

# **CAHIER DES CHARGES TECHNIQUE**

## **Projet : Plateforme Web Corporate Headless & Chatbot IA**

### **1. Présentation Générale**

- 1.1. Contexte et Objectifs du projet
- 1.2. Périmètre du projet (MVP)
- 1.3. Public cible et Expérience Utilisateur (UX)

### **2. Architecture Technique Globale**

- 2.1. Vue d'ensemble (Architecture "Decoupled")
- 2.2. Stack Technique Détailée
  - Frontend : Next.js, Tailwind CSS, Shadcn/ui
  - Backend : Django, DRF (Django REST Framework)
  - Base de données : PostgreSQL (via cPanel)
  - Stockage Médias : Cloudinary (CDN)
  - IA / LLM : OpenRouter (Proxy Backend)
- 2.3. Topologie de l'Hébergement & Flux Réseau
  - Backend (API) : Namecheap (Python App / cPanel)
  - Frontend (Client) : Vercel / Netlify
  - Gestion des DNS et Domaines (api. vs www.)

### **3. Spécifications Fonctionnelles : Frontend (Site Public)**

- 3.1. Structure et Navigation (Sitemap)
- 3.2. Internationalisation (i18n : FR / EN)
- 3.3. Pages "Statiques" Dynamiques (Accueil, À Propos)
  - Rendu des sections JSON
- 3.4. Pages Structurées (Portfolio, Services, Blog)
  - Listing, Filtrage, Pages de détail
- 3.5. UX/UI & Design System
  - Composants Shadcn/ui et Thème (Dark/Light mode)
  - Animations et Transitions

- 3.6. Performance & SEO (SSR, Metadata, Core Web Vitals)

#### **4. Spécifications Fonctionnelles : Backend (Dashboard Admin)**

- 4.1. Authentification & Rôles (SuperAdmin, Éditeur)
- 4.2. Gestion des Pages & "Page Builder"
  - CRUD des Sections (Ajout, Suppression, Ordre)
  - Édition de contenu JSON (Champs dynamiques)
- 4.3. Gestion des Contenus Structurés (Services, Projets, Blog)
- 4.4. Gestion des Médias (Intégration Cloudinary)
- 4.5. Traduction Assistée par IA
  - Workflow : Saisie manuelle + Bouton "Auto-Translate"
- 4.6. Paramètres Globaux (SEO, Méta-données, Contact)

#### **5. Spécifications Détaillées du Chatbot RAG (IA)**

- 5.1. Logique Fonctionnelle
  - Accès libre vs Capture de Leads (Scénario 3-5 messages)
- 5.2. Architecture RAG (Retrieval-Augmented Generation)
  - Indexation des contenus (Services/Projets)
  - Recherche vectorielle et Contexte
- 5.3. Interface Chat Widget (Frontend)
- 5.4. Sécurité et Proxy API (Backend)

#### **6. Modélisation des Données (PostgreSQL)**

- 6.1. Approche Hybride : Explication du concept
- 6.2. Schéma Relationnel (Entités Structurées)
  - Tables : User, Service, Project, Category
- 6.3. Schéma Dynamique (Entités Flexibles)
  - Table : Page, Section
  - Structure du champ JSONB pour le contenu des sections

#### **7. API REST & Interfaces**

- 7.1. Standards API (REST, JSON, Codes HTTP)

- 7.2. Authentification API (JWT)
- 7.3. Cartographie des Endpoints Principaux
  - Public (GET content)
  - Admin (CRUD content)
  - Chatbot (POST message)

## **8. Sécurité & Conformité**

- 8.1. Sécurité Applicative (CORS, CSRF, Injection SQL)
- 8.2. Gestion des Clés API (Variables d'environnement)
- 8.3. RGPD & Gestion des Leads

## **9. Déploiement & DevOps (Spécifique Namecheap/Vercel)**

- 9.1. Pré-requis Serveur (Version Python, Modules cPanel)
- 9.2. Pipeline de Déploiement Backend (FTP/SSH/Git)
- 9.3. Pipeline de Déploiement Frontend (Vercel Git Integration)
- 9.4. Stratégie de Backup (DB & Code)

## **10. Planning & Livrables**

- 10.1. Phases de développement (Sprint 1 à Sprint Final)
- 10.2. Liste des Livrables attendus (Code, Docs, Accès)
- 10.3. Tests & Recette (QA)

## **8. Sécurité & Conformité**

Dans une architecture découpée avec des clés API sensibles (IA) et des données clients, la sécurité n'est pas une option.

### **8.1. Sécurité Applicative**

- **CORS (Cross-Origin Resource Sharing) :**
  - Le Backend Django doit être configuré (via django-cors-headers) pour n'accepter **que** les requêtes provenant du domaine Frontend (<https://www.monentreprise.com>) et du domaine local (<http://localhost:3000> en phase de dev). Tout autre domaine est rejeté.
- **Protection CSRF (Cross-Site Request Forgery) :**
  - Bien que l'authentification se fasse via JWT, les formulaires standards de l'admin Django restent protégés par le token CSRF classique.
- **Injection SQL :**
  - Interdiction formelle d'écrire des requêtes SQL brutes (raw SQL). L'utilisation exclusive de l'ORM Django (Object-Relational Mapper) garantit une protection native contre les injections.
- **Rate Limiting (Anti-Abus) :**
  - Sur l'endpoint Chatbot (/api/chat/), limitation à **10 requêtes par minute par IP** pour éviter de drainer le crédit OpenRouter en cas d'attaque.

## 8.2. Gestion des Clés API (Variables d'environnement)

Aucune clé secrète ne doit jamais apparaître dans le code source (Git).

- **Méthode :** Utilisation d'un fichier .env non versionné (ajouté au .gitignore).
- **Liste des variables critiques :**
  - SECRET\_KEY (Django)
  - DEBUG (False en prod)
  - DB\_NAME, DB\_USER, DB\_PASSWORD (PostgreSQL)
  - OPENROUTER\_API\_KEY (Chatbot)
  - CLOUDINARY\_URL (Médias)
- **En Production (cPanel) :** Les variables doivent être définies soit dans l'interface "Setup Python App" (Application Environment variables), soit dans un fichier .env sécurisé à la racine du projet, hors du dossier public (public\_html).

## 8.3. RGPD & Gestion des Leads

- **Consentement Chatbot :** Une phrase d'avertissement doit être visible ("Ce chatbot utilise une IA, ne partagez pas d'infos sensibles").
- **Logs de conversation :**

- Les conversations anonymes sont purgées automatiquement après 30 jours (Cron job Django).
  - Les conversations liées à un Lead (email capturé) sont conservées 1 an ou jusqu'à demande de suppression.
  - **Droit à l'oubli** : L'admin doit pouvoir supprimer un Lead en un clic, ce qui anonymise instantanément toutes les conversations liées.
  - **Bandeau Cookies** : Implémentation sur le Frontend d'un bandeau simple (acceptation Google Analytics / Chatbot).
- 

## 9. Déploiement & DevOps (Spécifique Namecheap/Vercel)

Cette section détaille comment passer du code au site en ligne, en tenant compte des limitations de l'hébergement partagé.

### 9.1. Pré-requis Serveur (Namecheap cPanel)

- **Python** : Sélectionner la version **3.9** ou supérieure dans l'interface "Setup Python App".
- **PostgreSQL** : Créer la base et l'utilisateur via l'outil "PostgreSQL Databases" de cPanel.
- **Accès SSH** : Demander l'activation de l'accès SSH au support Namecheap (souvent désactivé par défaut) pour pouvoir exécuter les commandes pip install et python manage.py migrate.

### 9.2. Pipeline de Déploiement Backend (Méthode SSH/Git)

Pas de Docker possible ici. Le déploiement est semi-automatisé.

1. **Connexion** : Via Terminal SSH.
2. **Pull** : git pull origin main dans le dossier de l'app.
3. **Dépendances** : pip install -r requirements.txt (si nouvelles librairies).
4. **Base de données** : python manage.py migrate (pour appliquer les changements de schéma).
5. **Statiques** : python manage.py collectstatic (pour l'interface admin).
6. **Redémarrage** : touch tmp/restart.txt (Commande spécifique Passenger pour recharger l'app Python sans redémarrer le serveur).

