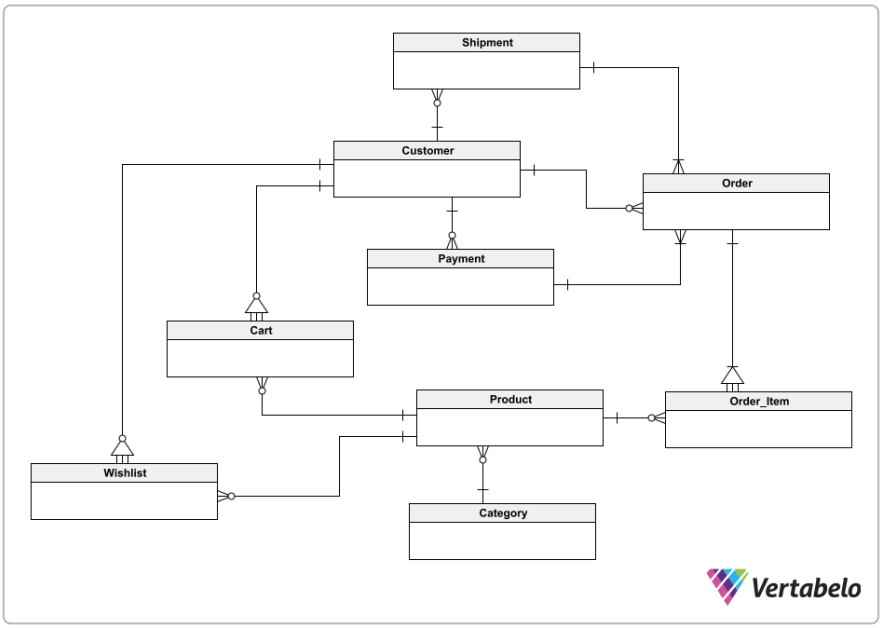
ChatGPT

Documentation Technique V0 – Marketplace pièces auto/moto (Tunisie)

1. Modèle de données



Le modèle conceptuel de la base de données inclut toutes les entités principales d'un marketplace e-commerce multivendeur. On y trouve notamment : **Utilisateur** (Customer) pour les clients inscrits, **Vendeur** (Vendor) avec ses informations (profil, société, statut), **Produit** (Product) appartenant à un vendeur et à une catégorie, **Catégorie** (Category) à structure hiérarchique, **Commande** (Order) comprenant plusieurs **Lignes de commande** (OrderItem), **Paiement** (Payment) lié à la commande, **Livraison** (Shipment) pour le suivi, ainsi que **Panier** (Cart) et **Liste de souhaits** (Wishlist) liés à l'utilisateur [1] [2]. On ajoute aussi les entités **Retour** (Return) pour la gestion des réclamations, **SupportClient** (ticket d'assistance), et **Log** (journal d'activité). Toutes les tables ont une clé primaire (souvent un `id` auto-incrémenté) et des clés étrangères pour représenter les relations (par exemple, `product.vendor_id → Vendor(id)`, `order.user_id → User(id)`, etc.). Les contraintes incluent l'unicité (par ex. email unique pour les utilisateurs), le non-null sur les champs obligatoires, et le respect des intégrités référentielles (ON DELETE CASCADE pour les entités dépendantes, etc.). La base doit être normalisée au minimum en **3e forme normale** pour garantir cohérence et performance transactionnelle [3].

- **Utilisateur (User)** : id (PK), nom, prénom, email (unique), mot de passe haché, rôle (`buyer`, `vendor`, `admin`), numéro de téléphone, date de création, etc. (Un utilisateur peut avoir plusieurs adresses, gérées dans une table **Address** liée).
- **Vendeur (Vendor)** : id (PK), user_id (FK → User.id), nom de l'entreprise, numéro fiscal, statut (validé, en attente), coordonnées bancaires, logo, date de création, etc.

