10
<b>INSERT INTO</b>	Colonnes, VALUES, NOW(), ON DUPLICATE KEY UPDATE	Ajouter de nouvelles données, gérer les doublons, ajouter une date automatiquement	INSERT INTO cart (user_id, createdAt) VALUES (?, NOW()) INSERT INTO productCart (fk_cart, fk_product, quantity) VALUES (?, ?, ?) ON DUPLICATE KEY UPDATE quantity = quantity + ?
<b>UPDATE</b>	SET, WHERE, CONCAT	Modifier des données existantes, concaténer	UPDATE user SET deleted_at = NOW(), username =

Type de requête	Mots-clés / Paramètres	Usage / Explication	Exemple
		des chaînes, filtrer les lignes à mettre à jour	CONCAT('deleted_user_', ?) WHERE id = ? UPDATE category SET name = ? WHERE id = ?
DELETE	WHERE	Supprimer des données spécifiques	DELETE FROM productCart WHERE fk_cart = ? AND fk_product = ? DELETE FROM category WHERE id = ?
JOIN	INNER JOIN, LEFT JOIN	Relier plusieurs tables pour combiner leurs données	SELECT pc.fk_product, p.name FROM productCart pc JOIN product p ON pc.fk_product = p.id WHERE pc.fk_cart = ?
LIMIT	LIMIT n	Limiter le nombre de résultats retournés	SELECT * FROM user WHERE deleted_at IS NULL LIMIT 20
AS	AS alias	Créer un alias pour une colonne ou table pour simplifier la lecture	SELECT name AS categoryName FROM category
NOW()	Fonction SQL	Récupère la date/heure actuelle	INSERT INTO contactMessage (name, email, subject, message, createdAt) VALUES (?, ?, ?, ?, NOW())
CONCAT	CONCAT(val1, val2)	Concaténer plusieurs chaînes dans une colonne	UPDATE user SET username = CONCAT('deleted_user_', ?) WHERE id = ?
ON DUPLICATE KEY UPDATE	ON DUPLICATE KEY UPDATE colonne = valeur	Met à jour une ligne existante si la clé unique est déjà présente	INSERT INTO productCart (fk_cart, fk_product, quantity) VALUES (?, ?, ?) ON DUPLICATE KEY UPDATE quantity = quantity + ?

## Prévention SQL Injection

Utilisation des paramètres « ? »

Dans les requêtes, je passe les valeurs via des paramètres « ? ».

Exemple :

```
await this.connection.execute('SELECT * FROM user WHERE email = ?', [email]);
```

Cela empêche l'injection de code SQL car les valeurs sont échappées automatiquement par MariaDB.

Les Models sont la passerelle entre l'application et la base de données.

- Ils permettent d'écrire des requêtes SQL sécurisées et réutilisables.
- Ils centralisent la logique de manipulation des données pour chaque entité.
- Grâce aux Models, les controllers n'ont pas besoin de connaître les détails de la base de données.

En résumé : les Models font le lien entre vos données et votre application, tout en gardant le code propre et maintenable.

## 6 - Front end

Le choix de la technologie front-end s'est porté sur Angular 19, un framework moderne et robuste développé par Google. Ce choix s'explique par plusieurs raisons. D'une part, Angular offre une architecture claire basée sur les composants, ce qui facilite la réutilisabilité du code, la maintenabilité et l'évolution de l'application à long terme. D'autre part, Angular intègre nativement de nombreuses fonctionnalités avancées, telles que la gestion des formulaires, le routage, l'injection de dépendances et la communication avec les API, ce qui permet de développer plus rapidement des interfaces complexes et dynamiques.

Un autre critère déterminant a été la volonté de proposer une interface responsive, afin d'assurer une expérience utilisateur optimale sur différents supports (ordinateurs, tablettes et smartphones). Si l'adaptabilité est rendue possible principalement grâce aux techniques CSS (media queries, flexbox, grid) et à l'utilisation de frameworks modernes, Angular s'intègre parfaitement avec ces outils et facilite la structuration de composants adaptés aux différents formats d'écran. Cette combinaison garantit une interface cohérente, maintenable et optimisée pour la mobilité.

En résumé, Angular 19 a été choisi non seulement pour sa stabilité et sa maturité, mais aussi pour sa capacité à accompagner le développement d'interfaces performantes, évolutives et adaptées aux usages mobiles.