### 9.3. Pipeline de Déploiement Frontend (Vercel Git Integration)

1. **Lien Repository** : Connecter le compte Vercel au repo GitHub du projet.

2. **Configuration Build** :

- Framework Preset : Next.js.
- Build Command : npm run build.

3. **Variables d'environnement (Vercel Dashboard)** :

- NEXT\_PUBLIC\_API\_URL : <https://api.tonsite.com/api/v1>

4. **Déploiement Continu (CD)** : Chaque git push sur la branche main déclenche automatiquement le build et la mise en ligne.

#### 9.4. Stratégie de Backup

- **Code** : Sécurisé sur GitHub (Private Repo).
- **Base de Données** : Configuration d'une tâche Cron sur cPanel pour faire un pg\_dump tous les jours à 03h00 du matin vers un dossier sécurisé non accessible publiquement.
- **Médias** : Cloudinary assure sa propre redondance. Un backup local mensuel des originaux est conseillé mais optionnel pour le MVP.

---

### 10. Planning & Livrables (Délai : 1 Mois)

Pour tenir le délai de **4 semaines**, une méthodologie Agile stricte est requise. Pas de dérive fonctionnelle ("Feature Creep") : on s'en tient au MVP.

#### 10.1. Phases de développement (4 Sprints d'une semaine)

- **Semaine 1 : Fondations & API (Backend Focus)**
  - Setup environnements (Repo, Django sur cPanel, Vercel).
  - Développement des modèles DB (Services, Projets, Pages, Sections).
  - Mise en place de l'Admin Django et de l'Auth (JWT).
  - *Livrable S1* : Accès au Dashboard Admin fonctionnel (sur serveur de prod).
- **Semaine 2 : "Page Builder" & Intégration (Backend + Frontend)**
  - Développement de la logique "Sections dynamiques" (API JSON).
  - Création des composants React (Hero, Texte/Image, Features).
  - Connexion API : Les pages du site s'affichent avec le contenu de la base.
  - *Livrable S2* : Site public "moche" mais fonctionnel (les textes remontent).

- **Semaine 3 : UI Design, Chatbot & Médias**
  - Intégration du Design System (Tailwind/Shadcn).
  - Développement du Chatbot (Connecteur OpenRouter + Widget Front).
  - Intégration Cloudinary pour les images.
  - *Livrable S3* : Site quasi-fini, Chatbot capable de répondre.
- **Semaine 4 : Polishing, SEO & Lancement**
  - Ajustements Mobile (Responsive).
  - Optimisation SEO (Metatags dynamiques).
  - Remplissage du contenu initial (Client).
  - Tests de sécurité et formulaires.
  - *Livrable S4 : Mise en production Finale (Go Live)*.

## 10.2. Liste des Livrables attendus

1. **Code Source** : Dépôt GitHub unique (Monorepo) ou deux dépôts (Front/Back), propriété du client.
2. **Accès Admin** : URL, Login, Mot de passe SuperAdmin.
3. **Documentation Technique Simplifiée** :
  - "Comment lancer le projet en local".
  - "Comment déployer une mise à jour sur cPanel".
4. **Guide Utilisateur (PDF ou Notion)** :
  - "Comment créer une nouvelle page".
  - "Comment ajouter un projet".
  - "Comment traduire une section via l'IA".

## 10.3. Tests & Recette (QA)

Vu le délai court, pas de tests unitaires exhaustifs (80% coverage impossible). On se concentre sur les **Tests End-to-End (E2E) Critiques** :

- Création d'une page + Ajout d'une section Hero (Admin).
- Affichage correct de cette page (Front).
- Envoi réussi d'un formulaire de contact (Email reçu).

- Dialogue Chatbot : Question → Réponse IA → Capture Email.
- Bascule FR/EN fonctionnelle.

## Projet : Plateforme Web Corporate Headless & Chatbot IA

### 1. Présentation Générale

Cette section définit la vision stratégique, les objectifs opérationnels et le périmètre fonctionnel du projet. Elle sert de référence pour aligner toutes les parties prenantes sur la finalité du produit.

#### 1.1. Contexte et Objectifs du projet

Le projet consiste à concevoir et développer une **plateforme web d'entreprise de nouvelle génération**. Loin d'un site vitrine statique classique ou d'un CMS monolithique (type WordPress), l'objectif est de déployer une architecture "**Headless**" (découplée) alliant la puissance d'un framework Frontend moderne (Next.js) à la robustesse d'un Backend Python (Django).

Le besoin central est l'indépendance totale : le propriétaire du site doit pouvoir modifier non seulement les textes, mais aussi la structure des pages, l'ordre des sections et les médias, sans jamais toucher une ligne de code, via une interface d'administration sur-mesure.

Les objectifs majeurs sont :

1. **Présenter l'entreprise et son savoir-faire** : Offrir une vitrine B2B haut de gamme, rapide et multilingue (FR/EN) pour crédibiliser l'offre de services.
2. **Autonomie et Modularité (Le concept "Site Builder")** : Fournir un Dashboard administrateur permettant de construire des pages dynamiquement (ajout/suppression de blocs, drag & drop) grâce à une architecture de données hybride (SQL structuré + JSON flexible).
3. **Acquisition et Conversion (Lead Gen)** : Transformer les visiteurs en prospects qualifiés via un Chatbot IA intelligent (RAG) capable de répondre aux questions sur les services et de capturer des emails.
4. **Performance et SEO** : Garantir un référencement optimal et des temps de chargement instantanés grâce au Server-Side Rendering (SSR) de Next.js.

5. **Optimisation des coûts d'infrastructure** : Exploiter l'hébergement existant (Namecheap cPanel) pour le backend tout en déportant le frontend et les médias vers des solutions Cloud performantes (Vercel, Cloudinary).

## 1.2. Périmètre du projet (MVP)

Le projet se concentre sur un **Minimum Viable Product (MVP)** robuste, incluant les fonctionnalités suivantes :

### A. Front-Office (Site Public - Next.js)

- **Pages "Libres" (Structure JSON)** : Accueil, À Propos, Landing Pages spécifiques. Ces pages sont composées de blocs réordonnables (Hero, CTA, Features, Stats, etc.).
- **Pages "Structurées" (Modèles SQL)** :
  - Portfolio / Projets (Listing filtrable + pages détails).
  - Services (Listing + pages détails).
  - Blog (Articles + Catégories) - *Optionnel en phase 1, mais architecture prévue.*
- **Fonctionnalités Transverses** :
  - Support multilingue natif (FR/EN).
  - Chatbot IA (Widget interactif connecté à l'API Backend).
  - Formulaires de contact et de capture de leads.
  - Mode Sombre / Mode Clair (Dark Mode).

### B. Back-Office (Dashboard Admin - Django)

- **Gestion des Pages** : Interface CRUD pour créer des pages et éditer leurs sections (Rich Text pour le contenu, upload médias).
- **Gestion des Entités** : Interfaces dédiées pour les Projets, Services et Catégories.
- **Médiathèque** : Intégration API Cloudinary pour l'upload et la sélection d'images/vidéos.
- **IA & Traduction** :
  - Bouton "Traduire" pour générer automatiquement les versions EN depuis le FR.
  - Configuration du "System Prompt" du Chatbot et indexation des contenus (RAG simple).
- **Sécurité & Système** : Authentification Admin, Logs d'activité, Proxy API pour OpenRouter.

### **1.3. Public cible et Expérience Utilisateur (UX)**

L'expérience utilisateur doit être traitée avec deux niveaux de priorité distincts : l'utilisateur final (le visiteur) et l'administrateur (le client).

#### **A. Pour le Visiteur (Front-Office)**

- **Cible** : Clients B2B, partenaires potentiels, investisseurs.
- **Profil** : Professionnels pressés, cherchant une information précise et rassurante.
- **Expérience visée** :
  - **Fluidité** : Navigation instantanée (SPA feel), pas de recharge de page visible.
  - **Clarté** : Design épuré, typographie lisible, hiérarchie visuelle forte (basée sur *shadcn/ui*).
  - **Accessibilité** : Site parfaitement "Responsive" (Mobile First) et compatible tous navigateurs modernes.
  - **Interaction** : Le chatbot doit être proactif mais non intrusif, agissant comme un assistant commercial virtuel.

