Table des matières

[Domaines principaux de la plateforme 3](#_Toc206595284)

[Technologies 8](#_Toc206595285)

[Architecture Microservices modulaires + Headless + Event-driven 9](#_Toc206595286)

[Modèle de conception : DDD (Domain-Driven Design) 9](#_Toc206595287)

[Modèle de communication : CQRS (Command Query Responsibility Segregation) 9](#_Toc206595288)

[Architecture technique : Microservices + Headless 10](#_Toc206595289)

[Communication inter-services : Event-Driven + Message Bus 10](#_Toc206595290)

[Sécurité & Authentification 10](#_Toc206595291)

[Intégration ERP/CRM 11](#_Toc206595292)

[Architecture globale de la solution 12](#_Toc206595293)

[Diagramme des dépendances 13](#_Toc206595294)

[Structure de la solution : 14](#_Toc206595295)

[Solutions à prendre comme exemple 15](#_Toc206595296)

[Architecture technique 15](#_Toc206595297)

[Fonctionnalités e-commerce essentielles 15](#_Toc206595298)

[Annexe 1 : Définition des fonctionnalités clés pour une application e-commerce moderne. 17](#_Toc206595299)

[Annexe 2 : Quelques types de plateformes e-commerce 19](#_Toc206595300)

[Annexe 3 : Zoho, Odoo, Xero, Sage, Akaunting, Netsuite, Dynamic et Central 21](#_Toc206595301)

[Annexe 4 : Vision d’une plateforme e-commerce type Shopify 23](#_Toc206595302)

[Annexe 5 : Solution Front-end 25](#_Toc206595303)

[Vue.js, React et Angular (Google) 25](#_Toc206595304)

[Autres options émergentes ou spécialisées 26](#_Toc206595305)

[Pour une plateforme e-commerce, que choisir ? 26](#_Toc206595306)

[Foire Aux Questions 27](#_Toc206595307)

[Pourquoi autant de couches ? 27](#_Toc206595308)

[Pourquoi ne pas tout mettre dans un seul projet ? 28](#_Toc206595309)

[Pourquoi la couche d’orchestration s’appelle “Application” ? 29](#_Toc206595310)

[Que peut-on faire si on veut enlever la couche Application ou doit-on déplacer les règles qui sont dedans ? 31](#_Toc206595311)

[Pourquoi ne pas relocalise la couche Application ailleurs ? 32](#_Toc206595312)

[Comment la perte de séparation stricte entre les couches est utile en microservices ? 34](#_Toc206595313)

[Est-ce que c’est d’une bonne idée d'utiliser CQRS ? 36](#_Toc206595314)

[Dans quelle situation peut-on opter pour le modèle CQRS ? 38](#_Toc206595315)

[C'est quoi DTOs ? 39](#_Toc206595316)

[C'est quoi les Transactions ACID ? 39](#_Toc206595317)

[C'est quoi la différence entre les handlers et les services ? 40](#_Toc206595318)

[A partir de quelle taille de l'équipe peut-on penser à une architecture Microservice ? 41](#_Toc206595319)

[C'est quoi la différence entre une logique métier et une logique technique ? 42](#_Toc206595320)

# Domaines principaux de la plateforme

Catalogue Produit

**Entités :** Produit, Catégorie, Variation

**Comportements :**

* Gestion des produits, attributs, images, descriptions, etc.
* Variantes (taille, couleur, etc.)
* Catégories et sous-catégories
* Prix, promotions, stock disponible.
* SEO produit, tags, filtres et recherche avancée.

Panier d’achat

**Entités :** Panier, LignePanier

**Comportements :**

* Ajout/suppression de produits.
* Mise à jour des quantités
* Calcul automatique des totaux, taxes, remises.
* Persistance du panier (Session ou localStorage).

Inventaire (Gestion des stocks)

**Entités :** Stock, MouvementStock

**Comportements :**

* Suivi des stocks par entrepôt.
* Alertes de stock faible : Réservations, seuils d’alerte.
* Mouvements de stock, réapprovisionnement.
* Mise à jour automatique après achat

Commande

**Entités :** Commande, LigneCommande, AdresseLivraison

**Comportements :**

* Création de commande à partir du panier (après paiement).
* Statuts (en attente, payé, expédié, livré, annulé).
* Historique, annulation, retour, suivi de livraison.

Paiement

**Entités :** Transaction, Abonnement

**Comportements :**

* Intégration avec Stripe, PayPal, ou autre passerelle.
* Paiement sécurisé (HTTPS, tokenisation).
* Gestion des statuts de paiement.
* Remboursements, paiements partiels.
* Gestion des erreurs et des validations.

Livraison

**Entités :** A venir

**Comportements :**

* Méthodes de livraison, tarifs, délais.
* Suivi des colis, intégration transporteurs.
* Points relais, adresses multiples.

Authentification et comptes clients

**Entités :** A venir

**Comportements :**

* Inscription / Connexion (JWT + cookies sécurisés)
* Historique des commandes
* Gestion des adresses de livraison

Notifications

**Entités :** Email, SMS, Webhook

**Comportements :**

* Confirmation
* Suivi de commande
* Promotions
* Alertes

Client (Utilisateur) :

**Entités :** Client, Compte, Adresse, Historique, Admin, Rôles, …

**Comportements :**

* Profil client, adresses, préférences.
* Historique d’achat, fidélité.
* Authentification, rôles, permissions, profil.

Marketing :

**Entités :** A venir

**Comportements :**

* Coupons, promotions, ventes flash.
* Emailing, relance panier abandonné.
* Programmes de fidélité, parrainage.

Rapports & Statistiques :

**Comportements :**

* Ventes, produits populaires, marges.
* Comportement client, taux de conversion.
* Export comptable, analytique.

Facturation & Comptabilité :

**Entités :** A venir

**Comportements :**

* Génération de factures, TVA.
* Intégration ERP (Odoo, Zoho, Dynamics).
* Gestion des devises, taxes locales.

Recommandation & IA :

**Entités :** A venir

**Comportements :**

* Produits similaires, cross-sell, upsell.
* Moteur de recherche intelligent.
* Personnalisation de l’expérience.

Dashboard admin :

**Entités :** Admin, Statistiques, Rapport

**Comportements :**

* Gestion des produits, commandes, utilisateurs
* Statistiques de ventes
* Export CSV / PDF

Administration :

**Entités :** A venir

**Comportements :**

* Gestion des utilisateurs internes.
* Paramétrage des modules.
* Logs, sécurité, audit.