## A - Arborescence

```
frontend/
| — angular.json
| — package.json
| — README.md
|
|—— public/
|   |—— favicon.ico
|   |—— logo.png
|   |—— (autres images...)
|   |—— Nouveautés/
```

```
|   └── (images nouveautés...)
```

```
|
```

```
└── src/
```

```
    ├── index.html
```

```
    ├── main.ts
```

```
    └── styles.css
```

```
    |
```

```
    |
```

```
    └── app/
```

```
        ├── app.component.*      # Composant racine
```

```
        ├── app.config.ts
```

```
        ├── app.routes.ts
```

```
        |
```

```
        |
```

```
        └── core/
```

```
            ├── interfaces/
```

```
            |   └── models.ts
```

```
            └── services/
```

```
                └── api.service.ts
```

```
                |
```

```
                |
```

```
                └── features/      # Pages principales
```

```
                    ├── home/
```

```
                    ├── product/
```

```
                    |   ├── product-list/
```

```
                    |   └── product-by-id/
```

```
                    ├── cart/
```

```
                    ├── contact/
```

```
                    ├── order/
```

```
                    |   ├── order-summary/
```

```
                    |   ├── order-detail/
```

```
                    |   └── order-confirmation/
```

```
                    └── info/
```

```
                        └── cgv/
```

```
                        └── delivery/
```

```
                        └── payment-condition/
```

```

|   |
|   |   |
|   |   |   layout/      # Composants communs
|   |   |   |
|   |   |   |   header/
|   |   |   |   footer/
|   |   |   |   compte/
|   |   |   |   login-modal/
|   |   |   |   register-modal/
|   |   |   |   menu-burger/
|   |   |   |   searchbar/
|   |   |   |   cookie-banner/
|   |   |
|   |   |
|   |   |   shared/      # Composants réutilisables
|   |   |   |
|   |   |   |   product-card/
|   |   |
|   |   |
|   |   environments/
|   |   |
|   |   environment.ts
|
└── (fichiers Angular auto-générés : tsconfig.* , .editorconfig, etc.)

```

L'architecture de mon application front-end a été pensée pour être claire, modulaire et facilement maintenable. J'ai choisi de m'appuyer sur une structure classique recommandée dans la communauté Angular, en séparant les éléments par rôle.

- Le dossier core/ contient les parties centrales de l'application, notamment les services et les interfaces. Par exemple, le service api.service.ts permet de gérer la communication avec l'API. Il centralise toutes les fonctions qui effectuent des requêtes via fetch, afin de récupérer ou envoyer des données. Cette approche garantit une meilleure organisation du code et évite de répéter la logique d'appel à l'API dans chaque composant. Les interfaces définissent les modèles de données utilisés dans l'ensemble du projet.
- Le dossier features/ regroupe les pages principales du site, comme l'accueil (home), la gestion des produits (product-list et product-by-id), le panier (cart), la page de contact ou encore le résumé de commande (order-summary). Cette organisation par fonctionnalité rend chaque module indépendant et facilite la maintenance ou l'ajout de nouvelles pages.
- Le dossier layout/ contient les composants communs à plusieurs pages, comme l'en-tête (header), le pied de page (footer), la barre de recherche, le menu burger ou encore les fenêtres modales de connexion et d'inscription. Cela permet de centraliser les éléments récurrents de l'interface et de garder une cohérence graphique sur tout le site.

- Le dossier shared/ est destiné aux composants réutilisables dans plusieurs sections de l'application. La carte produit (product-card) est utilisée dans les différentes listes de produits.

Enfin, le dossier public/ contient toutes les ressources statiques comme les images, icônes et logos, tandis que environments/ permet de gérer différentes configurations (par exemple développement ou production).

Cette organisation permet d'avoir une application claire, scalable et facile à maintenir, tout en séparant bien les responsabilités entre les différents éléments du projet.

## B - Installation

La première étape a donc consisté à installer Angular CLI, l'outil officiel en ligne de commande qui facilite la création et la gestion des projets Angular : `npm install -g @angular/cli`

Une fois Angular CLI installé, j'ai créé un nouveau projet grâce à la commande : `ng new frontend`

Cette commande génère la structure de base d'une application Angular avec tous les fichiers nécessaires : configuration, dépendances et structure initiale du code.

## Génération de composants et de service

Angular CLI permet également de générer rapidement des éléments du projet, ce qui garantit une organisation claire et cohérente. Par exemple, pour créer un composant : `ng generate component features/home` ou plus court : `ng g c features/home`

De la même manière, pour générer un service, il suffit d'utiliser : `ng generate service core/services/api`

Ces commandes créent automatiquement les fichiers TypeScript, HTML, CSS et tests associés, tout en déclarant les nouveaux composants dans l'application.

## Configuration des environnements

Angular propose un mécanisme de gestion des environnements, ce qui permet d'avoir des configurations différentes selon le contexte (développement ou production).

- `environment.ts` contient les variables pour le développement.
- `environment.development.ts` ou `environment.prod.ts` servent à adapter les paramètres en fonction du mode de compilation.

Cette approche permet notamment de définir des URLs d'API différentes entre le développement local et la mise en ligne.

## Lancement du projet

Une fois le projet configuré, il peut être lancé en local grâce à la commande : ng serve

Par défaut, l'application est alors accessible à l'adresse <http://localhost:4200>. Cette commande lance le serveur de développement d'Angular et met en place le rechargement automatique à chaque modification du code. Cela permet de travailler en continu sur l'interface tout en voyant directement le résultat dans le navigateur.