#### **B. Pour l'Administrateur (Back-Office)**

- **Cible** : L'équipe marketing ou le dirigeant de l'entreprise.
- **Profil** : Non-technique, habitué aux outils modernes (Notion, Airtable, etc.).
- **Expérience visée** :
  - **Simplicité** : Pas de JSON brut à éditer. Des formulaires clairs, des prévisualisations si possible.
  - **Efficacité** : Actions rapides (Drag & Drop de sections, duplication de pages).
  - **Sécurité** : Impossibilité de "casser" le site public par une erreur de manipulation (validation des données en amont).

## **2. Architecture Technique Globale**

### **2.1. Vue d'ensemble (Architecture "Decoupled")**

Le projet repose sur une architecture dite "**Headless**" (ou **Découplée**). Contrairement à un CMS monolithique classique (où le backend génère directement les pages HTML), ici,

**la gestion des données** (Backend) est strictement séparée de la **présentation visuelle** (Frontend).

Cette séparation physique et logique permet d'exploiter le meilleur de chaque environnement : la robustesse de Python pour la logique métier et la performance de Next.js pour l'expérience utilisateur.

Le système s'articule autour de trois couches distinctes communiquant exclusivement via API :

#### A. Le Backend (API Master & Logique Métier)

- **Rôle** : Il agit comme la "Source Unique de Vérité". Il ne se préoccupe pas du design. Il stocke les données, gère la sécurité, l'authentification et la logique complexe (proxy IA).
- **Technologie** : Django + Django REST Framework.
- **Hébergement** : Serveur **Namecheap (cPanel)**. L'application Python tourne en arrière-plan (via WSGI) et expose des points d'accès (Endpoints) RESTful.
- **Données** : Base de données **PostgreSQL** (hébergée localement sur cPanel) stockant à la fois des données relationnelles et des structures JSON dynamiques.

#### B. Le Frontend (Interface Client & Rendu)

- **Rôle** : Il consomme les données JSON fournies par le Backend pour construire l'interface utilisateur. Il assure le rendu visuel, les animations et l'interactivité.
- **Technologie** : Next.js (React Framework).
- **Hébergement** : Plateforme Cloud optimisée (**Vercel** ou Netlify).
- **Mode de Rendu** : Server-Side Rendering (SSR) et Static Site Generation (SSG) pour garantir un SEO optimal et des performances élevées, indépendamment de la puissance du serveur backend Namecheap.

#### C. Les Services Tiers (Satellites)

- **Stockage Médias** : **Cloudinary**. Afin de contourner les limitations de stockage et de bande passante du cPanel, les images et vidéos ne sont pas stockées sur le serveur backend. Le Backend ne stocke que les URL (liens), tandis que les fichiers physiques sont servis par le CDN mondial de Cloudinary.
- **Intelligence Artificielle** : **OpenRouter**. Le moteur LLM est interrogé par le Backend (qui détient les clés API secrètes) pour alimenter le Chatbot du Frontend.

**Flux de communication simplifié :**

**Utilisateur (Navigateur) ↔ Frontend (Next.js/Vercel)**  
↔ **HTTPS/JSON** ↔ **Backend (Django/Namecheap)** ↔ **Base de données (PostgreSQL)**.

## 2.2. Stack Technique Détaillée

Le choix des technologies privilégie la modernité, la performance et la maintenabilité à long terme. Chaque brique a été sélectionnée pour répondre à une contrainte spécifique du projet (SEO, flexibilité des données, hébergement partagé).

### A. Frontend (Interface Utilisateur)

L'application client est une Single Page Application (SPA) optimisée pour le référencement.

- **Framework Principal : Next.js (Dernière version stable, App Router)**
  - *Rôle* : Gestion du rendu (SSR/SSG), du routing, et de l'optimisation des performances.
  - *Justification* : Indispensable pour garantir un SEO performant (contrairement à React pur) et des temps de chargement rapides (Core Web Vitals) via le pré-rendu des pages.
- **Styling : Tailwind CSS**
  - *Rôle* : Framework CSS "Utility-first".
  - *Justification* : Permet un développement rapide, une cohérence visuelle stricte et une gestion native du Responsive Design.
- **Composants UI : shadcn/ui**
  - *Rôle* : Bibliothèque de composants réutilisables (Boutons, Modales, Inputs, Menus).
  - *Justification* : Offre un design professionnel, accessible et moderne par défaut. Contrairement à une librairie figée (MUI, Bootstrap), le code est modifiable, offrant un contrôle total sur le design. Supporte nativement le mode Sombre/Clair.
- **Gestionnaire de paquets : npm ou pnpm.**

### B. Backend (API & Administration)

Le serveur assure la logique métier, la sécurité et la distribution des données.

- **Framework : Django**
  - *Rôle* : Framework web Python de haut niveau.
  - *Justification* : Fournit "out of the box" une interface d'administration sécurisée et complète, essentielle pour le Dashboard client. Sécurité robuste (protection XSS, SQL Injection).
- **API : Django REST Framework (DRF)**
  - *Rôle* : Transformation des modèles de base de données en format JSON (Sérialisation) et gestion des Endpoints API.
  - *Justification* : Standard industriel pour créer des API RESTful avec Django. Gestion fine des permissions et de l'authentification.
- **Serveur d'application** : Phusion Passenger (via sélecteur "Setup Python App" de cPanel) assurant l'interface WSGI.

## C. Base de Données (Stockage)

- **Système : PostgreSQL**
  - *Rôle* : Persistance des données relationnelles et non-relationnelles.
  - *Justification critique* : Choisi spécifiquement pour son type de champ natif **JSONB**. C'est ce qui permet de stocker la structure flexible des pages (les sections dynamiques) de manière performante et requêtisable, ce que MySQL gère moins efficacement.
  - *Hébergement* : Instance locale sur le compte cPanel.

## D. Stockage Médias (Assets)

- **Service : Cloudinary**
  - *Rôle* : Hébergement des images et vidéos, CDN (Content Delivery Network) et optimisation à la volée.
  - *Justification* : Contourne les limitations de stockage et d'IO (Input/Output) du serveur mutualisé Namecheap. Cloudinary convertit automatiquement les images en formats modernes (WebP, AVIF) et les redimensionne selon l'écran du visiteur, améliorant drastiquement le score de performance Google PageSpeed.

## E. Intelligence Artificielle (Chatbot)

- **Service : OpenRouter**

- *Rôle* : Agrégateur d'API LLM (permet d'accéder à GPT-4o, Claude 3.5 Sonnet, Llama 3, etc. via une seule interface).
- *Architecture : Proxy Backend.*
  - Le Frontend n'appelle jamais OpenRouter directement.
  - Le Frontend appelle l'API Django (POST /api/chat/).
  - Django vérifie la requête, injecte la clé API (stockée en variable d'environnement serveur) et le contexte métier (RAG), interroge OpenRouter, et renvoie la réponse.
- *Justification* : Sécurité des clés API et contrôle des coûts.

### **2.3. Topologie de l'Hébergement & Flux Réseau**

Cette section décrit comment l'application est physiquement répartie sur internet et comment les différents serveurs communiquent entre eux. Cette séparation est critique pour assurer la sécurité et la performance du site.

#### **A. Backend : Le Serveur API (Hébergement Namecheap)**

Le backend est hébergé sur l'infrastructure existante (offre Stellar Plus). Il n'est jamais consulté directement par les visiteurs pour afficher le site, mais répond aux requêtes de données.

- **Type de déploiement** : Application Python (WSGI) via Phusion Passenger.
- **Adresse d'accès** : Sous-domaine dédié  
(ex: api.monentreprise.com ou admin.monentreprise.com).
- **Responsabilités** :
  - Héberger l'interface d'administration Django (HTML classique).
  - Exposer les Endpoints API (JSON) pour le Frontend.
  - Maintenir la connexion persistante avec la base de données locale PostgreSQL.
  - Gérer les tâches de fond (envoi d'emails, appels OpenRouter).