Boutique :

**Entités :** Boutique, Thème, Paramètres

**Comportements :**

* Création
* Personnalisation

Domaines transversaux :

**Entités :** A venir

**Comportements :**

* **Notification** : emails, SMS, push.
* **Sécurité** : authentification, autorisation, audit.
* **Fichiers & médias** : gestion des images, documents.
* **Intégration** : ERP, CRM, transporteurs, marketplaces.

# Technologies

**Frontend**

* **A venir**

**Backend**

* **ASP.NET Core 8** (Web API)
* **CQRS + MediatR**
* **Entity Framework Core** (multi-tenancy avec schéma par boutique)
* **JWT + cookies sécurisés**
* **Stripe SDK / PayPal SDK**

**Base de données**

* **PostgreSQL** ou **SQL Server**
* Multi-tenancy : schéma par boutique ou colonne TenantId

**DevOps / Infra**

* **Docker** (conteneurs par service)
* **Kubernetes** (optionnel pour scalabilité)
* **GitHub Actions** (CI/CD)
* **Azure / AWS / GCP** (hébergement)

# Architecture Microservices modulaires + Headless + Event-driven

## Modèle de conception : DDD (Domain-Driven Design)

* **Pourquoi ?** : Permet de structurer ton code autour des **domaines métier** (Produits, Commandes, Clients, Paiements, etc.).
* **Avantages** :
  + Code plus clair et maintenable.
  + Séparation des responsabilités.
  + Facilite l’évolution vers des microservices.

Exemple :  
/src

/Product

/Domain

/Application

/Infrastructure

/API

/Order

/Customer

## Modèle de communication : CQRS (Command Query Responsibility Segregation)

* **Pourquoi ?** : Séparer les opérations de lecture et d’écriture pour optimiser performance et scalabilité.
* **Avantages** :
  + Lecture rapide via projections.
  + Écriture sécurisée et validée.
  + Compatible avec Event Sourcing.

Exemple :

* Commandes : CreateOrderCommand, UpdateProductStockCommand
* Requêtes : GetProductByIdQuery, ListOrdersByCustomerQuery

## Architecture technique : Microservices + Headless

* **Pourquoi ?** : Permet de déployer chaque domaine indépendamment, avec une API centralisée.
* **Avantages** :
  + Scalabilité horizontale.
  + Flexibilité pour le front-end (Vue.js, React, etc.).
  + Intégration facile avec ERP ou CRM.

Exemple :

* Microservices : ProductService, OrderService, CustomerService
* API Gateway : pour centraliser les appels
* Front-end : Vue.js ou React en mode headless

## Communication inter-services : Event-Driven + Message Bus

* **Pourquoi ?** : Permet une communication asynchrone entre services.
* **Avantages** :
  + Découplage fort.
  + Traçabilité des événements.
  + Résilience.

Exemple :

* OrderPlacedEvent → déclenche UpdateInventoryCommand
* Utilise RabbitMQ, Kafka ou Azure Service Bus

## Sécurité & Authentification

* **JWT + Cookies sécurisés** (comme on l’a déjà fait)
* **OAuth2 / OpenID Connect** pour les intégrations tierces
* **RBAC (Role-Based Access Control)** pour les permissions

## Intégration ERP/CRM

Il faut prévoir des **connecteurs API** vers :

* + Odoo (open source)
  + Zoho (SaaS)
  + Dynamics 365 (entreprise)