## C - Les services

Les services jouent un rôle central pour gérer la logique métier et les échanges avec le backend. Plutôt que de mettre du code de requête HTTP directement dans les composants, nous centralisons toutes les interactions avec le serveur dans un service unique, ce qui rend l'application plus maintenable, cohérente et testable.

### Structure du service

Le service principal se trouve dans le dossier core/services/api.service.ts. Il est décoré avec `@Injectable({ providedIn: 'root' })`, ce qui permet de l'injecter dans tous les composants sans avoir à le déclarer plusieurs fois.

Il utilise également les interfaces définies dans core/interfaces/models.ts pour typer précisément les données échangées, ce qui permet à TypeScript de détecter les erreurs dès le développement. Par exemple, l'interface Product décrit un produit avec son identifiant, son nom, sa description, son prix, son image et sa quantité disponible :

```
export interface Product {
  id: number,
  name: string,
  description: string,
  price: number,
  imageURL: string,
  quantity: number,
  is_new: number
}
```

## Gestion des requêtes HTTP avec fetch

Dans notre service, nous utilisons fetch pour effectuer les requêtes vers le backend. Chaque requête est centralisée via une méthode utilitaire `handleResponse` qui se charge de parser la réponse en JSON et de gérer les erreurs HTTP de manière uniforme :

```
/**  
 * Fonction utilitaire pour parser la réponse et gérer les erreurs HTTP  
 * @param response - Réponse HTTP de fetch  
 * @returns Données parsées ou lève une erreur avec le message du backend  
 */  
private async handleResponse(response: Response): Promise<any> {  
    // Tente de parser la réponse en JSON, retourne un objet vide si erreur  
    const data = await response.json().catch(() => ({}));  
    // Si le code HTTP n'est pas OK (200-299), lève une erreur avec le message du backend (json)  
    if (!response.ok) {  
        const error = new Error(data?.error || response.statusText);  
        (error as any).status = response.status;  
        (error as any).body = data;  
        throw error;  
    }  
    // Sinon, retourne les données parsées  
    return data;  
}
```

Cette approche nous évite de répéter la gestion des erreurs dans chaque méthode et garantit une cohérence pour tous les appels API.

## Exemples de méthodes dans le service

- Récupérer tous les produits :

```
/**  
 * Récupère tous les produits (limite par défaut à 100)  
 */  
public async getProducts(limit: number = 100): Promise<Product[]> {  
    // Envoie une requête pour récupérer les produits  
    const response = await fetch(`${environment.baseURL}/products/all` + limit);  
    // Utilise la fonction utilitaire pour parser la réponse et gérer les erreurs  
    return this.handleResponse(response);  
}
```

- Connexion utilisateur :

Pour gérer la connexion d'un utilisateur, j'ai créé une méthode `login` qui prend en paramètre le `username` et le `password`. Cette méthode utilise `fetch` pour envoyer une requête au backend et récupérer les informations de l'utilisateur connecté.

```

/**
 * Connexion utilisateur
 */
public async login(username: string, password: string): Promise<User> {
    // Envoie une requête de connexion
    const response = await fetch(`.${environment.baseURL}/auth/login`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ "username": username, "password": password }),
        credentials: 'include' // Permet d'envoyer et de recevoir les cookies
    });

    return this.handleResponse(response);
}

```

Voici comment ça fonctionne et pourquoi j'utilise chaque option :

#### 1. URL de la requête

La requête est envoyée vers \${environment.baseURL}/auth/login. Cela permet de séparer l'URL de l'API du code, ce qui est pratique si je change d'environnement (développement, production...).

#### 2. Méthode POST

J'utilise POST car je dois envoyer des informations confidentielles (nom d'utilisateur et mot de passe). Contrairement à GET, les données ne seront pas exposées dans l'URL.

#### 3. Headers

J'ajoute headers: { 'Content-Type': 'application/json' } pour indiquer au serveur que les données envoyées sont au format JSON. Sans ce header, le backend pourrait ne pas comprendre correctement les informations.

#### 4. Body

Le body contient les données de connexion encodées en JSON : JSON.stringify({ username, password })

Cela convertit l'objet JavaScript en chaîne JSON, nécessaire pour que le serveur puisse le lire correctement.

#### 5. Credentials include

L'option credentials: 'include' permet d'envoyer les cookies avec la requête. C'est important car notre backend utilise des cookies pour gérer la session de l'utilisateur. Grâce à ça, le serveur peut créer ou maintenir la session de l'utilisateur et l'authentification reste persistante.

#### 6. Gestion de la réponse

Après avoir reçu la réponse du serveur, j'utilise this.handleResponse(response) pour :

- Vérifier que la requête a réussi (response.ok)

- Parser le JSON
- Lever une erreur si nécessaire avec le message du backend

En résumé, cette méthode me permet de sécuriser la connexion de l'utilisateur en envoyant correctement les informations au serveur, en respectant le format JSON, et en gérant la session grâce aux cookies.

- Gestion du panier :

Le service permet également de manipuler le panier, aussi bien côté serveur (addProductToCart, removeProductFromCart, clearCart) que côté client via des cookies (getCookieCart, setCookieCart). Cela permet de gérer un panier temporaire même pour les utilisateurs non connectés.

