Clone GPT – Rapport de Projet

Fait par ***Mwiseneza munezero Esther***

Dans le cadre du cours de projet-de-developpement-sgbd,

donné par le professeur ***Six Thibault Six***

1. Introduction

Contexte et objectifs

Dans le cadre du cours de projet-de-developpement-sgbd, le projet Clone GPT (ici appelé MiniGPT) vise à développer une application web permettant d'interagir avec une intelligence artificielle de type ChatGPT. L'objectif est de reproduire les principales fonctionnalités d'un assistant conversationnel basé sur un modèle de langage avancé, offrant une expérience utilisateur fluide et intuitive. Ce projet répond au besoin croissant d'intégrer l'IA dans des applications web modernes pour améliorer l'interaction homme-machine.

Périmètre du projet

Le périmètre couvre le développement d'une interface utilisateur avec Vue.js, la gestion des conversations et modèles via un backend Laravel, la persistance des données en base SQL, ainsi que l'intégration avec une API externe d'IA (OpenAI).

Technologies utilisées

Backend : Laravel 12

Frontend : Vue.js 3 (Composition API), TailwindCSS

Base de données : MySQL

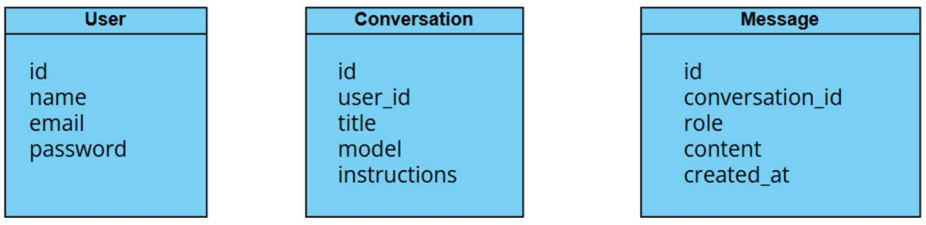
API IA : OpenAI GPT-4 (via clé API REST)

Tests : Laravel Dusk (tests fonctionnels simples)

2. Architecture et Conception

2.1 Base de données

Le diagramme UML comporte 3 tables principales:



- Users (id, nom, email, mot de passe)
- Conversations (id, user_id, titre)

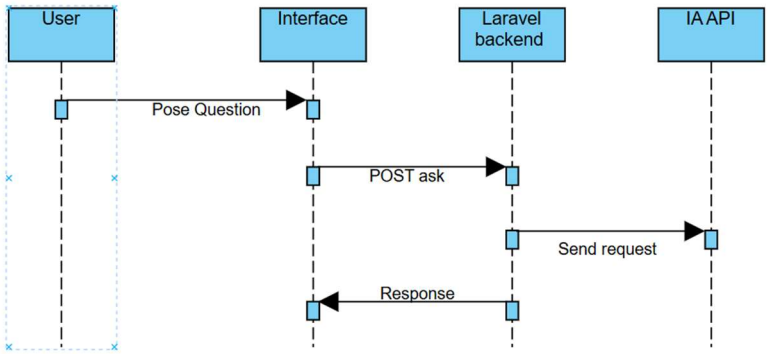
- Messages (id, conversation_id, contenu, rôle, date)

Relations :

- Un utilisateur peut avoir plusieurs conversations (relation 1-N).
- Une conversation contient plusieurs messages (relation 1-N).
- Les messages ont un rôle (utilisateur ou assistant) pour différencier l'émetteur.

Contraintes :

- Clés primaires et étrangères respectées.
- Suppression en cascade des messages lors de la suppression d'une conversation.
- Unicité sur l'email utilisateur.



