

5IDV3

Projet de développement SGBD

Thibault Six · Thomas Copin

62 chapitres

SUPPORTS DE COURS

https://cours.tsix.be/cours/projet-de-developpement-sgbd

Version 3 — Fiche UE 2026-2027 liée depuis la page d'accueil · 10/10/2026

PRÉSENTATION

Projet de développement SGBD

Programme

L'étudiant sera capable :

4.1. Laboratoire : Concepts de développement SGBD

en disposant d'une station informatique opérationnelle équipée d'un logiciel « Bases de données », d'un outil de développement,

en utilisant les dernières techniques d'accès aux bases de données,

- de se connecter sur une base de données par programmation ;
- d'ajouter, de modifier, de supprimer des données par programmation ;
- de vérifier l'intégrité des données par programmation ;
- de récupérer et de gérer les erreurs générées par la base de données par programmation ;
- d'implémenter une interface visuelle qui permet la gestion et la validation des données ;
- de programmer des transactions ;
- de découper la programmation en différentes couches (Data Access Layer, Business Object, Business Layer...) ;
- de débugger la programmation (breakpoint, Statut des variables, Step by step...).

4.2. Laboratoire : Projet de développement SGBD

en disposant d'une station informatique opérationnelle équipée d'un logiciel « Bases de données », d'un outil de développement et sur base d'un cahier des charges fourni par le chargé de cours,

en utilisant les dernières techniques d'accès aux bases de données,

- d'implémenter la base de données et l'intégrité des données ;
- d'implémenter et de débugger une interface visuelle qui gère et valide les données et leur intégrité.

Capacités terminales

Pour atteindre le seuil de réussite, l'étudiant sera capable :

dans le respect des consignes données, en disposant d'une station informatique opérationnelle équipée d'un logiciel « Bases de données », d'un outil de développement et sur base d'un cahier des charges fourni par le chargé de cours,

en utilisant les dernières techniques d'accès aux bases de données,

- d'élaborer et de défendre un dossier technique reprenant :
  - le schéma de la base de données,
  - l'expression des contraintes en langage usuel,
  - la documentation du code et la gestion des erreurs ;

- d'implémenter une base de données et l'intégrité des données;
- de programmer, de tester et de défendre la programmation de l'interface visuelle qui permet la gestion des données.

Pour la détermination du degré de maîtrise, il sera tenu compte des critères suivants :

- la qualité du code,
- le design et l'ergonomie du site,
- le niveau de fiabilité des tests,
- la qualité du rapport de projet

Fiche UE 2026-2027 (PDF) PDF . Grille d'évaluation (PDF) PDF

HORAIRE MINIMUM DE L'UNITE DE FORMATION

<table class="my-table" style="minWidth: 100px"> <colgroup><col><col><col><col></colgroup><tbody><tr> <td colspan="1" rowspan="1"> 3.1. Dénomination du cours </td><td colspan="1" rowspan="1"> Classement </td><td colspan="1" rowspan="1"> Code U </td><td colspan="1" rowspan="1"> Nombre de périodes </td></tr> <tr><td colspan="1" rowspan="1"> Laboratoire : Concepts de développement SGBD </td> <td colspan="1" rowspan="1"> CT </td><td colspan="1" rowspan="1"> B </td><td colspan="1" rowspan="1"> 30 </td></tr><tr><td colspan="1" rowspan="1"> Laboratoire : Projet de développement SGBD </td><td colspan="1" rowspan="1"> CT </td><td colspan="1" rowspan="1"> S </td><td colspan="1" rowspan="1"> 34 </td></tr><tr><td colspan="1" rowspan="1"> 3.2. Part d'autonomie </td><td colspan="1" rowspan="1"> </td><td colspan="1" rowspan="1"> P </td><td colspan="1" rowspan="1"> 16 </td></tr><tr><td colspan="1" rowspan="1"> Total des périodes </td><td colspan="1" rowspan="1"> </td><td colspan="1" rowspan="1"> </td><td colspan="1" rowspan="1"> 80 </td></tr></tbody> </table>

Sommaire

- Projet de développement SGBD

1 Framework
- 1.1 Framework et librairie
- 1.2 Laravel
- 1.3 Environnement de développement
- 1.4 Installation
- 1.5 MVC
- 1.6 Structure d'un projet Laravel
- 1.7 Routing
- 1.8 Controllers
- 1.9 Views
- 1.10 Model
- 1.11 TailwindCSS
- 1.12 Gestion d'utilisateurs
- 1.13 Todo List
- 1.14 Blog (part 1)
- 1.15 Blog (part 2)
- 1.16 Installer un projet Laravel existant

2 Vue.js
- 2.1 Introduction à Vue.js
- 2.2 TailwindCSS
- 2.3 Components
- 2.4 Vue router
- 2.5 Ref, v-model
- 2.6 V-if, v-for
- 2.7 Classes
- 2.8 Events
- 2.9 OnMounted
- 2.10 Templates refs
- 2.11 Exercices
- 2.12 Formulaire de paiement
- 2.13 API Pokémon

3 Inertia
3.1 Pourquoi ?
3.2 Installation
3.3 Shadcn
3.4 Pages
3.5 Routage
3.6 Wayfinder
3.7 Formulaire
3.8 Pagination
3.9 Policies
3.10 Exercices
3.11 CRUD projets
3.12 SEO
3.13 OG tags
3.14 Google Analytics
3.15 Exercice SEO
4 Mini chatGPT
4.1 Consignes Examen
4.2 Api
4.3 LLM Face-Off
4.4 Markdown
4.5 Exercice 1 : Réponse à des questions
4.6 Fenêtre de contexte
4.7 Prompt Engineering
4.8 Exercice 2 : Interface de Chat
4.9 Exercice 3 : Instructions personnalisées
4.10 Exercice 4 : Streaming
4.11 Reconnaissance d'images
4.12 Laravel Dusk
4.13 Outils et Agents
4.14 Annexe : Formatage des Messages pour l'API

CHAPITRE 1

Frameworks

Les frameworks sont des catalyseurs de productivité dans le domaine du développement de logiciels. Ils offrent une armature sur laquelle les développeurs peuvent s'appuyer pour assembler rapidement des applications robustes et évolutives. En utilisant un framework, on évite de répéter des tâches communes et on s'assure de suivre des pratiques éprouvées.



Avantages des Frameworks

- **Gain de temps**: Ils réduisent le temps de développement en fournissant des fonctionnalités standardisées.
- **Conventions**: Ils imposent des conventions qui aident à maintenir la cohérence du code.
- **Réutilisabilité**: Les composants préconstruits facilitent la réutilisation du code.

Framework vs Bibliothèque

- **Framework**: Il s'agit d'une structure qui dicte l'architecture de votre application. Vous adaptez votre code au framework.
- **Bibliothèque**: C'est une collection de fonctions et de routines que vous pouvez appeler selon vos besoins.

Laravel et le Modèle MVC

Nous explorerons Laravel, un framework PHP qui incarne le modèle MVC (Modèle-Vue-Contrôleur), en détaillant:

- **Routing**: Simplification de la navigation dans l'application.
- **Contrôleurs**: Centralisation de la logique de traitement des requêtes.
- **Vues**: Séparation de la logique et de la présentation pour une meilleure maintenance.
- **Modèles**: Facilitation de l'interaction avec la base de données.

Projets Pratiques

Nous mettrons en pratique nos connaissances à travers deux projets:

1. **Un Blog**: Pour comprendre les bases de la construction d'une application avec Laravel.
2. **Mini Instagram**: Pour explorer des fonctionnalités plus avancées et comprendre comment structurer une application plus complexe.

En résumé, ce chapitre vous préparera à naviguer dans l'écosystème du framework Laravel, vous permettant de construire des applications de manière plus efficace et structurée.

1.1 Framework et librairie

Introduction aux frameworks et aux librairies

Le développement logiciel moderne s'appuie sur des structures et des composants réutilisables pour accélérer la création et l'entretien des applications. Les frameworks et les librairies constituent les piliers de cette approche, bien qu'ils servent différents objectifs et offrent des niveaux d'encadrement distincts. Cette section délimite ces outils et en souligne les divergences.

Définitions des frameworks et des librairies

Un **framework** représente une charpente logicielle complète, fournissant un cadre et des directives pour le développement d'applications. Il s'agit d'une trame qui guide les programmeurs à travers un ensemble de composants interconnectés et de pratiques standardisées, facilitant ainsi la mise en œuvre de solutions complexes.

À l'opposé, une **librairie** incarne une collection de fonctionnalités accessibles, conçues pour être intégrées dans des programmes variés. Ces assortiments de routines, de procédures et d'opérations offrent une aide ponctuelle sans imposer un schéma global d'organisation du code.

Tableau synthétique : Frameworks vs Librairies

Critère	Frameworks	Librairies
Structure	Cadre complet pour le développement	Composants individuels réutilisables
Flexibilité	Conventions strictes	Intégration souple
Contrôle	Architecture dirigée par le framework	Choix dirigés par le développeur
Dépendances	Ensemble cohérent et dépendant	Modules indépendants
Utilisation	Conception d'applications complètes	Ajout de fonctionnalités spécifiques

Différences entre les frameworks et les librairies

La distinction fondamentale entre un framework et une librairie repose sur la manière dont ils sont incorporés dans un projet. Un framework enveloppe généralement le programme, dictant la structure et le flux, tandis qu'une librairie est invoquée par le programme selon les besoins.

- **Cadre vs Composants** : Les frameworks établissent un canevas global, les librairies proposent des éléments distincts.
- **Rigueur vs Souplesse** : Les frameworks imposent des conventions, alors que les librairies permettent une intégration plus libre.
- **Intégration** : Les frameworks requièrent souvent une adhésion à un écosystème spécifique, tandis que les librairies peuvent être choisies à la carte.

Dans les sections suivantes, nous allons examiner quelques frameworks web backend et frontend prédominants dans l'écosystème actuel du développement.

Très bien, poursuivons avec les sections sur les frameworks web backend et frontend.

Frameworks web backend connus

Les frameworks backend facilitent le développement de la logique côté serveur d'une application web. Ils offrent des outils pour gérer des aspects comme la base de données, les requêtes utilisateur, et la sécurité. Voici un aperçu de quelques-uns des frameworks backend les plus répandus.

Laravel

Laravel est un framework PHP qui simplifie des tâches courantes dans la plupart des projets web, telles que l'authentification, les sessions, le routage et la mise en cache.



Caractéristique	Description
Modèle MVC	Laravel suit le modèle de conception MVC, organisant clairement le code entre la logique métier, l'interface utilisateur et le contrôle de l'application.
Écosystème riche	Il dispose d'un écosystème riche avec des outils comme Eloquent ORM, Blade, Laravel Mix, et bien d'autres.
Communauté active	Une vaste communauté de développeurs contribue à son développement et à son support.

Symfony

Symfony est un ensemble de composants PHP réutilisables et un framework d'application web pour PHP qui est connu pour sa stabilité et sa modularité.



Caractéristique	Description
Flexibilité	Symfony permet aux développeurs de choisir les composants qu'ils souhaitent utiliser, sans être forcés d'adopter l'ensemble du framework.
Interopérabilité	Il adhère aux standards PHP et permet donc l'intégration avec des bibliothèques d'autres frameworks.
Performances	Optimisé pour le développement d'applications complexes, il est reconnu pour ses bonnes performances.

Django

Django est un framework Python de haut niveau qui encourage le développement rapide et propre, avec une conception pragmatique.



Caractéristique	Description
Tout-en-un	Django suit la philosophie "batteries incluses", offrant une large gamme de fonctionnalités intégrées.
Sécurité	Il met l'accent sur la sécurité en aidant les développeurs à éviter les erreurs de sécurité courantes.
ORM puissant	L'ORM de Django simplifie les tâches de manipulation de la base de données.

Express.js

Express.js est un framework de routage et de middleware web minimaliste et flexible pour Node.js, qui offre une couche fine de fonctionnalités web fondamentales.

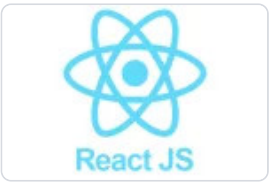
Caractéristique	Description
Simplicité	Express.js est minimaliste et flexible, permettant une mise en œuvre rapide.
Middleware	Il utilise un système de middleware robuste pour étendre ses fonctionnalités.
Communauté	Bénéficie d'une grande communauté et d'une large gamme de modules complémentaires.

Frameworks web frontend connus

En parallèle des frameworks backend, les frameworks frontend aident à concevoir l'interface utilisateur d'une application web. Ils se concentrent sur l'amélioration de l'expérience utilisateur en rendant les applications web plus réactives et interactives.

React.js

React.js est une bibliothèque JavaScript déclarative, efficace et flexible pour construire des interfaces utilisateur.



Caractéristique	Description
Composants	React favorise une architecture basée sur des composants réutilisables.
JSX	Utilise JSX, une syntaxe qui permet de mélanger HTML avec JavaScript.
Écosystème	Dispose d'un large écosystème avec des outils comme Redux pour la gestion d'état.

Angular

Angular est une plateforme et un framework pour construire des applications web à page unique efficaces et sophistiquées.



Caractéristique	Description
Complet	Angular est un framework complet qui inclut tout ce dont vous avez besoin pour construire une application frontend complexe.
TypeScript	Il est construit sur TypeScript, offrant ainsi des fonctionnalités avancées de programmation typée.
MVVM	Suit le modèle MVVM (Modèle-Vue-VueModèle), facilitant la liaison de données bidirectionnelle.

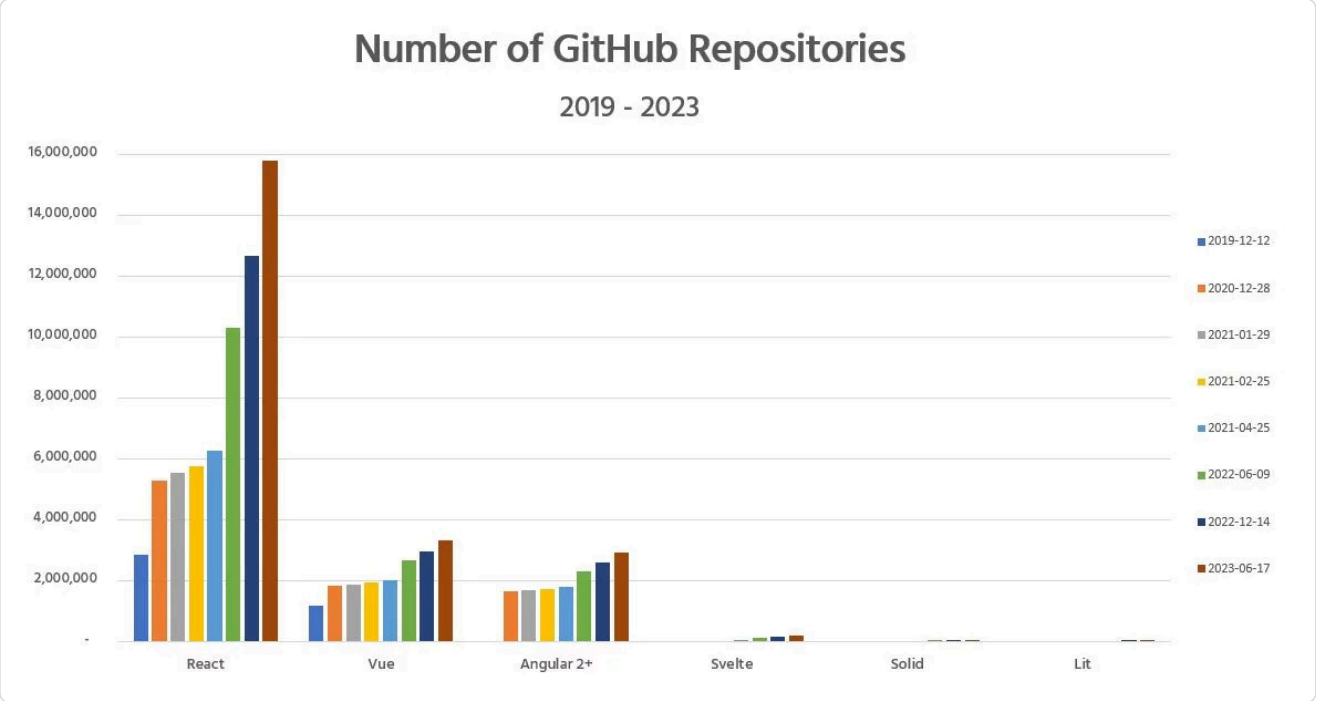
Vue.js

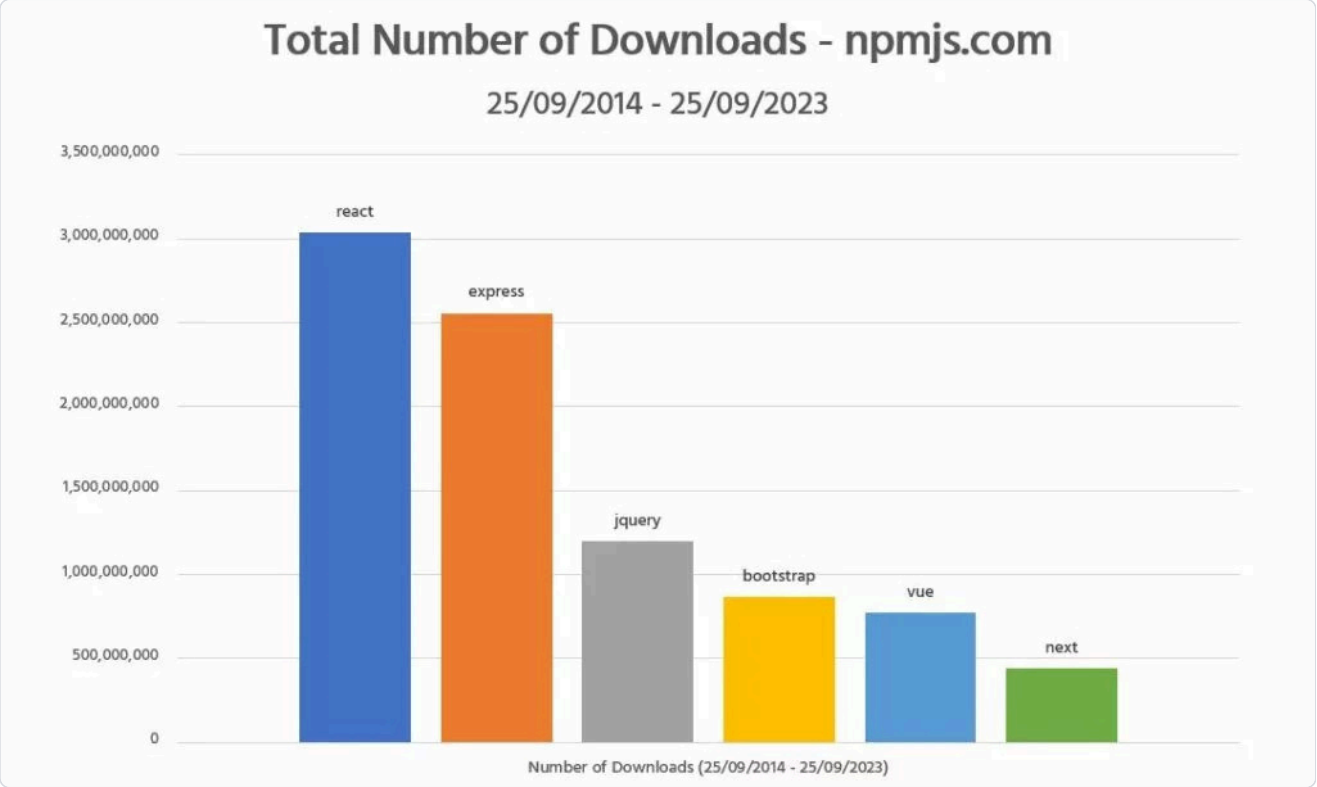
Vue.js est un framework progressif pour construire des interfaces utilisateur, qui se distingue par sa simplicité et sa facilité d'intégration.