- **Catégorie (Category)** : id (PK), nom, parent_id (FK vers Category.id, nullable) pour arbre de catégories.
- **Produit (Product)** : id (PK), vendor_id (FK → Vendor.id), category_id (FK → Category.id), nom, description, prix, quantité en stock, attributs (couleur, taille, etc.), images (liens), date de création, etc.
- **Commande (Order)** : id (PK), user_id (FK → User.id) acheteur, statut (`new`, `confirmed`, `shipped`, `delivered`, `cancelled`, etc.), montant total, adresse de livraison, mode de paiement, date de commande, date de mise à jour, etc.
- **OrderItem (Ligne de commande)** : id (PK), order_id (FK → Order.id), product_id (FK → Product.id), quantité, prix unitaire au moment de la commande, total ligne, etc.
- **Paiement (Payment)** : id (PK), order_id (FK → Order.id), montant payé, méthode (`credit_card`, `paypal`, `cash`...), statut (`pending`, `paid`, `failed`, `refunded`), référence transaction, date du paiement.
- **Livraison (Shipment)** : id (PK), order_id (FK → Order.id), transporteur, suivi (tracking number), statut de livraison, date d'envoi, date de livraison estimée.
- **Retour (Return)** : id (PK), order_item_id (FK → OrderItem.id) ou order_id si retour partiel/total, motif (`defectueux`, `erreur`, `non_livré`...), statut (`requested`, `approved`, `rejected`, `refunded`), date de demande, date de résolution, montant remboursé.
- **SupportClient (Ticket)** : id (PK), user_id (FK → User.id), sujet, message, statut (`open`, `pending`, `closed`), assigné à (user admin), date de création, date de clôture, etc.
- **Log** : id (PK), timestamp, user_id (FK → User.id, nullable), action (texte), détails (champ JSON ou texte libre), adresse IP, etc.

Les relations sont **un-à-plusieurs** : un utilisateur peut passer plusieurs commandes [4] ; une commande contient plusieurs `OrderItem` ; chaque commande a un paiement et une livraison (1-1) mais un paiement/expédition peut couvrir plusieurs commandes dans un système multi-commande [4] . Un produit appartient à une catégorie (1-1), et chaque catégorie peut avoir plusieurs sous-catégories. Les paniers et wishlists sont liés à un utilisateur et peuvent contenir plusieurs produits (relation M:N via tables intermédiaires CartItem/WishlistItem) [2] . Les clés étrangères assurent l'intégrité des dépendances. Ce schéma conceptuel est illustré ci-dessus : chaque entité correspond à une table SQL, et les liaisons entre entités sont matérialisées par des FK (diagramme ERD). On s'assurera de normaliser jusqu'à la 3NF pour les besoins transactionnels de la marketplace [3] .

2. Architecture UI/UX

L'interface utilisateur couvre trois profils principaux (acheteur, vendeur, admin) et doit être responsive (desktop + mobile). Les pages principales comprennent :

- **Accueil** : bannière promotionnelle, barre de recherche globale (moteur avec filtres avancés), sections mises en avant (nouvelles arrivées, promotions, catégories phares).
- **Catalogue/Recherche** : liste ou grille de produits filtrable (par catégorie, marque, prix, etc.) et triable (prix, nouveautés, popularité). Composants : barre de filtres latérale, barre de tri, pagination ou chargement infini.
- **Page Produit** : galerie d'images du produit, nom, description détaillée, spécifications techniques, options (variantes, quantités), prix, bouton "Ajouter au panier", avis clients. Afficher le profil du vendeur avec ses évaluations.