2.2 Architecture logicielle

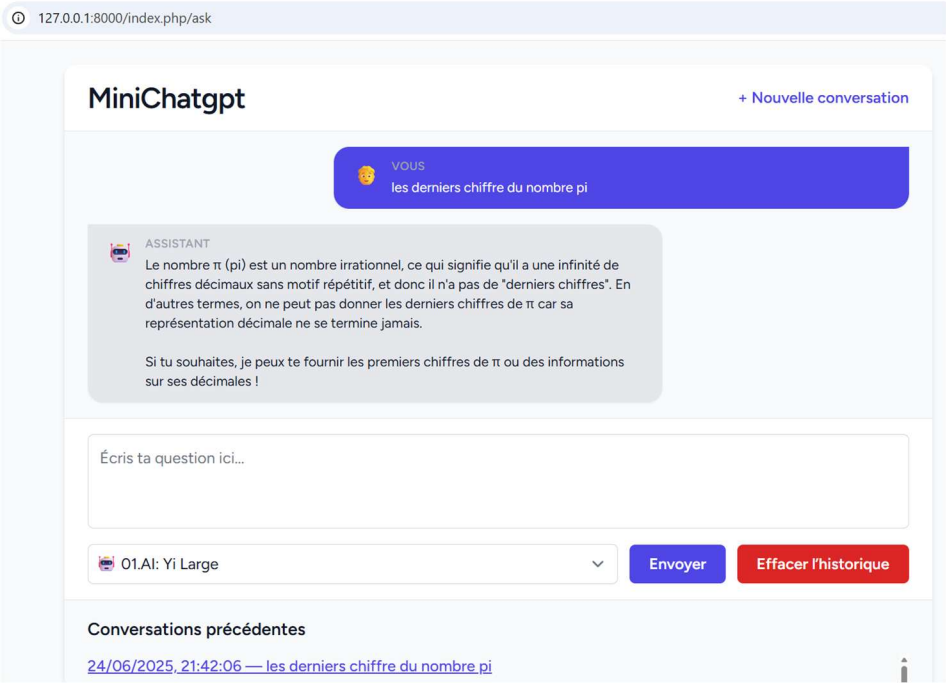
L'architecture repose sur une séparation claire des responsabilités entre :

- Le frontend (Vue 3 + Inertia.js + TailwindCSS)
- Le backend (Laravel 12)

Respectant les principes du modèle MVC, le code est organisé de façon modulaire et maintenable.

2.3 Frontend – Vue.js 3 + Inertia.js + TailwindCSS

Le composant principal de l'interface de chat est situé dans :
resources/js/Pages/Ask/Index.vue



Fonctionnalités intégrées:

- Envoi de messages à l'IA
- Sélection du modèle IA (GPT-3.5, GPT-4, etc.)
- Affichage de l'historique d'une conversation
- Chargement d'une conversation existante(plus haute dans la conversation globale)
- Création d'une nouvelle conversation
- Effacement complet de l'historique

Expérience utilisateur :

- Scroll automatique vers le bas lors de l'ajout de messages
- Animation progressive des messages
- Indicateur de chargement animé
- Design simple et grâce à TailwindCSS

Dans l'interface que j'ai développée, j'ai fait en sorte que l'utilisateur puisse accéder directement à l'historique de ses conversations dès la page d'accueil, sans devoir naviguer vers une section séparée. Cela m'a permis de me focaliser davantage sur la

logique backend et le fonctionnement global de l'application, tout en assurant une expérience utilisateur simple et efficace.

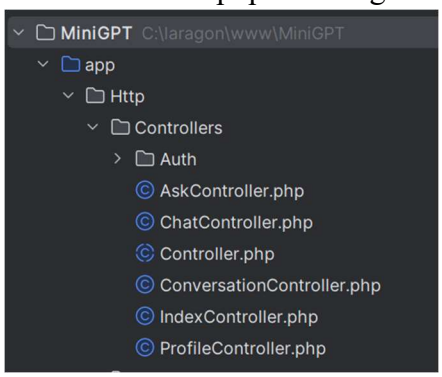
Les titres des conversations sont générés automatiquement en fonction du premier message envoyé, ce qui facilite leur identification.