#### **B. Frontend : Le Serveur de Rendu (Vercel ou Netlify)**

Le frontend est hébergé sur une infrastructure "Edge" spécialisée pour les frameworks JS modernes comme Next.js.

- **Type de déploiement** : Serverless Functions & Static Edge Network.

- **Adresse d'accès :** Domaine principal (ex: www.monentreprise.com et monentreprise.com).
- **Responsabilités :**
  - Recevoir le trafic public (visiteurs).
  - Construire les pages (HTML/CSS) en récupérant les données depuis le Backend.
  - Servir les fichiers statiques optimisés.

### C. Configuration DNS et Routage

Une configuration DNS rigoureuse est nécessaire pour diriger le trafic vers la bonne infrastructure.

Type d'enregistrement	Hôte (Host)	Destination (Value)	Cible
A Record (ou CNAME)	@ (root)	IP Vercel / Netlify	Site Public
CNAME	www	cname.vercel-dns.com	Site Public
A Record	api (ou admin)	IP Serveur Namecheap	Backend / Admin
TXT	@	Validation SPF/DKIM	Délivrabilité Emails

### D. Flux de communication et Sécurité (CORS & SSL)

Puisque le Frontend (www.) et le Backend (api.) sont sur des domaines différents, des mesures de sécurité réseau spécifiques s'appliquent :

1. **HTTPS Obligatoire :** Les deux environnements doivent disposer de certificats SSL valides (Let's Encrypt sur Namecheap, géré automatiquement sur Vercel). Aucune communication API ne se fera en HTTP clair.
2. **CORS (Cross-Origin Resource Sharing) :** Le Backend Django doit être configuré (via django-cors-headers) pour accepter **uniquement** les requêtes provenant du domaine du Frontend (<https://www.monentreprise.com>). Toute autre origine sera rejetée.
3. **Proxy de fichiers :** Le serveur Frontend ne stocke aucun fichier média. Les balises <img> du site pointeront directement vers les URLs du CDN Cloudinary.

### **3. Spécifications Fonctionnelles : Frontend (Site Public)**

Cette section décrit le comportement, l'apparence et la logique interne de l'interface utilisateur (Next.js).

#### **3.1. Structure et Navigation (Sitemap)**

Le site est organisé selon une hiérarchie claire, optimisée pour le maillage interne (SEO).

##### **A. Arborescence Principale (URLs Canoniques)**

- / : Page d'accueil (Assemblage dynamique de sections).
- /a-propos : Présentation de l'entreprise (Assemblage dynamique).
- /services : Listing des services (Grille de cartes).
  - /services/{slug} : Page de détail d'un service spécifique.
- /projets : Portfolio / Cas clients (Grille filtrable).
  - /projets/{slug} : Page de détail d'un projet (Galerie, résultats).
- /blog (*Optionnel MVP*) : Listing des articles.
  - /blog/{slug} : Lecture d'un article.
- /contact : Page dédiée (Formulaire + Infos + Map).
- /legal/\* : Pages légales (Mentions légales, Politique de confidentialité, CGU).

##### **B. Navigation (Header & Footer)**

- **Header (Global) :**
  - Logo (SVG optimisé).
  - Menu de navigation (liens gérés via l'API, non hardcodés).
  - Sélecteur de Langue (FR/EN).
  - Bouton CTA principal (ex: "Demander un devis").

- *Comportement* : Sticky (reste visible au scroll), changement de couleur de fond au scroll (transparent → solide).
- *Mobile* : Menu Burger avec animation d'ouverture fluide (Side Panel).
- **Footer (Global) :**
  - Rappel Logo et slogan.
  - Colonnes de liens (Navigation rapide, Services, Légal).
  - Réseaux sociaux (Icônes).
  - Copyright dynamique (Année auto-updatée).

### **3.2. Internationalisation (i18n : FR / EN)**

Le site utilise le **Routing International** natif de Next.js.

- **Structure des URLs :**
  - Langue par défaut (FR) : domaine.com/services
  - Langue secondaire (EN) : domaine.com/en/services
- **Détection :**
  - À la première visite, détection de la langue du navigateur (Accept-Language header).
  - Si EN détecté → redirection auto vers /en. Sinon, reste sur /.
- **Logique de récupération des données (Data Fetching) :**
  - Chaque requête API vers le Backend inclut le paramètre de langue (?lang=fr ou ?lang=en).
  - Le Frontend ne stocke pas de fichiers de traduction statiques (type fr.json) pour le contenu éditorial. Tout le texte vient de l'API.
  - Seuls les textes d'interface fixes (Boutons "Envoyer", "Suivant", labels de formulaire) sont gérés via des fichiers de traduction locaux (next-intl ou react-i18next).
- **Persistiance :**
  - Le choix de l'utilisateur est stocké dans un cookie (NEXT\_LOCALE) pour les visites futures.

### **3.3. Pages "Statiques" Dynamiques (Accueil, À Propos)**

Ces pages n'ont pas de template fixe. Elles sont des conteneurs vides qui s'adaptent au JSON renvoyé par l'API.

## A. Logique de Rendu (Component Factory)

1. **Appel API** : Le frontend appelle GET /api/pages/{slug}/.

2. **Réception** : L'API renvoie un tableau d'objets sections:

codeJSON

[

```
{"type": "hero_video", "content": {...}, "order": 1},  
 {"type": "features_grid", "content": {...}, "order": 2},  
 {"type": "cta_banner", "content": {...}, "order": 3}
```

]

3. **Mapping** : Un composant "Switch" (SectionRenderer) boucle sur ce tableau et instancie le composant React correspondant :

- hero\_video → <HeroVideoComponent data={content} />
- features\_grid → <FeaturesComponent data={content} />

4. **Fallback** : Si un type de section est inconnu (ex: ajouté dans le back mais pas encore codé dans le front), le système l'ignore silencieusement ou affiche un placeholder (en mode dev uniquement) pour éviter le crash.

## B. Catalogue des Sections (Composants Requis)

Le développeur devra créer les composants React suivants, configurables via props :

- **Hero Section** : Titre H1, sous-titre, 2 boutons, image ou vidéo de fond.
- **Text & Image** : Bloc 50/50 (Image à gauche/Texte à droite et inversement).
- **Features / Services Grid** : Grille d'icônes avec titre et résumé.
- **Stats Counter** : Chiffres animés (ex: "50+ Projets").
- **Testimonials** : Carrousel d'avis clients.
- **Logo Cloud** : Bandeau de logos partenaires (défilement infini).
- **CTA (Call to Action)** : Bloc pleine largeur pour inciter au contact.
- **FAQ (Accordéon)** : Liste de questions/réponses dépliables.

## 3.4. Pages Structurées (Portfolio, Services, Blog)

Contrairement aux pages dynamiques, ces pages suivent un gabarit précis (Template).

### A. Page Listing (ex: /projets)

- **Layout** : En-tête (Titre/Intro) + Zone de filtres + Grille de résultats.
- **Système de Filtrage** :
  - Filtres par Catégorie (ex: Web, Mobile, Design).
  - Filtres "Client-side" pour une réactivité instantanée (si < 100 éléments) ou "Server-side" (API Call avec query params) si volume important.
  - *UX* : Les filtres modifient l'URL (?category=web) pour permettre le partage de lien filtré.
- **Cartes (Cards)** :
  - Composant <ProjectCard /> : Image à la une (hover zoom), Titre, Tags, lien "Voir plus".

## B. Page de Détail (ex: /projets/mon-super-projet)

- **Routing dynamique** : Utilisation de [slug].js ou [slug]/page.tsx (Next.js App Router).
- **Génération Statique (SSG/ISR)** :
  - Utilisation de generateStaticParams pour pré-générer les pages au build.
  - Activation de l'**ISR (Incremental Static Regeneration)** : La page est mise en cache mais se régénère automatiquement (ex: toutes les 60s) si le contenu a changé dans le backend.
- **Structure visuelle** :
  - Hero Header (Image du projet en full-width).
  - Sidebar ou Bandeau métadonnées (Client, Année, Technologies).
  - Contenu riche (Description, Challenge, Solution).
  - Galerie ou Vidéo de démo.
  - Bloc "Projets similaires" en bas de page (Cross-linking).