Caractéristique	Description
Progressif	Vue est conçu pour être adopté de manière incrémentale, et peut facilement être utilisé dans des projets avec d'autres bibliothèques.
Performances	Il est optimisé pour la vitesse et la flexibilité, offrant des performances élevées.
Écosystème ouvert	Vue bénéficie d'un écosystème ouvert et en croissance, avec des outils comme VueX pour la gestion d'état.

Ces tableaux offrent une synthèse des caractéristiques principales de chaque framework, permettant de comparer leurs atouts et d'envisager leur utilisation selon les besoins spécifiques d'un projet.





Source : https://bluebirdinternational.com/frontend-frameworks/

1.2 Laravel

Laravel est un framework PHP populaire qui facilite et accélère le développement d'applications web modernes en fournissant un ensemble riche de fonctionnalités et une syntaxe expressive. Il est conçu pour les développeurs qui ont besoin d'un outil simple mais puissant pour créer des sites web ou des services robustes et maintenables.

Versions de Laravel

- Laravel 1 - Juin 2011
- Laravel 2 - Septembre 2011
- Laravel 3 - Février 2012
- Laravel 4 - Mai 2013
- Laravel 5 - Février 2015
- Laravel 6 - Septembre 2019
- Laravel 7 - Mars 2020
- Laravel 8 - Septembre 2020
- Laravel 9 - Février 2022
- Laravel 10 - Février 2023
- Laravel 11 - Mars 2024
- Laravel 12 - Février 2025

Philosophie de Laravel

Laravel vise à rendre le processus de développement agréable pour le développeur sans sacrifier les fonctionnalités d'application. Il valorise l'élégance, la simplicité et la lisibilité du code, tout en étant puissant et flexible pour les grandes applications.

Documentation officielle

https://laravel.com/docs/11.x

Ressources utiles

https://laravel.sillo.org/

https://laracasts.com/

https://benjamincrozat.com/

https://laravel-news.com/

https://laraveldaily.com/

1.3 Environnements de développement PHP

Windows : Laragon, ton meilleur pote

Laragon, c'est le couteau suisse pour développer en PHP sous Windows. Plus rapide qu'un café un lundi matin, Laragon embarque PHP, Apache, MySQL et même Node.js en mode zéro prise de tête. Et tout ça, dans un dossier que tu peux déplacer à volonté, parce que toi aussi t'as le droit d'être bordélique.

Pourquoi Laragon est cool :

- **URLs sexy** : Oublie les `localhost/mon-super-projet` tout claqués, maintenant c'est `mon-super-projet.test`.
- **Isolation** : Garde ton Windows tout propre, enfin presque.
- **Facilité extrême** : Installer une nouvelle version de PHP ? Deux clics, même ton chat peut y arriver.

Installation rapide :

1. Télécharge Laragon sur le site officiel.
2. Lance l'installeur, choisis ton dossier favori (`C:\laragon` par défaut).
3. Clique sur "Installer", attends, prends une pause-café, et voilà !

macOS : Laravel Herd, l'environnement chic et rapide

Si tu préfères les pommes aux fenêtres, Laravel Herd est fait pour toi. Plus rapide que tes excuses quand ton code plante, Herd est spécialement conçu pour Laravel et PHP sur macOS.

Pourquoi Laravel Herd déchire :

- **Ultra-rapide** : PHP, nginx, dnsmasq boostés aux binaires statiques.
- **Prêt à coder** : Composer, Laravel et Node.js directement accessibles dans le terminal.
- **Pro attitude** : Gestion simplifiée des bases de données MySQL, Postgres et Redis avec la version Pro.

Installation simplissime :

1. Télécharge Laravel Herd sur le site officiel.
2. Ouvre le fichier DMG, glisse Laravel Herd dans le dossier Applications.
3. Lance Herd, donne-lui les permissions (allez, fais confiance), et c'est prêt à l'emploi.

Extensions indispensables pour VS Code (Laravel Edition)

Pour coder comme un vrai pro de Laravel sans avoir l'air trop noob, voici les extensions VS Code incontournables :

- **Laravel Extension Pack** : Le starter-pack ultime avec Blade Snippets, coloration syntaxique, autocomplétion, tout-en-un.
- **Intelephense** : Auto-complétion intelligente pour PHP, parce que coder en aveugle, c'est cringe.
- **PHP Namespace Resolver** : Gère les namespaces PHP automatiquement, car tu as mieux à faire.
- **Prettier** : Formatte ton code automatiquement pour éviter les débats inutiles avec tes collègues.
- **SQLite Viewer** : Visualise facilement tes bases SQLite sans sortir de VS Code, pour les flemmards qui s'assument.
- **Tailwind CSS IntelliSense** : Autocomplétion des classes Tailwind CSS, histoire de coder à la vitesse de la lumière.
- **Laravel VSCode** : Support natif Laravel pour VS Code, t'en rêvais, ils l'ont fait.

Voilà, t'es équipé maintenant. Essaie de pas tout faire crasher !

1.4 Installation de Laravel 12 avec Composer

Prérequis

Avant de foncer installer Laravel 12 comme un ninja du web, vérifie tes bases, sinon tu risques d'avoir des surprises pas franchement drôles :

- **PHP** : Version 8.2 minimum. Si tu traînes sur une vieille version, fais-toi violence et mets à jour !
- **Composer** : C'est obligatoire. Si tu l'as pas, tu peux rentrer chez toi.
- **Serveur Web** : Pas besoin de lancer `artisan serve`, Laragon ou Laravel Herd s'en occupent déjà comme des grands.

Installation avec Composer (terminal Laragon ou macOS)

Ouvre ton terminal favori :

- Sur **Windows**, utilise le terminal intégré de Laragon.
- Sur **macOS**, utilise ton bon vieux Terminal natif.

1. **Déplace-toi vers ton dossier des projets :**

```bash
cd /chemin/vers/ton/dossier
```

2. **Crée un projet Laravel 12 via Composer :**

```bash
composer create-project --prefer-dist laravel/laravel nom-du-projet
```

Remplace évidemment `nom-du-projet` par un nom explicite (évite "test", merci).

3. **Entre dans ton nouveau projet :**

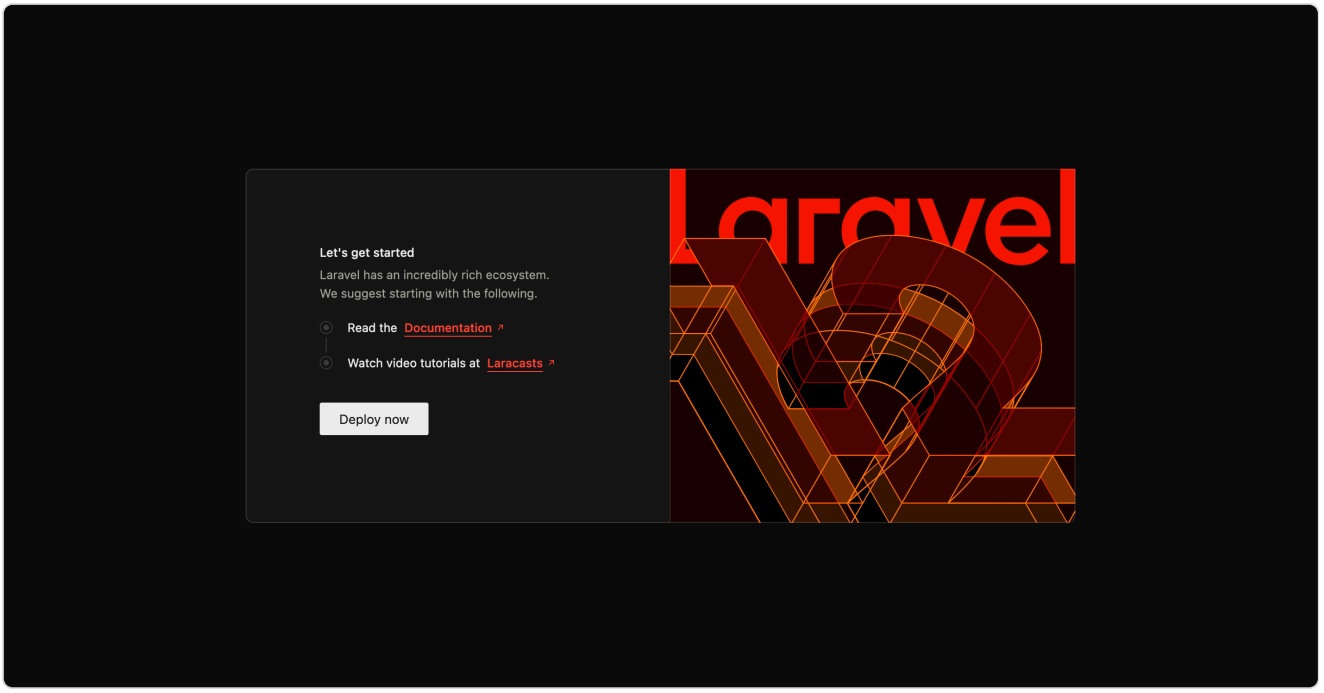
```bash
cd nom-du-projet
```

Et c'est tout ! Laravel s'est occupé du reste automatiquement (merci Taylor Otwell 🙌).

Vérifie ton installation dans ton navigateur

Ouvre ton navigateur préféré, entre l'adresse suivante pour vérifier que tout roule correctement :

```bash
http://nom-du-projet.test
```



Si tu vois la magnifique page d'accueil de Laravel, félicitations, t'as réussi quelque chose pour une fois ! Sinon, retourne pleurer à l'étape 1. 🤡

Ton Laravel est désormais prêt et configuré, plus aucune excuse pour procrastiner. 💪✨

1.5 Modèle-Vue-Contrôleur (MVC)

Introduction

Le **Modèle-Vue-Contrôleur** (**MVC**) est un motif d'architecture logicielle qui divise une application en trois composants principaux, chacun ayant un rôle distinct. Cette structure favorise la **séparation des préoccupations**, facilitant ainsi la maintenance et l'évolution des applications web.

Les composants du MVC

Modèle

Le **modèle** représente la structure des données, leur organisation et la logique métier. Il gère les interactions avec la base de données, assurant l'intégrité et la cohérence des données.

Vue

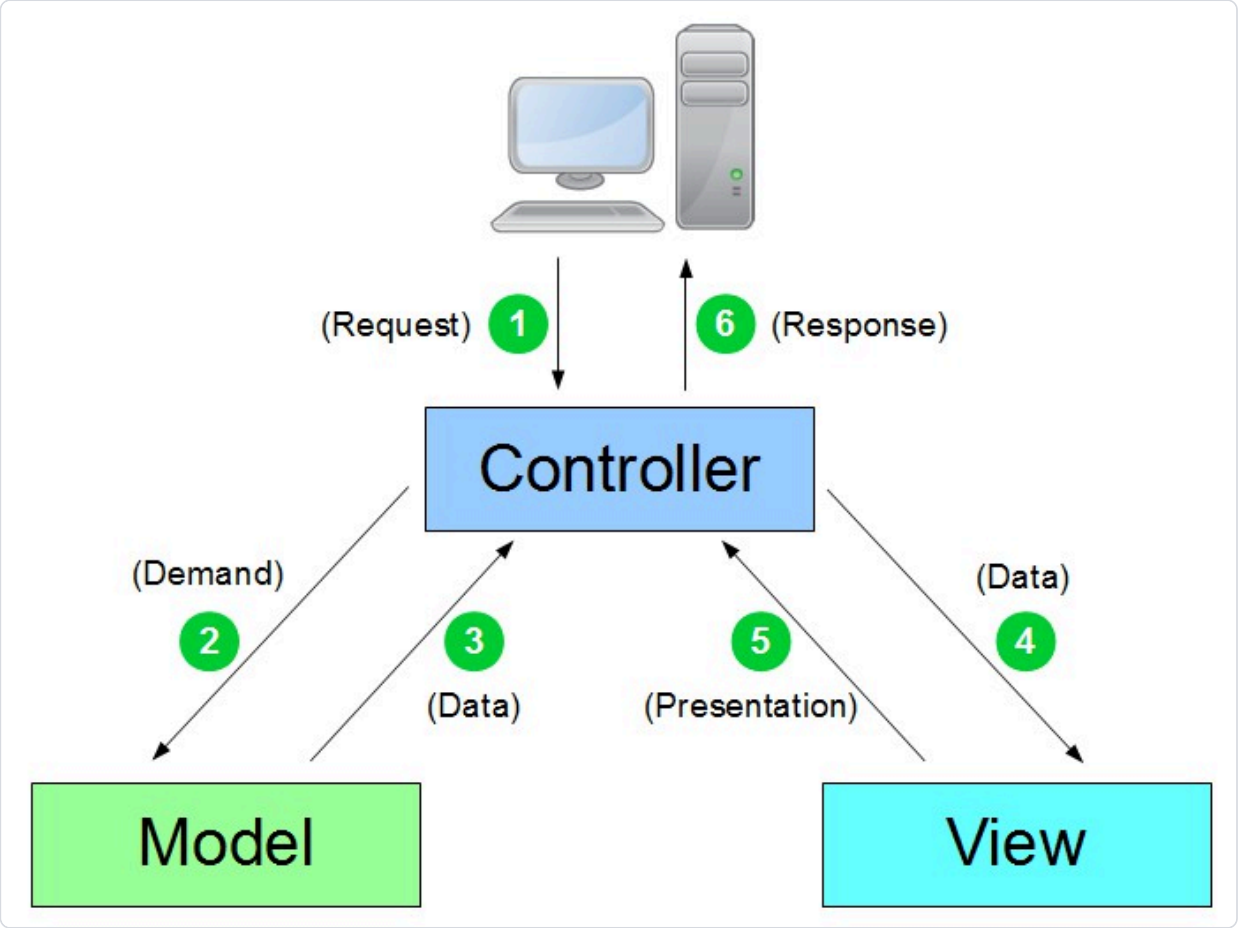
La **vue** est la couche de présentation. Elle affiche les données à l'utilisateur et recueille ses actions, qui sont transmises au contrôleur.

Contrôleur

Le **contrôleur** est l'intermédiaire entre l'utilisateur et le système. Il interprète les actions de l'utilisateur, manipule le modèle et met à jour la vue.

Routeur

Le **routeur** dirige les requêtes des utilisateurs vers le bon contrôleur, jouant le rôle de dispatcheur dans l'architecture MVC.



Source : https://adventy.org/fr/mvc

Cycle de vie d'une requête dans MVC

1. L'**utilisateur** (🧔) demande une action via le **navigateur** (🌐).
2. Le **navigateur** (🌐) transmet la requête au **routeur** (🛣️).
3. Le **routeur** (🛣️) oriente vers le **contrôleur** (🎛️) concerné.
4. Le **contrôleur** (🎛️) sollicite le **modèle** (💾) pour les données.
5. Le **modèle** (💾) interroge la **base de données** (💿).
6. Les données sont renvoyées au **contrôleur** (🎛️).
7. Le **contrôleur** (🎛️) transmet les données à la **vue** (📊).
8. La **vue** (📊) envoie le résultat au **navigateur** (🌐).
9. L'**utilisateur** (🧔) voit le résultat sur le **navigateur** (🌐).

Tableau récapitulatif des composants MVC

Composant	Rôle
Modèle	Gestion des données et logique métier.
Vue	Présentation des données et interaction utilisateur.
Contrôleur	Traitement des requêtes, manipulation du modèle et mise à jour de la vue.
Routeur	Orientation des requêtes vers le contrôleur adéquat.

Exemple pratique MVC avec Laravel

Pour mieux comprendre le MVC, voici un mini-exemple pratique d'affichage d'une liste d'articles :

1. La route (`routes/web.php`)

La route dirige la requête vers le contrôleur :

```php
use App\Http\Controllers\ArticleController;

Route::get('/articles', [ArticleController::class, 'index']);
```

2. Le contrôleur (`app/Http/Controllers/ArticleController.php`)

Le contrôleur récupère les données via le modèle et appelle la vue :

```php
namespace App\Http\Controllers;

use App\Models\Article;

class ArticleController extends Controller
{
    public function index()
    {
        $articles = Article::all(); // On récupère tous les articles depuis le modèle
        return view('articles.index', compact('articles')); // On passe les articles à la vue
    }
}
```

3. Le modèle (`app/Models/Article.php`)

Le modèle représente la logique métier (ici, très simple) et l'accès à la base de données :

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    protected $fillable = ['title', 'content']; // Colonnes autorisées à être modifiées
}
```

4. La vue (`resources/views/articles/index.blade.php`)

La vue présente les données à l'utilisateur :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Liste des articles</title>
</head>
<body>
    <h1>Articles récents</h1>
    <ul>
        @foreach(articlesasarticle)
            <li><strong>{{ article→title }}</strong> - {{ article→content }}</li>
        @endforeach
    </ul>
</body>
</html>
```

Maintenant, quand tu accèdes à `http://ton-projet.test/articles`, Laravel exécute tout ce beau cycle MVC en arrière-plan pour afficher ta liste d'articles.

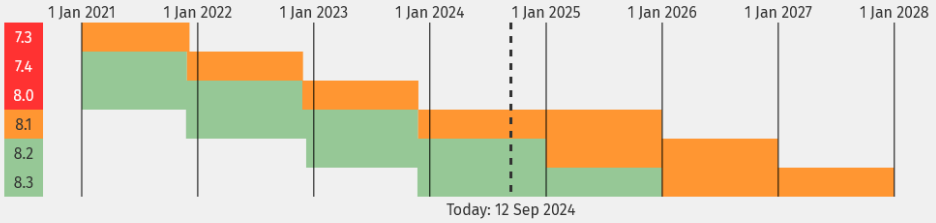
1.6 Vue d'ensemble de la structure d'un projet Laravel

Laravel est un framework MVC (Modèle-Vue-Contrôleur) qui permet de structurer de manière propre et logique les différents composants d'une application web. Chaque fichier et dossier à la racine d'un projet Laravel a un rôle spécifique, et comprendre cette structure permet de naviguer plus facilement entre les composants et de maintenir un code propre et cohérent.

Currently Supported Versions

Branch	Initial Release		Active Support Until		Security Support Until	
8.1	25 Nov 2021	*2 years, 9 months ago*	25 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*
8.2	8 Dec 2022	*1 year, 9 months ago*	31 Dec 2024	*in 3 months*	31 Dec 2026	*in 2 years, 3 months*
8.3	23 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*	31 Dec 2027	*in 3 years, 3 months*

Or, visualised as a calendar:



Key

Active support	A release that is being actively supported. Reported bugs and security issues are fixed and regular point releases are made.
Security fixes only	A release that is supported for critical security issues only. Releases are only made on an as-needed basis.
End of life	A release that is no longer supported. Users of this release should upgrade as soon as possible, as they may be exposed to unpatched security vulnerabilities.

Fichiers racine du projet Laravel

La structure d'un projet Laravel repose sur plusieurs dossiers et fichiers principaux, chacun ayant une mission bien définie. Voyons d'abord les fichiers qui se trouvent directement à la racine du projet avant de nous attarder sur les dossiers qui structurent l'application.

```
├── .env                # Configuration de l'environnement : base de données, services externes, etc.
├── .env.example        # Exemple de configuration de l'environnement
├── artisan             # Utilitaire en ligne de commande pour exécuter des tâches Laravel
├── composer.json       # Liste des dépendances PHP du projet
├── package.json        # Liste des dépendances JavaScript du projet
├── phpunit.xml         # Configuration des tests unitaires
├── README.md           # Documentation initiale sur le projet
├── vite.config.js      # Configuration pour Vite, outil de build pour les assets (CSS, JS)
```

`.env` – Fichier d'environnement

Le fichier `.env` contient toutes les variables d'environnement nécessaires au bon fonctionnement du projet. C'est ici qu'on spécifie les paramètres comme les informations de connexion à la base de données, la clé d'API de services tiers, ou les paramètres de mail.