Cependant, faute de temps, je n'ai pas pu mettre en place une gestion de conversations réellement séparées avec un historique dédié. Actuellement, les échanges s'enchaînent dans un seul fil. Une amélioration future consisterait à permettre la création et la navigation entre plusieurs conversations distinctes.

2.4 Backend – Laravel 12 (MVC)

Contrôleurs principaux :

- AskController.php : index(), send(), newConversation(), clear()
- ConversationController.php : show(Conversation $conversation)
- IndexController.php : Redirige vers la page par défaut du chat.



Utilisation du service ChatService

Dans le cadre du projet, j'ai utilisé un service dédié appelé ChatService, tel que présenté dans le cours. Ce service a pour rôle central d'encapsuler la logique d'interaction avec l'API OpenAI, permettant de séparer proprement les appels réseau de la gestion des données et de la logique métier dans les contrôleurs.

Concrètement, ChatService gère :

- La construction des requêtes vers l'API GPT, en incluant les paramètres comme le modèle choisi et le contenu des messages.
- L'envoi des requêtes à OpenAI via la clé API sécurisée.
- Le traitement des réponses reçues, notamment le formatage et la gestion des erreurs potentielles.

Cette abstraction facilite la maintenance du code, améliore sa lisibilité, et permet de centraliser les modifications liées à l'API dans un seul endroit.

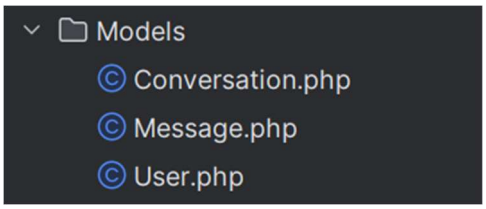
Ainsi, dans le contrôleur AskController, au lieu d'appeler directement l'API, j'utilise le service ChatService pour envoyer les messages et récupérer les réponses, ce qui améliore la séparation des responsabilités et la testabilité de l'application.

2.5 Modèles et base de données

Modèles principaux :

- Conversation.php
- Message.php

Relations : Chaque conversation liée à plusieurs messages.



2.6 Sécurité – Policies

L'accès aux conversations est normalement protégé via des politiques d'autorisation Laravel, ce qui garantit que seul l'utilisateur propriétaire peut consulter ses données. Par exemple, on utilise généralement la commande suivante dans les contrôleurs :

```
$this->authorize('view', $conversation);
```

Cependant, en raison d'erreurs techniques rencontrées lors du développement, notamment liées à la gestion des policies et à la synchronisation avec Inertia.js, j'ai été contraint de commenter temporairement cette ligne. Cette décision a été prise pour assurer le bon fonctionnement global de l'application et éviter des blocages lors de l'affichage des conversations.

Cette mesure provisoire sera corrigée dans une future version afin de garantir pleinement la sécurité et le respect des droits d'accès des utilisateurs.

2.7 Routes (web.php)

Toutes les routes sont regroupées dans un middleware auth et verified :

```php
// Page d'accueil redirigée vers le chat si l'utilisateur est connecté
Route::get( uri: '/', [IndexController::class, 'index'])
    ->middleware(['auth', 'verified'])
    ->name( name: 'index');
```

```php
// Routes accessibles uniquement après authentification
Route::middleware(['auth', 'verified'])->group(function () {

    // Nouvelle conversation
    Route::post( uri: '/ask/new-conversation', [AskController::class, 'newConversation'])->name( name: 'ask.newConversation');

    // Page principale du chat
    Route::get( uri: '/ask', [AskController::class, 'index'])->name( name: 'ask.index');

    // Envoi d'un message à l'IA
    Route::post( uri: '/ask', [AskController::class, 'send'])->name( name: 'ask.send');

    // Affichage d'une conversation spécifique
    Route::get( uri: '/conversations/{conversation}', [ConversationController::class, 'show'])->name( name: 'conversations.show');

    // Suppression de l'historique complet
    Route::delete( uri: '/history/clear', [AskController::class, 'clear'])->name( name: 'history.clear')->middleware( middleware: 'auth');

    // Page "Chat"  /ask  (peux la supprimer si inutile)
    Route::get( uri: '/chat', [IndexController::class, 'index'])->name( name: 'chat');

    // Dashboard
    Route::get( uri: '/dashboard', fn () => Inertia::render( component: 'Dashboard'))->name( name: 'dashboard');

    // Gestion du profil utilisateur
    Route::get( uri: '/profile', [ProfileController::class, 'edit'])->name( name: 'profile.edit');
    Route::patch( uri: '/profile', [ProfileController::class, 'update'])->name( name: 'profile.update');
    Route::delete( uri: '/profile', [ProfileController::class, 'destroy'])->name( name: 'profile.destroy');
});
```

```php
// Authentification (Jetstream ou Breeze)
require __DIR__ . '/auth.php';
```