## Observables et partage de données

Pour synchroniser le panier entre différents composants de l'application, le service utilise un BehaviorSubject :

```
/**
 * Observable du panier partagé pour synchroniser le panier entre les composants.
 * Utiliser `cart$` pour s'abonner aux changements du panier.
 */
private cartSubject = new BehaviorSubject<CartItem[]>([]);

/**
 * Observable du panier partagé.
 */
cart$ = this.cartSubject.asObservable();

/**
 * Met à jour le panier partagé et notifie tous les abonnés.
 * À appeler après chaque modification du panier (ajout, suppression, vidage).
 * @param cart Tableau des produits du panier à partager
 */
public updateCart(cart: CartItem[]) {
  this.cartSubject.next(cart);
}
```

Chaque composant peut s'abonner à cart\$ pour recevoir automatiquement les mises à jour lorsque le panier change. Cela garantit une expérience utilisateur fluide et réactive.

En centralisant toutes les interactions avec le backend dans ApiService, nous obtenons plusieurs avantages :

- Réutilisabilité : toutes les requêtes API sont accessibles depuis n'importe quel composant.

- Maintenance facilitée : toute modification côté backend ne nécessite qu'une mise à jour dans le service.
- Typage strict grâce aux interfaces, réduisant les erreurs.
- Gestion uniforme des erreurs et des données, grâce à handleResponse.
- Synchronisation du panier et partage de données avec les observables.

En résumé, ce service constitue le cœur de la communication entre le front-end et le back-end de notre application Angular, tout en garantissant un code propre et structuré.

## D - Les composants

Un composant Angular est une unité réutilisable qui regroupe :

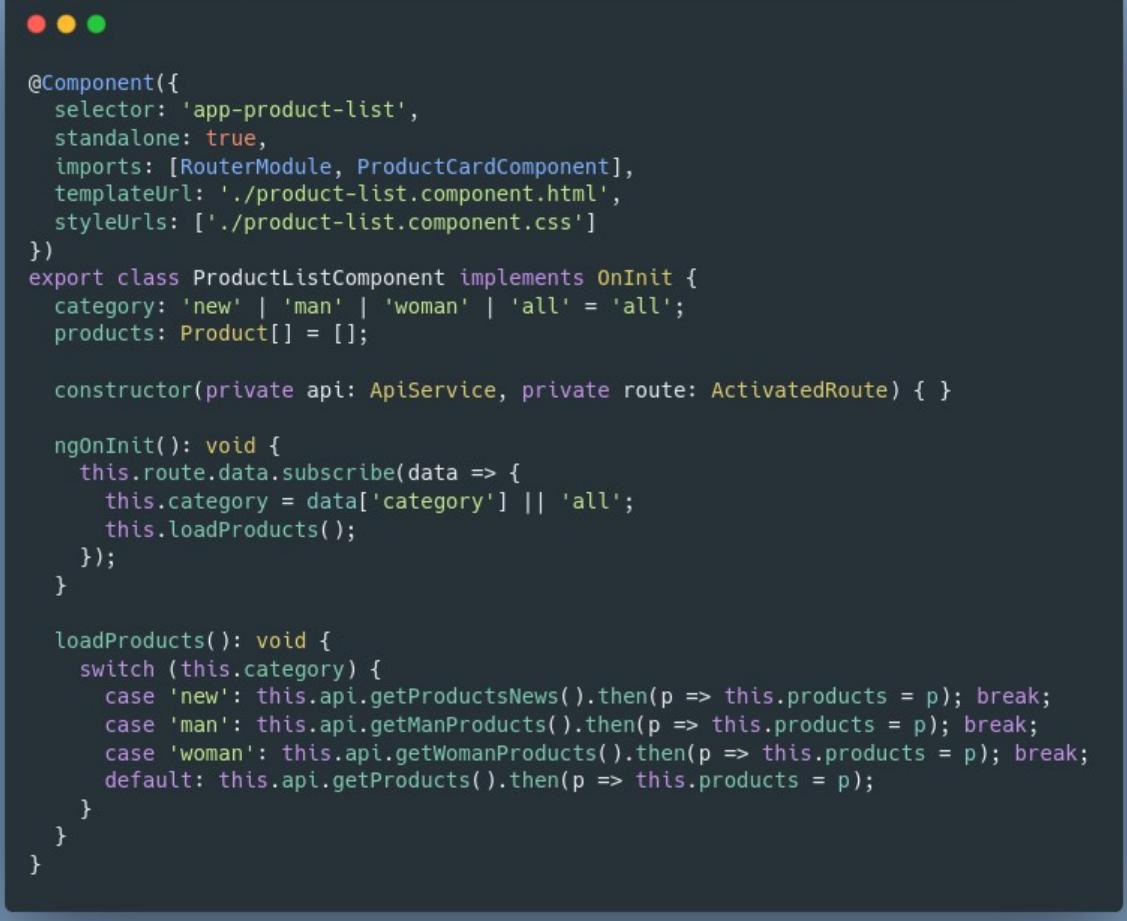
- Le template HTML (l'affichage)
- Le code TypeScript (la logique)
- Le CSS (le style)
- Optionnellement, les tests unitaires (.spec.ts)

Le composant permet d'isoler une fonctionnalité ou une partie de l'interface afin de la réutiliser facilement dans l'application.