Exemple de contenu typique du fichier `.env` :

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
DB_CONNECTION=sqlite
DB_DATABASE=database.sqlite
```

Le fichier `.env` est essentiel, car il permet de rendre le projet flexible à différents environnements (développement, production) sans modifier le code.

`.env.example` – Exemple de configuration d'environnement

Le fichier `.env.example` est une version modèle du fichier `.env`. Il sert de référence pour les développeurs qui veulent configurer le projet sur leur propre machine. Lorsqu'un développeur clone un projet, il peut simplement copier `.env.example` vers `.env` et le personnaliser avec les valeurs spécifiques à son environnement.

`artisan` – Console de commande

Le fichier `artisan` est un script PHP qui permet d'accéder à la console de commande Laravel. Cette console est très utile pour exécuter des tâches variées, comme la création de nouvelles entités (contrôleurs, modèles, etc.), la gestion des migrations de base de données, ou encore l'exécution de tâches planifiées.

Commandes courantes utilisables avec `artisan` :

```
php artisan migrate        # Appliquer les migrations de base de données
php artisan serve          # Lancer un serveur local de développement, ne pas l'utiliser si vous
avez un logiciel comme Laragon ou Mamp installé qui remplit déjà ce rôle.
php artisan make:controller MyController  # Créer un nouveau contrôleur nommé MyController
```

L'outil `artisan` est un des points forts de Laravel, car il permet de faciliter beaucoup de tâches de développement.

`composer.json` – Gestionnaire de dépendances PHP

Ce fichier décrit toutes les dépendances PHP du projet. **Composer** est l'outil qui permet de gérer ces dépendances. Par exemple, si vous avez besoin d'utiliser une librairie externe, `composer.json` va la lister, et la commande `composer install` va installer cette dépendance.

Voici un extrait typique du fichier `composer.json` :

```json
{
    "require": {
        "php": "^8.0",
        "laravel/framework": "^9.0",
        "fideloper/proxy": "^4.0"
    },
    "autoload": {
        "psr-4": {
            "App\\": "app/"
        }
    }
}
```

Ce fichier est indispensable pour toute mise à jour de dépendances ou pour synchroniser l'environnement de développement avec celui de production.

`package.json` – Gestionnaire de dépendances JavaScript

Le fichier `package.json` est l'équivalent de `composer.json`, mais pour les dépendances JavaScript. Laravel utilise souvent des outils JavaScript comme **Vue.js** ou **React** pour la partie front-end. Ce fichier permet de lister les bibliothèques JavaScript nécessaires au projet, telles que `axios` ou `lodash`.

Exemple d'extrait du fichier `package.json` :

```json
{
    "devDependencies": {
        "vite": "^4.0.0",
        "laravel-vite-plugin": "^0.7.0"
    },
    "dependencies": {
        "vue": "^3.0.0",
        "axios": "^1.0.0"
    }
}
```

`package.json` est essentiel pour gérer les scripts de compilation et les dépendances front-end du projet.

`vite.config.js` – Configuration de Vite

Ce fichier est une configuration pour **Vite**, qui est utilisé pour le traitement des fichiers CSS et JavaScript. **Vite** est un outil de build moderne qui remplace **Webpack** et permet une compilation plus rapide et efficace des assets. Le fichier `vite.config.js` contient les paramètres de compilation, comme les alias de chemin et les plugins utilisés.

Exemple de contenu de `vite.config.js` :

```javascript
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: true,
        }),
    ],
});
```

Ce fichier permet de définir la manière dont les assets sont gérés et compilés lors du développement.

`phpunit.xml` – Configuration des tests

Le fichier `phpunit.xml` contient la configuration nécessaire pour exécuter les tests unitaires via PHPUnit. Laravel est prééquipé avec PHPUnit pour que les développeurs puissent écrire des tests unitaires sur leurs méthodes et contrôleurs.

Autres fichiers importants

- `README.md` : Un fichier de documentation initiale du projet. Idéalement, il contient des informations sur la façon de configurer et de lancer le projet.

Ces fichiers à la racine sont essentiels pour configurer et exploiter pleinement un projet Laravel. Ils participent à la gestion de l'environnement, de la documentation, des tests, des dépendances et des assets.

Les dossiers principaux du projet Laravel

Laravel organise la structure de son projet en plusieurs dossiers principaux, chacun jouant un rôle spécifique. Cette organisation favorise une séparation claire des responsabilités et permet une évolution maintenable de l'application. Voyons ensemble les dossiers les plus importants et leur utilité.

```
├── app/                # Contient la logique métier de l'application
├── bootstrap/          # Initialisation du framework
├── config/             # Configuration globale de l'application
├── database/           # Gestion de la base de données (migrations, seeders, etc.)
├── public/             # Point d'entrée de l'application (serveur web)
├── resources/          # Contient les vues, les assets, et les fichiers de localisation
├── routes/             # Définition des routes de l'application
├── storage/            # Stockage des logs, des fichiers, et du cache
├── tests/              # Tests unitaires et fonctionnels
├── vendor/             # Bibliothèques externes installées par Composer
```

Dossier `app/`

Le dossier `app/` est le cœur de l'application. Il contient toute la logique métier et les composants qui structurent l'application en utilisant le modèle MVC. Ce dossier est divisé en plusieurs sous-dossiers :

- **Http/** : Contient les contrôleurs, les middlewares et les requêtes. Par exemple, le sous-dossier `Controllers/` gère la logique de traitement des requêtes des utilisateurs.
- **Models/** : Contient les modèles, qui représentent les entités de la base de données (par exemple, `User.php`).
- **Providers/** : Contient les classes qui servent à lier des services à l'application, comme `AppServiceProvider.php`.

Exemple de structure :

```bash
app/
├── Http/
│   ├── Controllers/    # Logique des contrôleurs
│   ├── Middleware/     # Traitement des requêtes avant les contrôleurs
│   └── Requests/       # Validation des requêtes utilisateur (créé dynamiquement)
├── Models/             # Représentation des entités, ex : User.php
└── Providers/          # Services liés à l'application
```

Dossier `bootstrap/`

Le dossier `bootstrap/` contient le fichier `app.php`, qui est responsable de charger le framework et de préparer l'application pour l'exécution. Ce dossier ne doit pas être modifié directement, sauf pour des besoins très spécifiques.

Dossier `config/`

Ce dossier regroupe tous les fichiers de configuration de l'application. Chaque aspect de Laravel possède son propre fichier de configuration :

- **app.php** : Contient la configuration générale de l'application (nom, fuseau horaire, langue, etc.).
- **database.php** : Gère la connexion aux bases de données.
- **mail.php** : Configure les paramètres d'envoi de mails.

Exemple d'une partie de `app.php` :

```php
return [
    'name' => env('APP_NAME', 'Laravel'),
    'env' => env('APP_ENV', 'production'),
    'debug' => (bool) env('APP_DEBUG', false),
    'timezone' => 'UTC',
];
```

Ces fichiers de configuration permettent d'adapter le comportement de Laravel en fonction des besoins du projet.

Dossier `database/`

Le dossier `database/` est destiné à la gestion de la base de données. Il contient :

- **migrations/** : Définit la structure des tables de la base de données.
- **seeders/** : Remplit la base avec des données initiales ou de test.
- **factories/** : Génère des données factices, souvent utilisées pour les tests unitaires.

Structure typique :

```
database/
├── factories/          # Génération de fausses données
├── migrations/         # Définition des tables de la base de données
└── seeders/            # Remplissage des tables avec des données de test
```

Dossier `public/`

Le dossier `public/` est le point d'entrée du projet. C'est là où se trouve le fichier `index.php`, qui est la première étape d'exécution de l'application Laravel lorsque les utilisateurs accèdent au site via leur navigateur. Ce dossier contient également les assets publics comme les images et les fichiers JavaScript compilés.

Dossier `resources/`

Le dossier `resources/` contient les fichiers de vues, les fichiers CSS et JavaScript non compilés, ainsi que les fichiers de localisation.

- **views/** : Contient les fichiers Blade (`.blade.php`), qui sont utilisés pour générer l'interface utilisateur dynamique.
- **lang/** : Contient les fichiers de localisation pour les traductions de l'application.

Structure :

```
resources/
├── views/              # Vues Blade pour générer l'interface utilisateur
├── css/                # Fichiers CSS non compilés
├── js/                 # Fichiers JavaScript non compilés
└── lang/               # Localisation et traductions
```

Dossier `routes/`

Le dossier `routes/` est là où les routes de l'application sont définies. Chaque fichier correspond à un groupe de routes :

- **web.php** : Contient les routes qui sont accessibles via le web (interfaces utilisateur).
- **api.php** : Contient les routes destinées à l'API de l'application.

Dossier `storage/`

Ce dossier est utilisé pour stocker tout ce qui est généré par l'application et qui doit être conservé, comme les logs, les sessions, ou encore les fichiers générés par les utilisateurs.

- **logs/** : Enregistre les logs de l'application.
- **framework/** : Contient le cache, les sessions, et les fichiers temporaires.

Dossier `tests/`

Le dossier `tests/` contient les tests unitaires et fonctionnels de l'application. Laravel est livré avec PHPUnit pour l'écriture des tests. Ce dossier est divisé en plusieurs sous-dossiers :

- **Feature/** : Contient les tests de fonctionnalités, qui vérifient des aspects larges de l'application, par exemple si une fonctionnalité complète fonctionne comme prévu.
- **Unit/** : Contient les tests unitaires, qui vérifient des morceaux de code isolés (fonctions, classes).
- **Browser/** : Peut contenir des tests de navigation (end-to-end) utilisant Laravel Dusk, qui vérifient le comportement de l'application dans un navigateur.
- **TestCase.php** : Ce fichier est la classe de base que tous les tests héritent. Il configure l'environnement de test, les dépendances, et fournit des méthodes utilitaires pour les tests.

Pour une meilleure organisation, il est souvent conseillé de reproduire la structure des dossiers `app/` dans `tests/` afin de rendre la navigation plus intuitive. Par exemple, un test pour un contrôleur situé dans `app/Http/Controllers/` serait placé dans `tests/Feature/Http/Controllers/`.

Structure typique :

```
tests/
├── Feature/            # Tests de fonctionnalités
│   ├── Http/           # Tests liés aux contrôleurs HTTP
│   └── ...
├── Unit/               # Tests unitaires
├── Browser/            # Tests de navigation avec Laravel Dusk (optionnel)
└── TestCase.php        # Classe de base pour les tests
```

Dossiers créés dynamiquement par Laravel

Certains dossiers peuvent être créés automatiquement par Laravel lorsque certaines fonctionnalités sont utilisées :

- **Requests/** : Ce dossier est créé dans `app/Http/` lorsque des form requests sont générés via des commandes Artisan. Les form requests sont utilisés pour la validation des données avant qu'elles n'atteignent les contrôleurs.
- **Events/** : Ce dossier est créé dans `app/` lorsque des événements sont utilisés dans l'application. Les événements permettent de découpler des parties de l'application, par exemple lorsqu'une action (comme la création d'un utilisateur) doit déclencher d'autres processus (comme l'envoi d'un email).
- **Notifications/** : Ce dossier est créé dans `app/` lorsqu'on utilise le système de notifications de Laravel. Les notifications sont souvent utilisées pour informer les utilisateurs par email, SMS, ou d'autres moyens.

Dossiers créés manuellement

Il est courant de créer des dossiers supplémentaires pour organiser des aspects spécifiques de l'application qui ne sont pas prévus par Laravel par défaut. Ces dossiers peuvent être ajoutés manuellement par les développeurs selon les besoins du projet :

- **Services/** : Ce dossier est généralement créé dans `app/` pour y mettre des classes qui font appel à des services externes, tels que des API tierces (par exemple, l'API d'OpenAI, Twilio pour envoyer des SMS, etc.). Créer un dossier `Services` est une convention courante dans la communauté Laravel pour organiser proprement les appels aux services externes et faciliter la maintenabilité du code.
- **Autres dossiers** : Libre à chaque développeur d'ajouter d'autres dossiers selon les besoins de l'application pour structurer le code de manière logique et efficace.

Dossier `vendor/`

Le dossier `vendor/` est généré automatiquement par Composer et contient toutes les bibliothèques externes dont dépend Laravel. Il n'est jamais modifié manuellement.

Synthèse

Ces dossiers sont essentiels pour bien organiser un projet Laravel. Ils permettent de garder une structure cohérente et de maintenir la séparation des responsabilités, ce qui est crucial pour le développement et la maintenabilité d'une application. Les dossiers créés dynamiquement par Laravel et ceux ajoutés manuellement viennent enrichir cette structure en fonction des besoins spécifiques du projet.

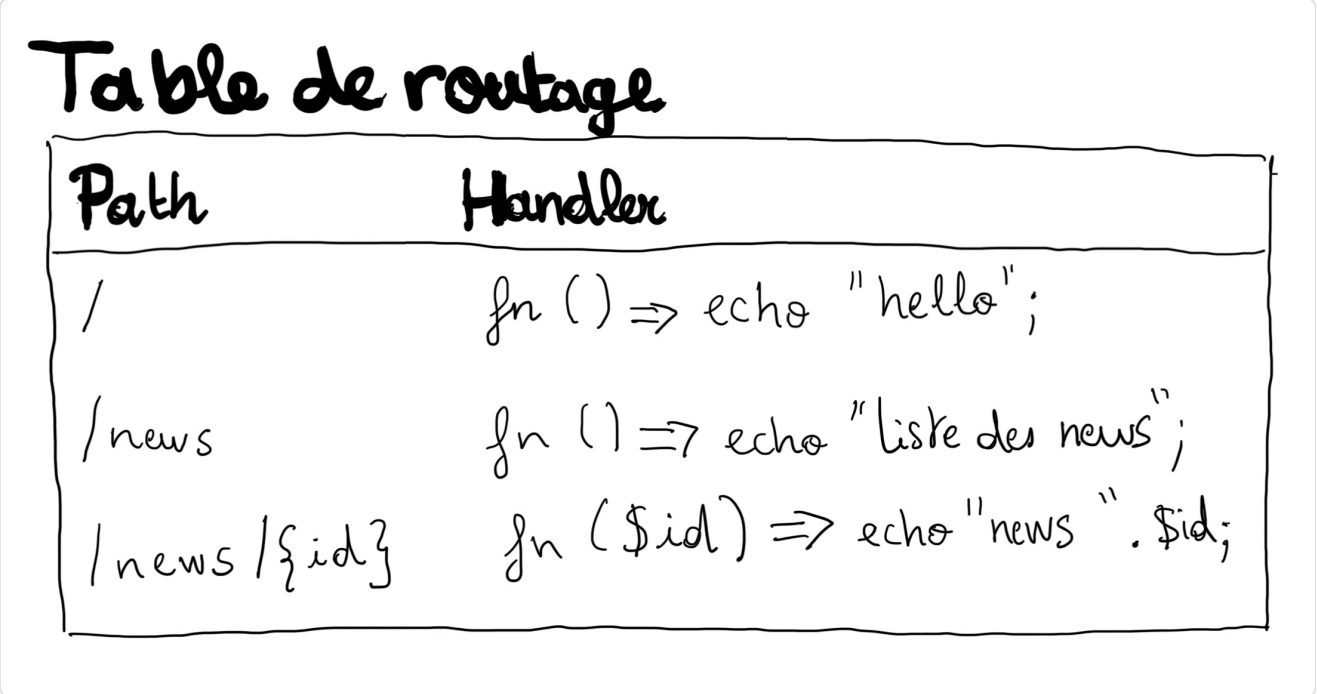
1.7 Routage

Définition et rôle d'un routeur

Un routeur est un composant clé dans le développement web qui sert d'intermédiaire entre les requêtes des utilisateurs et les réponses du serveur. Il interprète l'URL demandée et déclenche l'action appropriée, souvent en dirigeant la requête vers une méthode de contrôleur.

Mécanisme de routage

Le routeur examine l'URL fournie par l'utilisateur et la compare avec sa table de routage pour identifier l'action à exécuter, qui peut être la méthode d'un contrôleur ou une fonction anonyme.



Introduction au routage dans Laravel

Laravel, un framework PHP réputé, simplifie la création d'applications web en offrant des outils et une structure prédéfinis. Le routage est un aspect crucial de Laravel et est géré à travers des fichiers spécifiques situés dans le répertoire `routes`, notamment `web.php` pour les routes web.

Configuration des routes avec Laravel

Dans Laravel, les routes sont définies en utilisant des méthodes qui correspondent aux différentes méthodes HTTP. Voici une illustration de la syntaxe :

```php
Route::get(uri,uri, uri,callback);
Route::post(uri,uri, uri,callback);
// et d'autres méthodes HTTP...
```

Les verbes HTTP et leur utilisation dans le routage

Explication des verbes HTTP

Les verbes HTTP sont des méthodes de requête qui indiquent l'action souhaitée pour une ressource donnée. Ils jouent un rôle crucial dans le routage, définissant la nature de la requête et l'opération à effectuer.

Les verbes HTTP courants

Voici les verbes HTTP les plus utilisés dans le routage avec Laravel :

- **GET** : Utilisé pour récupérer des ressources. Dans le contexte du routage, une route GET est souvent utilisée pour afficher une page ou des données.
- **POST** : Employé pour créer de nouvelles ressources. Une route POST est généralement associée à la soumission de formulaires.
- **PUT/PATCH** : Utilisé pour mettre à jour des ressources existantes. PUT est souvent employé pour les mises à jour complètes, tandis que PATCH est pour les mises à jour partielles.
- **DELETE** : Comme son nom l'indique, ce verbe sert à supprimer des ressources.

Tableau comparatif des verbes HTTP

Verbe HTTP	Usage	Action CRUD correspondante
GET	Récupérer une ou plusieurs ressources	Read (Lire)
POST	Créer une nouvelle ressource	Create (Créer)
PUT/PATCH	Mettre à jour une ressource existante	Update (Mettre à jour)
DELETE	Supprimer une ressource	Delete (Supprimer)

Ces verbes sont directement utilisés dans la définition des routes Laravel pour indiquer comment les requêtes à différentes URLs doivent être traitées.

Association des routes aux contrôleurs

Pour associer une route à un contrôleur, Laravel permet d'utiliser la notation `[Controller::class, 'method']` :

```php
Route::get('/articles', [ArticleController::class, 'index']);
```

Paramétrage des routes

Il est possible d'intégrer des paramètres aux routes pour passer des données aux méthodes de contrôleur :

```php
Route::get('/articles/{id}', [ArticleController::class, 'show']);
```

Nomination des routes

Attribuer un nom aux routes facilite leur référence ultérieure dans l'application :

```php
Route::get('/contact', [ContactController::class, 'show'])->name('contact.show');
```

Exemples concrets

Exemple de déclaration de routes

Voici comment les routes sont déclarées dans le fichier `web.php` en utilisant la notation préférée :

```php
// Accueil
Route::get('/', function () {
    return view('welcome');
});

// Contact
Route::get('/contact', [ContactController::class, 'show']);