## Architecture globale de la solution

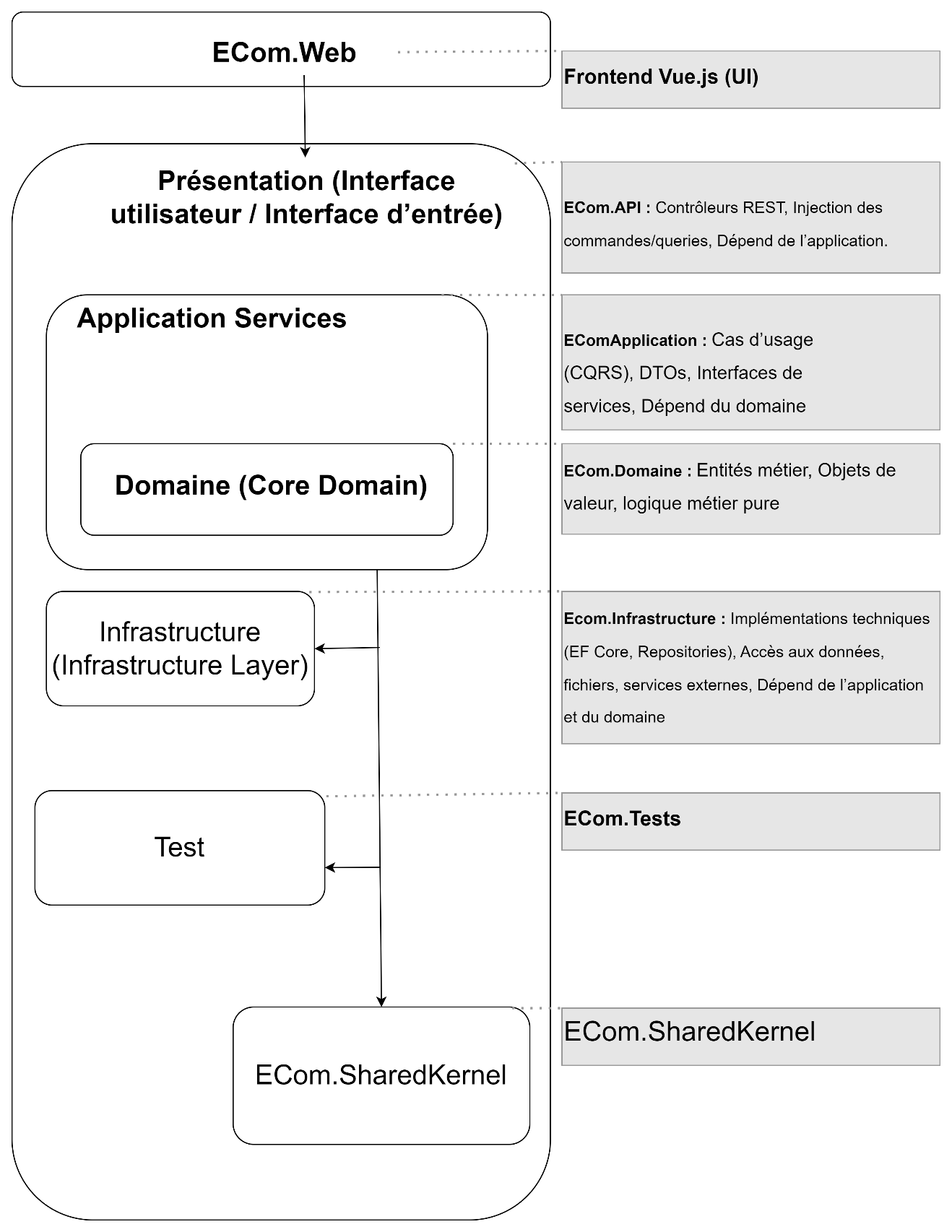


Figure 1 : Architecture

## Diagramme des dépendances

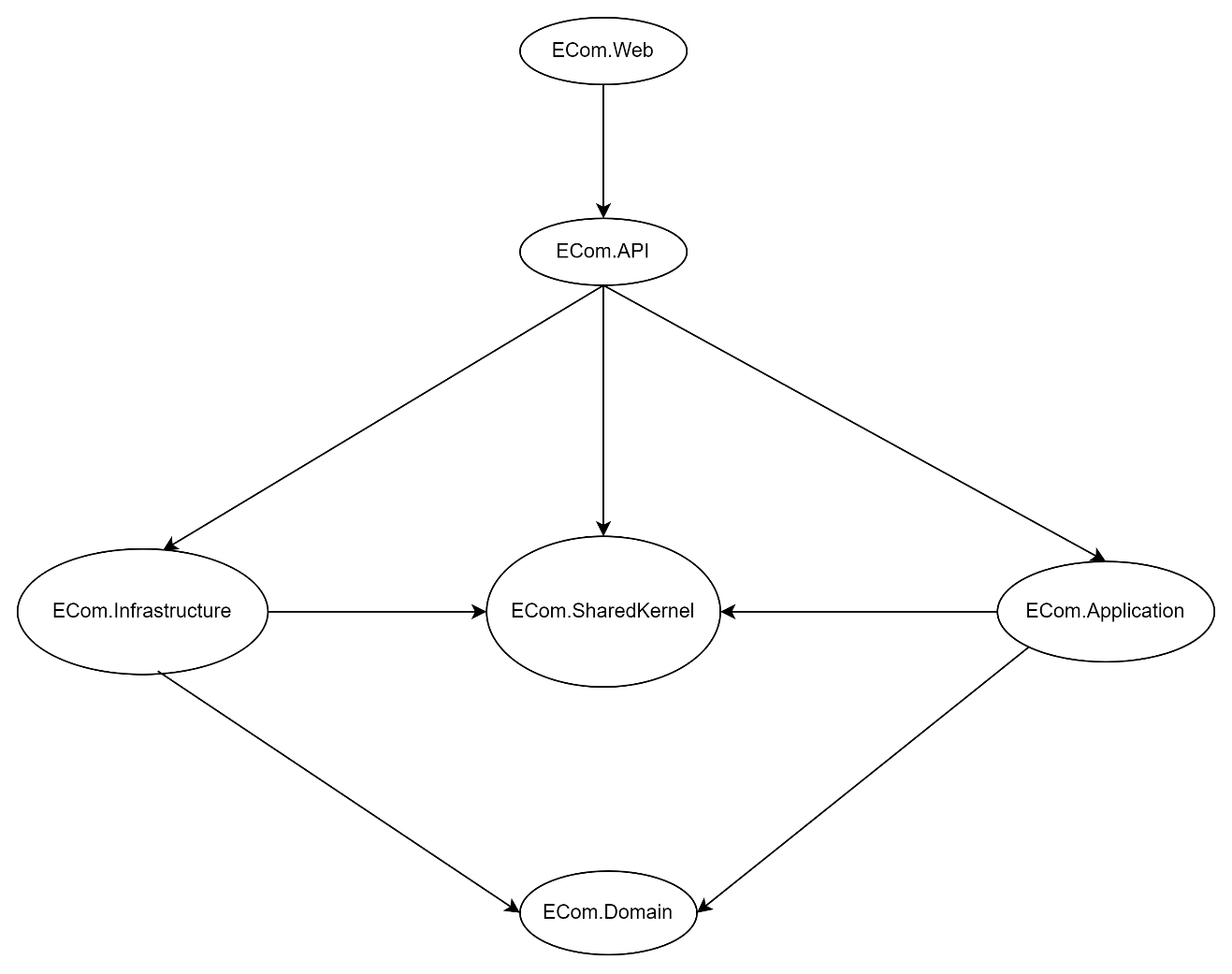


Figure 2 : Diagramme de dépendance

## Structure de la solution :

ECom/

├── ECom.API/ # Projet API ASP.NET Core ()

│ ├── ProductCatalog/ # Le domaine catalogue produit

│ │ └── Controllers/ # Contrôleurs REST

│ └── Program.cs # Point d'entrée de l'application

│

├── ECom.Application/ # Pour séparer la logique d'application de l'API

et du domaine.

│ └── ProductCatalog /

│ ├── Commands/ # Commandes CQRS

│ ├── DTOs/ # Objets de transfert de données

│ ├── Handlers/ # handlers CQRS

│ ├── Interfaces/

│ ├── Mapping/

│ ├── Queries/ # Requêtes CQRS

│ └── Validators/

│

├── ECom.Domain/ # Projet de domaine (DDD)

│ ├── Common/

│ └── ProductCatalog /

│ ├── Entities/ # Entités du domaine

│ ├── Events/ # Événements du domaine

│ ├── Exceptions/ # Gestion des exceptions

│ ├── Interfaces/ # Interfaces pour les services

│ ├── Services/

│ └── ValueObjects/ # Objets de valeur

│

├── ECom.Infrastructure/ # Projet d'infrastructure

│ ├── ProductCatalog /

│ │ ├── Configurations/ # Configurations de la base de données (Mapping Fluent API)

│ │ ├── Data/ # Accès aux données: DbContext, configuration EF Core

│ │ ├── Interfaces/

│ │ ├── Migrations/ # Migrations EF Core

│ │ ├── Persistence/

│ │ └── Repositories/ # Référentiels

│

├── ECom.SharedKernel/ # Pour les objets partagés

(Ex.: StatutPeriode, TacheDto)

├── DTOs/ # Data Transfer Object: Utilisé pour transporter les

données entre les différentes couches.