Chaque composant Angular se compose de 4 fichiers principaux :

1. nom-composant.component.ts → la logique du composant
2. nom-composant.component.html → le template (HTML)
3. nom-composant.component.css → le style
4. nom-composant.component.spec.ts → tests unitaires

Composant ProductListComponent :



```
@Component({
  selector: 'app-product-list',
  standalone: true,
  imports: [RouterModule, ProductCardComponent],
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {
  category: 'new' | 'man' | 'woman' | 'all' = 'all';
  products: Product[] = [];

  constructor(private api: ApiService, private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.data.subscribe(data => {
      this.category = data['category'] || 'all';
      this.loadProducts();
    });
  }

  loadProducts(): void {
    switch (this.category) {
      case 'new': this.api.getProductsNews().then(p => this.products = p); break;
      case 'man': this.api.getManProducts().then(p => this.products = p); break;
      case 'woman': this.api.getWomanProducts().then(p => this.products = p); break;
      default: this.api.getProducts().then(p => this.products = p);
    }
  }
}
```

Le composant `ProductListComponent` est défini comme `standalone`, ce qui signifie qu'il n'a pas besoin d'être déclaré dans un module Angular classique (`NgModule`). Cela permet de le rendre autonome et plus facile à réutiliser ou à importer directement dans d'autres composants.

Dans la section `imports`, on précise les autres composants ou modules dont ce composant dépend. Ici, on importe le `RouterModule` pour pouvoir utiliser des directives comme `routerLink` et `ProductCardComponent` pour afficher chaque produit via une carte réutilisable.

Le constructeur `constructor(private api: ApiService, private route: ActivatedRoute)` permet d'injecter des dépendances dans le composant: `ApiService` pour communiquer avec l'API et récupérer les produits, et `ActivatedRoute` pour accéder aux données associées à la route actuelle, comme la catégorie de produits à afficher.

La méthode `ngOnInit()` fait partie du cycle de vie Angular et est exécutée une seule fois après l'initialisation du composant. Ici, elle s'abonne aux données de la route (`this.route.data.subscribe(...)`) pour récupérer la catégorie courante et appelle ensuite `loadProducts()` afin de charger la liste des produits correspondante.

Enfin, la méthode `loadProducts()` utilise un `switch` sur la catégorie pour appeler le service approprié de l'API. Selon la catégorie (new, man, woman ou all), elle met à jour la propriété `products` qui sera ensuite affichée dans le template via une boucle sur `ProductCardComponent`.

## Types de composants :

### Features

Ce sont les composants représentant les pages principales de l'application : home, produits, panier, contact...

Exemple : ProductListComponent

- Affiche la liste des produits selon une catégorie (man, woman, new, all)
- Utilise le composant ProductCardComponent pour chaque produit

HTML du composant produit :

```
<!--  
Affiche une carte produit en utilisant le composant ProductCardComponent.  
Le produit à afficher est passé en entrée via la propriété [product].  
Cette ligne est utilisée à l'intérieur d'une boucle (@for) pour afficher chaque produit de la liste.  
-->  
<div class="container">  
  @for (product of products; track product.id) {  
    <app-product-card [product]="product"></app-product-card>  
  }  
</div>
```

### Layout

Ce sont les composants partagés sur toutes les pages : header, footer, menu, modales, etc.

Exemple : HeaderComponent

- Gère le menu burger, la barre de recherche, le panier, la connexion/inscription
- Peut interagir avec d'autres composants via `@ViewChild` ou les bindings (`[isCopen]`, `(quantityChanged)`)

```
@ViewChild(CartComponent) cartComponent?: CartComponent;  
  
openCart() {  
  this.isCartOpen = !this.isCartOpen;  
  if (this.isCartOpen) setTimeout(() => this.cartComponent?.loadCart(), 0);  
}
```

## Shared

Composants réutilisables dans plusieurs pages, comme ProductCardComponent :

```
@Component({
  selector: 'app-product-card',
  standalone: true,
  imports: [RouterModule],
  templateUrl: './product-card.component.html',
  styleUrls: ['./product-card.component.css']
})
export class ProductCardComponent {
  @Input() product!: Product; // Permet de recevoir un objet Product depuis le composant parent (liaison de données descendante)
}

// @Input() indique que la propriété product peut être renseignée depuis le composant parent via un binding [product] = "...".
// Le ! : signifie à TypeScript que la propriété sera bien initialisée par Angular.
// Le type Product assure que seules des données conformes à l'interface Product peuvent être passées.
```

```
<div class="card" [routerLink]="['/product', product.id]">
  <img [src]="product.imageURL" [alt]="product.name" />
  <h3>{{ product.name }}</h3>
  <p class="price">{{ product.price }} €</p>
  @if (product.is_new) {
    <div class="badges">
      <span class="badge new">Nouveau</span>
    </div>
  }
</div>
```

## Composant central : AppComponent

Le composant racine contient la structure globale de l'application :

```
<app-header></app-header>

<main class="page-content">
  <router-outlet></router-outlet>
</main>

<app-cookie-banner></app-cookie-banner>

<app-footer></app-footer>
```

<router-outlet> affiche la page correspondant à la route active.