2.8 Authentification

Basée sur Laravel Jetstream/Breeze avec Inertia.js, offrant :

- Inscription, connexion, vérification d'email
- Gestion native des sessions

3. Fonctionnalités développées

Fonctionnalités

- Envoi d'un message utilisateur
- Requête à l'API OpenAI
- Affichage de la réponse

- Gestion simple de conversation
- Responsive design simple

Défis techniques et solutions

- Gestion asynchrone des appels API
- Affichage des réponses après réception
- Synchronisation frontend/backend

4. Tests et Qualité

Stratégie de tests

Pour valider les fonctionnalités principales de l'espace de discussion, j'ai mis en place une série de tests fonctionnels dans le fichier AskTest. Ces tests visent à garantir la stabilité et la sécurité des principales interactions côté utilisateur connecté ou non.

J'ai notamment vérifié que :

- un utilisateur connecté peut accéder à la page d'accueil du chat (ask.index) et que le bon composant Vue est bien chargé (Ask/Index) ;
- un utilisateur non connecté est automatiquement redirigé vers la page de connexion, ce qui confirme la protection des routes par middleware d'authentification ;
- un utilisateur peut créer une nouvelle conversation, testant ainsi la route ask.newConversation ;
- l'envoi d'un message et la réception de la réponse depuis le modèle IA fonctionnent correctement, sans erreurs de validation.

Grâce à cette stratégie de tests ciblés, j'ai pu m'assurer que les flux principaux de l'application étaient robustes et conformes aux attentes fonctionnelles, tout en couvrant les cas d'usage courants.

Tests implémentés

- Test de connexon utilisateur

```php
#[Test]
public function un_utilisateur_connecte_peut_acceder_a_la_page_ask()
{
    $user = User::factory()->create();

    $response = $this->actingAs($user)->get(route( name: 'ask.index'));

    $response->assertStatus( status: 200);

    $response->assertJsonFragment(['component' => 'Ask/Index']);
}
```

```php
#[Test]
public function un_utilisateur_non_connecte_est_redirige()
{
    $response = $this->get(route( name: 'ask.index'));
    $response->assertStatus( status: 302);
    $response->assertRedirect(route( name: 'login'));
}
```

- Test d'envoi question IA

```php
#[Test]
public function un_message_est_envoye_et_reponse_recue()
{
    $user = User::factory()->create();

    $response = $this->actingAs($user)->post(route( name: 'ask.store'), [
        'question' => 'Bonjour IA',
        'model' => config( key: 'openrouter.models')[0]['id'] ?? 'default-model',
        'history' => [],
    ]);

    $response->assertStatus( status: 200); // ou 302 si redirection prévue
    $response->assertSessionHasNoErrors();
}
```