// Article spécifique
Route::get('/articles/{id}', [ArticleController::class, 'show']);
```

Correspondances URL/Route

Le tableau suivant illustre des exemples d'URLs et les routes associées avec le gestionnaire correspondant :

URL	Route	Gestionnaire
`/`	`Route::get('/')`	Closure (retourne view 'welcome')
`/contact`	`Route::get('/contact')`	`ContactController@show`

URL	Route	Gestionnaire
`/articles/42`	`Route::get('/articles/{id}')`	`ArticleController@show`

Routes de type ressource

Présentation des routes ressource

Laravel propose un moyen rapide de créer plusieurs routes pour gérer des ressources complètes avec une seule ligne de code. Ces routes ressource prennent en charge les opérations CRUD (Create, Read, Update, Delete) pour une ressource donnée.

Déclaration d'une route ressource

Pour déclarer une route ressource, on utilise la méthode `resource` :

```php
Route::resource('articles', ArticleController::class);
```

Tableau de correspondance des routes ressource

Ce tableau montre la correspondance entre les actions CRUD et les routes générées automatiquement par Laravel :

Action	URL	Route	Gestionnaire	HTTP Method
Index	`/articles`	`Route::resource('articles')`	`ArticleController@index`	GET
Create	`/articles/create`	`Route::resource('articles')`	`ArticleController@create`	GET
Store	`/articles`	`Route::resource('articles')`	`ArticleController@store`	POST
Show	`/articles/{article}`	`Route::resource('articles')`	`ArticleController@show`	GET
Edit	`/articles/{article}/edit`	`Route::resource('articles')`	`ArticleController@edit`	GET
Update	`/articles/{article}`	`Route::resource('articles')`	`ArticleController@update`	PUT/PATCH
Destroy	`/articles/{article}`	`Route::resource('articles')`	`ArticleController@destroy`	DELETE

Exercices pratiques sur le routage

Voici quelques exercices pratiques simples pour que tu puisses maîtriser les bases du routage Laravel. Pas de contrôleur, pas de base de données, juste toi, `routes/web.php`, et ton talent (ou son absence). 😉

Exercice 1 : Route simple (GET)
Crée une route `/hello` qui affiche dans le navigateur :
« Hello World ! Ceci est ma première route Laravel. »

Exercice 2 : Route avec paramètre
Crée une route `/user/{name}` qui affiche :
« Bienvenue sur ton profil, [name] ! » en remplaçant `[name]` par la valeur fournie dans l'URL.

Exemple : `/user/Alice` affiche « Bienvenue sur ton profil, Alice ! »

Exercice 3 : Route avec deux paramètres
Crée une route `/calcul/{nombre1}/{nombre2}` qui affiche la somme des deux nombres passés en paramètre.

Exemple : `/calcul/4/6` affiche « Le résultat est : 10 ».

Exercice 4 : Route optionnelle
Crée une route `/bienvenue/{langue?}` :

- Si aucune langue n'est spécifiée : affiche « Bienvenue ! »
- Si la langue est "en" : affiche « Welcome! »
- Si la langue est "es" : affiche « ¡Bienvenido! »
- Pour toute autre langue : affiche « Langue non supportée. »

Exercice 5 : Méthode POST fictive
Crée une route `/formulaire` en POST qui renvoie simplement la phrase suivante :
« Formulaire envoyé avec succès ! ».

(Tu ne pourras pas encore la tester directement via le navigateur, mais prépare déjà le terrain pour la suite !)

Exercice 6 : Route de redirection
Crée une route `/ici` qui redirige automatiquement vers `/hello` (la première route créée).

Si tu survis à ces exercices, on pourra éventuellement envisager des choses sérieuses. Bonne chance !

1.8 Les Controllers

Les controllers sont au cœur du traitement logique d'une application web. Ils interceptent les requêtes entrantes, communiquent avec les modèles et renvoient des réponses sous forme de vues ou de données JSON (dans le cas des API).

Exemple de Controller simple

Prenons un exemple concret d'un controller qui affiche une liste de tâches :

Déclaration de la route

```php
// routes/web.php

use App\Http\Controllers\TaskController;

Route::get('/tasks', [TaskController::class, 'index']);
```

Création du Controller

```php
// app/Http/Controllers/TaskController.php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TaskController extends Controller
{
    public function index()
    {
        // Simulons des tâches pour l'exemple
        $tasks = [
            ['id' => 1, 'title' => 'Acheter du café'],
            ['id' => 2, 'title' => 'Coder comme un boss'],
            ['id' => 3, 'title' => 'Faire une pause (ou pas)'],
        ];

        return view('tasks.index', compact('tasks'));
    }
}
```

Vue associée

```
<!— resources/views/tasks/index.blade.php —>

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Liste des tâches</title>
</head>
<body>
    <h1>Tâches à accomplir</h1>
    <ul>
        @foreach($tasks as $task)
            <li>{{ $task['title'] }}</li>
        @endforeach
    </ul>
</body>
</html>
```

Ici, le controller récupère des tâches fictives (aucune base de données pour l'instant) et les envoie à la vue pour affichage.

Controller Resource : exemple avec une ressource "User"

Laravel facilite la création de controllers en suivant le modèle RESTful. Un controller resource prend en charge automatiquement les opérations CRUD (Create, Read, Update, Delete).

Déclaration d'une route ressource

```php
// routes/web.php

use App\Http\Controllers\UserController;

Route::resource('users', UserController::class);
```

Exemple complet d’un Controller Resource

PHP

```php
// app/Http/Controllers/UserController.php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Request;

class UserController extends Controller
{
    // Afficher tous les utilisateurs
    public function index()
    {
        users = User::all();
        return view('users.index', compact('users'));
    }

    // Afficher le formulaire pour créer un utilisateur
    public function create()
    {
        return view('users.create');
    }

    // Stocker un nouvel utilisateur
    public function store(Request request)
    {
        validated=validated = validated=request→validate([
            'name' ⇒ 'required|string|max:255',
            'email' ⇒ 'required|email|unique:users,email',
        ]);

        User::create(validated);

        return redirect()→route('users.index');
    }

    // Afficher un utilisateur précis
    public function show(User user)
    {
        return view('users.show', compact('user'));
    }

    // Afficher le formulaire d'édition
    public function edit(User
user)returnview(’users.edit’,compact(’user’));//Mettrea`jourunutilisateurpublicfunctionupdate(Requ
estuser)
    {
        return view('users.edit', compact('user'));
    }

    // Mettre à jour un utilisateur
    public function update(Request
user)returnview(’users.edit’,compact(’user’));//Mettrea`jourunutilisateurpublicfunctionupdate(Requ
estrequest, User user)
    {
        $validated = $request→validate([
            'name' ⇒ 'required|string|max:255',
            'email' ⇒ 'required|email|unique:users,email,' . $user→id,
```

```
        ]);

        $user->update($validated);

        return redirect()->route('users.index');
    }

    // Supprimer un utilisateur
    public function destroy(User user)
    {
        $user->delete();

        return redirect()->route('users.index');
    }
}
```

Chaque méthode correspond précisément à une action CRUD. Plus aucune excuse pour t'embrouiller.

Réponse JSON pour les API

Dans un contexte API, le controller renvoie des données JSON plutôt que des vues HTML :

Déclaration de la route API

PHP

```php
// routes/api.php

use App\Http\Controllers\Api\UserController;

Route::get('/users', [UserController::class, 'index']);
```

Exemple de Controller JSON

PHP

```php
// app/Http/Controllers/Api/UserController.php

namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
    public function index()
    {
        users=User::all();returnresponse()->json(users = User::all();
        return response()->json(users=User::all();returnresponse()->json(users);
    }
}
```

Laravel gère automatiquement la sérialisation JSON des données.

Grâce à ces exemples détaillés, même toi tu devrais comprendre comment fonctionnent les controllers dans Laravel.

1.9 Views

Introduction à Blade

Blade est le moteur de templates intégré à Laravel. Il te permet d'utiliser du code PHP de manière simplifiée dans tes fichiers de vue, tout en offrant une syntaxe propre et expressive. Les fichiers Blade se terminent par l'extension `.blade.php` et sont généralement stockés dans le répertoire `resources/views`.

Rendre une Vue et Passer des Données

Pour afficher une vue, tu peux utiliser la fonction globale `view` dans tes routes ou contrôleurs. Par exemple :

```php
Route::get('/', function () {
    return view('accueil', ['nom' => 'Jean']);
});
```

Dans cet exemple, la vue `accueil` recevra une variable `$nom` avec la valeur 'Jean'. Dans ta vue Blade, tu peux afficher cette donnée ainsi :

```
<p>Bienvenue, {{ $nom }} !</p>
```

Les Composants Anonymes

Les composants anonymes sont des morceaux réutilisables de code Blade qui n'ont pas besoin d'une classe PHP associée. Ils permettent de simplifier la structure de ton application en évitant la création de classes inutiles.

Création d'un Composant Anonyme

Pour créer un composant anonyme, il te suffit de créer un fichier Blade dans le répertoire `resources/views/components`. Par exemple, pour un composant d'alerte :

```
<!-- resources/views/components/alerte.blade.php -->
<div class="alerte alerte-{{ type }}">
    {{ message }}
</div>
```

Ici, le composant utilise deux variables : `$type` et `$message`.

Utilisation d'un Composant Anonyme

Pour utiliser ce composant dans une vue, tu peux l'appeler ainsi :

```
<x-alerte :type="type" :message="message" />
```

Les deux-points `:` avant les noms d'attributs indiquent à Blade que ces attributs doivent être évalués comme du code PHP. Pas besoin de préfixer de `:` lorsqu'on passe une string au composant.

Passage de Données aux Composants

Pour passer des données à un composant, tu peux utiliser la directive `@props` en haut du fichier du composant :

```
<!-- resources/views/components/alerte.blade.php -->
@props(['type' => 'info', 'message'])

<div class="alerte alerte-{{ type }}">
    {{ message }}
</div>
```

Ici, `@props` définit les variables `$type` et `$message` avec des valeurs par défaut. Si aucune valeur n'est passée lors de l'appel du composant, `$type` sera 'info' par défaut.

Les Layouts et les Slots

Les layouts sont des modèles de base que tes vues peuvent étendre, ce qui te permet de maintenir une structure cohérente à travers ton application. Les slots sont des espaces réservés dans ces layouts où le contenu spécifique à chaque vue sera inséré.

Création d'un Layout avec Slots

Imaginons un layout de base situé dans `resources/views/layouts/app.blade.php` :

```
<!-- resources/views/layouts/app.blade.php -->
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>{{ title ?? 'Mon Application' }}</title>
</head>
<body>
    <header>
        <!-- Contenu de l'en-tête -->
    </header>

    <main>
        {{ slot }}
    </main>

    <footer>
        <!-- Contenu du pied de page -->
    </footer>
</body>
</html>
```

Ici, `{{ $slot }}` représente l'endroit où le contenu spécifique de chaque vue sera inséré. La variable `$title` est optionnelle et peut être définie dans chaque vue.

Utilisation d'un Layout dans une Vue

Pour utiliser ce layout dans une vue, tu peux faire :

```
<!-- resources/views/accueil.blade.php -->
<x-layouts.app>
    <x-slot:title>
        Page d'accueil
    </x-slot:title>

    <h1>Bienvenue sur la page d'accueil</h1>
    <p>Ceci est le contenu de la page.</p>
</x-layouts.app>
```

Ici, `x-slot:title` permet de définir la valeur de la variable `$title` dans le layout. Le contenu entre les balises `<x-layouts.app>` sera inséré à la place de `{{ $slot }}` dans le layout.

Autres Fonctionnalités Utiles de Blade

- **Directives Conditionnelles** : Blade offre des directives comme `@if`, `@elseif`, `@else` et `@unless` pour gérer les conditions.

  ```
  @if ($utilisateur->estAdmin())
      <p>Bienvenue, administrateur.</p>
  @else
      <p>Bienvenue, utilisateur.</p>
  @endif
  ```

- **Boucles** : Pour itérer sur des collections ou des tableaux, tu peux utiliser `@foreach`, `@for`, `@while` et `@forelse`.

  ```
  @foreach ($articles as $article)
      <h2>{{ $article->titre }}</h2>
      <p>{{ $article->contenu }}</p>
  @endforeach
  ```

- **Inclusion de Vues** : Avec la directive `@include`, tu peux inclure une vue à l'intérieur d'une autre.

  ```
  @include('partials.entete')
  ```

En adoptant ces fonctionnalités de Blade, tu pourras créer des vues Laravel propres, modulaires et maintenables, ce qui facilitera le développement et améliorera la qualité globale de ton application.

Pour approfondir tes connaissances sur Blade, consulte la documentation :

- https://laravel.com/docs/12.x/blade

Ouais, parfaitement clair (pour une fois, ça change 😉).
Voici une liste révisée des exercices pour entraîner tes étudiants débutants à créer des vues Blade simples sans s'embourber avec une DB ou des layouts compliqués. Let's go 🚀 :

Exercices pratiques sur les vues Blade

Exercice 1 : Page d'Accueil simple

Objectif : Créer et afficher une vue basique.

- Crée une route `/accueil` qui affiche une vue `accueil.blade.php`.
- Dans cette vue, affiche un titre et un paragraphe de bienvenue.

Exercice 2 : Page "À propos"

Objectif : Passer et afficher des données simples dans une vue.

- Crée une route `/a-propos` qui affiche une vue `a-propos.blade.php`.
- Passe à la vue un tableau contenant plusieurs informations sur l'entreprise (nom, année de création, mission).
- Affiche ces informations dans la vue sous forme de liste.

Exercice 3 : Affichage d'une liste avec conditions

Objectif : Utiliser les boucles et les conditions Blade.

Crée une route `/membres` qui transmet à la vue `membres.blade.php` un tableau de membres suivant :

```php
$membres = [
    ['prenom' => 'Alice', 'age' => 29],
    ['prenom' => 'Bob', 'age' => 34],
    ['prenom' => 'Chloé', 'age' => 17],
    ['prenom' => 'Damien', 'age' => 42],
];
```

- Affiche chaque membre dans une liste.
- Si le membre est majeur (18+), affiche son âge. Sinon, affiche "Mineur".

Exercice 4 : Création d'un composant anonyme "Card"

Objectif : Créer et réutiliser un composant Blade anonyme.

- Crée un composant anonyme nommé `card.blade.php` dans `resources/views/components/`.
- Ce composant doit afficher un titre et un texte descriptif.
- Crée une route `/services` avec une vue qui affiche trois cartes différentes présentant des services fictifs (ex : Design, Développement, Marketing).

Indice : Passe les données au composant via des attributs.

Exercice 5 : Liste de produits avec lien vers une vue détaillée

Objectif : Créer des vues dynamiques avec des liens et afficher des données spécifiques.

- Crée une route `/produits` qui passe ce tableau de produits à la vue `produits.blade.php` :

```php
$produits = [
    ['id' => 1, 'nom' => 'PC Portable', 'prix' => 1200, 'en_promo' => true],
    ['id' => 2, 'nom' => 'Souris Gaming', 'prix' => 80, 'en_promo' => false],
    ['id' => 3, 'nom' => 'Clavier Mécanique', 'prix' => 150, 'en_promo' => true],
    ['id' => 4, 'nom' => 'Casque Audio', 'prix' => 100, 'en_promo' => false],
];
```

- Affiche chaque produit avec son nom et son prix.
- Si un produit est en promo, affiche à côté "🔥 Promo !".
- Chaque produit doit contenir un lien vers une route `/produits/{id}`.

Ensuite :

- Crée la route `/produits/{id}` pour afficher une vue `produit.blade.php` présentant uniquement les détails du produit choisi.

Exercice 6 : Composant Blade pour produits

Objectif : Réutiliser du code Blade en créant un composant pour afficher chaque produit.

- Reprends la liste des produits fournie précédemment (Exercice 5).
- Crée un composant anonyme `produit.blade.php` qui affiche proprement un produit (nom, prix, promo éventuelle).
- Modifie la vue précédente pour utiliser ton composant à chaque produit de la liste.

1.10 Les Modèles dans Laravel

À quoi servent les modèles?

Dans le cadre du développement d'applications avec Laravel, les modèles jouent un rôle crucial en tant que partie intégrante de l'architecture MVC (Modèle-Vue-Contrôleur). Le modèle est responsable de la gestion des données et des règles métier de l'application. Il communique directement avec la base de données et sert d'intermédiaire entre la base de données et les contrôleurs.

- **Représentation des données** : Chaque modèle représente une table dans la base de données, où chaque instance du modèle correspond à une ligne de cette table.
- **Abstraction des opérations** : Les modèles permettent d'effectuer des opérations CRUD (Création, Lecture, Mise à jour, Suppression) sans avoir à écrire de requêtes SQL complexes.

Utilisation d'Eloquent au lieu de PDO

Laravel propose une approche simplifiée pour interagir avec les bases de données grâce à Eloquent, son ORM (Object-Relational Mapping) intégré. Eloquent permet aux développeurs de travailler avec les données de la base comme s'ils manipulaient des objets PHP.

- **Abstraction et simplicité** : Eloquent élimine le besoin d'écrire des requêtes SQL manuellement, ce qui réduit la complexité du code et les erreurs potentielles.
- **Opérations intuitives** : Avec Eloquent, les opérations de base de données sont effectuées via des méthodes simples et expressives.

Comprendre l'ORM

Un ORM est un outil qui crée un "pont" entre les objets du langage de programmation (dans notre cas, PHP) et les tables de base de données. Il permet aux développeurs de manipuler les données de la base de données en utilisant des classes et des objets, au lieu de requêtes SQL.

- **Eloquent** : C'est l'ORM par défaut de Laravel qui suit le pattern Active Record, permettant à chaque modèle de correspondre à une table dans la base de données et à chaque instance de modèle à une ligne dans cette table.

Exemples de code avec un modèle `User`

Voici un exemple simple de la façon dont un modèle `User` peut être utilisé dans Laravel :

```php
// Création d'un nouvel utilisateur
user=newUser();user = new User();
user=newUser();user→name = 'John Doe';
user->email=’john@example.com’;user→email = 'john@example.com';
user->email=’john@example.com’;user→save();

// Lecture d'un utilisateur par son id
user=User::find(1);//Misea`jourd’unutilisateuruser = User::find(1);

// Mise à jour d'un utilisateur
user=User::find(1);//Misea`jourd’unutilisateuruser→email = 'john.doe@example.com';
user->save();//Suppressiond’unutilisateuruser→save();

// Suppression d'un utilisateur
user->save();//Suppressiond’unutilisateuruser→delete();
```

Ces opérations montrent comment les modèles Eloquent permettent de manipuler les données de manière élégante et efficace.

Création d'un modèle avec Artisan

Laravel utilise une interface en ligne de commande appelée Artisan qui facilite diverses tâches de développement, y compris la création de modèles. Pour créer un modèle, vous utilisez la commande suivante :

```bash
php artisan make:model Animal
```

Cette commande crée un fichier de modèle dans le répertoire `app/Models`.

Exemple de fichier de modèle créé

Voici à quoi ressemble le fichier de modèle `Animal` créé par la commande Artisan :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Animal extends Model
{
    use HasFactory;

    // Vous pouvez ajouter des propriétés et des méthodes personnalisées ici
}
```

Explication du contenu du modèle

- **Namespace** : Le modèle appartient à l'espace de noms `App\Models`, ce qui aide à organiser le code de l'application.
- **Base Class** : Le modèle `Animal` étend la classe `Model` de Eloquent, ce qui lui permet d'hériter de toutes les fonctionnalités ORM d'Eloquent.
- **Trait** `HasFactory` : Ce trait est utilisé pour intégrer la capacité de générer des instances de modèle via des factories, utile pour les tests ou le seeding de votre base de données.

Personnalisation du modèle

Vous pouvez personnaliser le modèle en ajoutant des propriétés telles que `$fillable` ou `$guarded` pour gérer l'assignation en masse, ou en définissant des relations comme `belongsTo` ou `hasMany` pour décrire les relations entre les tables.

```php
class Animal extends Model
{
    use HasFactory;

    protected $fillable = ['name', 'type', 'age'];

    public function owner()
    {
        return $this->belongsTo(Owner::class);
    }
}
```

Dans cet exemple, le modèle `Animal` peut maintenant avoir des champs `name`, `type`, et `age` sécurisés pour l'assignation en masse, et il est lié à un modèle `Owner` via une relation `belongsTo`.