## 3.5. UX/UI & Design System

L'interface repose sur une stack moderne assurant cohérence et rapidité de développement.

### A. Frameworks & Librairies

- **Tailwind CSS** : Pour le styling atomique. Configuration d'un fichier tailwind.config.js avec les couleurs de la marque (Primary, Secondary, Accent) et les polices.

- **Shadcn/ui** : Utilisation des composants primitifs (Button, Dialog, Input, Select, Sheet, Accordion). Ces composants sont copiés dans le code source (/components/ui) et personnalisés.

## B. Gestion du Thème (Dark Mode)

- Utilisation de next-themes pour gérer la classe .dark sur la balise <html>.
- **Switch** : Bouton dans le Header (Icône Soleil/Lune).
- **Système** : Respect de la préférence système (prefers-color-scheme) par défaut.
- **Implémentation** : Toutes les couleurs sont définies via des variables CSS (--background, --foreground) pour basculer instantanément sans recharge.

## C. Animations et Transitions

- Utilisation de **Framer Motion** pour des micro-interactions soignées :
  - *Fade-in up* : Les éléments apparaissent doucement en remontant lors du scroll.
  - *Staggering* : Les éléments d'une liste (grille services) apparaissent les uns après les autres avec un léger délai.
  - *Page Transition* : Transition subtile (opacité) lors du changement de route (via template.tsx dans Next.js App Router).
- **Contrainte** : Les animations doivent être désactivables si l'utilisateur a configuré "Reduce Motion" dans son OS.

## 3.6. Performance & SEO (Core Web Vitals)

La performance technique est un pré-requis, pas une option.

### A. Optimisation des Images

- Utilisation du composant natif <Image /> de Next.js.
- **Loader** : Configuration d'un loader personnalisé pour générer les URLs **Cloudinary** avec les paramètres d'optimisation (format f\_auto, qualité q\_auto, largeur w\_...).
- **LCP (Largest Contentful Paint)** : L'image du Hero (haut de page) doit avoir la propriété priority={true} pour être pré-chargée.
- **CLS (Layout Shift)** : Toutes les images doivent avoir un ratio d'aspect défini (width/height) pour réservé l'espace avant le chargement.

### B. SEO Technique (Metadonnées)

- **Dynamic Metadata** : Chaque page appelle l'API pour récupérer son <title> et sa <meta name="description">.
- **Open Graph (OG)** : Génération automatique des balises pour le partage social (Facebook, LinkedIn, Twitter) incluant l'image à la une.
- **Données Structurées (JSON-LD)** : Injection de scripts Schema.org pour :
  - Organization (Page d'accueil).
  - Service (Pages services).
  - BreadcrumbList (Fil d'Ariane).
- **Sitemap** : Génération automatique d'un sitemap.xml dynamique listant toutes les routes publiques lors du build ou à la demande.

## C. Polices et Scripts

- **Next/Font** : Hébergement des polices Google Fonts (ex: Inter, Poppins) directement par Next.js (pas de requête externe vers Google au chargement).
- **Script Loading** : Les scripts tiers (Analytics, Chatbot, Pixel) sont chargés via le composant <Script /> avec la stratégie lazyOnload ou afterInteractive pour ne pas bloquer le rendu principal.

## 4. Spécifications Fonctionnelles : Backend (Dashboard Admin)

Le Backend ne se contente pas de servir des données ; il fournit une interface d'administration complète (Back-Office) permettant au client de piloter l'intégralité du site sans intervention technique. Cette interface repose sur l'admin native de Django, fortement personnalisée pour l'ergonomie.

### 4.1. Authentification & Rôles (SuperAdmin, Éditeur)

L'accès au Dashboard est strictement sécurisé et cloisonné. Le système utilise le module d'authentification natif de Django (`django.contrib.auth`) étendu pour les besoins du projet.

#### A. Gestion des Accès

URL d'accès : Le dashboard est accessible sur une route sécurisée (ex: <https://api.tonsite.com/admin-control/> - l'URL par défaut /admin/ sera masquée pour la sécurité).

Mécanisme : Authentification par Session (Cookies sécurisés HttpOnly et Secure).

Sécurité Renforcée :

Mise en place de Rate Limiting (blocage après 5 tentatives échouées) pour prévenir les attaques Bruteforce.

Déconnexion automatique après 30 minutes d'inactivité.

#### B. Hiérarchie des Rôles (RBAC)

Deux niveaux de priviléges sont définis via le système de "Groupes" Django :

Le SuperAdmin (Propriétaire / Développeur)

Droits : Accès total (RWX).

Capacités exclusives :

Gestion des utilisateurs (créer/supprimer des éditeurs).

Configuration technique (Clés API OpenRouter, Identifiants Cloudinary).

Modification des types de sections disponibles (Configuration système).

Accès aux logs d'erreurs et de sécurité.

#### L'Éditeur (Marketing / Content Manager)

Droits : Lecture et Écriture sur le contenu, restriction sur la configuration.

Capacités :

CRUD (Créer, Lire, Mettre à jour, Supprimer) sur les Pages, Articles de Blog, Projets, Services.

Gestion de la médiathèque.

Visualisation des messages du formulaire de contact.

Restrictions : Ne peut pas supprimer la page d'accueil, ne peut pas changer les configurations DNS ou API.

#### 4.2. Gestion des Pages & "Page Builder"

C'est la fonctionnalité centrale du CMS Headless. Elle permet de construire des pages complexes en assemblant des blocs, un peu comme des "Lego".

##### A. Modélisation : Entité "Page"

Chaque page est un objet conteneur comportant les métadonnées globales :

Titre (Interne) : Pour l'organisation dans le dashboard.

Slug (URL) : Ex: nos-services.

SEO Méta-titre (FR/EN) : Le titre bleu dans Google.

SEO Méta-description (FR/EN) : Le résumé dans Google.

Statut : Brouillon / Publié.

## B. Modélisation : Entité "Section" (Le cœur du système)

Les pages ne contiennent pas de champ "body" HTML unique. Elles sont liées (relation One-to-Many) à une liste d'objets Section.

Structure de la table Section (PostgreSQL) :

page\_id (ForeignKey) : La page parente.

type (ChoiceField) : Le modèle de section (ex: hero\_header, text\_image, pricing\_table, cta).

order (Integer) : La position dans la page (1, 2, 3...).

is\_active (Boolean) : Permet de masquer une section temporairement sans la supprimer.

content (JSONField) : Contient toutes les données variables (textes, ids des images, liens).

## C. CRUD et Expérience Utilisateur (UX) dans l'Admin

Pour éviter que l'administrateur ne manipule du code JSON brut, l'interface Django Admin sera adaptée :

Ajout & Ordre (Inline Admin) :

Dans l'écran d'édition d'une Page, les Sections apparaissent sous forme de liste "Inline".

Utilisation de la librairie django-admin-sortable2 : L'administrateur peut réorganiser l'ordre des sections par simple Drag & Drop (glisser-déposer) visuel.

Édition du contenu (JSON Editor) :

Le champ content ne s'affiche pas comme un bloc de texte brut.

Intégration d'un widget JSON convivial (ex: django-json-widget ou un éditeur de formulaire dynamique).

L'éditeur affiche des champs clairs : "Titre (FR)", "Titre (EN)", "Image", "Lien Bouton".

À la sauvegarde, ces champs sont serialisés en un objet JSON valide stocké dans PostgreSQL.

## D. Exemple de structure de données (Stockée en base)

C'est ce format que l'API renverra au Frontend. Notez la structure multilingue native.

code

JSON

```
// Exemple pour une section de type "Hero"
```

```
{
  "title": {
    "fr": "Transformez votre entreprise",
    "en": "Transform your business"
  },
  "subtitle": {
    "fr": "Des solutions digitales sur mesure.",
    "en": "Tailor-made digital solutions."
  },
  "background_image": "https://res.cloudinary.com/demo/image/upload/v1/hero-bg.jpg",
  "cta_primary": {
    "label": {"fr": "Commencer", "en": "Get Started"},
    "url": "/contact"
  }
}
```

## E. Validation des Données

Le Backend applique une validation stricte via les Serializers DRF avant la sauvegarde :

Vérification que les champs obligatoires pour un type de section donné sont présents (ex: une section video doit avoir une video\_url).