- Les composants header et footer sont présents sur toutes les pages.

## Points clés

- Les features contiennent la logique spécifique à chaque page.
- Les layouts gèrent l'interface commune et l'interaction avec plusieurs composants.
- Les shared sont des composants génériques, réutilisables.
- @Input() permet de passer des données depuis le parent vers le composant enfant.
- @ViewChild() permet au parent de contrôler un composant enfant.

## E - Les routes

Le fichier de routes app.routes.ts définit la navigation de notre application Angular. Chaque route associe une URL à un composant qui sera affiché lorsque l'utilisateur accède à cette URL.

- La route " correspond à la page d'accueil et affiche le HomeComponent.
- Les routes comme 'nouveautes', 'bracelet-homme' et 'bracelet-femme' utilisent toutes ProductListComponent mais avec une donnée supplémentaire data qui permet de filtrer la catégorie de produits (new, man ou woman). Cela permet de réutiliser le même composant pour plusieurs pages sans créer de composants séparés.
- Les pages d'information comme 'livraison', 'payment-condition' et 'cgv' affichent les composants DeliveryComponent, PaymentConditionComponent et CGVComponent.
- La route 'contact' ouvre le ContactComponent.
- La route 'product/:id' est une route paramétrée, où :id correspond à l'identifiant du produit. Cela permet au composant ProductByIdComponent d'afficher les détails d'un produit spécifique.
- La route 'searchbar' affiche la barre de recherche avec SearchbarComponent.
- Les routes 'mon-compte' et 'panier' permettent d'accéder au compte utilisateur et au panier via CompteComponent et CartComponent.
- Les pages de commande sont gérées par OrderSummaryComponent pour le récapitulatif, OrderDetailComponent pour les détails d'une commande spécifique (:orderId), et OrderConfirmationComponent pour afficher la confirmation ou l'annulation de paiement (success ou cancel).

Ce système de routes permet de centraliser la navigation et de rendre l'application plus modulaire et réactive, en affichant le bon composant selon l'URL visitée.

```

export const routes: Routes = [
  { path: '', component: HomeComponent }, // page d'accueil
  { path: 'nouveautes', component: ProductListComponent, data: { category: 'new' } }, // page nouveautes
  { path: 'bracelet-homme', component: ProductListComponent, data: { category: 'man' } }, // page bracelet-homme
  { path: 'bracelet-femme', component: ProductListComponent, data: { category: 'woman' } }, // page bracelet-femme
  { path: 'livraison', component: DeliveryComponent }, // page livraison
  { path: 'payment-condition', component: PaymentConditionComponent }, // page paiement condition
  { path: 'cgv', component: CGVComponent }, // page CGV
  { path: 'contact', component: ContactComponent }, // page contact
  { path: 'product/:id', component: ProductByIdComponent }, // page produit detail
  { path: 'searchbar', component: SearchbarComponent }, // barre de recherche
  { path: 'mon-compte', component: CompteComponent }, // Compte
  { path: 'panier', component: CartComponent }, // Panier
  { path: 'commande', component: OrderSummaryComponent }, // Commande recap
  { path: 'commande/:orderId', component: OrderDetailComponent }, // Commande detail
  { path: 'success', component: OrderConfirmationComponent }, // Confirmation de paiement réussi
  { path: 'cancel', component: OrderConfirmationComponent } // Annulation de paiement
];

```



## F - Responsive design

Le responsive design permet à l'application de s'adapter automatiquement à toutes les tailles d'écran, que ce soit sur un ordinateur, une tablette ou un smartphone. L'objectif est d'offrir une expérience utilisateur optimale, quel que soit l'appareil utilisé.

Pour cela, nous utilisons principalement :

- Les media queries CSS pour appliquer des styles différents selon la largeur de l'écran. Par exemple, on peut réduire la taille des images ou empiler les cartes produits sur mobile.

```

/* Tablettes */
@media (min-width: 601px) and (max-width: 1024px) {
  .menu {
    top: 8vh;
    left: -100%;
  }
}

/* Smartphones */
@media (max-width: 600px) {
}

```

- Les grilles et flexbox pour créer des mises en page flexibles qui s'ajustent automatiquement aux dimensions de l'écran.
- Les unités relatives comme em, rem ou % plutôt que des valeurs fixes en pixels, afin que les éléments puissent grandir ou rétrécir proportionnellement.