Cette structure vous permet de commencer à travailler avec des données de manière orientée objet très rapidement et efficacement dans votre application Laravel.

Système de migration

Le système de migration de Laravel permet de définir la structure de la base de données de l'application en PHP, ce qui facilite le partage et la mise à jour de la structure de la base de données.

- **Création de migrations** : Laravel utilise des migrations pour créer et modifier les tables de la base de données.
- **Exemple de migration pour la table** `users` :

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Cette migration crée une nouvelle table `users` avec quelques champs de base comme `id`, `name`, `email` et `timestamps`.

Méthodes Eloquent utiles

Sélection de données

Eloquent fournit une variété de méthodes pour récupérer et filtrer les données de manière simple et expressive. Par exemple, pour récupérer tous les utilisateurs d'une base de données, on peut simplement utiliser :

```php
$users = User::all();
```

Pour des requêtes plus spécifiques, Eloquent permet d'utiliser des méthodes comme `find`, `where`, `orderBy`, permettant une grande flexibilité. Par exemple, pour trouver un utilisateur par son email :

```php
$user = User::where('email', 'john@example.com')->first();
```

Mise à jour de données

La mise à jour des données est également simplifiée avec Eloquent. Après avoir récupéré un modèle, on peut modifier ses propriétés et le sauvegarder :

```php
user=User::find(1);user = User::find(1);
user=User::find(1);user->email = 'new.email@example.com';
$user->save();
```

Pour les mises à jour en masse, Laravel fournit la méthode `update` qui peut être utilisée directement sur une requête :

```php
User::where('active', 0)->update(['active' => 1]);
```

Suppression de données

La suppression de modèles est tout aussi intuitive. Pour supprimer un modèle, il suffit d'appeler la méthode `delete` sur une instance de modèle :

```php
user=User::find(1);user = User::find(1);
user=User::find(1);user->delete();
```

Pour une suppression en masse, on peut combiner `where` avec `delete` :

```php
User::where('active', 0)->delete();
```

Relations entre les modèles

Types de relations

Laravel Eloquent supporte plusieurs types de relations qui reflètent les relations entre les tables dans une base de données. Comprendre ces relations est essentiel pour modéliser efficacement les interactions entre les données.

- **hasOne** : Utilisée pour indiquer qu'un modèle possède une relation un-à-un avec un autre modèle. Par exemple, un utilisateur a une adresse.
- **belongsTo** : L'inverse de `hasOne` et `hasMany`. Elle est utilisée pour indiquer qu'un modèle appartient à un autre modèle. Par exemple, une adresse appartient à un utilisateur.
- **hasMany** : Indique qu'un modèle possède une relation un-à-plusieurs avec un autre modèle. Par exemple, un utilisateur peut avoir plusieurs articles de blog.
- **belongsToMany** : Utilisée pour les relations plusieurs-à-plusieurs. Par exemple, un utilisateur peut avoir plusieurs rôles et un rôle peut être attribué à plusieurs utilisateurs.

hasOne

Création dans la base de données avec migrations Laravel

Pour définir une relation `hasOne` entre `User` et `Profile`, voici comment vous pourriez structurer vos migrations :

```php
// Migration pour la table 'users'
Schema::create('users', function (Blueprint table) {
    $table->id();
    $table->string('name');
    $table->string('email')->unique();
    $table->timestamps();
});

// Migration pour la table 'profiles'
Schema::create('profiles', function (Blueprint table) {
    table->id();table->id();
    table->id();table->unsignedBigInteger('user_id');
    table->text('bio');table->text('bio');
    table->text('bio');table->timestamps();

    $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
});
```

On peut aussi écrire la migration de la table profiles comme ceci :

```php
Schema::create('profiles', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')->constrained()->onDelete('cascade');
    $table->string('bio');
    $table->timestamps();
});
```

On peut ajouter la clé étrangère ainsi que les contraintes d'intégrité en une seule ligne.

Définition de la relation dans la classe User

Pour exploiter la relation `hasOne`, ajoutez la méthode suivante dans la classe `User` :

```php
public function profile() {
    return $this->hasOne(Profile::class);
}
```

Simple : Récupération des données d'un utilisateur et de son profil.

```php
$user = User::with('profile')->find(1);
```

Ajout d'un profil à un utilisateur : Pour créer un nouveau profil et l'associer à un utilisateur existant.

```php
user=User::find(1);user = User::find(1);
user=User::find(1);profile = new Profile(['bio' => 'Développeur Laravel passionné']);
user->profile()->save(user->profile()->save(user->profile()->save(profile);
```

Avancée : Chargement des données d'un utilisateur avec une condition spécifique sur le profil.

```php
user=User::whereHas('profile',function(user = User::whereHas('profile',
function(user=User::whereHas('profile',function(query) {
    $query->where('bio', 'like', '%Laravel%');
})->first();
```

Création d'un utilisateur et de son profil simultanément : Pour créer un utilisateur et son profil :

```php
user=User::create(['name'=>'Thibault','email'=>'thibault@example.com']);user =
User::create(['name' => 'Thibault', 'email' => 'thibault@example.com']);
user=User::create(['name'=>'Thibault','email'=>'thibault@example.com']);profile = new
Profile(['bio' => 'Enseignant en informatique et développeur Laravel']);
user->profile()->save(user->profile()->save(user->profile()->save(profile);
```

belongsTo

C'est la relation inverse de hasOne définie dans le modèle `User`.

> *Le profil appartient à l'utilisateur.*

Création dans la base de données avec migrations Laravel

La clé étrangère est déjà définie dans la migration de la table `profiles`.

Définition de la relation dans la classe Profile

Pour exploiter la relation `belongsTo`, ajoutez la méthode suivante dans la classe `Profile` :

```php
public function user() {
    return $this->belongsTo(User::class);
}
```

Requêtes avec belongsTo et Manipulation de Données

Simple : Récupérer l'utilisateur associé à un profil.

```php
profile=Profile::find(1);profile = Profile::find(1);
profile=Profile::find(1);user = $profile->user;
```

Ajout d'un nouvel objet à la relation : Créer un nouvel utilisateur et l'associer à un profil.

```php
user=newUser(['name'⇒'NouvelUtilisateur','email'⇒'email@example.com']);user = new User(['name'
⇒ 'Nouvel Utilisateur', 'email' ⇒ 'email@example.com']);
user=newUser(['name'⇒'NouvelUtilisateur','email'⇒'email@example.com']);profile =
Profile::find(1);
profile->user()->associate(profile→user()→associate(profile->user()->associate(user);
$profile→save();
```

Avancée : Charger tous les profils avec leur utilisateur respectif, en filtrant les utilisateurs selon un critère.

```php
profiles=Profile::whereHas('user',function(profiles = Profile::whereHas('user',
function(profiles=Profile::whereHas('user',function(query) {
    $query→where('name', 'like', '%John%');
})→get();
```

Création d'un profil et association à un utilisateur existant : Supposons que nous avons un utilisateur existant auquel nous voulons associer un nouveau profil.

```php
user=User::find(1);//Supposonsquecetutilisateurestdeˊjaˋexistantuser = User::find(1); // Supposons
que cet utilisateur est déjà existant
user=User::find(1);//Supposonsquecetutilisateurestdeˊjaˋexistantprofile = new Profile(['bio' ⇒
'Biographie du profil', 'birthdate' ⇒ '2000-01-01']);
user->profile()->save(user→profile()→save(user->profile()->save(profile);
```

hasMany

Création dans la base de données avec migrations Laravel

La relation `hasMany` relie un modèle à plusieurs instances d'un autre modèle. Par exemple, un utilisateur (`User`) peut avoir plusieurs posts (`Post`). Voici comment créer une migration pour la table `posts` :

```php
// Migration pour la table 'posts'
Schema::create('posts', function (Blueprint $table) {
    $table→id();
    $table→unsignedBigInteger('user_id');
    $table→string('title');
    $table→text('body');
    $table→timestamps();

    $table→foreign('user_id')→references('id')→on('users')→onDelete('cascade');
});
```

Définition des relations dans les modèles

Pour compléter les migrations et établir les relations entre nos modèles `User` et `Post` dans Laravel, voici comment procéder :

Dans le modèle `User`, ajoutez la méthode suivante pour définir la relation `hasMany` avec le modèle `Post` :

```php
// Dans le modèle User
public function posts() {
    return $this->hasMany(Post::class);
}
```

Cette méthode signifie qu'un utilisateur peut avoir plusieurs posts. Elle utilise la méthode `hasMany` de Eloquent, le système de gestion de base de données ORM de Laravel, pour établir cette relation.

D'autre part, dans le modèle `Post`, vous devez définir la relation inverse `belongsTo` pour spécifier qu'un post appartient à un utilisateur :

```php
// Dans le modèle Post
public function user() {
    return $this->belongsTo(User::class);
}
```

Cette méthode établit qu'un post est associé à un seul utilisateur. En utilisant `belongsTo`, on crée une relation inverse, permettant de naviguer de chaque post à son utilisateur propriétaire.

Requêtes avec hasMany

Simple : Récupérer tous les posts d'un utilisateur.

```php
user=User::find(1);user = User::find(1);
user=User::find(1);posts = $user->posts;
```

Ajout d'un nouvel objet à la relation : Créer un nouveau post et l'associer à un utilisateur.

```php
post=newPost(['title'⇒'MonnouveauPost','body'⇒'Contenudupost']);post = new Post(['title' ⇒ 'Mon
nouveau Post', 'body' ⇒ 'Contenu du post']);
post=newPost(['title'⇒'MonnouveauPost','body'⇒'Contenudupost']);user = User::find(1);
user->posts()->save(user->posts()->save(user->posts()->save(post);
```

Avancée : Récupérer tous les utilisateurs avec leurs posts, en filtrant les posts selon un critère.

```php
users=User::with(['posts'⇒function(users = User::with(['posts' ⇒
function(users=User::with(['posts'⇒function(query) {
    $query->where('title', 'like', '%Laravel%');
}])->get();
```

Création de plusieurs posts et association à un utilisateur existant : Pour créer plusieurs posts et les associer à un utilisateur existant, on peut procéder comme suit :

```php
user=User::find(1);//Supposonsquecetutilisateurestde´ja`existantuser = User::find(1); // Supposons que cet utilisateur est déjà existant
user=User::find(1);//Supposonsquecetutilisateurestde´ja`existantposts = [
    new Post(['title' => 'Post 1', 'body' => 'Contenu du post 1']),
    new Post(['title' => 'Post 2', 'body' => 'Contenu du post 2'])
];
user->posts()->saveMany(user→posts()→saveMany(user->posts()->saveMany(posts);
```

Gestion des relations many-to-many avec Laravel

Mise en place dans la base de données

La relation `belongsToMany`, illustrée par la liaison entre `User` et `Role`, nécessite une table intermédiaire, souvent nommée table pivot. Voici comment la créer grâce aux migrations Laravel :

```php
// Création de la table pivot 'role_user'
Schema::create('role_user', function (Blueprint $table) {
    $table→unsignedBigInteger('user_id'); // Clé étrangère pour User
    $table→unsignedBigInteger('role_id'); // Clé étrangère pour Role

    // Déclaration des contraintes de clé étrangère
    $table→foreign('user_id')→references('id')→on('users')→onDelete('cascade');
    $table→foreign('role_id')→references('id')→on('roles')→onDelete('cascade');

    // Définition de la clé primaire composée
    $table→primary(['user_id', 'role_id']);
});
```

Création et association de modèles

Pour exploiter pleinement la relation `belongsToMany`, il faut définir cette relation dans les modèles concernés. Voici comment procéder :

1. **Modèle User** : Intégrer une méthode `roles` retournant la relation.

```php
public function roles() {
    return $this→belongsToMany(Role::class);
}
```

2. **Modèle Role** : Ajouter une méthode similaire pour accéder aux utilisateurs associés.

```php
public function users() {
    return $this→belongsToMany(User::class);
}
```

Pour associer un rôle à un utilisateur (et inversement), Laravel offre des méthodes telles que `attach`, `detach` et `sync` permettant de manipuler facilement les enregistrements dans la table pivot.

- **Attachement** d'un rôle à un utilisateur :

```php
user->roles()->attach(user→roles()→attach(user->roles()->attach(roleId); // Associe un rôle par son ID
```

- **Détachement** d'un rôle :

```php
user->roles()->detach(user→roles()→detach(user->roles()->detach(roleId); // Supprime l'association avec le rôle
```

- **Synchronisation** des rôles :

```php
user->roles()->sync([user→roles()→sync([user->roles()->sync([roleId1, $roleId2]); // Synchronise les rôles, en supprimant les anciennes associations et en ajoutant les nouvelles.
```

Ces méthodes facilitent la gestion dynamique des relations entre les modèles dans des applications complexes.

Exploitation de la relation `belongsToMany`

- **Basique** : Pour récupérer les rôles associés à un utilisateur donné :

```php
user=User::find(1);//Trouvel'utilisateuravecl'ID1user = User::find(1); // Trouve l'utilisateur avec l'ID 1
user=User::find(1);//Trouvel'utilisateuravecl'ID1roles = $user→roles; // Récupère les rôles liés
```

- **Avancé** : Pour trouver les utilisateurs ayant le rôle "Admin" :

```php
users=User::whereHas('roles',function(users = User::whereHas('roles',
function(users=User::whereHas('roles',function(query) {
    $query→where('name', 'Admin'); // Filtre sur le nom du rôle
})→get();
```

Relations polymorphiques

Création dans la base de données avec migrations Laravel

Les relations polymorphiques permettent à un modèle d'appartenir à plus d'un autre modèle sur une seule association. Par exemple, si vous avez des `Comment` pouvant appartenir à `Post` ou `Video`, vous structurerez vos migrations comme suit :

```php
// Migration pour la table 'comments'
Schema::create('comments', function (Blueprint $table) {
    $table→id();
    $table→string('body');
    $table→morphs('commentable'); // Cela crée 'commentable_type' et 'commentable_id'
    $table→timestamps();
});
```

Notation dans les modèles

Vous devez définir la relation dans les modèles `Comment`, `Post`, et `Video` :

```php
// Dans le modèle Comment
class Comment extends Model
{
    public function commentable()
    {
        return this→morphTo();
    }
}

// Dans les modèles Post et Video
class Post extends Model
{
    public function comments()
    {
        return this→morphMany(Comment::class, 'commentable');
    }
}

class Video extends Model
{
    public function comments()
    {
        return $this→morphMany(Comment::class, 'commentable');
    }
}
```

Requêtes avec relations polymorphiques

Simple : Accéder aux commentaires d'un post.

```php
post=Post::find(1);post = Post::find(1);
post=Post::find(1);comments = $post→comments;
```

Avancée : Charger tous les commentaires pour un type spécifique de "commentable" avec un critère spécifique.

```php
$comments = Comment::where('commentable_type', Post::class)
                    →where('body', 'like', '%Laravel%')
                    →get();
```

Exemples de requêtes avancées

Pour approfondir l'utilisation des relations dans Laravel, voici quelques exemples de requêtes avancées qui montrent la puissance et la flexibilité d'Eloquent.

Filtrage avec des relations profondes

Imaginons que vous voulez récupérer tous les utilisateurs qui ont posté des commentaires contenant le mot "Laravel" sur des posts. Vous pouvez utiliser la méthode `whereHas` de manière imbriquée :

```php
users=User::whereHas('posts.comments',function(users = User::whereHas('posts.comments', function
(users=User::whereHas('posts.comments',function(query) {
    $query->where('body', 'like', '%Laravel%');
})->get();
```

Factories et Seeders

Pour tester efficacement une application, il est souvent nécessaire de générer des données factices. Laravel fournit deux outils puissants pour cela : les **Factories** et les **Seeders**.

Factories

Les **Factories** permettent de définir des modèles de données pour générer automatiquement des enregistrements de test. Elles utilisent la bibliothèque Faker pour générer des données réalistes.

Création d'une Factory

Pour créer une factory pour le modèle `User`, utilisez la commande Artisan :

```bash
php artisan make:factory UserFactory --model=User
```

Cela crée un fichier dans `database/factories` où vous pouvez définir la structure de vos données factices :

```php
use Faker\Generator as Faker;

factory->define(User::class,function(Fakerfactory->define(User::class, function (Faker
factory->define(User::class,function(Fakerfaker) {
    return [
        'name' => faker->name,'email'=>faker->name,
        'email' => faker->name,'email'=>faker->unique()->safeEmail,
        'password' => bcrypt('secret'), // ou Hash::make('secret')
    ];
});
```

Utilisation d'une Factory

Pour générer une instance de modèle sans la persister en base de données :

```php
$user = factory(User::class)->make();
```

Pour créer une instance de modèle et la persister :

```php
$user = factory(User::class)->create();
```

Seeders

Les **Seeders** permettent de peupler votre base de données avec des données de test en utilisant les factories ou en insérant directement des données.

Création d'un Seeder

Pour créer un seeder pour le modèle `User`, utilisez la commande Artisan :

```bash
php artisan make:seeder UsersTableSeeder
```

Dans le fichier créé dans `database/seeders`, vous pouvez définir comment peupler votre table :

```php
use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
{
    public function run()
    {
        // Créer 10 utilisateurs
        factory(User::class, 10)->create();
    }
}
```

Exécution des Seeders

Pour exécuter un seeder particulier :

```bash
php artisan db:seed --class=UsersTableSeeder
```

Ou pour exécuter tous les seeders définis :

```bash
php artisan db:seed
```

Pour détruire la base de données, la recréer et la peupler :

```bash
php artisan migrate:fresh --seed
```

Exemples de requêtes avancées avec Factories et Seeders

Relations avec Factories

Si vous avez des relations entre vos modèles, par exemple, un utilisateur ayant plusieurs posts, vous pouvez définir cela dans vos factories :

```php
factory->define(Post::class,function(Fakerfactory→define(Post::class, function (Faker
factory->define(Post::class,function(Fakerfaker) {
    return [
        'user_id' ⇒ factory(User::class),
        'title' ⇒ faker->sentence,'body'⇒faker→sentence,
        'body' ⇒ faker->sentence,'body'⇒faker→paragraph,
    ];
});
```

Et dans vos seeders, vous pouvez utiliser les relations pour créer des données cohérentes :

```php
user=factory(User::class)->create();user = factory(User::class)→create();
user=factory(User::class)->create();posts = factory(Post::class, 5)→create(['user_id' ⇒ $user
→id]);
```

Ces outils offrent une flexibilité incroyable pour générer des données de test, permettant de tester efficacement votre application dans divers scénarios.

Bonnes pratiques avec Eloquent

L'utilisation efficace d'Eloquent dans Laravel n'est pas seulement une question de savoir comment créer des modèles ou exécuter des requêtes. Il est également crucial d'adopter de bonnes pratiques pour assurer la sécurité, les performances et la maintenabilité de votre application. Voici quelques conseils essentiels :

Gestion de l'Assignation Massive

L'**Assignation Massive** représente une méthode efficace pour attribuer simultanément plusieurs propriétés à un modèle. Toutefois, elle requiert une manipulation prudente pour éviter les vulnérabilités de sécurité.