- **Panier (Cart)** : liste des articles sélectionnés, modification des quantités, total, résumé des frais (livraison/taxes), bouton “Passer à la caisse”.
- **Paiement/Checkout** : formulaire de saisie/confirmation des adresses, choix du mode de paiement, récapitulatif final, bouton de validation. Inclure étapes claires et sécurisées (exemple de fil d’Ariane).
- **Profil Utilisateur** : gestion du compte (nom, email, mot de passe, adresses), historique des commandes (statuts, tracking), liste de souhaits, configuration notifications. Onglet pour **Support** (créer un ticket d’aide, voir statut).
- **Espace Vendeur** : dashboard interne avec accès sécurisé. Pages clés : tableau de bord (chiffre d’affaires, nombre de commandes en cours), gestion des produits (création/édition/archivage), gestion des commandes reçues (liste des commandes, mise à jour de statut, impression de bordereaux), gestion du stock, gestion de la boutique (horaires, politique de retour, banque). Composants : formulaires de saisie de produit (texte, image), listes de produits éditables, filtres d’état de commande.
- **Espace Admin** : contrôle global du site. Pages pour gérer les utilisateurs (clients/vendeurs), modérer les produits, configurer les catégories du catalogue, consulter les logs d’activité. Actions d’admin sur statut des vendeurs (activer/bannir), traitement manuel de litiges.
- **Pages Supplémentaires** : Connexion/Inscription, Profil, FAQ/Aide, mentions légales et contact.

Les wireframes (desktop et mobile) doivent illustrer les parcours : un acheteur navigue du catalogue au panier au checkout, tandis qu’un vendeur fait un post de nouveau produit puis suit les commandes. On veillera à une UI claire et intuitive (par ex. menus burger sur mobile), avec appels à l’action visibles (boutons d’ajout au panier, validation). L’UX doit minimiser le nombre de clics et guider l’utilisateur pas-à-pas. Le design utilise des composants réutilisables (carte produit, barre de recherche, menu de navigation) pour cohérence sur toutes les pages.

3. Backend Django

On adopte une architecture **monolithique modulaire** en Django, avec plusieurs *apps* dédiées. Par exemple : `accounts` (gestion des utilisateurs, profils, authentification), `vendors` (profil vendeur), `catalogue` (produits, catégories), `orders` (panier, commande, orderitems), `payments`, `shipping`, `returns`, `support` (tickets), `logs` (journalisation), etc. Chaque app contient ses *models.py* correspondants aux entités ci-dessus, des *serializers.py* (DRF) pour traduire modèles ↔ JSON, et des *viewsets.py* ou *views.py* (ex: `ModelViewSet`) pour l’API REST.

DRF facilite la configuration des routes via les **ViewSet** et **Router** : au lieu d’écrire manuellement chaque URL, on crée des `ViewSet` (CRUD type) et on les enregistre auprès d’un `DefaultRouter`, qui génère automatiquement les endpoints REST correspondants [5]. Par exemple, `router.register(r'products', ProductViewSet)` crée les routes `/api/products/`, `/api/products/{id}/` (avec méthodes GET/POST/PUT/DELETE selon les actions). Chaque `ViewSet` référence son `queryset` et son `serializer_class`. On définira également des routes personnalisées au besoin via le décorateur `@action` de DRF pour des opérations particulières (ex. confirmer une commande).

Permissions & Authentification : On utilise **JWT** pour l’auth (via *Django REST Framework SimpleJWT*). SimpleJWT est un backend d’authentification Django REST qui prend en charge les scénarios courants de JWT [6]. On configure dans `settings.py` :