- La hiérarchie et la lisibilité : les textes, boutons et images sont adaptés pour rester lisibles et cliquables sur tous les appareils.

Grâce au responsive design, le site reste pratique, esthétique et facile à naviguer, que l'utilisateur consulte une page produit sur son téléphone ou un récapitulatif de commande sur un grand écran.

## G - Fonctionnalités principales

Le site propose plusieurs fonctionnalités principales qui permettent une navigation fluide et une expérience complète pour l'utilisateur. Ces fonctionnalités sont directement liées aux composants et routes que nous avons définis dans le projet.

### 1. Navigation et pages produit

L'utilisateur peut parcourir différentes catégories de produits : nouveautés, bracelets hommes ou femmes.

Chaque page charge dynamiquement les produits correspondant à la catégorie sélectionnée grâce au composant ProductListComponent.

### 2. Détail d'un produit

En cliquant sur un produit, l'utilisateur accède à une page de détail où il peut consulter les informations complètes.

### 3. Recherche et filtres

Le site propose une barre de recherche pour retrouver rapidement un produit. Le composant SearchbarComponent permet de filtrer les résultats en fonction des mots-clés saisis.

### 4. Panier et gestion des commandes

Les utilisateurs peuvent ajouter des produits à leur panier (CartComponent), consulter un récapitulatif de commande (OrderSummaryComponent) et suivre le détail de leurs commandes (OrderDetailComponent)

Le site intègre également un paiement sécurisé via Stripe. Lors du paiement, une session Stripe est créée côté backend et l'utilisateur est redirigé vers Stripe Checkout :

```

/**
 * Lance le paiement Stripe pour le panier.
 * Crée une session Stripe côté backend puis redirige l'utilisateur vers Stripe Checkout.
 */
async onPay() {
    // Appelle ton backend pour créer une session Stripe
    try {
        const { sessionId } = await this.api.createStripeSession(this.cart, this.user);
        // On reçoit SessionId du back et du coup redirige vers Stripe Checkout
        // Initialise Stripe.js avec la clé publique Stripe (clé commençant par pk_)
        // Stripe.js doit être chargé dans index.html pour que window.Stripe soit disponible
        const stripe = (window as any).Stripe('pk_test_51RqZlyHzhcSqHsruPjVy9VX0ITrF28BSjyX5fUJ7q2ZQTwUPBjxu9mauXN');
        // Redirige l'utilisateur vers la page de paiement Stripe Checkout avec l'id de session reçu du backend
        await stripe.redirectToCheckout({ sessionId });
    } catch (error) {
        alert('Erreur lors du paiement');
        console.error(error);
    }
}
}

```

## 5. Compte utilisateur

Chaque utilisateur peut accéder à son compte, consulter ses informations personnelles et ses commandes passées via le composant CompteComponent

## 6. Informations légales et services

Des pages d'information comme la livraison, les conditions de paiement et les CGV sont accessibles pour rassurer et informer l'utilisateur.

Le site affiche également un bannière cookie pour informer l'utilisateur sur l'usage des cookies strictement nécessaires :

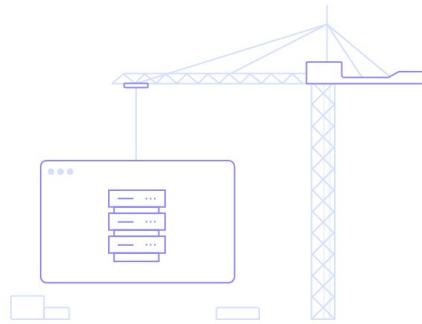
```

@if (showCookieBanner) [
<div class="cookie-banner">
    <span>
        Ce site utilise des cookies strictement nécessaires à son fonctionnement (panier, connexion, sécurité).
        <strong>Aucun cookie publicitaire ou de tracking n'est utilisé.</strong>
        En poursuivant votre navigation, vous acceptez leur utilisation.
    </span>
    <button (click)="acceptCookies()">OK</button>
</div>
]

```

# Déploiement

Pour le déploiement, j'ai commencé par acquérir un VPS chez Hostinger, ce qui m'a permis d'avoir un serveur dédié pour héberger à la fois le backend et le frontend de mon application.



## Configuration de votre VPS

Le processus prendra environ 10 minutes. Vous pouvez [quitter](#) cette page.  
Vous recevez un email lorsque votre VPS sera prêt.



J'ai utilisé Docker et Docker Compose pour simplifier la gestion des services et garantir une configuration reproductible. Dans le fichier docker-compose.yml, j'ai défini plusieurs services :

- mariadb pour la base de données, avec les variables d'environnement définies dans un fichier .env.
- backend, un service Node.js exposant le port 4004, uniquement accessible à l'intérieur du réseau Docker.
- frontend, application Angular compilée avec ng build --configuration production, ce qui génère le dossier dist/. Celui-ci est ensuite servi par Nginx, qui joue deux rôles :

Servir efficacement les fichiers statiques (cache headers, compression).

Gérer le fallback vers index.html pour le routage côté Single Page Application.