Configuration des Attributs Autorisés

Afin de se prémunir contre les risques d'assignation massive indésirable, il est crucial de spécifier clairement quels attributs peuvent être assignés de cette manière dans le modèle. Ceci se fait comme suit :

```php
protected $fillable = ['nom', 'email', 'mot_de_passe'];
```

Exemples d'Utilisation

Création d'un nouvel enregistrement :

Lors de l'utilisation de la méthode `create`, Laravel permet d'insérer un nouvel enregistrement dans la base de données en utilisant les attributs définis comme `fillable`.

```php
$user = User::create([
    'nom' => 'Jean Dupont',
    'email' => 'jean.dupont@example.com',
    'mot_de_passe' => Hash::make('password123'),
]);
```

Mise à jour d'un enregistrement existant :

Pour la mise à jour, la méthode `update` permet de modifier plusieurs attributs d'un modèle existant en une seule fois, en respectant toujours les règles définies par `fillable`.

```php
$user->update([
    'nom' => 'Jean Dupont',
    'email' => 'jean.nouveau@example.com',
]);
```

Ces exemples démontrent comment manipuler l'assignation massive de manière sécurisée, en prenant soin de définir précisément quels attributs peuvent être assignés ou modifiés de cette façon.

Conventions de nommage

Respecter les conventions de nommage de Laravel facilite la compréhension du code par d'autres développeurs et par vous-même dans le futur.

- **Modèles** : Singulier avec la première lettre en majuscule (ex. `User`, `Post`).
- **Tables de base de données** : Pluriel et en minuscules (ex. `users`, `posts`).
- **Clés étrangères** : Nom du modèle en singulier suivi de `_id` (ex. `user_id`, `post_id`).

Sécurisation des requêtes

Eloquent protège contre les injections SQL grâce à la préparation des requêtes. Cependant, il est toujours important d'être vigilant, surtout lorsque vous utilisez des requêtes SQL brutes ou des clauses whereRaw.

Validation des entrées

Toujours valider les données entrantes, surtout si elles sont utilisées dans des requêtes SQL, pour éviter toute manipulation malveillante :

```php
$request->validate([
    'email' => 'required|email',
    'password' => 'required|min:8',
]);
```

Optimisation des requêtes : Éliminer les N+1 queries

Les **N+1 queries** désignent une inefficacité courante en base de données, spécialement avec les ORM comme Eloquent. Imaginez que vous avez une liste d'utilisateurs et que pour chaque utilisateur, vous souhaitez récupérer ses publications. Sans optimisation, vous lancez une première requête pour obtenir tous les

utilisateurs, puis, pour chaque utilisateur, une requête supplémentaire pour ses publications. Si vous avez 10 utilisateurs, cela équivaut à 1 (pour les utilisateurs) + 10 (une pour chaque utilisateur pour ses publications) = 11 requêtes au total.

Solution : Le eager loading

Pour contourner ce piège, Laravel propose le **eager loading** avec la méthode `with`, permettant de récupérer toutes les données en une fois :

```php
// Requête inefficace : N+1 queries
users=User::all();foreach(users = User::all();
foreach (users=User::all();foreach(users as user) {
    $posts = $user->posts; // Une requête par utilisateur
}

// Requête optimisée avec chargement eager
users = User::with('posts')->get(); // Une seule requête pour tout récupérer
```

Cette approche réduit considérablement le nombre de requêtes, améliorant la performance de votre application.

Conclusion

En adoptant ces bonnes pratiques, vous maximiserez les avantages d'utiliser Eloquent dans vos projets Laravel. Cela contribuera non seulement à la sécurité et aux performances de vos applications mais aussi à leur maintenabilité sur le long terme.

Création complète avec Artisan

Laravel permet de simplifier le processus de développement en offrant la possibilité de créer simultanément un modèle, sa migration associée, un seeder, une factory, et un contrôleur de type resource. Cette fonctionnalité est particulièrement utile pour mettre en place rapidement les composants nécessaires à une partie de votre application.

Commande d'exemple

Pour créer tous ces éléments en une seule commande, vous pouvez utiliser la commande `make:model` d'Artisan avec plusieurs options :

```bash
php artisan make:model Animal --all
```

Ou, si vous préférez spécifier chaque composant individuellement :

```bash
php artisan make:model Animal -mfsr
```

Options disponibles

Voici les options que vous pouvez utiliser avec la commande `make:model` :

- `-m`, `--migration` : Crée une migration pour le modèle.

- `-f`, `--factory` : Crée une factory pour le modèle.
- `-s`, `--seed` : Crée un seeder pour le modèle.
- `-c`, `--controller` : Crée un contrôleur pour le modèle.
- `-r`, `--resource` : Indique que le contrôleur généré doit être un contrôleur de ressource, qui inclut des méthodes pour les opérations CRUD (Create, Read, Update, Delete).
- `--api` : Similaire à `--resource`, mais le contrôleur sera optimisé pour une API, excluant les méthodes qui ne sont pas nécessaires comme celles pour afficher des vues.

Ces outils combinés offrent une base solide pour développer une partie fonctionnelle de votre application, en respectant les conventions de Laravel et en accélérant le processus de développement initial.

Documentation

- https://laravel.com/docs/11.x/eloquent
- https://laravel.com/docs/11.x/eloquent-relationships
- https://laravel.com/docs/11.x/eloquent-factories
- https://laravel.com/docs/11.x/migrations
- https://laravel.com/docs/11.x/seeding

1.11 Tailwind CSS v3 – Bases & Exemples Pratiques

1. À propos de Tailwind CSS

Tailwind CSS est un **framework CSS utility-first**, c'est-à-dire qu'il propose des **classes prédéfinies** à utiliser directement dans le HTML pour créer rapidement des interfaces stylées, sans écrire de CSS traditionnel.

Avec Tailwind, fini les prises de tête pour nommer ses classes ou jongler entre plusieurs fichiers CSS : tout se fait directement dans votre balisage HTML. Résultat ? Vous développez **plus vite**, votre code est **plus lisible**, et vos interfaces restent **cohérentes**.

Exemple concret rapide :

Au lieu d'écrire du CSS traditionnel comme ceci :

```css
/* CSS traditionnel (ennuyeux) */
.bouton-primaire {
  background-color: #3490dc;
  color: white;
  padding: 10px 20px;
  border-radius: 6px;
}
```

Puis l'utiliser en HTML :

```xml
<button class="bouton-primaire">Cliquez-moi</button>
```

Avec Tailwind, on écrit simplement :

```xml
<button class="bg-blue-500 text-white px-4 py-2 rounded">
  Cliquez-moi
</button>
```

Pas de fichier CSS externe, pas de nom de classe à inventer, juste des utilitaires simples à combiner.

Ce que vous allez apprendre dans ce chapitre :

- Comment utiliser efficacement les classes utilitaires Tailwind pour styliser vos composants (boutons, formulaires, etc.).
- Construire rapidement des mises en page (Flexbox et Grid).
- Rendre vos interfaces responsive et interactives sans douleur.

On passe tout de suite à la pratique ! 😉🔥

2. Les bases incontournables

Ici, tu vas apprendre les fondamentaux pour créer des composants stylés avec Tailwind CSS. Chaque section est illustrée par plusieurs exemples dans des styles très variés (moderne, brutaliste, skeuomorphe). Tu vas vite piger que Tailwind, c'est la simplicité ET la flexibilité.

2.1 Boutons

Un bouton réussi combine une bonne couleur, du padding, un texte lisible et des effets d'interaction simples (hover, focus). Voici trois variantes complètement différentes pour t'inspirer :

Exemple 1 – Moderne (flat, minimaliste)

```xml
<button class="bg-indigo-500 text-white px-4 py-2 rounded-lg hover:bg-indigo-600 transition">
  Bouton Moderne
</button>
```

Exemple 2 – Brutaliste (contrastes forts, style agressif)

```xml
<button class="border-4 border-black bg-yellow-300 text-black px-6 py-3 uppercase font-extrabold
hover:bg-yellow-400 shadow-brutalist transition-transform hover:-translate-x-1 hover:-translate-y-
1">
  Clique-moi fort
</button>
```

(Tu peux définir l'ombre "brutaliste" toi-même plus tard via config Tailwind.)

Exemple 3 – Skeuomorphisme (effet réaliste avec ombres et relief)

```xml
<button class="bg-gradient-to-b from-gray-200 to-gray-400 text-gray-800 px-4 py-2 rounded-lg
shadow-inner border border-gray-500 hover:shadow-xl transition-shadow">
  Bouton 3D
</button>
```

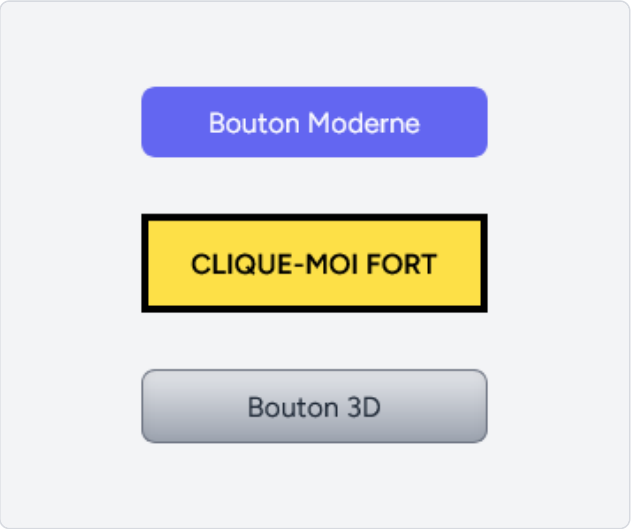


Tableau récap des classes clés pour les boutons :

Classe	Utilité	Exemple
`bg-{couleur}`	Couleur de fond	`bg-indigo-500`
`text-{couleur}`	Couleur texte	`text-white`
`px-{n}`, `py-{n}`	Padding (horiz/vert)	`px-4 py-2`
`rounded`	Coins arrondis	`rounded-lg`, `rounded-full`
`border`	Bordure	`border-2 border-black`
`hover:`	Style au survol	`hover:bg-indigo-600`
`shadow`	Ombre portée	`shadow-inner`, `shadow-xl`

2.2 Champs de formulaire (inputs)

Créer des inputs stylés avec Tailwind, c'est rapide. Voici trois exemples bien différents pour capter le truc :

Exemple 1 – Moderne (simple et épuré)

```xml
<input class="border border-gray-300 rounded px-3 py-2 focus:border-indigo-500 focus:outline-none"
placeholder="Nom complet">
```

Exemple 2 – Brutaliste (épais et massif)

```xml
<input class="border-4 border-black bg-white px-4 py-3 text-lg font-bold focus:bg-yellow-100
focus:outline-none" placeholder="TON NOM EN MAJUSCULES">
```

Exemple 3 – Skeuomorphe (effet relief)

```xml
<input class="border border-gray-400 rounded-lg px-4 py-2 shadow-inner bg-gray-200 focus:bg-white
focus:shadow-lg focus:outline-none" placeholder="Pseudo 3D">
```



Tableau récap des classes clés pour les inputs :

Classe	Utilité	Exemple
`border`	Bordure	`border border-gray-300`
`rounded`	Coins arrondis	`rounded`, `rounded-lg`
`px-{n}`, `py-{n}`	Padding intérieur	`px-4 py-2`
`focus:`	Apparence quand sélectionné	`focus:border-indigo-500`
`shadow`	Ombre intérieure/extérieure	`shadow-inner`, `focus:shadow-lg`

2.3 Mise en page (Layouts : Flexbox & Grid)

Deux façons principales de placer des éléments facilement :

Exemple Flexbox – centrer horizontalement/verticalement

```xml
<div class="flex items-center justify-center h-48 bg-gray-100">
  <div class="bg-blue-500 text-white p-4 rounded">Je suis centré !</div>
</div>
```



Exemple Grid – grille simple 3 colonnes égales

```xml
<div class="grid grid-cols-3 gap-4">
  <div class="bg-green-300 p-4">Bloc 1</div>
  <div class="bg-green-400 p-4">Bloc 2</div>
  <div class="bg-green-500 p-4">Bloc 3</div>
</div>
```



Tableau récap Layout :

Classe	Utilité	Exemple
`flex`, `items-center`	Conteneur Flexbox + alignement	`flex items-center`
`justify-center`	Alignement horizontal (flex)	`justify-center`
`grid`, `grid-cols-{n}`	Créer une grille	`grid grid-cols-3`
`gap-{n}`	Espacement grille ou flex	`gap-4` (1rem)

2.4 Typographie basique

Voici comment contrôler facilement tes textes :

Exemples rapides :

```xml
<h1 class="text-3xl font-bold">Titre principal</h1>
<p class="text-gray-700 leading-relaxed">Paragraphe confortable à lire.</p>
```

```xml
<p class="text-2xl font-black uppercase text-black">BRUTALISTE AGRESSIF</p>
```



Tableau récap typo :

Classe	Utilité	Exemple
`text-{taille}`	Taille police	`text-xl`, `text-3xl`
`font-{poids}`	Graisse texte	`font-bold`, `font-black`
`leading-{esp}`	Hauteur ligne	`leading-relaxed`
`uppercase`	Texte majuscule	`uppercase`

2.5 Couleurs & arrière-plans

Exemple basique de coloration rapide :

```xml
<div class="bg-red-500 text-white p-4">Fond rouge, texte blanc</div>
```

```xml
<div class="bg-black text-yellow-300 p-4">Contraste brutaliste</div>
```

```xml
<div class="bg-gradient-to-b from-gray-100 to-gray-300 text-gray-900 p-4 shadow">Effet
réaliste</div>
```



Tableau récap couleurs :

Classe	Utilité	Exemple
`bg-{couleur}`	Couleur arrière-plan	`bg-indigo-500`
`text-{couleur}`	Couleur texte	`text-white`
`bg-gradient-to-{dir}`	Dégradé	`bg-gradient-to-b` (vers bas)

Avec ça, tu as largement de quoi démarrer et tester plein de choses directement. Ensuite, on attaque le responsive et les variantes interactives.

3. Responsive Design en 2 minutes chrono 🤔

Rendre un design responsive, ça consiste juste à adapter ta page pour qu'elle soit agréable sur n'importe quel écran (du vieux smartphone claqué de ton pote jusqu'au gros écran gamer de riche). Avec Tailwind, t'as juste à ajouter des petits préfixes (`sm:` , `md:` , `lg:` , etc.) devant tes classes. C'est tout, fin du cours, tu peux rentrer chez toi (non).

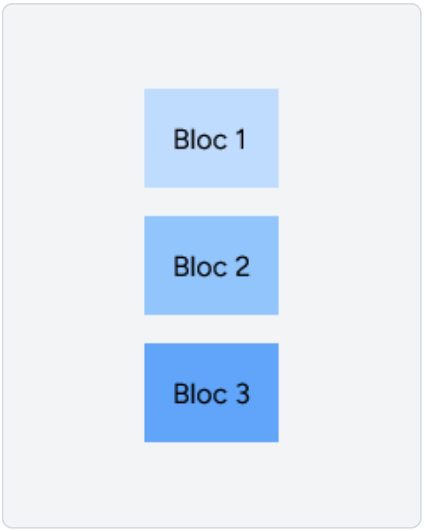
⚡ Exemple responsive simple et rapide :

Ce layout est vertical sur mobile et devient horizontal sur tablette/desktop :

```xml
<div class="flex flex-col md:flex-row gap-4">
  <div class="bg-blue-200 p-4">Bloc 1</div>
  <div class="bg-blue-300 p-4">Bloc 2</div>
  <div class="bg-blue-400 p-4">Bloc 3</div>
</div>
```

Sur mobile :

- Les blocs sont empilés verticalement grâce à `flex-col` .



Dès la taille `md` (tablette) :

- Les blocs s'affichent côte à côte horizontalement avec `md:flex-row`.



📏 Tableau récap des Breakpoints :

Préfixe	Écrans concernés	Taille min
Aucun	Mobile	0px
`sm:`	Grands mobiles	640px
`md:`	Tablettes	768px
`lg:`	Ordinateurs	1024px
`xl:`	Grands écrans desktop	1280px
`2xl:`	Écrans XXL	1536px

🚩 Règle simple à retenir :

- Définis **d'abord** le style mobile (**sans préfixe**).
- Ajoute ensuite les variantes pour écrans plus grands (**avec préfixes**).

Exemple ultra-concret d'un texte qui s'adapte progressivement :

```xml
<p class="text-sm sm:text-base md:text-lg lg:text-xl">
  Texte qui grossit avec la taille de l'écran.
</p>
```

Sur mobile, texte petit (`text-sm`), puis ça augmente progressivement.

Allez, on continue avec les variantes interactives avant que tu décroches complètement (▓'-')▓ 🫠

4. Variantes d'état (hover, focus, disabled...) 💅

Tailwind CSS gère les états interactifs sans que t'aies à écrire du CSS séparé. Tu mets juste des préfixes (`hover:`, `focus:`, `disabled:`...) devant tes classes pour gérer les interactions en 2 secondes. Pas mal, hein ?

🚨 Exemple interactif ultra-simple :

Voici un bouton qui réagit joliment au survol et à la sélection :

```xml
<button class="bg-green-500 text-white px-4 py-2 rounded hover:bg-green-600 focus:ring-4
focus:ring-green-200 focus:outline-none transition">
  Clique-moi stp
</button>
```

- Au **survol** (`hover:`), il devient plus foncé (`bg-green-600`).
- Au **focus** (clic ou sélection clavier), il affiche un petit anneau clair (`focus:ring-4 focus:ring-green-200`).

🚧 Exemple de bouton désactivé :

```xml
<button class="bg-gray-400 text-gray-700 px-4 py-2 rounded cursor-not-allowed opacity-50"
disabled>
  Bouton désactivé
</button>
```

Avec `disabled:`, tu peux gérer facilement le style de boutons inactifs (ici, rendu grisé avec `opacity-50` et curseur barré).

📌 Tableau récap des variantes utiles :

Variante	Utilité	Exemple
`hover:`	Au survol souris	`hover:bg-blue-600`
`focus:`	Lorsque sélectionné (clic, clavier)	`focus:ring-2`, `focus:outline-none`
`active:`	Pendant clic	`active:scale-95`
`disabled:`	Quand élément désactivé	`disabled:opacity-50`

💡 Bonus : combiner variantes responsive & interactives !

Oui, ça marche aussi :

```XML
<button class="bg-red-500 sm:hover:bg-red-700 md:hover:bg-red-800 transition">
  Je change selon la taille et l'interaction !
</button>
```

- Sur mobile : Rouge classique.
- À partir de `sm` (tablette), rouge foncé au survol.
- À partir de `md` (grand écran), encore plus foncé au survol.

Voilà, t'as les bases solides pour coder du style interactif sans galérer dans tes CSS. Maintenant, au boulot !

Sources :

- https://v3.tailwindcss.com/

1.12 Exercice : Gestion d'utilisateurs

Introduction

Ce tutoriel vous guidera dans la création d'une application de gestion d'utilisateurs avec Laravel 12. Vous apprendrez à :

- Installer Laravel et configurer l'environnement de développement (Laragon pour Windows, Laravel Herd pour Mac).
- Utiliser Laravel Breeze pour l'authentification et Tailwind CSS pour le style.
- Créer, modifier, supprimer et consulter des utilisateurs.
- Gérer la base de données SQLite et générer des données fictives avec des seeders.

Les sujets couverts incluent :

- Routes, contrôleurs, vues, et gestion de base de données.
- Pagination et recherche des utilisateurs.

Des exercices pratiques vous permettront d'appliquer les concepts expliqués : création d'une vue de détail, édition, suppression et ajout de filtres par rôle.

L'objectif est de comprendre le fonctionnement de Laravel tout en acquérant de l'autonomie pour développer des applications web bien structurées.

Pré-requis et Installation

Pour suivre ce tutoriel, il est nécessaire de disposer d'un environnement de développement configuré correctement. Voici les étapes à suivre :

Pré-requis

- PHP 8.2+ installé sur votre machine.
- Composer (gestionnaire de dépendances PHP).
- Un éditeur de code (comme Visual Studio Code).