- Test de creation d'une nouvelle conversation

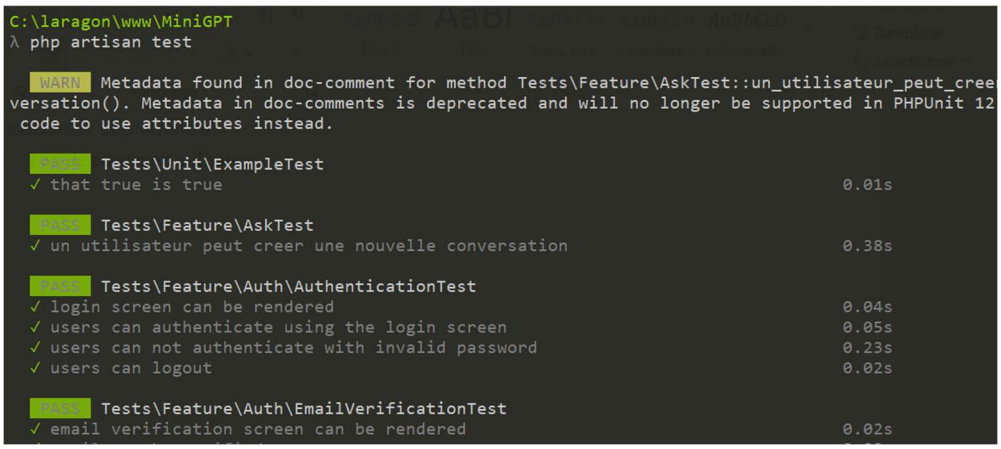
```php
no usages
#[Test]
public function un_utilisateur_peut_creer_une_nouvelle_conversation()
{
    $user = User::factory()->create();

    $response = $this->actingAs($user)->post(route( name: 'ask.newConversation'));

    $response->assertRedirect(); // redirection vers la nouvelle conversation
}
```

Résultats

- Tests du fichier AskTest valides
  Pas de blocage majeur



Perspectives d'amélioration

- Couverture tests API et validations
- Automatisation CI
- Tests frontend plus complets

5. Difficultés et Solutions

Gestion des routes et contrôleurs

Problèmes d'appels corrigés par liaison correcte.

Solution : Réorganisation des fichiers de contrôleurs et bonne utilisation des noms de routes (name()), avec regroupement dans le middleware auth et verified.

Mauvais chemin de composant Vue

Correction des chemins dans routes après déplacement.

Solution : Mise à jour systématique des imports dans les routes Inertia.js après chaque refactorisation ou déplacement de fichiers.

Sélection de modèle (LLM)

Ajout colonne model en migration et transmission correcte.

Solution : Ajout d'une colonne model dans la migration conversations, passage correct via le frontend, et persistance dans chaque échange

Erreurs base de données

Gestion des migrations doublons.

Solution : Suppression ou rollback des migrations problématiques, puis regroupement des colonnes nécessaires dans une seule migration propre.

Tests de rendu & liaison back/frontend

Synchronisation historique, modèle et affichage.

Solution : Synchronisation des données via props Inertia, stockage clair du modèle, de la conversation et du message utilisateur.

Historique & affichage dynamique

Rendu Markdown, coloration syntaxique, interface fluide.

Solution : Intégration de Markdown pour le rendu (via marked ou markdown-it) + coloration syntaxique avec highlight.js, pour une interface conversationnelle moderne.

Problèmes rencontrés

Gestion erreurs API, latence IA.

Latence IA : temps de réponse parfois long selon les modèles.

Erreurs API : appel échoué ou réponse malformée.

Solutions apportées

Affichage de messages d'erreur utilisateur en cas d'échec d'appel API.

Ajout d'un loader animé pendant le traitement de la réponse IA pour améliorer l'expérience utilisateur.

Prévention des doubles envois avec désactivation temporaire du bouton d'envoi.

Améliorations possibles

Intégration du streaming de réponse (mot par mot comme ChatGPT).

Gestion des multi-conversations plus avancée (titres personnalisés, épinglage, recherche).

Choix persistant du modèle IA entre les sessions.

Ajout de tests Laravel Dusk automatisés pour l'interface frontend.

6. Utilisation des outils IA

ChatGPT pour génération de code, OpenAI API pour backend.