Sa configuration est simplifiée : il ne gère plus l'API, puisque Caddy intercepte déjà les requêtes /api.

- caddy, le serveur web principal, qui s'occupe de gérer les requêtes HTTP/HTTPS et de faire le reverse proxy vers le frontend et le backend selon les chemins.

```

👉 docker-compose.yml > ...
1   version: '3.9'
    ▷Run All Services
2   services:
        ▷ Run Service
3     mariadb:
4       image: mariadb:11
5       container_name: mariadb
6       restart: unless-stopped
7       env_file:
8         - ./.env
9       environment:
10      MySQL_ROOT_PASSWORD: ${DB_ROOT_PASSWORD}
11      MySQL_DATABASE: ${DB_NAME}
12      MySQL_USER: ${DB_USER}
13      MySQL_PASSWORD: ${DB_PASSWORD}
14     volumes:
15       - mariadb_data:/var/lib/mysql
16     ports:
17       - "${DB_PORT}:${DB_PORT}"
18
    ▷ Run Service
19   backend:
20     build: ./backend
21     container_name: backend
22     restart: unless-stopped
23     env_file:
24       - ./.env

```

```

# Étape 1 : Build Angular
FROM node:18 AS build

WORKDIR /app

# Copie package.json et package-lock.json
COPY package*.json .

# Installe les dépendances
RUN npm install

# Copie le reste du code
COPY . .

# Build Angular en production
RUN npm run build -- --configuration production
# Le -- transmet --configuration production à ng build

# Étape 2 : Serveur Nginx pour l'Angular build
FROM nginx:stable-alpine

```

```
frontend > ⚙ nginx.conf
1   server {
2     listen 80;
3     server_name _;
4
5     root /usr/share/nginx/html;
6     index index.html;
7
8     location / {
9       try_files $uri /index.html;
10    }
11
12  }
13 }
```

Le fichier Caddyfile m'a permis de configurer Caddy pour :

- Servir le frontend sur toutes les requêtes sauf celles commençant par /api/.
- Rediriger les requêtes /api/\* vers le backend.
- Gérer automatiquement les certificats SSL grâce à Let's Encrypt, en sécurisant ainsi toutes les communications via HTTPS.

#### ≡ Caddyfile

```
1
2 # Adresse du site
3 wrapvintage.duckdns.org {
4   encode gzip
5   log
6
7   # Redirige toutes les requêtes /api/* vers le backend
8   handle_path /api/* {
9     reverse_proxy backend:4004
10  }
11
12  # Tout le reste → frontend
13  handle {
14    reverse_proxy frontend:80
15  }
16 }
17 }
```

Pour que mon serveur soit accessible depuis Internet avec un domaine dynamique, j'ai utilisé DuckDNS. Cela m'a permis d'associer un nom de domaine (wrapvintage.duckdns.org) à l'adresse IP publique de mon VPS. Grâce à DuckDNS, même si l'IP de mon VPS change, le domaine reste toujours valide.

The screenshot shows the Duck DNS web interface. At the top, there's a navigation bar with links for 'spec', 'about', 'why', 'install', 'faqs', and 'logout'. On the right, it says 'logged in with quoc-nam.ngo@laplateforme.io'. The main header is 'Duck DNS' with a yellow rubber duck icon to its left. Below the header, account information is displayed: 'account' (quoc-nam.ngo@laplateforme.io), 'type' (free), and 'token' (a long string of characters). It also shows 'token generated' (1 day ago) and 'created date' (22 Aug 2025, 17:38:26). The main content area is titled 'domains' (1/5). It has a search bar with 'http:// sub domain .duckdns.org add domain'. A table lists one domain: 'wrapvintage' with 'current ip' (redacted), 'ipv6' (redacted), and 'changed' (1 day ago). Buttons for 'update ip' and 'update ipv6' are shown next to the respective columns. A red 'delete domain' button is at the bottom right of the table row.

Enfin, après avoir démarré tous les services avec docker-compose up -d, j'ai pu vérifier que :

- L'API Node.js répond correctement aux requêtes via HTTPS (/api/\*).
- Le frontend Angular est accessible via le domaine principal.
- La communication est sécurisée avec des certificats SSL valides.

Cette configuration me permet d'avoir un déploiement complet, sécurisé et maintenable, tout en séparant clairement le frontend, le backend et la base de données.

Voici le lien du site :

[WrapVintage](https://wrapvintage.duckdns.org)

# Conclusion

Pour conclure, ce projet m'a permis de prendre pleinement conscience de l'importance de l'autonomie et de la rigueur dans le développement web. Travailler sur un projet complet, alliant front et back, m'a rassuré sur mes compétences acquises durant la formation et m'a donné confiance dans ma capacité à mettre en pratique mes connaissances. Cette expérience m'a également appris la patience nécessaire pour résoudre des problèmes techniques et l'importance de documenter et structurer son code. Enfin, je tiens à remercier mes formateurs et toute l'équipe pédagogique pour leur accompagnement, leur soutien et leurs conseils précieux tout au long de ce projet.