Nettoyage des entrées textes (Sanitization) pour empêcher l'injection de scripts malveillants (XSS).

## **4.3. Gestion des Contenus Structurés (Services, Projets, Blog)**

Contrairement aux "Pages" flexibles basées sur des blocs JSON, ces contenus nécessitent une structure de base de données relationnelle rigoureuse (SQL standard) pour permettre le filtrage, le tri et des requêtes complexes.

### **A. Module "Services"**

Gère l'offre commerciale de l'entreprise.

- **Modèle de Données (Champs) :**

- Nom (FR/EN) : Champ texte court.
- Slug : Identifiant URL unique (auto-généré).
- Icône : Upload SVG ou sélection depuis une librairie (ex: Lucide/FontAwesome).
- Résumé (FR/EN) : Texte court pour les cartes d'aperçu sur l'accueil.
- Description Complète (FR/EN) : **Rich Text** (HTML simplifié : gras, listes, titres) pour la page de détail.
- Ordre d'affichage : Entier pour trier la liste manuellement.

## B. Module "Projets" (Portfolio)

Gère les cas clients et réalisations.

- **Modèle de Données (Champs) :**

- Titre du projet : Texte.
- Client : Nom du client ou de l'entreprise.
- Date de réalisation : Date (permet le tri chronologique).
- Image de couverture : Visuel principal pour le listing.
- Galerie d'images : Relation *Many-to-Many* ou champ multiple pour le carrousel de détail.
- Catégorie : Relation vers une table Category (ex: Web, Mobile, SEO) pour le filtrage.
- Tags : Mots-clés (ex: Python, React) affichés sous forme de badges.
- Challenge & Solution (FR/EN) : Zones de texte riche pour le storytelling.
- Lien externe : URL vers le site du client (optionnel).

## C. Module "Blog" (CMS Editorial)

- **Modèle de Données (Champs) :**

- Titre (H1), Slug, Auteur, Date de publication.
- Contenu : Éditeur WYSIWYG complet (ex: **CKEditor** ou **TipTap**) permettant l'insertion d'images au fil du texte.
- Statut : Brouillon / Publié / Archivé.

## D. Fonctionnalités de Liste (Admin)

Pour ces trois modules, l'interface de liste Django propose :

- **Barre de recherche** : Recherche par titre ou client.
- **Filtres latéraux** : Filtrer par date, statut ou catégorie.
- **Actions de masse** : "Supprimer la sélection" ou "Publier la sélection".

#### **4.4. Gestion des Médias (Intégration Cloudinary)**

Le Backend ne stocke **aucun fichier lourd**. Il agit comme un gestionnaire de références vers Cloudinary.

##### **A. Intégration Technique**

- Utilisation du package **django-cloudinary-storage**.
- Configuration dans settings.py pour remplacer le DEFAULT\_FILE\_STORAGE de Django.

##### **B. Expérience Utilisateur (Upload)**

1. **Widget d'Upload** : Dans l'admin, les champs ImageField classiques sont remplacés par un champ d'upload Cloudinary.
2. **Processus :**
  - L'administrateur sélectionne un fichier sur son ordinateur.
  - Le fichier est envoyé directement du navigateur vers Cloudinary (ou via le backend en stream si nécessaire pour la validation).
  - Cloudinary renvoie un objet JSON contenant l'ID public, l'URL, le format et les dimensions.
  - Django ne sauvegarde que la référence (ex: image/upload/v161234/mon-image.jpg) dans la base PostgreSQL.
3. **Prévisualisation** : L'admin affiche une miniature de l'image (thumbnail) à côté du champ d'upload pour confirmer visuellement le fichier choisi.

##### **C. Optimisation Automatique**

Le Backend n'a pas besoin de redimensionner les images. C'est le Frontend (Next.js) qui demandera le format idéal en ajoutant des paramètres à l'URL stockée (ex: .../upload/w\_800,q\_auto,f\_auto/...).

#### **4.5. Traduction Assistée par IA**

Pour accélérer la gestion du multilinguisme (FR ↔ EN), une assistance IA est intégrée directement dans le workflow d'édition.

##### **A. Workflow "Human-in-the-loop"**

L'IA ne remplace pas l'humain, elle le pré-maché.

1. **Saisie** : L'éditeur remplit tous les contenus en **Français** uniquement.
2. **Action** : Il clique sur un bouton "Générer les traductions (EN)" présent dans la barre d'outils de l'objet ou de la page.
3. **Traitement (Backend)** :
  - Django détecte les champs EN vides.
  - Il envoie les textes FR correspondants à **OpenRouter** (modèle GPT-4o-mini ou Claude 3 Haiku pour la rapidité/coût).
  - Prompt système : "*Tu es un traducteur professionnel B2B. Traduis ce contenu JSON du français vers l'anglais en gardant le ton corporate et la structure HTML/Markdown intacte.*"
4. **Résultat** : Les champs EN sont remplis automatiquement dans le formulaire.
5. **Validation** : La page se recharge, l'éditeur relit les traductions anglaises, corrige si besoin, et sauvegarde.

## B. Scope de traduction

Cette fonctionnalité s'applique :

- Aux champs textes simples (Titres, résumés).
- Au contenu des blocs JSON des sections dynamiques.
- Aux champs Rich Text des articles/projets.

## 4.6. Paramètres Globaux (Singleton)

Certains réglages concernent l'ensemble du site et ne doivent pas être dupliqués. On utilise un **Pattern Singleton** (un seul objet "Configuration" existe en base).

### A. Module "Paramètres Généraux"

Accessible via un lien unique "Configuration du site" dans le menu Admin.

- **Identité Visuelle** :
  - Upload Logo (Header).
  - Upload Logo (Footer/Inversé).
  - Upload Favicon.
- **Coordonnées (Contact)** :
  - Email de contact (destinataire des formulaires).
  - Numéro de téléphone (affiché dans le header).
  - Adresse physique (affichée dans le footer et la map).

- Lien Google Maps.
- **Réseaux Sociaux :**
  - Liste de liens (Facebook, LinkedIn, X, Instagram). Si un champ est vide, l'icône ne s'affiche pas sur le site.
- **SEO & Technique Global :**
  - Suffixe Titre : (ex: " | NomEntreprise").
  - Image OG par défaut : Image de partage utilisée si une page n'en a pas de spécifique.
  - Scripts Header : Champ texte pour coller les codes de tracking (Google Analytics, Pixel FB) qui seront injectés dans le <head> du Frontend.

## 5. Spécifications Détaillées du Chatbot RAG (IA)

Le Chatbot n'est pas un simple gadget "discutant". Il est conçu comme un **Assistant Commercial Virtuel**. Il doit comprendre le métier de l'entreprise (grâce à ses données) et convertir les visiteurs curieux en prospects qualifiés (leads).

### 5.1. Logique Fonctionnelle

Pour maximiser la conversion, le chatbot adopte une stratégie d'engagement progressif ("Give then Take").

#### A. Le Scénario d'Engagement (The "Soft-Wall")

Le système suit une machine à états finis gérée par le Backend (via l'ID de session ou un cookie) :

1. **Phase "Séduction" (Messages 1 à 3) :**
  - L'accès est libre et anonyme.
  - L'utilisateur pose des questions ("Quels sont vos tarifs ?", "Faites-vous du SEO ?").
  - L'IA répond instantanément en utilisant les données de l'entreprise.
2. **Phase "Conversion" (Message 4) :**
  - L'utilisateur envoie son message.

- Le système bloque la réponse de l'IA.
- Une interface (Modale ou Message système) apparaît : "Pour continuer cet échange détaillé et recevoir une réponse précise, merci de nous indiquer votre email."

### 3. Phase "Qualification" (Post-Email) :

- Dès que l'email est validé, il est enregistré en base (Table Lead).
- L'IA génère la réponse à la question qui était en attente.
- La conversation reprend sans limite, avec un contexte enrichi.