Utilisation : codage accéléré, assistance redaction .

7. Conclusion

Bilan du projet

Ce projet MiniGPT m'a permis de mieux comprendre les enjeux liés à l'intégration d'une intelligence artificielle dans une application web. J'ai pu réaliser un prototype fonctionnel

qui valide les choix techniques et architecturaux que j'ai faits. Cette expérience constitue une base solide pour le développement futur d'outils conversationnels basés sur des modèles de langage avancés.

Apprentissages

Tout au long du développement, plusieurs compétences techniques clés ont été renforcées :

- La gestion des appels API asynchrones, essentielle pour communiquer efficacement avec l'IA et assurer une bonne expérience utilisateur.
- La maîtrise du framework Laravel pour structurer le backend selon le modèle MVC, garantissant une architecture solide et maintenable.
- L'utilisation de Vue.js avec la Composition API pour construire une interface frontend réactive et moderne.
- La compréhension approfondie de la synchronisation entre frontend et backend via Inertia.js, facilitant la fluidité des échanges et la cohérence des données.

Perspectives

Ce projet constitue une base solide, mais plusieurs axes d'amélioration sont envisagés pour en accroître la richesse fonctionnelle et la robustesse :

- La gestion multi-conversations pour permettre à l'utilisateur de naviguer entre plusieurs échanges sans perte de contexte.
- L'intégration du streaming des réponses, afin d'afficher les résultats en temps réel et améliorer la réactivité perçue.
- Le renforcement de l'interface utilisateur avec des animations plus avancées, une meilleure ergonomie, ainsi que le développement de tests unitaires et fonctionnels plus complets pour garantir la qualité et la fiabilité du système.

8. Corrections apportées suite aux remarque de la 1re sessions

Étape 1 — Erreur « Namespace declaration… »

- **Cause identifiée :** certains fichiers contenaient un BOM UTF-8 avant <?php, ce qui provoquait l'erreur de namespace. D'autres fichiers étaient corrompus avec des ... ou du code recopié.
- **Corrections appliquées :**
    - Suppression du BOM dans :
        - app/Http/Middleware/HandleInertiaRequest.php
        - database/seeders/UserSeeder.php
    - Réécriture propre des fichiers corrompus :
        - app/Http/Middleware/HandleInertiaRequests.php
        - app/Http/Controllers/AskController.php
        - app/Http/Controllers/ChatController.php
        - app/Http/Controllers/ConversationController.php
    - Ajout de fichiers manquants :
        - app/Http/Controllers/MessageController.php (suppression de message)
        - app/Services/ChatService.php (service minimal pour futur streaming)
        - app/Models/Conversation.php (reconstruit avec casts)
    - Nettoyage des fichiers de configuration :
        - routes/web.php (routes cohérentes)
        - database/migrations/2025_06_24_130746_create_conversations_table.php (vraie table conversations)

Étape 2 — Seeders : doublon d'email

- **Cause identifiée :** le seeder/factory ne garantissait pas l'unicité des emails.
- **Corrections appliquées :**

database/factories/UserFactory.php → utilisation de fake()->unique()->safeEmail()

database/seeders/DatabaseSeeder.php → création/maj d'un utilisateur test avec updateOrCreate + génération de 10 utilisateurs via factory

database/seeders/UserSeeder.php → nettoyé et corrigé avec même logique (updateOrCreate)

Étape 3 — UI : bouton « Nouvelle conversation »

- **Problème :** le bouton ne déclenchait aucune action.
- **Correction appliquée :**

Ajout d'une fonction Vue (createConversation) qui effectue un Inertia.post(route('conversations.store')) pour créer une nouvelle conversation.

Étape 4 — Suppression de messages

- **Problème :** la suppression n'était pas conforme à Inertia.
- **Correction appliquée :**

MessageController@destroy → renvoie maintenant redirect()->back()->with('success', 'Message supprimé').

Côté Vue → suppression via Inertia.delete(route('messages.destroy', message.id)).