Environnement de Développement

- **Windows** : Utilisez **Laragon** pour simplifier l'installation et la configuration des services nécessaires (PHP, SQLite, etc.).
- **Mac** : Utilisez **Laravel Herd**, une solution facile pour gérer votre environnement Laravel.

Installation de Laravel 12

1. Ouvrez votre terminal.
2. Exécutez la commande suivante pour créer un nouveau projet Laravel :

```bash
composer create-project laravel/laravel nom-du-projet
```

Installation de Laravel Breeze

Laravel Breeze permet d'ajouter rapidement l'authentification à votre application.

1. Installez Laravel Breeze via Composer :

```bash
composer require laravel/breeze --dev
```

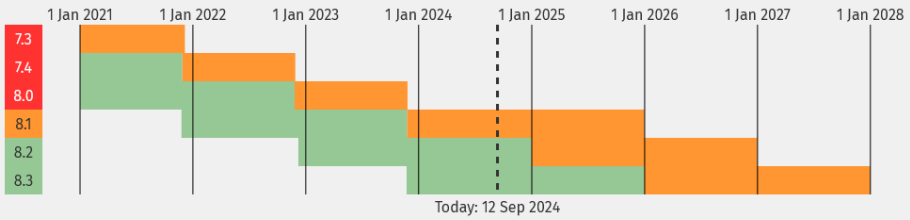
2. Exécutez la commande pour installer les fichiers de Breeze :

```bash
php artisan breeze:install
```

Currently Supported Versions

Branch	Initial Release		Active Support Until		Security Support Until	
8.1	25 Nov 2021	*2 years, 9 months ago*	25 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*
8.2	8 Dec 2022	*1 year, 9 months ago*	31 Dec 2024	*in 3 months*	31 Dec 2026	*in 2 years, 3 months*
8.3	23 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*	31 Dec 2027	*in 3 years, 3 months*

Or, visualised as a calendar:



Key

Active support	A release that is being actively supported. Reported bugs and security issues are fixed and regular point releases are made.
Security fixes only	A release that is supported for critical security issues only. Releases are only made on an as-needed basis.
End of life	A release that is no longer supported. Users of this release should upgrade as soon as possible, as they may be exposed to unpatched security vulnerabilities.

Attention ! N'installez pas le dark mode ! Si vous l'installez, vous devrez écrire le code Tailwind pour le mode clair ET le mode sombre. C'est 2 fois plus de travail pour un exercice dont ce n'est pas l'objectif.

Votre environnement de développement est maintenant prêt pour commencer à travailler sur la gestion des utilisateurs.

Génération des Données de Démonstration (Seeders)

Dans cette section, nous allons générer des données fictives pour peupler notre application avec une liste d'utilisateurs. Cela nous permettra de tester facilement les fonctionnalités de notre application. Pour cela, nous utiliserons des seeders Laravel.

Modification du DatabaseSeeder

1. **Modifier le DatabaseSeeder** : Nous devons modifier le fichier `DatabaseSeeder.php` pour générer 200 utilisateurs aléatoires et notre compte utilisateur de test.
   - Ouvrez le fichier `DatabaseSeeder.php` situé dans le dossier `database/seeders`.
   - Modifiez la méthode `run()` comme suit :

```php
public function run()
{
    \App\Models\User::factory(200)->create();

    User::factory()->create([
        'name' => 'Test User',
        'email' => 'test@example.com',
    ]);
}
```

2. **Exécuter le Seeder** : Enfin, exécutez le seeder pour remplir la base de données avec les utilisateurs générés.

- Dans le terminal, lancez la commande :

```bash
php artisan db:seed
```

- Cette commande créera 201 utilisateurs fictifs dans votre base de données.

Vérification des Données Générées

- Une fois le seeder exécuté, vous pouvez vérifier les données générées dans votre base de données avec un outil comme **phpMyAdmin** ou **HeidiSQL**.

Ces données fictives nous permettront de tester et de valider les fonctionnalités de notre application de manière simple et efficace. Passons maintenant à la création des contrôleurs et des routes nécessaires pour gérer ces utilisateurs.

Ajout d'une Colonne "Rôle" aux Utilisateurs

Pour permettre une gestion plus flexible des utilisateurs, nous allons ajouter une colonne "rôle" à notre table `users`. Cela nous permettra de distinguer les utilisateurs "admin" des utilisateurs classiques "user".

Création d'une Nouvelle Migration

1. **Générer la Migration** : Utilisez Artisan pour créer une nouvelle migration qui ajoutera la colonne `role` à la table `users`. Exécutez la commande suivante dans le terminal :

```bash
php artisan make:migration add_role_to_users_table --table=users
```

Cette commande créera un nouveau fichier de migration dans le dossier `database/migrations`.

2. **Modifier la Migration** : Ouvrez le fichier de migration nouvellement créé et ajoutez le code nécessaire pour ajouter la colonne `role` :

```php
public function up()
{
    Schema::table('users', function (Blueprint table) {
        $table->string('role')->default('user');
    });
}

public function down()
{
    Schema::table('users', function (Blueprint table) {
        $table->dropColumn('role');
    });
}
```

- La colonne `role` sera ajoutée avec une valeur par défaut de "user".

3. **Exécuter la Migration** : Pour appliquer cette modification à la base de données, lancez la commande suivante :

```bash
php artisan migrate
```

Mise à Jour de la Factory

Pour attribuer des rôles aux utilisateurs, nous allons maintenant mettre à jour la factory `UserFactory.php` :

1. **Modifier la Factory** : Ouvrez le fichier `UserFactory.php` situé dans le dossier `database/factories`.
   - Ajoutez un champ `role` et utilisez Faker pour que seulement **1 utilisateur sur 10** soit administrateur :

     ```php
     'role' => $this->faker->boolean(10) ? 'admin' : 'user',
     ```

   - Cette ligne attribuera le rôle "admin" à environ 1 utilisateur sur 10, les autres ayant le rôle "user".

Vérification des Modifications

- **Exécuter les Migrations et les Seeders** : Pour appliquer toutes les modifications et regénérer les utilisateurs, utilisez la commande suivante :

  ```bash
  php artisan migrate:fresh --seed
  ```

  - Cette commande va recréer la base de données et exécuter le seeder pour générer de nouveaux utilisateurs avec les rôles mis à jour.

- **Vérifier les Données** : Vérifiez dans votre base de données que les utilisateurs ont bien des rôles assignés ("user" ou "admin") et que la distribution respecte bien la règle de 1 administrateur pour 10 utilisateurs.

Avec cette mise à jour, nous sommes maintenant prêts à utiliser ces rôles pour gérer les permissions et personnaliser les fonctionnalités en fonction des types d'utilisateurs.

Création du UserController et des Routes

Maintenant que nous avons une base de données peuplée d'utilisateurs avec des rôles distincts, nous allons créer un contrôleur pour gérer ces utilisateurs et définir les routes nécessaires pour accéder aux différentes actions.

Création du UserController

1. **Générer le Contrôleur** : Utilisez Artisan pour générer un contrôleur de type ressource qui inclura toutes les actions nécessaires à la gestion des utilisateurs. Dans votre terminal, exécutez la commande suivante :

   ```bash
   php artisan make:controller UserController --resource
   ```

   Cette commande créera un fichier `UserController.php` dans le dossier `app/Http/Controllers`.

2. **Comprendre les Méthodes Générées** : Le contrôleur ressource généré contient les méthodes suivantes :

- `index()` : Afficher la liste des utilisateurs.
- `create()` : Afficher le formulaire de création d'un utilisateur.
- `store()` : Sauvegarder un nouvel utilisateur dans la base de données.
- `show($id)` : Afficher les détails d'un utilisateur spécifique.
- `edit($id)` : Afficher le formulaire de modification d'un utilisateur.
- `update($id)` : Mettre à jour les informations d'un utilisateur.
- `destroy($id)` : Supprimer un utilisateur.

Ces méthodes vont nous permettre de gérer toutes les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur les utilisateurs.

Définition des Routes

1. **Ajouter les Routes Ressources** : Pour permettre l'accès aux différentes actions du `UserController`, nous devons définir des routes. Ouvrez le fichier `routes/web.php` et ajoutez la ligne suivante :

```php
Route::resource('users', UserController::class)->middleware('auth');
```

   - Cette ligne crée automatiquement toutes les routes nécessaires pour les opérations CRUD sur les utilisateurs.
   - Le middleware `auth` s'assure que seules les personnes connectées peuvent accéder à ces routes.

2. **Vérifier les Routes** : Pour vérifier que les routes ont été correctement ajoutées, vous pouvez exécuter la commande suivante dans le terminal :

```bash
php artisan route:list
```

   - Cette commande affichera toutes les routes disponibles dans votre application, y compris celles associées au `UserController`.

Mise en Place des Permissions de Base

1. **Restreindre l'Accès** : Pour l'instant, nous allons simplement restreindre l'accès aux routes des utilisateurs aux personnes connectées grâce au middleware `auth`. Plus tard, nous pourrons affiner ces permissions en utilisant les rôles "admin" et "user".

Avec ces routes et ce contrôleur en place, nous avons les fondations nécessaires pour gérer la liste des utilisateurs. La prochaine étape consistera à créer les vues associées pour permettre l'interaction avec notre application via une interface utilisateur conviviale.

Création de la Vue de la liste des utilisateurs

Nous allons maintenant créer une interface pour gérer la liste des utilisateurs, en commençant par l'affichage de cette liste. Nous allons implémenter la méthode du contrôleur, créer la vue correspondante, et configurer la route pour que cette fonctionnalité soit opérationnelle.

Affichage de la Liste des Utilisateurs

1. Lancez le serveur de développement PHP et JavaScript.
    1. Lancez votre serveur Laragon et pour le front-end, lancez la commande de dev dans un deuxième terminal :

```
npm run dev
```

1. **Méthode Index dans le UserController**
    - Ouvrez le fichier `UserController.php` dans le dossier `app/Http/Controllers`.
    - Ajoutez la méthode `index()` pour récupérer tous les utilisateurs et les passer à la vue :

```PHP
public function index()
{
    users=User::paginate(10);returnview(’users.index’,[’users’⇒users = User::paginate(10);
    return view('users.index', ['users' ⇒ users=User::paginate(10);returnview(’users.index’,
[’users’⇒users]);
}
```

        - Cette méthode récupère tous les utilisateurs en utilisant une pagination de 10 utilisateurs par page et les transmet à la vue `index`.

2. **Créer la Vue Index**

- Pour afficher la liste des utilisateurs, créez une vue `index.blade.php` dans le dossier `resources/views/users`.
- Créez d'abord le dossier `users` dans `resources/views`.
- Ensuite, créez le fichier `index.blade.php` dans le dossier `resources/views/users`.

1. **Ajouter la Structure HTML**
    - Ouvrez le fichier `index.blade.php` et ajoutez la structure de base de la page pour afficher la liste des utilisateurs :

```XML
<x-app-layout>
    <div class="container mx-auto py-8">
        <h1 class="text-2xl font-bold mb-6">Liste des Utilisateurs</h1>
```

Nom	Email	Rôle	Actions
{{ $user->name }}	{{ $user->email }}	{{ $user->role }}	[Voir]({{ route('users.show', user→id) }}) [Modifier]({{ route('users.edit', user→id) }})

```
        <div class="mt-6">
            {{ $users->links() }}
        </div>
    </div>
</x-app-layout>
```

- Ce code crée une table qui affiche les utilisateurs, leur email, leur rôle, et propose des actions pour chaque utilisateur (voir, modifier).

2. **Ajouter la Route Index**
   - Le `UserController` étant un contrôleur de type ressource, la route pour accéder à la méthode `index` est déjà incluse. Inutile d'ajouter une nouvelle route, car elle est générée automatiquement par la ligne suivante dans `web.php` :

     ```php
     Route::resource('users', UserController::class)->middleware('auth');
     ```

3. **Vérifier la Fonctionnalité**
   - Vous pouvez maintenant accéder à la liste des utilisateurs via l'URL `/users`. Vérifiez que la pagination fonctionne bien et que les utilisateurs sont correctement affichés.

Avec cette fonctionnalité complète en place, les utilisateurs connectés peuvent visualiser la liste des utilisateurs avec une pagination de 10 utilisateurs par page.

Exercice : Création de la Vue Show

Maintenant que la vue index est en place, le premier exercice consistera à créer une vue pour afficher les détails d'un utilisateur spécifique.

1. **Méthode Show dans le UserController**
   - Implémentez la méthode `show($id)` dans le `UserController` pour récupérer un utilisateur spécifique.
   - Transmettez cet utilisateur à la vue `show`.

2. **Créer la Vue Show**
   - Créez la vue `show.blade.php` dans le dossier `resources/views/users`.
   - Ajoutez-y les informations de l'utilisateur, comme son nom, son email, et son rôle.

L'objectif est de rendre la fonctionnalité de visualisation d'un utilisateur complet à partir de la vue index. La prochaine étape consistera à implémenter la fonctionnalité de création d'un utilisateur, en suivant la même approche pour rendre la fonctionnalité entièrement opérationnelle.

Création d'un Nouvel Utilisateur

Dans cette section, nous allons implémenter la fonctionnalité permettant de créer un nouvel utilisateur. Nous allons développer la méthode `create()` et `store()` dans le contrôleur, créer la vue correspondante pour le formulaire de création, et nous assurer que la fonctionnalité est entièrement opérationnelle.

Méthode Create et Store dans le UserController

1. **Méthode Create**
   - Ouvrez le fichier `UserController.php` dans le dossier `app/Http/Controllers`.
   - Ajoutez la méthode `create()` pour retourner la vue permettant de créer un utilisateur :

```php
public function create()
{
    return view('users.create');
}
```

     - Cette méthode renvoie la vue `create` qui contient le formulaire de création d'un utilisateur.

2. **Méthode Store**
   - Ajoutez la méthode `store()` pour enregistrer un nouvel utilisateur dans la base de données :

```php
public function store(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|email|unique:users,email',
        'password' => 'required|string|min:8',
        'role' => 'required|in:user,admin',
    ]);

    User::create([
        'name' => $validatedData['name'],
        'email' => $validatedData['email'],
        'password' => bcrypt($validatedData['password']),
        'role' => $validatedData['role'],
    ]);

    return redirect()->route('users.index')->with('success', 'Utilisateur créé avec
succès.');
}
```

     - Cette méthode valide les données entrantes, crée un nouvel utilisateur, et redirige vers la liste des utilisateurs avec un message de succès.

Création de la Vue Create

1. **Créer le Fichier de Vue Create**
   - Créez manuellement une nouvelle vue `create.blade.php` dans le dossier `resources/views/users` ou avec cette commande :

```bash
touch resources/views/users/create.blade.php
```

2. **Ajouter le Formulaire de Création**

- Ouvrez `create.blade.php` et ajoutez le formulaire pour créer un utilisateur :

```xml
<x-app-layout>
    <div class="container mx-auto py-8">
        <h1 class="text-2xl font-bold mb-6">Créer un Nouvel Utilisateur</h1>

        <form action="{{ route('users.store') }}" method="POST">
            @csrf
            <div class="mb-4">
                <label class="block text-sm font-bold mb-2">Nom</label>
                <input type="text" name="name" class="w-full border border-gray-300 px-3 py-
2" value="{{ old('name') }}">
                @error('name')
                    <div class="text-red-500 text-sm mt-1">{{ message }}</div>
                @enderror
            </div>
            <div class="mb-4">
                <label class="block text-sm font-bold mb-2">Email</label>
                <input type="email" name="email" class="w-full border border-gray-300 px-3
py-2" value="{{ old('email') }}">
                @error('email')
                    <div class="text-red-500 text-sm mt-1">{{ message }}</div>
                @enderror
            </div>
            <div class="mb-4">
                <label class="block text-sm font-bold mb-2">Mot de passe</label>
                <input type="password" name="password" class="w-full border border-gray-300
px-3 py-2">
                @error('password')
                    <div class="text-red-500 text-sm mt-1">{{ message }}</div>
                @enderror
            </div>
            <div class="mb-4">
                <label class="block text-sm font-bold mb-2">Rôle</label>
                <select name="role" class="w-full border border-gray-300 px-3 py-2">
                    <option value="user">Utilisateur</option>
                    <option value="admin">Administrateur</option>
                </select>
                @error('role')
                    <div class="text-red-500 text-sm mt-1">{{ message }}</div>
                @enderror
            </div>
            <button type="submit" class="bg-blue-500 text-white px-4 py-2">Créer</button>
        </form>
    </div>
</x-app-layout>
```

- Ce formulaire permet de créer un utilisateur avec les champs nom, email, mot de passe, et rôle. Les messages d'erreur s'affichent sous chaque champ si les validations échouent.

Vérifier la Fonctionnalité

1. **Accéder à la Vue de Création**

- D'abord, connectez-vous à l'application avec ces identifiants :
  - email : test@example.com
  - mdp : password

- Pour retrouver les identifiants, cherchez dans les fichiers `DatabaseSeeder` et `UserFactory`.
- Pour accéder au formulaire de création, utilisez l'URL `/users/create`. Cette route est déjà incluse dans le `UserController` grâce à la déclaration de route ressource.

2. **Tester la Création**
   - Remplissez le formulaire et créez un utilisateur. Vous devriez être redirigé vers la liste des utilisateurs avec un message de succès affiché.

Avec cette fonctionnalité complète, nous avons désormais la possibilité de créer de nouveaux utilisateurs à travers une interface conviviale. La prochaine étape consistera à implémenter la fonctionnalité de modification d'un utilisateur, en laissant une partie sous forme d'exercice pour renforcer la compréhension.

Exercice : Modification d'un Utilisateur

Pour cet exercice, vous allez implémenter la fonctionnalité permettant de modifier un utilisateur existant. Cette fonctionnalité vous permettra de consolider vos compétences en implémentant des méthodes dans le contrôleur, en créant une vue, et en testant la mise à jour d'un enregistrement.

Objectifs de l'Exercice

- Ajouter la méthode `edit()` dans le `UserController` pour retourner le formulaire de modification d'un utilisateur spécifique.
- Ajouter la méthode `update()` dans le `UserController` pour enregistrer les modifications dans la base de données.
- Créer une vue `edit.blade.php` pour permettre la modification des informations de l'utilisateur.

Étapes à Suivre

1. **Méthode Edit dans le UserController**
   - Implémentez la méthode `edit($id)` pour récupérer l'utilisateur à modifier et le passer à la vue `edit`.
   - Utilisez la méthode `findOrFail()` pour vous assurer que l'utilisateur existe.

2. **Créer la Vue Edit**
   - Créez la vue `edit.blade.php` dans le dossier `resources/views/users`.
   - Ajoutez un formulaire similaire à celui de création (`create.blade.php`), mais pré-remplissez les champs avec les valeurs actuelles de l'utilisateur.
   - Le formulaire doit utiliser la méthode `POST` et la directive `@method('PUT')` en plus de la directive `@csrf`.
   - Utilisez la fonction `old()` combinée aux valeurs actuelles de l'utilisateur pour remplir les champs. Par exemple :
     - Pour le champ nom :

```xml
<input type="text" name="name" value="{{ old('name', $user->name) }}" class="w-full border border-gray-300 px-3 py-2">
```

    - Cela permet de conserver la valeur entrée en cas d'erreur de validation.

3. **Méthode Update dans le UserController**
    - Implémentez la méthode `update($id)` pour valider les données et mettre à jour l'utilisateur dans la base de données.
    - Validez les champs comme pour la méthode `store()`, et assurez-vous de prendre en compte l'utilisateur existant pour éviter les doublons d'email.
    - Redirigez vers la liste des utilisateurs avec un message de succès après la mise à jour.