## B. Personnalité du Bot (System Prompt)

Une instruction système (non visible par l'utilisateur) définit le comportement de l'IA :

- *Rôle* : "Tu es un expert consultant senior de l'entreprise [Nom]."
- *Ton* : Professionnel, concis, serviable, orienté solution.
- *Contrainte* : "Si tu ne trouves pas la réponse dans le contexte fourni, admetts-le et propose de contacter l'équipe humaine. N'invente jamais de faits."

## 5.2. Architecture RAG (Retrieval-Augmented Generation)

Pour éviter les "hallucinations" et garantir que l'IA connaisse les projets spécifiques de l'entreprise, nous utilisons une architecture **RAG**.

### A. Indexation des Contenus (L'ingestion)

Un processus automatique s'exécute à chaque fois qu'un admin modifie une page Service ou Projet dans le Dashboard Django.

1. **Extraction** : Le texte pertinent (Titre, Description, Challenge, Solution) est extrait.
2. **Chunking** : Le texte est découpé en morceaux (chunks) de ~500 caractères.
3. **Vectorisation (Embeddings)** :
  - Le Backend envoie ces morceaux à l'API d'OpenRouter (modèle d'embedding type text-embedding-3-small ou équivalent).
  - L'API renvoie un vecteur mathématique (une liste de nombres flottants) représentant le *sens* du texte.
4. **Stockage** :
  - Les vecteurs sont stockés dans PostgreSQL.
  - *Note technique cPanel* : Si l'extension pgvector n'est pas disponible sur l'hébergement mutualisé, les vecteurs seront stockés dans un champ JSONB et

la recherche de similarité (Cosine Similarity) sera effectuée en Python (via NumPy/Scikit-learn) étant donné le volume de données modeste (< 1000 chunks).

## B. Recherche Vectorielle et Construction de Réponse (Le Run)

Lorsqu'un utilisateur pose une question :

1. **Vectorisation Query** : La question de l'utilisateur est convertie en vecteur.
2. **Recherche (Retrieval)** : Le système compare ce vecteur à ceux stockés en base pour trouver les 3 morceaux de texte les plus "proches" (pertinents).
3. **Injection de Contexte** : Le Backend construit le prompt final envoyé au LLM :

"Voici des informations sur l'entreprise : [Morceau 1] [Morceau 2] [Morceau 3].

En utilisant ces infos, réponds à la question suivante : [Question Utilisateur]"

### 5.3. Interface Chat Widget (Frontend)

Le composant React (`<ChatWidget />`) doit être léger, non intrusif et fluide.

#### A. UI / UX

- **Launcher** : Bouton flottant (FAB) en bas à droite de l'écran avec une icône de bulle. Badge de notification rouge possible pour attirer l'attention ("1").
- **Fenêtre de Chat** :
  - Header : Nom du bot + Bouton fermer.
  - Body : Liste des messages (Bulles grises à gauche pour l'IA, Bulles bleues à droite pour l'utilisateur). Support du Markdown (listes à puces, gras) pour les réponses de l'IA.
  - Input Zone : Champ texte + Bouton envoi.
- **État "Typing"** : Affichage de trois points animés (...) pendant que le Backend traite la demande.

### B. Gestion de l'état (State Management)

- L'historique de la conversation est stocké localement dans le localStorage du navigateur (pour retrouver la conversation si l'utilisateur change de page) ET synchronisé avec le backend si l'utilisateur est un Lead identifié.

### 5.4. Sécurité et Proxy API (Backend)

C'est la clé de voûte de la sécurité du système IA.

#### A. Proxy Django (Le Douanier)

Le Frontend n'a **JAMAIS** accès à la clé API OpenRouter.

1. Le Frontend fait un POST vers <https://api.tonsite.com/api/chat/message/>.
2. **Validation** : Django vérifie que le message n'est pas vide et ne dépasse pas une certaine taille.
3. **Rate Limiting** : Django vérifie l'IP de l'utilisateur (via django-ratelimit) pour empêcher les abus (ex: max 10 messages / minute).
4. **Appel LLM** : Django contacte OpenRouter avec la clé secrète stockée dans les variables d'environnement (.env).
5. **Logging** : Django enregistre la question et la réponse (si l'option logs est activée) pour audit ultérieur.

## B. Protection des Coûts

Pour éviter une facture OpenRouter explosive :

- Limitation de la longueur des réponses (paramètre max\_tokens).
- Choix d'un modèle performant mais économique pour le MVP (ex: gpt-4o-mini ou meta-llama-3-70b).
- Budget cap (limite de dépense mensuelle) configuré directement côté OpenRouter.

## 6. Modélisation des Données (PostgreSQL)

### 6.1. Approche Hybride : Explication du concept

Le projet refuse de choisir entre la rigidité d'une base de données SQL classique et le chaos d'une base NoSQL. Nous adoptons une **architecture hybride** pour répondre aux deux besoins contradictoires du site :

#### 1. Le besoin de Structure (SQL Relationnel) :

Certaines données sont critiques, doivent être triées, filtrées et liées entre elles. Un "Projet" appartient à une "Catégorie". Un "Utilisateur" a des "Permissions". Ces données nécessitent des contraintes d'intégrité (Clés Étrangères) et des colonnes typées fixes.

#### 2. Le besoin de Flexibilité (JSONB / NoSQL) :

La mise en page d'une page d'accueil ou d'une landing page change tout le temps. Ajouter un bouton, changer un titre, intervertir une image et une vidéo ne doit pas nécessiter une modification de la structure de la table (migration de base de

données). Ces données de *présentation* sont stockées sous forme de documents JSON binaires.

**L'avantage technique :** PostgreSQL permet d'indexer les champs JSONB. Cela signifie que nous pourrons effectuer des recherches performantes même à l'intérieur des structures flexibles.

## 6.2. Schéma Relationnel (Entités Structurées)

Ces tables définissent le "squelette" immuable de l'application. Elles suivent les conventions Django (Snake\_case pour les champs).

### A. Table : User (Utilisateurs système)

Hérite de AbstractUser de Django.

- id (PK) : UUID ou Auto-increment.
- username : Identifiant de connexion.
- email : Unique, obligatoire.
- role : Enum (SuperAdmin, Editeur).
- is\_active : Booléen (Banissement/Désactivation).

### B. Table : Category (Taxonomie)

Permet de classer les Projets et les Articles de blog.

- id (PK) : Integer.
- name : Varchar (ex: "Développement Web", "Design"). *Note: Peut être dupliqué en name\_fr / name\_en.*
- slug : SlugField (Unique, indexé). ex: développement-web.

### C. Table : Service (Offre commerciale)

- id (PK) : Integer.
- slug : SlugField (Unique).
- title\_fr / title\_en : Varchar(255).
- short\_description\_fr / short\_description\_en : Text (pour les cartes).
- content\_fr / content\_en : RichText (HTML).
- icon\_name : Varchar (Référence à une icône Lucide/FontAwesome).
- is\_visible : Boolean (Default: True).
- sort\_order : Integer (Pour gérer l'affichage : 10, 20, 30...).

#### D. Table : Project (Portfolio)

- id (PK) : Integer.
- category\_id (FK) : Clé étrangère vers Category (ON\_DELETE=PROTECT).
- slug : SlugField (Unique).
- title : Varchar(255).
- client\_name : Varchar(255).
- completion\_date : Date.
- cover\_image\_url : Varchar (URL Cloudinary).
- gallery\_images : **ArrayField** (Postgres) ou **JSONB**. Liste d'URLs d'images secondaires.
- description\_fr / description\_en : RichText (Challenge, Solution, Résultats).
- technologies : JSONB (Liste de tags ex: ["React", "Django", "AWS"]).
- external\_link : URL (Optionnel).
- seo\_meta\_title / seo\_meta\_description : Varchar (Optimisation SEO).

### 6.3. Schéma Dynamique (Entités Flexibles)

C'est ici que réside la logique "Page Builder".

#### A. Table : Page (Le Conteneur)

Représente une URL publique du site.

- id (PK) : Integer.
- slug : SlugField (Unique, ex: accueil, a-propos, contact).
- title\_internal : Varchar (Nom pour l'admin).
- is\_published : Boolean.
- created\_at, updated\_at : DateTime.

#### B. Table : Section (Les Blocs de construction)

Chaque ligne de cette table est un "étage" d'une page.