Étape 5 — Absence de streaming LLM

- **Problème :** les réponses étaient envoyées en bloc (pas en flux).
- **Correction appliquée :**

Backend : ajout d'une méthode ChatController@stream utilisant StreamedResponse et envoi de chunks SSE (data: {...}).

Service : ChatService::stream() reconstruit pour renvoyer les morceaux de réponse du LLM.

Frontend Vue : mise en place d'un fetch() avec lecture du ReadableStream et affichage en temps réel de la réponse.

Cependant, les modifications en backend, n'ont pas réglé le soucis du streaming dont je n'ai pas pu trouver la solution, pour ce rapport.

Étape 6 — Instructions personnalisées (Cahier des charges)

- **Amélioration appliquée :**

Étape 7 — Gestion des conversations (esquisse)

- **Améliorations prévues :**

Génération automatique de titre (déjà esquissée).

Fonctions de renommage, duplication, archivage, recherche et pagination (à venir).

Bénéfices :
Ces corrections ont permis d'éliminer les erreurs de namespace et de seeders, de fiabiliser les migrations et seeders, de rendre le backend compatible avec Inertia, et d'avoir une compréhension à propos de l'implémentation du **streaming en temps réel des réponses LLM**, rapprochant l'application d'une expérience type ChatGPT.

Rapport de problèmes rencontrés avec Jetstream/Laravel

Voici les raisons pour lesquelles le projet n'a pas été réalisé avec jetstream.

Lors de l'installation et de la configuration de Jetstream avec Inertia, Vue.js, Vite et TailwindCSS, j'ai rencontré plusieurs difficultés techniques principalement liées à la gestion des dépendances front-end.

1. Conflit de versions de Vite

Je me suis rapidement heurté à des conflits de versions. Jetstream et ses plugins front-end étaient incompatibles avec la version de Vite installée par défaut dans le projet.
Initialement, j'avais une version trop récente de Vite (Vite 7), alors que la majorité des plugins (comme @vitejs/plugin-vue, @tailwindcss/vite et laravel-vite-plugin) ne géraient que Vite 4 ou 5.
Résultat : l'installation des modules échouait systématiquement avec des erreurs npm sur les dépendances ("peer dependencies").

2. Mauvais alignement des peer-dependencies

J'ai aussi constaté que certains plugins imposaient des versions très précises de Vite, rendant leur coexistence difficile. Je recevais régulièrement des messages tels que "could not resolve dependency" ou "conflicting peer dependency", qui bloquaient tout le processus d'installation.

3. Tentatives de nettoyage

Pour tenter de résoudre le problème, j'ai supprimé le dossier node_modules et le fichier package-lock.json, avant de relancer l'installation, mais sans succès. Le véritable problème venait du mauvais alignement des versions dans le package.json.

4. Dépannage avec npm legacy-peer-deps

Après plusieurs essais infructueux, j'ai finalement réussi à installer les dépendances en utilisant la commande :

npm install --legacy-peer-deps

Cela a fonctionné, en "forçant" npm à installer les dépendances malgré les conflits de versions, même si ce n'est pas la solution la plus propre à long terme.

5. Avertissements de vulnérabilité

Suite à cette installation, npm m'a signalé des vulnérabilités de sécurité de sévérité modérée. Ces avertissements sont courants en développement, mais ils signalent que certains packages ne sont pas parfaitement à jour ou sûrs.

Conclusion

Mes principales difficultés étaient liées à la compatibilité entre Vite et ses plugins dans l'écosystème Laravel/Jetstream. J'ai dû forcer l'installation des dépendances pour pouvoir avancer, et je reste attentif à d'éventuels soucis qui pourraient en découler lors du développement.

Solution : Si je veux une base vraiment stable, il sera judicieux d'aligner manuellement les versions dans le package.json avant d'aller plus loin. Ce qui pourrai me permettre de poursuivre le développement de l'application avec Laravel/Jetstream.