├── Enums/

├── Exceptions/

└── Interfaces/

# Solutions à prendre comme exemple

## Architecture technique

* **Shopify / BigCommerce** : architecture SaaS modulaire, API REST/GraphQL bien documentées, approche headless possible.
* **Magento / Odoo** : architecture monolithique ou microservices, très personnalisable, mais plus complexe à maintenir.
* **Microsoft Dynamics / NetSuite** : architecture orientée services, intégration native avec CRM, ERP, etc.

**Inspiration** :

* Utiliser une **architecture modulaire avec microservices**.
* Implémente une **API REST ou GraphQL** pour le front-end.
* Pense à une **approche headless** pour séparer le front-end du back-end.

## Fonctionnalités e-commerce essentielles

|  |  |
| --- | --- |
| **Fonction** | **Inspiration** |
| Catalogue produits | Shopify, WooCommerce, Magento |
| Panier & paiement | BigCommerce, Stripe, PayPal |
| Gestion des stocks | Odoo, PrestaShop, Dynamics |
| Multi-langue / multi-devise | Magento, Shopify |
| SEO & marketing | Shopify, Wix, Zoho |
| CRM client | Zoho CRM, Odoo, Dynamics |
| Intégration ERP | Odoo, NetSuite, Dynamics 365 |

**Back-office & ERP**

* **Odoo** : très bon modèle modulaire, chaque app est un domaine métier.
* **Zoho** : intégration fluide entre CRM, inventaire, comptabilité.
* **Dynamics 365** : logique métier bien définie, extensible via Power Platform.

**Inspiration** :

* Structure ton projet en **domaines métier** (ex. Produits, Commandes, Clients, Paiements).
* Utilise **CQRS + DDD** comme on l’a déjà fait pour SIE-Core.
* Prévois une **intégration future avec un ERP** (Odoo ou Dynamics).

**UX/UI et front-end**

* **Shopify & Wix** : simplicité, design responsive, personnalisation via thèmes.
* **Magento** : plus complexe mais très flexible.
* **Squarespace** : design épuré, bon pour les créatifs.

**Inspiration** :

* Utiliser **une solution** rapide et réactive (Vois **Solutions front-end**).
* Prévois une **interface admin** intuitive pour les commerçants.

**Intelligence & automatisation**

* **Zoho & Dynamics** : automatisation des workflows, IA pour recommandations.
* **Shopify** : apps tierces pour upsell, email marketing, etc.

**Inspiration** :

* Intègre des **règles métier automatisées** (ex. relance panier abandonné).
* Prévois une **API pour plugins tiers** (comme Shopify App Store).

# Annexe 1 : Définition des fonctionnalités clés pour une application e-commerce moderne.

**Catalogue Produits**

* **Ajout / modification / suppression de produits**
* **Catégories et sous-catégories**
* **Images, descriptions, prix, stock**
* **Variantes** (taille, couleur, etc.)
* **Filtres et recherche avancée**

**Panier d’achat**

* Ajouter / retirer des produits
* Mise à jour des quantités
* Calcul automatique des totaux
* Sauvegarde du panier (localStorage ou session)

**Paiement**

* Intégration avec Stripe, PayPal, ou autre passerelle
* Paiement sécurisé (HTTPS, tokenisation)
* Gestion des erreurs et des validations

**Gestion des stocks**

* Suivi des quantités disponibles
* Mise à jour automatique après achat
* Alertes de stock faible

**Authentification et comptes clients**

* Inscription / Connexion (JWT + cookies sécurisés)
* Historique des commandes
* Gestion des adresses de livraison

**Commandes**

* Création de commande après paiement
* Statuts : En attente, Expédiée, Livrée, Annulée
* Suivi de livraison (optionnel)

**Notifications**

* Email de confirmation
* Suivi de commande
* Promotions

**Dashboard admin**

* Gestion des produits, commandes, utilisateurs
* Statistiques de ventes
* Export CSV / PDF

# Annexe 2 : Quelques types de plateformes e-commerce

1. **SaaS (Software as a Service)** : hébergement inclus, gestion simplifiée, abonnement mensuel.
   * Exemples : **Shopify**, **BigCommerce**, **Wix**, **Squarespace**, **Salesforce Commerce Cloud**, **WiziShop**, **Jimdo**.
2. **Open Source** : hébergement séparé, personnalisation poussée, maintenance à gérer soi-même.
   * Exemples : **WooCommerce**, **PrestaShop**, **Magento (Adobe Commerce)**, **OpenCart**, **Sylius**.
3. **Headless Commerce** : séparation du front-end et du back-end, très flexible, souvent utilisé par les grandes marques.
   * Exemples : **Magento avec CMS**, **Shopify avec Hydrogen**, **BigCommerce headless**.

**Comparatif des plateformes populaires**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Plateforme** | **Type** | **Avantages** | **Inconvénients** | **Prix** |
| **Shopify** | SaaS | Facile à utiliser, rapide à déployer, bon support, vente multicanale | Frais de transaction, personnalisation limitée sans apps | Dès 25 €/mois |
| **WooCommerce** | Open Source | Gratuit, très personnalisable, SEO puissant | Maintenance technique, pas de support centralisé | Gratuit + extensions payantes |
| **BigCommerce** | SaaS | Pas de frais de transaction, très scalable, bon SEO | Coût élevé, courbe d’apprentissage | Dès 39,95 €/mois |
| **PrestaShop** | Open Source | Très flexible, bon pour les grandes boutiques | Complexité technique, hébergement à gérer | Gratuit + modules payants |
| **Magento (Adobe Commerce)** | Open Source / Headless | Ultra puissant, adapté aux grandes entreprises | Très complexe, coûteux à maintenir | $$$ (souvent entreprise) |
| **Wix eCommerce** | SaaS | Drag & drop, simple pour débutants | Limité pour les grandes boutiques, SEO basique | Dès 20 €/mois |
| **Squarespace** | SaaS | Design élégant, facile à utiliser | Personnalisation limitée, pas adapté aux grandes boutiques | Dès 16 €/mois |
| **Ecwid** | SaaS | Intégration facile à un site existant | Moins complet qu’un CMS dédié | Gratuit à partir, puis abonnements |
| **Salesforce Commerce Cloud** | SaaS / Headless | Solution entreprise, intégration CRM | Très coûteux, complexe | $$$ (niveau entreprise) |
| **OpenCart** | Open Source | Léger, simple à installer | Moins de communauté, moins d’extensions | Gratuit |