- id (PK) : Integer.
- page\_id (FK) : Clé étrangère vers Page (related\_name='sections').
- section\_type : Varchar / Enum (ex: hero, features, cta, stats, testimonials). Ce champ indique au Frontend quel composant React charger.
- order : Integer (Indispensable pour le Drag & Drop. Indexé).

- `is_visible` : Boolean.
- `settings` : **JSONB**. Paramètres de style non-textuels.
  - *Exemple* : `{"background_color": "bg-gray-100", "padding_y": "py-12", "text_align": "center"}`.
- `content` : **JSONB**. Le contenu éditorial pur.

### C. Structure détaillée du champ JSONB (content)

Le contenu de ce champ varie selon le `section_type`. PostgreSQL ne valide pas ce schéma par défaut, la validation se fait via les *Serializers Django*.

*Exemple 1 : Type = hero*

codeJSON

```
{
  "title": {
    "fr": "L'innovation à votre portée",
    "en": "Innovation within reach"
  },
  "subtitle": {
    "fr": "Nous créons des solutions digitales uniques.",
    "en": "We craft unique digital solutions."
  },
  "media_type": "image", // ou "video"
  "media_url": "https://res.cloudinary.com/.../hero.jpg",
  "cta_button": {
    "text": {"fr": "Voir nos projets", "en": "See our work"},
    "link": "/projets"
  }
}
```

*Exemple 2 : Type = features\_grid (Liste d'éléments)*

codeJSON

```
{
```

```

"section_title": {
    "fr": "Pourquoi nous choisir ?",
    "en": "Why choose us?"
},
"items": [
{
    "icon": "zap",
    "title": {"fr": "Rapidité", "en": "Speed"},
    "desc": {"fr": "Déploiement en 24h", "en": "24h Deployment"}
},
{
    "icon": "shield",
    "title": {"fr": "Sécurité", "en": "Security"},
    "desc": {"fr": "Données cryptées", "en": "Encrypted data"}
}
]
}

```

#### **Note importante pour le développeur Backend :**

L'utilisation de JSONB permet de faire des requêtes du type : "*Trouve-moi toutes les sections dont le titre FR contient le mot 'Innovation'*".

Syntaxe SQL : SELECT \* FROM app\_section WHERE content->'title'->>'fr' ILIKE '%Innovation%'.

## **7. API REST & Interfaces**

L'application repose intégralement sur une architecture **API First**. Le Backend expose une interface programmable (API) conforme aux standards REST (Representational State Transfer).

## 7.1. Standards API (REST, JSON, Codes HTTP)

Pour garantir la prédictibilité et la maintenance, l'API respecte des conventions strictes.

### A. Format et Protocole

- **Protocole :** HTTPS uniquement (TLS 1.2+).
- **Format d'échange :** JSON (Content-Type: application/json).
- **Encodage :** UTF-8 (Support total des caractères accentués et emojis).
- **Date et Heure :** Format ISO 8601 UTC (ex: 2023-12-25T14:30:00Z).
- **Préfixe d'URL :** Toutes les routes sont préfixées par /api/v1/ pour gérer le versioning futur.

### B. Structure de Réponse Standard

Toute réponse (succès ou erreur) doit suivre une enveloppe cohérente ou des standards de codes HTTP clairs.

- **Succès (200 OK) :** Retourne directement l'objet ou la liste demandée.
- **Pagination (Listes) :** Pour les listes (Services, Projets), utilisation du standard LimitOffset ou PageNumber.

codeJSON

```
{  
  "count": 42,  
  "next": "https://api...?page=3",  
  "previous": "https://api...?page=1",  
  "results": [ ... ]  
}
```

### C. Codes HTTP (Signification)

Le Frontend doit écouter ces codes pour adapter l'interface (ex: afficher un toaster d'erreur).

- 200 OK : Requête traitée avec succès.
- 201 Created : Ressource créée (après un POST).
- 204 No Content : Suppression réussie ou requête traitée sans réponse attendue.
- 400 Bad Request : Erreur de validation (ex: email invalide, champ manquant). Le corps de réponse contient le détail des champs en erreur ({"email": ["Ce champ est requis."]}).

- 401 Unauthorized : Token JWT invalide ou manquant (Utilisateur non connecté).
- 403 Forbidden : Token valide mais droits insuffisants (ex: un Editeur qui essaie de supprimer un user).
- 404 Not Found : Ressource inexistante (Page ou ID introuvable).
- 500 Server Error : Bug côté Python.

## 7.2. Authentification API (JWT)

L'API est "Stateless" (sans état). Le serveur ne garde pas de session utilisateur en mémoire RAM (préférable sur un hébergement partagé comme cPanel). Nous utilisons **JWT (JSON Web Tokens)** via la librairie `djangorestframework-simplejwt`.

### A. Mécanisme

1. **Login** : L'admin envoie username + password via POST `/api/v1/token/`.
2. **Délivrance** : Le serveur renvoie deux tokens :
  - `access_token` (Durée de vie courte : 15 min). Utilisé pour authentifier chaque requête.
  - `refresh_token` (Durée de vie longue : 24h ou 7j). Utilisé pour obtenir un nouveau `access_token` sans se reconnecter.
3. **Transport** : Le Frontend stocke ces tokens de manière sécurisée (Idéalement Cookie HttpOnly pour le refresh token, ou en mémoire/localStorage pour l'`access token`).
4. **Requête Authentifiée** : Chaque appel vers une route protégée doit inclure le Header : `Authorization: Bearer <votre_access_token>`

## 7.3. Cartographie des Endpoints Principaux

Voici la liste non-exhaustive des routes à implémenter.

### A. API Publique (Lecture seule - Accès libre)

Ces routes sont utilisées par Next.js pour construire le site (SSR/SSG).

Méthode	Endpoint (URL relative)	Description
GET	<code>/pages/{slug}/</code>	Récupère une page complète (méta-données + liste des <b>sections</b> ordonnées).
GET	<code>/services/</code>	Liste des services (titre, icône, résumé).

<b>GET</b>	/services/{slug}/	Détail complet d'un service.
<b>GET</b>	/projects/	Liste des projets (filtrable via ?category=x).
<b>GET</b>	/projects/{slug}/	Détail complet d'un projet (galerie incluse).
<b>GET</b>	/settings/global/	Récupère le singleton "Configuration" (Logo, Liens réseaux, Contact info).
<b>GET</b>	/navigation/	Récupère l'arbre du menu principal et du footer.

## B. API Admin (Lecture/Écriture - Protégé JWT)

Ces routes alimentent le Dashboard. Seuls les SuperAdmin/Éditeurs y ont accès.

Méthode	Endpoint	Description
<b>POST</b>	/media/upload/	Upload d'un fichier vers Cloudinary (Proxy) et création de l'objet média.
<b>POST</b>	/ai/translate/	Envoie un objet JSON (contenu FR), appelle OpenRouter, retourne la version EN.
<b>POST</b>	/pages/{id}/sections/reorder/	Reçoit une liste d'IDs de sections [12, 5, 8] pour mettre à jour leur ordre en masse.
<b>CRUD</b>	/admin/pages/	Gestion des pages.
<b>CRUD</b>	/admin/sections/	Gestion individuelle d'une section.
<b>CRUD</b>	/admin/projects/	Gestion des projets.
<b>GET</b>	/admin/leads/	Liste des emails capturés par le chatbot.

## C. API Interactive & Chatbot

Routes permettant l'interaction utilisateur.

Méthode	Endpoint	Description
---------	----------	-------------

<b>POST</b>	/contact/form/	Réception du formulaire de contact classique. Envoi d'email via SMTP + Log en base. Payload: {name, email, message}.
<b>POST</b>	/chat/session/	Initialise une nouvelle session de chat (retourne un session_id).
<b>POST</b>	/chat/message/	Envoi d'un message utilisateur.   <b>Input:</b> {session_id, message_text}.  <b>Process:</b> RAG Search + OpenRouter Call.  <b>Output:</b> {reply_text, sources[]} (Stream possible ou réponse bloquante).
<b>POST</b>	/chat/lead/	Associe un email à une session de chat existante (Conversion du prospect).