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    ...
}
```

On impose `IsAuthenticated` par défaut (toutes les API nécessitent un token valide, sauf celles explicitement en *AllowAny* comme l'enregistrement/utilisateur ou le login), conformément aux préconisations OWASP [7]. Les tokens JWT (accès + rafraîchissement) sont générés lors du login, par exemple via le point d'API `/api/token/`. FlutterFlow récupère alors ce token pour les appels suivants (voir section Front). Les **middlewares Django** utiles incluent `corsheaders.middleware.CorsMiddleware` (pour autoriser le frontend), `AuthenticationMiddleware`, et potentiellement un middleware de logging (ou Sentry) pour audit.

Throttle & Pagination : Afin de limiter la charge, on peut configurer des throttles DRF : p.ex. `UserRateThrottle` pour limiter à 60 requêtes/minuté par utilisateur, 1000/jour, etc. (« toll » permet de réguler le trafic d'API [8]). Dans `settings.py` on mettra des classes de throttle dans `DEFAULT_THROTTLE_CLASSES` (ex. `'rest_framework.throttling.UserRateThrottle'`) et leurs `THROTTLE_RATES`. La pagination est également activée globalement (p.ex. pagination Limit/Offset) via `DEFAULT_PAGINATION_CLASS = 'rest_framework.pagination.LimitOffsetPagination'` et `PAGE_SIZE = 50` pour couper les gros jeux de données en pages de 50 items [9]. Ainsi la réponse JSON inclura des champs de pagination (page, count, liens prev/next) ou un format `'results': [...]`.

Filtrage et Requêtage : On utilise `django-filter` avec DRF : ajouter `rest_framework.filters.SearchFilter` et `DjangoFilterBackend` pour permettre des requêtes par paramètres. Par exemple, sur la vue produit on peut définir `filter_backends = [DjangoFilterBackend, SearchFilter]` et `filterset_fields = ['category', 'in_stock', 'vendor']` pour filtrer par catégorie ou stock, et un champ de recherche sur le nom. Cela génère automatiquement des filtres GET (ex: `/api/products/?category=3&search=frein` pour chercher "frein" en catégorie 3) [10].

Endpoints clés et views : On expose des routes RESTful telles que `/api/users/`, `/api/vendors/`, `/api/products/`, `/api/categories/`, `/api/orders/`, `/api/orders/{id}/items/`, `/api/payments/`, `/api/shipments/`, `/api/returns/`, `/api/support/tickets/`, etc. Les points de terminaison pour les applications vendeurs/admin sont protégés par des permissions spécifiques (par ex. un vendeur ne peut mettre à jour que ses propres produits, défini via `permissions.IsAuthenticated & IsVendorOwner`). Les vues utilisent les mixins DRF (`ListModelMixin`, `RetrieveModelMixin`, etc.) ou des `ModelViewSet`. On appliquera `check_object_permissions(request,obj)` dans `get_object()` pour vérifier l'accès (OWASP A1) [11].

Logique des statuts : On définit des statuts d'énumérations (choices) pour les commandes (`'new'`, `'confirmed'`, `'shipped'`, `'delivered'`, `'cancelled'`), paiements (`'pending'`, `'success'`, `'refunded'`), livraisons (`'in_transit'`, `'delivered'`, etc.), retours (`'requested'`, `'approved'`, etc.), tickets (`'open'`, `'closed'`). La logique métier dans le backend gère la transition d'état (ex: après paiement réussi, commande passe à `confirmed` puis `shipped`). Les serializers vérifient aussi l'intégrité des données (ne pas exposer plus de champs que nécessaire pour éviter les **excessive data exposure** [12]).

4. Infrastructure

L'architecture de déploiement utilise des conteneurs Docker pour scalabilité. Exemples :

- **Dockerfile** de base (Django) :

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . .
ENV DJANGO_SETTINGS_MODULE=project.settings.prod
RUN python manage.py collectstatic --noinput
CMD ["gunicorn", "project.wsgi:application", "--bind", "0.0.0.0:8000"]
```

- **docker-compose.yml** minimal :

```
version: '3.8'
services:
  web:
    build: .
    ports: - "8000:8000"
    env_file: .env
    depends_on:
      - db
      - redis
    volumes:
      - static_volume:/app/static
  db:
    image: postgres:14
    env_file: .env
    volumes:
      - pgdata:/var/lib/postgresql/data
  redis:
    image: redis:7
  celery:
    build: .
    command: celery -A project worker -l info
```

```
    depends_on:
      - db
      - redis
  nginx:
    image: nginx:stable
    ports: - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - web
volumes:
  pgdata:
  static_volume:
```