**Comment choisir ?**

* **Débutant ou PME** → Shopify, Wix, Squarespace, WooCommerce (si WordPress déjà utilisé)
* **Entreprise avec équipe technique** → PrestaShop, Magento, BigCommerce
* **Projet sur mesure ou headless** → Magento + CMS, BigCommerce headless, Shopify Hydrogen
* **Budget serré** → WooCommerce, OpenCart, Ecwid (freemium)

# Annexe 3 : Zoho, Odoo, Xero, Sage, Akaunting, Netsuite, Dynamic et Central

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Solution** | **Type** | **Points forts** | **Limites** | **Idéal pour** |
| **Odoo** | ERP open source | Très modulaire, personnalisable, riche en fonctionnalités (CRM, RH, stock, compta, e-commerce) | Demande des compétences techniques, configuration initiale complexe | PME/ETI avec équipe technique  1 |
| **Zoho** | Suite SaaS | Interface intuitive, intégration facile, bon rapport qualité/prix, Zoho One très complet | Moins personnalisable, structure plus rigide | Startups, petites entreprises  1 |
| **Xero** | Comptabilité SaaS | Très bon pour la comptabilité, automatisation bancaire, multi-devises, API ouverte | Pas un ERP complet, peu adapté à la gestion d’inventaire complexe | Freelances, petites entreprises |
| **Sage** | ERP/Compta | Solution mature, forte en comptabilité, gestion RH, paie, inventaire | Interface vieillissante, coût élevé, moins agile | PME traditionnelles, secteurs réglementés |
| **Akaunting** | Comptabilité open source | Gratuit, hébergé ou auto-hébergé, simple à utiliser | Fonctionnalités limitées, peu d’intégrations natives | Freelances, micro-entreprises |
| **NetSuite (Oracle)** | ERP cloud | Très puissant, complet, adapté aux grandes entreprises, e-commerce natif | Coût élevé, complexité de mise en œuvre | Grandes entreprises, multinationales |
| **Microsoft Dynamics 365 Business Central** | ERP cloud | Intégration avec Microsoft 365, très complet, bon support, extensible | Coût et complexité, dépendance à l’écosystème Microsoft | PME/ETI avec besoins avancés |

**Pour un projet e-commerce, que choisir ?**

* **Petite structure ou startup** : **Zoho** ou **Odoo (hébergé)** sont les plus adaptés. Zoho est plus rapide à mettre en place, Odoo plus personnalisable.
* **Entreprise en croissance** : **Odoo**, **Microsoft Dynamics 365**, ou **Sage** selon ton environnement technique.
* **Grandes entreprises ou projets complexes** : **NetSuite** ou **Dynamics 365** sont les plus robustes.
* **Si on veut juste gérer la comptabilité** : **Xero** ou **Akaunting** suffisent, mais ne couvrent pas les besoins ERP.

# Annexe 4 : Vision d’une plateforme e-commerce type Shopify

**Fonctionnalités principales**

**1. Gestion des boutiques**

* Création de boutique personnalisée
* Nom de domaine personnalisé
* Thèmes et templates

**2. Gestion des produits**

* CRUD produits, variantes, catégories
* Import/export CSV
* Stock et inventaire

**3. Gestion des commandes**

* Suivi des commandes
* Statuts : payé, expédié, livré, annulé
* Facturation

**4. Paiement**

* Intégration Stripe, PayPal, etc.
* Gestion des abonnements (pour ta plateforme)
* Paiement des clients finaux

**5. Multi-tenancy (multi-boutiques)**

* Chaque boutique a ses propres données
* Isolation des bases ou schéma partagé

**6. Back-office pour les marchands**

* Dashboard de ventes
* Gestion des clients
* Statistiques

**7. Front-office pour les clients**

* Page d’accueil, catalogue, panier, paiement
* Responsive design

**8. Marketplace de plugins/thèmes (optionnel)**

* Extensions pour SEO, marketing, etc.
* Personnalisation avancée

**Architecture recommandée**

**🔹 Microservices + Multi-tenancy**

* **Services** : Auth, Produits, Commandes, Paiement, Boutique, Frontend
* **Base de données** : PostgreSQL ou SQL Server avec schéma par tenant
* **API Gateway** : pour router les requêtes
* **Event Bus** : RabbitMQ ou Azure Service Bus
* **Frontend** : Vue.js + Tailwind + Vite
* **Backend** : ASP.NET Core 8 avec DDD + CQRS

**Domaines DDD pour ta plateforme**

|  |  |  |
| --- | --- | --- |
| **Domaine** | **Entités** | **Services** |
| **Boutique** | Boutique, Thème, Paramètres | Création, personnalisation |
| **Produit** | Produit, Variante, Catégorie | CRUD, import/export |
| **Commande** | Commande, LigneCommande | Paiement, suivi |
| **Client** | Client, Adresse | Authentification, historique |
| **Paiement** | Transaction, Abonnement | Stripe, PayPal |
| **Utilisateur** | Marchand, Admin | Gestion des rôles |
| **Statistiques** | Rapport, Vue | Dashboard, KPIs |

# Annexe 5 : Solution Front-end

## Vue.js, React et Angular (Google)

**Vue.js**

* **Avantages** : Léger, facile à apprendre, syntaxe claire, intégration rapide dans des projets existants.
* **Inconvénients** : Moins soutenu par les grandes entreprises, écosystème plus jeune.
* **Idéal pour** : Petites à moyennes applications, MVP, projets avec délais courts.

**React (Meta/Facebook)**

* **Avantages** :
  + Très populaire et bien soutenu.
  + Architecture basée sur les composants.
  + DOM virtuel performant.
  + Écosystème riche (Next.js, Redux, React Query, etc.).
* **Inconvénients** :
  + Nécessite souvent des bibliothèques tierces pour des fonctionnalités de base.
  + Courbe d’apprentissage modérée.
* **Idéal pour** : Applications complexes, headless e-commerce, projets avec forte interactivité.

**Angular (Google)**

* **Avantages** :
  + Framework complet avec tout intégré (routing, formulaires, services, etc.).
  + Utilise TypeScript pour plus de robustesse.
  + Architecture MVC bien structurée.
* **Inconvénients** :
  + Courbe d’apprentissage plus raide.
  + Moins flexible pour les petits projets.
* **Idéal pour** : Applications d’entreprise, projets à grande échelle avec équipe expérimentée.

## Autres options émergentes ou spécialisées

|  |  |  |
| --- | --- | --- |
| **Framework** | **Description** | **Usage recommandé** |
| **Svelte** | Compile en JavaScript natif, très rapide, syntaxe simple | Projets légers, haute performance |
| **SolidJS** | Inspiré de React mais plus performant | Applications interactives |
| **Qwik** | Conçu pour le chargement ultra-rapide (partiel) | Sites e-commerce ultra-optimisés |
| **Alpine.js** | Minimaliste, pour des interactions simples | Petits composants, sans framework complet |
| **Lit** | Web Components natifs | Intégration dans des projets multi-frameworks |

## Pour une plateforme e-commerce, que choisir ?

* **Vue.js** : parfait pour un MVP ou une plateforme modulaire.
* **React + Next.js** : excellent pour un site headless, SEO-friendly, scalable.
* **Angular** : adapté si on veut une solution tout-en-un avec une équipe TypeScript.
* **Svelte / Qwik** : à considérer si on veut innover sur la performance et le temps de chargement.

# Foire Aux Questions

## Pourquoi autant de couches ?

Séparation des responsabilités (SRP)

Chaque couche a une responsabilité unique, ce qui rend le code :

* Plus lisible
* Plus testable
* Plus maintenable
* Plus évolutif

|  |  |
| --- | --- |
| Couche | Rôle |
| **API** | Expose les endpoints HTTP, reçoit les requêtes, retourne les réponses |
| **Application** | Orchestration métier, logique d’usage, CQRS, DTOs, validation. |
| **Domain** | Cœur métier pur, entités, règles métier, services de domaine. |
| **Infrastructure** | Accès aux données, EF Core, implémentations concrètes. |
| **SharedKernel** | Objets partagés entre les couches ou domaines. |
| **Web (Vue.js)** | Interface utilisateur |

## Pourquoi ne pas tout mettre dans un seul projet ?

C’est possible pour un petit projet, mais cela devient vite un **cauchemar** à maintenir dès que :

* Le projet grandit
* Plusieurs développeurs travaillent dessus
* Si on veut tester ou réutiliser des parties du code
* Si on veut migrer vers des microservices
* L’API peut changer (REST, GraphQL, gRPC…) mais la logique d’usage reste.
* On peut réutiliser la couche Application dans un autre contexte (ex : CLI, batch, tests).
* On peut tester les cas d’usage sans dépendre du transport HTTP.

**Avantages de cette architecture**

1. **Testabilité** : On peut tester la logique métier sans dépendre de la base de données ou de l’API.
2. **Évolutivité** : On peut ajouter de nouveaux domaines (ex. : Congés, Paie) sans casser l’existant.
3. **Flexibilité** : On peut changer la base de données ou le frontend sans toucher au domaine.
4. **Préparation aux microservices** : chaque domaine peut devenir un service indépendant.

**Analogie simple**

Imaginant une entreprise :

|  |  |
| --- | --- |
| Rôle | Couche équivalente |
| Réceptionniste | API (reçoit les demandes) |
| Chef de projet | Application (oriente les demandes) |
| Expert métier | Domain (prend les décisions métier) |
| Archiviste | Infrastructure (stocke les données) |
| Fournisseur commun | SharedKernel (outils partagés) |

## Pourquoi la couche d’orchestration s’appelle “Application” ?

Il faut distinguer deux types de logique :

|  |  |  |
| --- | --- | --- |
| **Type de logique** | **Description** | **Où elle devrait être** |
| **Logique métier pure** | Règles métier immuables, indépendantes de l’infrastructure (ex : « une commande ne peut être expédiée que si elle est payée »). | Domaine |
| **Logique d’orchestration** | Coordination des opérations, appels aux services, validation, mapping, etc. | Application |

Donc, **la logique métier est bien dans le domaine**, mais elle est **appelée** depuis la couche Application (le handler qui sert pour orchestrer).

**Pourquoi cette séparation est utile ?**

* **Domain** : contient des règles métier **pures**, testables sans base de données ni dépendances.
* **Application** : agit comme un **chef d’orchestre**, enchaîne les appels, gère les erreurs, les DTOs, les validations.

Cela permet de :

* Tester le domaine indépendamment
* Réutiliser la logique métier dans d’autres contextes (ex : batch, microservice, etc.)
* Garder le code clair et modulaire

**Pourquoi on l'Appelle "Application" ? Puisque la couche API (Application Programming Interface).**

Dans l’architecture **Clean Architecture** ou **DDD**, le mot **“Application”** ne désigne pas l’interface utilisateur ni l’API, mais plutôt la **logique d’usage** de l’application.

**Elle contient :**

* Les **cas d’usage** (use cases)
* Les **commandes et requêtes** (CQRS)
* Les **handlers**
* Les **DTOs**
* Les **validations**
* Les **interfaces de services**

**Son rôle :**

Orchestrer les actions métier sans implémenter les règles métier elles-mêmes.

**Et l’API alors ?**

Le mot **API** signifie **Application Programming Interface**, mais ici, il désigne une **interface technique** pour exposer l’application à l’extérieur (souvent via HTTP).

|  |  |
| --- | --- |
| Couche | Rôle |
| Api | Interface d’entrée (HTTP, REST, etc.) |
| Appliocation | Orchestration des cas d’usage |
| Domain | Règles métier pures |

**Analogie simple**

Imaginant une entreprise :

|  |  |
| --- | --- |
| Rôle | Couche |
| Réceptionniste (reçoit les demandes) | API |
| Chef de projet (oriente les demandes) | Application |
| Expert métier (prend les décisions métier) | Comain |

En résumé,

* Le mot **“Application”** dans **Clean Architecture** ≠ “Application” dans “API”.
* La **couche Application** est le **cerveau opérationnel** de ton système.
* La **couche API** est juste une **porte d’entrée**.