- **Nginx** en front sert de reverse proxy devant Gunicorn, gérant le TLS et les fichiers statiques/media (collectés dans le volume `static_volume`) [13]. Le fichier `nginx.conf` redirige `/static/` et `/media/` vers le volume partagé, et passe le reste vers `web:8000`.
- **Base de données** : PostgreSQL conteneurisé, avec volume persistant et des backups réguliers (script ou service).
- **Cache & Files** : Redis est utilisé pour Celery (brokering) et pour le caching (optionnel). Les fichiers statiques/médias sont idéalement stockés sur un stockage objet S3-compatible (AWS S3, MinIO, etc.) via Django-Storages [14]. Cela décharge le serveur et permet de servir les ressources médias via un CDN ou par Nginx.
- **CI/CD** : On préconise une chaîne d'intégration continue (GitHub Actions/GitLab CI) pour *build/test/linter* et déploiement automatique sur AWS (ECS, EKS ou EC2+Docker) ou autre cloud. Le pipeline peut effectuer : tests unitaires, build docker, push image vers registry, puis déploiement (ex. `docker-compose pull && docker-compose up -d`).
- **Monitoring/Sécurité** : Utiliser un outil de monitoring (Datadog, Prometheus, CloudWatch) pour superviser CPU, RAM, latence DB. Logging centralisé (ELK, Papertrail ou CloudWatch Logs) pour agréger les logs applicatifs (ex. avec `django-logging` ou Sentry pour les erreurs). Assurer les meilleures pratiques de sécurité : exiger HTTPS (TLS), désactiver `DEBUG=False` en prod, des secrets (SECRET_KEY, DB_PASS, JWT_SECRET) stockés dans des variables d'environnement ou un vault (jamais dans le code) [15]. Configurer des règles de firewall pour exposer uniquement les ports nécessaires (80/443). Appliquer des mises à jour régulières (OS, dépendances) pour éviter les vulnérabilités connues [15].

5. FlutterFlow / Frontend Web

Le frontend est développé avec **FlutterFlow**, générant un site Flutter Web responsive. L'architecture FlutterFlow comprend : des *Data Models* pour les objets REST (Product, Category, Order, User, Vendor, etc.) ; des Pages (correspondant aux écrans listés ci-dessus) ; et des Widgets (Text, Images, ListView, Forms, Boutons) pour la structure visuelle. Chaque page FlutterFlow intègre des *API Calls* configurées pour communiquer avec le backend Django REST. Par exemple : un appel GET `/api/products/` remplit le widget de liste de produits; un POST `/api/orders/` crée une commande au checkout. FlutterFlow permet de paramétrer les en-têtes HTTP : on ajoute un header `Authorization: Bearer [auth_token]` pour envoyer dynamiquement le JWT obtenu lors du login [16].

Pour l'authentification, on utilise le mode **Custom Auth** (JWT). L'utilisateur saisit ses identifiants dans un formulaire de login : FlutterFlow appelle une API dédiée (ex. `/api/token/`), récupère le token JWT, puis le stocke dans une variable d'état persistée. Ce token est ensuite automatiquement ajouté aux appels ultérieurs via `Authorization: Bearer [auth_token]` [17] [16]. Les pages de l'app vérifient si l'utilisateur est connecté (présence du token) pour contrôler l'accès (ex. empêcher un acheteur d'accéder à l'espace vendeur).

La conception UI/UX fait un usage intensif du mode *Responsive* de FlutterFlow : on adapte la disposition (widgets `Row` / `Column`, `Flex`) selon la taille de l'écran (mobile vs desktop). Des templates de blocs sont réutilisés (header avec barre de recherche, card produit, footer). Les interactions et mises à jour d'état (ex. ajouter au panier modifie la somme totale) sont gérées via les *Actions* de FlutterFlow (Update Document, Navigate, etc.). En résumé, FlutterFlow sert de constructeur visuel pour assembler les écrans, tout en permettant l'intégration fine de la logique API REST et de gestion de l'état JWT, sans écrire manuellement tout le code Flutter.

6. Connecteurs ERP & Fichiers plats

La marketplace doit se synchroniser avec différents ERP (Odoo, Sage, « Mosaique Auto », etc.) pour le catalogue produits et le traitement des commandes. **Schéma de connexion** : chaque ERP dispose soit d'une API SOAP/REST (ex. API JSON-RPC d'Odoo), soit d'une interface d'import/export. On développe des connecteurs (typiquement des tâches backend Django) qui effectuent :

- **Authentification API** : connexion sécurisée à l'ERP (token OAuth, API key, etc.).
- **Synchronisation Catalogue** : récupération périodique des données produit (références, stock, prix, nouveautés) depuis l'ERP via API, puis mise à jour de la base locale (ou insertion de nouveaux produits/vendeurs). Par exemple, interroger Odoo pour synchroniser les catégories et produits du stock.
- **Synchronisation Commandes** : transmission des commandes validées du marketplace vers l'ERP pour traitement (création de devis/factures dans Sage ou Odoo).
- **Gestion des fichiers CSV/Excel** : certains ERP acceptent l'import de fichiers plats. On met en place un module Django qui surveille un dépôt (par ex. SFTP ou bucket S3) pour de nouveaux fichiers CSV/Excel. Ces fichiers sont parsés (avec `python csv` ou `pandas`), validés (contrôle de la structure : entêtes attendues, types de champ), et intégrés en base (mise à jour de stocks, tarifs, ou traitements de commandes). Les erreurs de parsing sont consignées dans un log spécifique, avec alertes si nécessaire.

Pour l'automatisation, on utilise **Celery + Celery Beat** (plutôt que cron) : par exemple, une tâche journalière `sync_catalogue` connecte l'ERP, télécharge les données, et effectue les mises à jour. Selon la volumétrie, on planifie ces tâches toutes les X minutes/heure via Celery Beat [18]. En cas d'erreur, la tâche renvoie un échec, et un système d'alerte (mail ou tableau de bord de logs) informe l'équipe.

Les fichiers plats sont traités dans des tâches Celery asynchrones également : par exemple, lorsqu'un fichier CSV arrive (upload manuel ou ingestion d'API), on déclenche une task `process_csv_file` qui lit ligne par ligne, mappe dans des modèles Django, et applique les changements. Les erreurs (par ex. format incorrect, référence produit inconnue) sont journalisées en base et/ou envoyées par mail à l'administrateur.

Enfin, tous les échanges (réussis ou échoués) avec les ERP sont tracés dans des logs de synchronisation (table **ERPSyncLog**) avec date, type d'opération, statut et message d'erreur. Cela permet d'auditer la fiabilité de l'intégration.

7. Annexes

- **Conventions de code** : Respecter PEP8 pour Python : noms de variables/fonctions en minuscules avec underscore [19], noms de classes en CamelCase [20], 88 caractères max par ligne (Black), commentaires en Français sur les cas métier spécifiques. Fichiers, modules et endpoints utilisent des noms descriptifs et cohérents (ex. `orders/views.py`, `/api/products/` en REST). Les commits Git suivent un modèle précis (par ex. `[APP] Sujet clair`).
- **Format des réponses API** : On suit les bonnes pratiques REST. Par exemple, un **GET** d'un objet unique renvoie HTTP 200 avec l'objet JSON direct [21], tandis qu'un **GET** de liste renvoie HTTP 200 et un tableau JSON (avec pagination) [22]. Un **POST** créant une ressource renvoie HTTP 201 avec un en-tête `Location` vers la nouvelle ressource et/ou un message de succès [23]. Un **PUT** qui met à jour peut renvoyer 200 avec l'objet modifié, ou 204 sans corps [24] [25]. Les erreurs utilisent des codes standards (404 si non trouvé avec message d'erreur JSON, 400 pour validation [26] [27]). Par exemple, `{ "message": "L'élément n'existe pas" }`. Les réponses JSON doivent être concises : ne pas renvoyer d'attributs inutiles (voir OWASP Excessive Data Exposure [12]).
- **Nomenclature des endpoints REST** : On utilise des noms en minuscules et pluriel pour les ressources (RESTful) [28]. Exemples : `/api/v1/users/`, `/api/v1/products/`, `/api/v1/orders/{id}/items/`. On évite les verbes dans l'URL (l'action est dans la méthode HTTP) [29]. Les paramètres sont passés en query string pour filtrage et tri (`?category=2&search=frein`). Les versions d'API (`/v1/`) sont prévues pour évoluer sans casser les clients. On sépare les ressources hiérarchiquement : ex. `/users/{user_id}/orders` pour liste des commandes d'un utilisateur, ou `/vendors/{vendor_id}/products` pour produits d'un vendeur.
- **Logiques métier clés** : Statuts commande (`new→confirmed→shipped→delivered`), traitement des annulations et retours, calcul de la commission (par exemple prendre 10% sur chaque vente et la verser au marketplace). Le stock produit est décrémenté lors de la commande confirmée. La gestion des droits : seuls les admins peuvent changer les statuts "annulé" ou "remboursé". On gère les codes promotionnels (coupons), éventuellement la TVA selon les zones.
- **Sécurité** :
- Utiliser HTTPS partout.
- Protéger contre les attaques classiques (CSRF n'est pas nécessaire pour API stateless JWT, mais on s'assure de ne pas exposer les tokens).
- Ne jamais logger d'informations sensibles (mots de passe, tokens) [30].
- Conserver les logs d'accès et d'erreurs (authentifications ratées, accès refusés) dans un système sécurisé [30].
- Appliquer le principe de moindre privilège : permissions ORM basées sur le rôle utilisateur, ne jamais laisser un endpoint public par erreur (suivre la guideline OWASP sur DEFAULT_PERMISSION_CLASSES [7]).
- Configurer un CORS strict (autoriser seulement le domaine du frontend).
- Garder SECRET_KEY hors code source (env. vars) et DEBUG=False en production [15].
- Enfin, valider et filtrer systématiquement les données entrantes pour se prémunir des injections SQL/NoSQL [31].