## Que peut-on faire si on veut enlever la couche Application ou doit-on déplacer les règles qui sont dedans ?

Dans la couche application, on a les dossiers suivants :

* Les **Commandes** et **Requêtes** (CQRS)
* Les **Handlers** (logique d’orchestration)
* Les **DTOs**
* Les **Validators**
* Les **Mappers**
* Les **Interfaces de services**

On pourrait faire la relocalisation comme suit :

|  |  |  |
| --- | --- | --- |
| Élément | Destination | Pourquoi |
| **Handlers** | Dans les **contrôleurs API** ou dans des **services d’orchestration** dans Infrastructure | Si on veut simplifier, on peut appeler directement les méthodes du domaine depuis les contrôleurs |
| **DTOs** | Dans un dossier Models ou ViewModels dans API ou SharedKernel | Pour garder une séparation entre les entités et les objets exposés |
| **Validators** | Dans API/Validators ou directement dans les contrôleurs avec [ApiController] + DataAnnotations | Pour simplifier la validation côté API |
| **Mappers** | Dans une classe statique dans API ou Infrastructure | Ou utiliser AutoMapper si on veut automatiser |
| **Interfaces de services** | À fusionner avec les interfaces du domaine ou les supprimer si inutiles | Si elles ne sont pas utilisées ailleurs |

En résumé, supprimer la couche Application :

* **Simplifie** la structure pour un petit projet
* Mais **couple** fortement l’API au domaine et à l’infrastructure
* Rend plus difficile la **migration vers des microservices** ou la **réutilisation** de la logique métier ailleurs

**Cependant, elle est acceptable dans les cas suivants :**

* Pour un **prototype rapide**
* Pour une **application simple** avec peu de logique métier
* Si on est **seul développeur** et que on veut aller vite

## Pourquoi ne pas relocalise la couche Application ailleurs ?

Au lieu de supprimer la couche « Application », on va juste la relocalise ailleurs. Comme ça on aurait le même nombre de couche, mais juste moins de projets ?

Actuellement on a :

* ECom.API
* ECom.Application
* ECom.Domain
* ECom.Infrastructure
* ECom.SharedKernel

Cela donne **5 projets backend**. On pourrait les **regrouper en 1 ou 2 projets** tout en gardant les **couches logiques**.

Avantages :

* **Moins de projets à gérer** dans Visual Studio
* **Compilation plus rapide**
* **Navigation plus fluide**
* Toujours **modulaire** grâce aux dossiers

Cependant :

* On perd un peu de **séparation stricte** entre les couches (utile en microservices).
* Il faut bien **nommer et organiser les dossiers** pour garder la lisibilité.
* On doit éviter les **références croisées** entre les sous-dossiers (ex : Domain ne doit pas dépendre d’Infrastructure).

En résumé, on peut tout à fait :

* **Garder la couche Application** (CQRS, orchestrations)
* **La déplacer dans le même projet que l’API ou le Backend**
* **Réduire le nombre de projets** sans sacrifier la qualité architecturale

Cependant, avec la fusion des couches offre les propriétés suivantes :

* Simple à gérer.
* Moins de projets.
* Mais elle on augmente le risque de couplage.
* On obtient moins de clarté.

En revanche, avec la séparation stricte est :

* Modulaire et évolutive,
* Prépare aux Microservices
* Contrôle mieux les dépendances
* Offre un meilleur maintenabilité

## Comment la perte de séparation stricte entre les couches est utile en microservices ?

Dans une architecture bien découpée, chaque couche ou projet :

* A ses **propres responsabilités**
* Est **indépendant** des autres (dans la mesure du possible)
* Peut être **testé**, **déployé**, ou **remplacé** séparément

En fusionnant les couches, on crée certaines complexités :

**Couplage technique**

* Les couches **Domain**, **Application**, **Infrastructure** sont dans le même projet.
* Cela rend plus difficile de **contrôler les dépendances** : par exemple, le domaine pourrait accidentellement utiliser EF Core ou des DTOs.

**Difficulté de migration vers microservices**

* Si on veut extraire FeuilleDeTemps en microservice, on devrat **re-séparer** les couches, afin que chaque projet pourrait être déployé indépendamment, Versionné séparément et Testé en isolation.
* On perd la **modularité** qui permet de transformer chaque domaine en service indépendant.

**Moins de clarté pour les nouveaux développeurs**

* Dans un projet unique, il est plus difficile de **voir la structure logique**.
* Les responsabilités sont **moins explicites** (ex : où mettre une règle métier ? un mapping ? une validation ?).

Cependant, si on respecte les **principes de séparation logique**, on garde malgré tout :

* Utiliser des **dossiers bien nommés** (Domain/, Application/, Infrastructure/)
* Respecter les **dépendances directionnelles** :
  + Domain ne dépend de rien
  + Application dépend de Domain
  + Infrastructure dépend de la couche « Application » et la couche « Domain ».
* Utiliser des **interfaces** pour découpler les implémentations

**Nommage des couches**

**Tu pourrais renommer ECom.Application en :**

* ECom.UseCases : si on veut insister sur les scénarios métier
* ECom.Orchestration : si on veut souligner le rôle de coordination
* ECom.Services : plus générique, mais parfois ambigu
* ECom.AppLogic : clair et simple

Le nom **Domain** est très bien choisi car il reflète :

* Les **entités métier**
* Les **règles métier**
* Les **services de domaine**
* Les **exceptions métier**
* ECom.Core : utilisé dans certains projets, mais moins explicite
* ECom.Model : à éviter, car souvent confondu avec les modèles de données
* ECom.DomainModel : plus explicite, mais un peu verbeux

La couche **Infrastructure** contient :

* Les **implémentations techniques** (EF Core, Repositories, DbContext)
* Les **accès aux données**
* Les **services externes**
* ECom.DataAccess : si elle ne contient que l’accès aux données
* ECom.Persistence : si on veut insister sur la persistance
* ECom.Technical : plus large, mais moins conventionnel

## Est-ce que c’est d’une bonne idée d'utiliser CQRS ?

CQRS est une bonne idée si…

1. **Notre domaine métier est complexe**
   * On a des règles métier comme la soumission, l’approbation, le refus de périodes.
   * On veut bien séparer les **commandes** (actions) des **requêtes** (consultations).
2. On veut évoluer vers des microservices
   * CQRS facilite la séparation des responsabilités et des bases de données par service.
3. **On veut optimiser les performances de lecture**
   * On pourrait avoir une base optimisée pour les lectures (ex : vues matérialisées, projections).
4. **On veut une architecture évolutive et testable**
   * Les Handlers sont faciles à tester.
   * On veutajouter des événements, des validations, des workflows.

Cependant, CQRS peut être trop lourd si…

1. **Notre application est simple**
   * Si on n’a que quelques entités et peu de logique métier, CQRS peut être trop verbeux.
2. **On est juste un pu deux développeurs et on veut aller vite**
   * CQRS demande plus de code, plus de structure, plus de discipline.
3. **On n’a pas besoin de scalabilité extrême**
   * CQRS est souvent utilisé dans des systèmes à forte charge ou à haute disponibilité.