Cette documentation couvre l'ensemble des aspects techniques initiaux. Elle sert de base pour démarrer le développement : chaque module peut être créé dans le dépôt Git avec sa structure, et l'équipe pourra s'y référer pour implémenter les modèles, APIs et intégrations de manière cohérente. Les extraits et exemples ci-dessus sont accompagnés de sources pour approfondir ces pratiques [1] [6] [5] [9] [8] [28] [21] [7] .

---

[1] [2] [3] [4] Ecommerce Database Design: ER Diagram for Online Shopping | Vertabelo Database Modeler
https://vertabelo.com/blog/er-diagram-for-online-shop/

[5] 6 - Viewsets and routers - Django REST framework
https://www.django-rest-framework.org/tutorial/6-viewsets-and-routers/

[6] Simple JWT — Simple JWT 5.2.2.post30+gfaf92e8 documentation
https://django-rest-framework-simplejwt.readthedocs.io/

[7] [11] [12] [15] [30] [31] Django REST Framework - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Django_REST_Framework_Cheat_Sheet.html

[8] Throttling - Django REST framework
https://www.django-rest-framework.org/api-guide/throttling/

[9] Pagination - Django REST framework
https://www.django-rest-framework.org/api-guide/pagination/

[10] Filtering - Django REST framework
https://www.django-rest-framework.org/api-guide/filtering/

[13] Dockerizing Django with Postgres, Gunicorn, and Nginx | TestDriven.io
https://testdriven.io/blog/dockerizing-django-with-postgres-gunicorn-and-nginx/

[14] Amazon S3 - django-storages 1.14.6 documentation
https://django-storages.readthedocs.io/en/latest/backends/amazon-S3.html

[16] API Calls | FlutterFlow Documentation
https://docs.flutterflow.io/resources/backend-logic/rest-api/

[17] JWT Token | FlutterFlow Documentation
https://docs.flutterflow.io/integrations/authentication/firebase/jwt-auth/

[18] How To Schedule Periodic Tasks Using Celery Beat
https://www.axelerant.com/blog/how-to-schedule-periodic-tasks-using-celery-beat

[19] [20] PEP 8 – Style Guide for Python Code | peps.python.org
https://peps.python.org/pep-0008/

[21] [22] [23] [24] [25] [26] [27] REST API response format based on some of the best practices · GitHub
https://gist.github.com/igorjs/407ffc3126f6ef2a6fe8f918a0673b59

[28] [29] Best Practices for Naming REST API Endpoints
https://blog.dreamfactory.com/best-practices-for-naming-rest-api-endpoints