En revanche on a :

* Une logique métier bien définie
* Des entités avec des états et des transitions
* Une vision vers les microservices
* Une séparation déjà en place entre commandes et requêtes
* On veut maintenir une architecture propre
* On veut ajouter des domaines (Congés, Paie, etc.)
* On veut préparer une scalabilité future

Donc, CQRS, s’adapte mieux à notre cas, mais :

* **Il faut simplifier là où c’est possible** (ex : pas besoin de projection complexe pour les lectures simples)
* **Il faut éviter l’excès de verbosité** (ne faut pas créer un handler pour chaque petite action triviale)
* **Il faut bien documente** pour que d’autres développeurs comprennent la structure

## Dans quelle situation peut-on opter pour le modèle CQRS ?

Domaines complexes avec logique métier riche : Si notre application a des règles métier complexes (comme les statuts de période, validation par superviseur, etc.), CQRS permet de mieux organiser cette logique :

* On peut centraliser la logique métier dans les handlers de commande.
* On peut valider les règles métier dans les services de domaine (PeriodeDomainService) sans polluer la logique de lecture.
* On évite les effets de bord dans les lectures.

Scalabilité et performance :

* On peut optimiser les lectures et écritures séparément.
* Par exemple, les lectures peuvent être mises en cache ou distribuées, tandis que les écritures peuvent être traitées de manière transactionnelle :<
* Les lectures peuvent être optimisées :
  + Mise en cache (Redis, MemoryCache).
  + Réplication de base de données pour lecture seule.
  + Projection vers des modèles simplifiés (DTOs, vues matérialisées).
* Les écritures sont traitées de manière stricte :
  + Validation métier.
  + Transactions ACID.
  + Sauvegarde dans une base centrale.

Évolutivité vers les microservices : CQRS facilite la transition vers des microservices, car chaque service peut gérer ses propres commandes et requêtes.

Séparation des responsabilités : Cela améliore la maintenabilité du code, car chaque handler a une seule responsabilité.

Audit et traçabilité : En combinant CQRS avec Event Sourcing, On peut garder une trace de toutes les modifications du système:

* CQRS facilite l’audit si on ajoute Event Sourcing :
  + Chaque commande génère un événement.
  + On stocke ces événements dans un journal.
  + On peut rejouer l’historique pour reconstruire l’état.
* Avantages :
  + On sait qui a fait quoi et quand.
  + On peut restaurer l’état à un moment donné.
  + On peut analyser les comportements (ex : combien de commande a été annulées par mois).

## C'est quoi DTOs ?

Un DTO (Data Transfer Object) est un objet simple utilisé pour transporter des données entre différentes couches d’une application, sans exposer la logique métier ou les entités du domaine.

Objectifs :

* Séparer les modèles de domaine des modèles exposés à l’extérieur (API, UI).
* Contrôler les données exposées (ex : ne pas envoyer des champs sensibles).
* Optimiser les performances (en envoyant uniquement les données nécessaires).
* Faciliter la sérialisation/désérialisation (ex : JSON dans une API REST).

## C'est quoi les Transactions ACID ?

A (Atomicité) : Toutes les opérations d’une transaction réussissent ou échouent ensemble. Il n’y a pas d’état intermédiaire.

C (Cohérence) : La base reste dans un état valide avant et après la transaction, en respectant toutes les règles métier et contraintes.

I (Isolation) : Les transactions concurrentes ne se perturbent pas entre elles. Chaque transaction est comme si elle était seule.

D (Durabilité) : Une fois la transaction validée (commit), les données sont sauvegardées de façon permanente, même en cas de panne.

Supposons que on veut payer une commande :

* Modifier le statut de la commande.
* Enregistrer la modification dans la base.
* Envoyer une notification au client.

Si une étape échoue (ex : la sauvegarde), aucune modification ne doit être appliquée. C’est l’atomicité.

## C'est quoi la différence entre les handlers et les services ?

Les handlers sont des composants qui :

* Reçoivent une commande ou une requête via MediatR.
* Orchestrent l’exécution de cette commande ou requête.
* Appellent les services de domaine ou d’infrastructure si nécessaire.
* Retourne un résultat (souvent un DTO).

Les services encapsulent la logique métier ou la logique technique :

* Services de domaine : contiennent des règles métier complexes qui ne peuvent pas être placées dans une entité.
* Services d’application : orchestrent plusieurs opérations métier ou techniques.
* Services d’infrastructure : accèdent à des ressources externes (base de données, fichiers, API).

## A partir de quelle taille de l'équipe peut-on penser à une architecture Microservice ?

Le passage à une architecture microservices ne dépend pas uniquement de la taille de l’équipe, mais aussi de la complexité du projet, des besoins métier, et des objectifs techniques. Cela dit, la taille de l’équipe joue un rôle important dans la capacité à gérer cette complexité.

1 à 3 développeurs : Monolithique modulaire

4 à 8 développeurs : Microservices possibles, mais attention à la surcharge

8+ développeurs : Microservices fortement recommandés

Microservices demandent :

* Des compétences en DevOps (CI/CD, conteneurs, orchestration).
* Une gestion des communications inter-services (API Gateway, messaging).
* Une observabilité (logs, monitoring, tracing).
* Une coordination entre les équipes (versioning, contrats d’API).

Si l’équipe est trop petite :

* On risque de passer plus de temps à gérer l’infrastructure qu’à développer des fonctionnalités.
* La complexité peut ralentir le développement au lieu de l’accélérer.

## C'est quoi la différence entre une logique métier et une logique technique ?

Logique technique : C’est la logique liée à la technologie, à l’infrastructure ou aux outils utilisés pour faire fonctionner l’application.

Exemples :

* Connexion à la base de données (EF Core, SQL Server).
* Sérialisation JSON.
* Authentification JWT.
* Envoi de courriels ou de notifications.
* Mise en cache, logging, etc.

Logique métier : C’est la logique qui reflète les règles et processus du domaine fonctionnel de ton application.

Exemples :

* Une période ne peut être soumise que si elle est en brouillon.
* Un superviseur peut approuver une période uniquement si elle est soumise.
* Une ligne de période doit avoir une date de fin après la date de début.
* Un employé ne peut pas soumettre deux périodes qui se chevauchent.