Vérification de la Fonctionnalité

- **Accéder à la Vue de Modification** : Utilisez l'URL `/users/{id}/edit` pour accéder au formulaire de modification d'un utilisateur spécifique.
- **Tester la Modification** : Remplissez le formulaire avec les nouvelles informations et enregistrez les modifications. Vérifiez que l'utilisateur est mis à jour avec succès et que vous êtes redirigé vers la liste des utilisateurs avec un message de confirmation.

Points à Retenir

- Utilisez la fonction `old()` pour maintenir les valeurs saisies par l'utilisateur en cas d'erreur de validation.
- La méthode `update()` est similaire à `store()`, mais elle prend en compte l'utilisateur existant et met à jour les champs nécessaires.

Cet exercice vous permettra de mieux comprendre comment gérer la mise à jour des ressources dans Laravel. Prenez le temps de tester et de vous assurer que toutes les validations fonctionnent correctement.

Exercice : Suppression d'un Utilisateur

Dans cet exercice, vous allez implémenter la fonctionnalité permettant de supprimer un utilisateur existant.

Objectifs de l'Exercice

- Ajouter un bouton de suppression à la vue `index.blade.php` pour chaque utilisateur.
- Ajouter la méthode `destroy()` dans le `UserController` pour supprimer un utilisateur.
- Assurer la protection de la route de suppression avec un formulaire utilisant la méthode `POST` et un token CSRF.

Étapes à Suivre

1. **Ajouter un Bouton de Suppression à la Vue Index**
    - Ouvrez la vue `index.blade.php` et ajoutez un bouton de suppression pour chaque utilisateur.
    - Assurez-vous que le bouton est placé dans un formulaire avec la méthode `POST` et que la directive `@method('DELETE')` est utilisée.
    - Utilisez la directive `@csrf` pour inclure un token CSRF et protéger la requête contre les attaques CSRF.

2. **Méthode Destroy dans le UserController**

- Implémentez la méthode `destroy($id)` dans le `UserController` pour supprimer l'utilisateur.
- Utilisez la méthode `findOrFail()` pour récupérer l'utilisateur à supprimer et appelez la méthode `delete()` sur l'utilisateur.
- Redirigez vers la liste des utilisateurs avec un message de succès après la suppression.

3. **Vérifier la Fonctionnalité**
   - **Tester la Suppression** : Accédez à la liste des utilisateurs, cliquez sur le bouton de suppression pour un utilisateur et vérifiez que celui-ci est bien supprimé.
   - **Vérifier la Redirection** : Assurez-vous que vous êtes redirigé vers la liste des utilisateurs avec un message confirmant la suppression.

Points à Retenir

- Utilisez un formulaire avec une méthode `POST` et la directive `@method('DELETE')` pour effectuer la suppression d'un utilisateur.
- La directive `@csrf` est essentielle pour protéger votre application des attaques CSRF.

Cet exercice vous permettra de mieux comprendre comment gérer la suppression des ressources dans Laravel tout en assurant la sécurité des opérations.

Exercice : Ajout de la Recherche sur le Nom et l'Email

Dans cette section, nous allons implémenter une fonctionnalité de recherche pour permettre aux utilisateurs de rechercher des noms ou des emails dans la liste des utilisateurs sur la page index. Cette recherche sera envoyée à l'aide d'un formulaire avec un paramètre `GET`.

Objectifs

- Ajouter un formulaire de recherche sur la vue `index.blade.php`.
- Modifier la méthode `index()` dans le `UserController` pour inclure la logique de recherche.
- Gérer la recherche via un paramètre `GET`.

Étapes à Suivre

1. **Ajouter le Formulaire de Recherche à la Vue Index**
   - Ouvrez la vue `index.blade.php` et ajoutez un formulaire de recherche en haut de la page.
   - Ce formulaire doit contenir un champ `input` pour la recherche, ainsi qu'un bouton pour soumettre le formulaire.
   - Utilisez la méthode `GET` dans le formulaire pour que les critères de recherche apparaissent dans l'URL.
   - Exemple : placez un champ de texte avec un bouton "Rechercher" à côté pour soumettre la recherche.

2. **Méthode Index dans le UserController**
   - Modifiez la méthode `index()` dans le `UserController` pour inclure la logique de recherche.
   - Récupérez le paramètre `search` à partir de la requête et utilisez-le pour filtrer les utilisateurs.

- Utilisez la méthode `where()` sur le modèle `User` pour rechercher par nom ou par email en fonction de la valeur du paramètre `search`.
- Assurez-vous de conserver la pagination, même lors de l'application du filtre de recherche.

3. **Afficher les Critères de Recherche**
   - Lorsque les résultats de recherche sont affichés, le champ de recherche doit préserver la valeur de la recherche en cours pour indiquer ce que l'utilisateur a recherché.
   - Utilisez la fonction `old()` ou `request()->input()` pour pré-remplir le champ de recherche avec la valeur précédemment saisie.

Conseils pour la Recherche

- La recherche devrait être insensible à la casse pour rendre l'expérience utilisateur plus agréable.
- Vous pouvez utiliser la méthode `orWhere()` pour permettre de rechercher à la fois sur le nom et sur l'email.
- Pensez à ajouter un message indiquant à l'utilisateur s'il n'y a aucun résultat correspondant aux critères de recherche.

Vérification de la Fonctionnalité

- **Tester la Recherche** : Accédez à la liste des utilisateurs et effectuez une recherche en entrant un nom ou un email dans le champ de recherche. Cliquez sur le bouton "Rechercher" et vérifiez que la liste des utilisateurs se met à jour correctement.
- **Tester l'Absence de Résultats** : Entrez une valeur qui ne correspond à aucun utilisateur et vérifiez que le message d'absence de résultats s'affiche correctement.

Points à Retenir

- La recherche doit être intuitive pour l'utilisateur, avec une interface claire et un retour approprié lorsque aucun résultat n'est trouvé.
- Utiliser des paramètres `GET` permet de partager facilement les critères de recherche via l'URL.

Cette fonctionnalité de recherche vous permettra de rendre votre application plus pratique et plus efficace pour l'utilisateur final.

1.13 Todo List

Création d'une Todo List

Dans ce chapitre, nous allons apprendre à créer une Todo List, une application qui permet de gérer une liste de tâches.

Instructions de l'exercice

- Créer un formulaire pour ajouter une nouvelle tâche à la Todo List.
- Afficher la liste des tâches.
- Permettre à l'utilisateur de marquer une tâche comme étant terminée en utilisant une checkbox.
- Implémenter un mécanisme de suppression des tâches avec un bouton et un modal de confirmation.

- Enregistrer les données dans une base de données.
- L'application doit être multi-utilisateurs



1.14 Blog

Installation

Pour installer Laravel, nous allons utiliser la commande `composer create-project` à partir de la ligne de commande. Cette commande va créer une nouvelle application Laravel pour nous et installer toutes les dépendances nécessaires.

Voici la commande à utiliser pour créer notre nouveau blog Laravel :

```bash
composer create-project laravel/laravel blog
```

Cette commande va télécharger la dernière version de Laravel et l'installer dans un dossier nommé **blog**. Une fois l'installation terminée, nous pouvons entrer dans le répertoire **blog** avec la commande suivante :

```bash
cd blog
```

Maintenant que notre application Laravel est installée, nous pouvons commencer à la configurer pour notre blog.

Base de données

Configuration

La prochaine étape dans la construction de notre application de blog consiste à configurer notre base de données. Tout d'abord, nous devons créer une base de données pour notre blog à l'aide d'un outil tel que phpMyAdmin ou MySQL Workbench. Une fois que nous avons créé notre base de données, nous devons modifier le fichier .env situé à la racine de notre projet.

Dans ce fichier, nous devons définir les informations de connexion à notre base de données. Les informations requises comprennent le nom de l'hôte, le nom d'utilisateur, le mot de passe et le nom de la base de données. Voici un exemple de configuration de base de données pour une installation locale :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nom_de_votre_base_de_donnees
DB_USERNAME=nom_d_utilisateur
DB_PASSWORD=mot_de_passe
```

Modélisation de la base de données

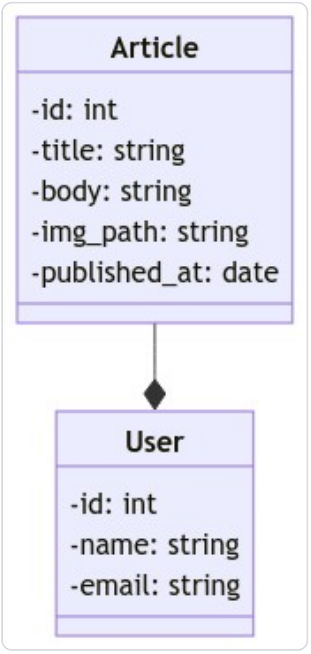
La première étape pour modéliser la base de données de notre application de blog est de déterminer les modèles dont nous avons besoin et les relations entre ces modèles. Dans notre cas, nous aurons besoin d'un modèle Article et d'un modèle User.

Le modèle Article aura les champs suivants :

- title: titre de l'article
- body: corps de l'article
- img_path: chemin de l'image associée à l'article
- published_at: date de publication de l'article

Nous utiliserons le modèle User fourni par Laravel pour gérer les utilisateurs de notre application.

Les modèles Article et User seront en relation en utilisant l'association "appartient à". Cela signifie que chaque article appartiendra à un seul utilisateur. Nous pouvons représenter cette relation en utilisant un diagramme de classe :



Notez que c'est une relation de composition, car un article ne peut pas exister sans un utilisateur. Cela signifie que si nous supprimons un utilisateur, tous les articles associés à cet utilisateur seront également supprimés.

Modèle et Migration

Dans Laravel, les modèles sont utilisés pour représenter les données dans la base de données. Il est très simple de créer un modèle pour notre relation de composition entre l'utilisateur et l'article en utilisant la commande suivante :

```bash
php artisan make:model Article -mf
```

Cela créera 3 fichiers. Un fichier `Article.php` dans le répertoire `app/Models`. Il créera également un fichier de migration dans le répertoire `database/migrations/` et un fichier de factory dans le répertoire `database/factories/`.

Dans ce fichier, vous pouvez définir la relation de composition entre l'utilisateur et l'article en ajoutant le code suivant dans la classe `Article` :

```php
class Article extends Model
{
    // ...

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Nous devons aussi définir la relation inverse dans le modèle `User` :

```php
class User extends Authenticatable
{
    // ...

    public function articles()
    {
        return $this->hasMany(Article::class);
    }
}
```

Laravel utilise les migrations pour gérer la structure de la base de données. Pour créer une migration pour notre modèle `Article`, nous aurions pu utiliser la commande `php artisan make:migration create_articles_table`. Cela créera un fichier de migration dans le répertoire `database/migrations/` qui peut être utilisé pour définir les champs pour la table `articles`.

Vous devez ajouter les champs à la méthode `up` de la migration :

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class() extends Migration {
    /**
     * Run the migrations.
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('body');
            $table->string('img_path')->nullable();
            $table->timestampTz('published_at')->nullable();
            $table->foreignId('user_id')->constrained()->onDelete('cascade');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
};
```

Enfin, pour exécuter la migration, utilisez la commande `php artisan migrate`. Cela créera la table `articles` dans votre base de données en fonction de la définition du fichier de migration. Cette commande vous proposera de créer la base de données si elle n'existe pas encore.

Création d'une fabrique pour les articles

Pour générer des articles fictifs dans notre base de données, nous allons utiliser les fabriques de Laravel.

Dans le fichier `ArticleFactory.php`, nous allons définir les champs que nous voulons que notre fabrique génère pour chaque article fictif. Voici un exemple :

```php
<?php

namespace Database\Factories;

use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Article>
 */
class ArticleFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'title' => fake()->bs(),
            'body' => fake()->realTextBetween(minNbChars=500,minNbChars = 500,
minNbChars=500,maxNbChars = 2000),
            'img_path' => function () {
                randomName=Str::uuid();randomName = Str::uuid();
                randomName=Str::uuid();imageUrl = "https://picsum.photos/1024/768.webp?random=
{randomName}";
                path = "images/{randomName}.webp";
                Storage::disk('public')->put(path, file_get_contents(imageUrl));returnimageUrl));

                return imageUrl));returnpath;
            },
            'published_at' => fake()->dateTimeBetween('-2 months', '+ 1 month'),
            'user_id' => User::get()->random()->id,
        ];
    }
}
```

Nous devons aussi créer le dossier qui va contenir nos images de test qui seront générées par faker. Créez le dossier `images` dans le dossier `storage/app/public/`.

Soyez patients, la commande peut prendre plus d'une minute pour enregistrer les images, dépendamment de votre configuration hardware.

Remplissage de la base de données avec le `DatabaseSeeder`

Avec Laravel, nous pouvons utiliser la classe `DatabaseSeeder` pour remplir notre base de données avec des données fictives. Pour ce faire, nous allons utiliser les **factories** que nous avons créées plus tôt. Voici comment cela peut être fait :

Dans le terminal, nous exécutons la commande :

```
php artisan make:seeder ArticlesTableSeeder
```

Cela créera un nouveau **fichier de semeur** dans le répertoire `database/seeds`.

Dans ce nouveau fichier, nous allons ajouter le code suivant :

```php
use App\Models\Article;
use Illuminate\Database\Seeder;

class ArticlesTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Article::factory()
            ->count(50)
            ->create();
    }
}
```

Nous allons maintenant enregistrer notre nouveau semeur dans le fichier `DatabaseSeeder`. Pour ce faire, nous ajoutons la ligne suivante dans la méthode `run` :

```php
$this->call(ArticlesTableSeeder::class);
```

N'oubliez pas décommenter les lignes qui créent des utilisateurs de test dans le fichier `DatabaseSeeder.php` :

```php
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run()
    {
        \App\Models\User::factory(10)->create();

        \App\Models\User::factory()->create([
            'name' => 'Test User',
            'email' => 'test@example.com',
        ]);

        $this->call([
            ArticlesTableSeeder::class,
        ]);
    }
}
```

Enfin, nous pouvons exécuter la commande `php artisan db:seed` pour **remplir** notre base de données avec des utilisateurs et des **articles fictifs**.

Et voilà ! Maintenant, notre base de données est **complète** avec des articles fictifs et nous pouvons commencer à construire notre application de blog.

Pour recréer la base de données et la remplir avec des données fictives, nous pouvons utiliser la commande :

```bash
php artisan migrate:fresh --seed
```

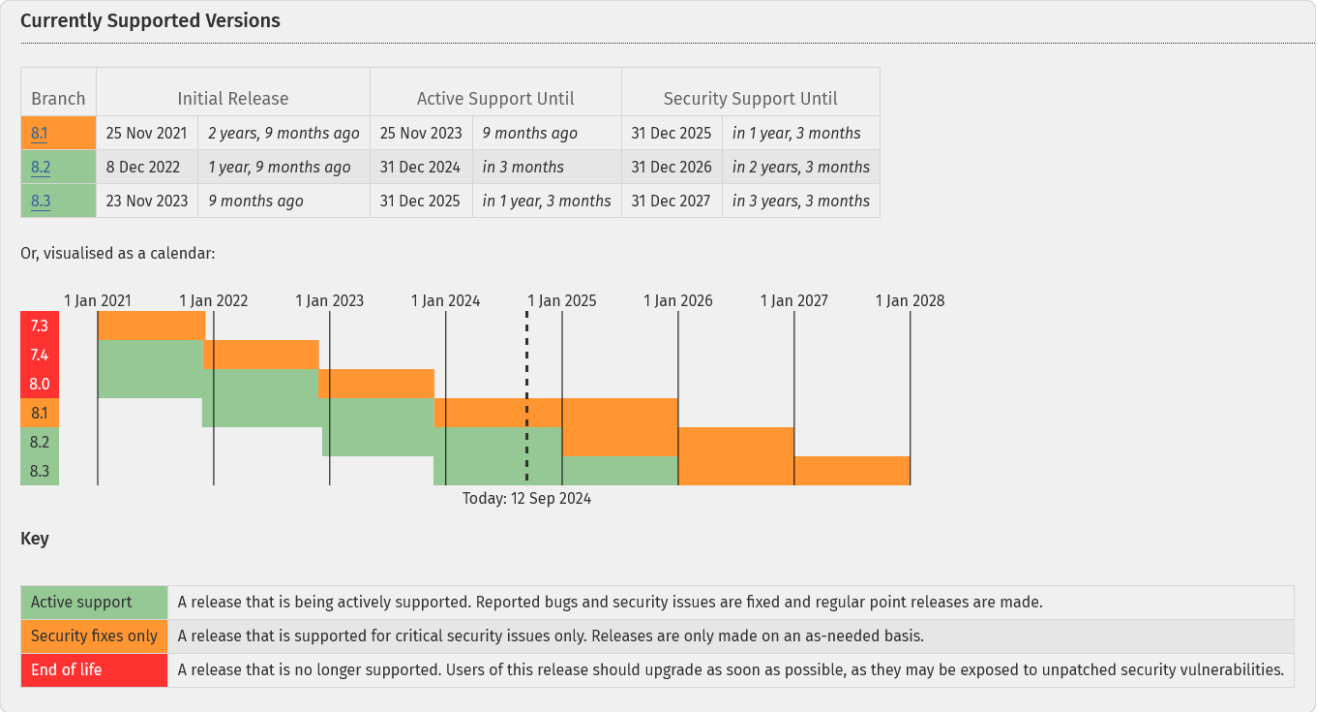
Authentification avec Laravel Breeze

Pour installer Laravel Breeze, nous pouvons utiliser la commande suivante :

```bash
composer require laravel/breeze --dev
```

Une fois l'installation terminée, nous pouvons générer les fichiers nécessaires à l'authentification en utilisant la commande :

```bash
php artisan breeze:install
```



Currently Supported Versions

Branch	Initial Release		Active Support Until		Security Support Until	
8.1	25 Nov 2021	*2 years, 9 months ago*	25 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*
8.2	8 Dec 2022	*1 year, 9 months ago*	31 Dec 2024	*in 3 months*	31 Dec 2026	*in 2 years, 3 months*
8.3	23 Nov 2023	*9 months ago*	31 Dec 2025	*in 1 year, 3 months*	31 Dec 2027	*in 3 years, 3 months*

Or, visualised as a calendar:

Key

Active support	A release that is being actively supported. Reported bugs and security issues are fixed and regular point releases are made.
Security fixes only	A release that is supported for critical security issues only. Releases are only made on an as-needed basis.
End of life	A release that is no longer supported. Users of this release should upgrade as soon as possible, as they may be exposed to unpatched security vulnerabilities.

Cette commande va créer les vues, les contrôleurs et les routes nécessaires à l'authentification. Nous pouvons maintenant commencer à utiliser Laravel Breeze pour gérer les utilisateurs de notre application.

Création de la page d'accueil

Configuration de la Homepage

Nous allons maintenant créer notre contrôleur `HomepageController` qui aura la responsabilité de gérer la page d'accueil de notre blog. Pour ce faire, nous allons utiliser la commande de génération de contrôleur de Laravel :

```bash
php artisan make:controller HomepageController
```

Cela va créer un nouveau fichier `app/Http/Controllers/HomepageController.php`. Ouvrons-le et écrivons la méthode `index` qui sera en charge de récupérer les articles à afficher sur la page d'accueil :

```php
namespace App\Http\Controllers;

use App\Models\Article;

class HomepageController extends Controller
{
    public function index()
    {
        articles=Article::all();returnview('homepage.index',['articles'⇒articles =
Article::all();

        return view('homepage.index', [
            'articles' ⇒ articles=Article::all();returnview('homepage.index',
['articles'⇒articles,
        ]);
    }
}
```

Maintenant, nous allons modifier la route de base de Laravel pour qu'elle appelle notre méthode `index` (pour rappel elle se trouve dans le fichier `routes/web.php` :

```php
Route::get('/', [HomepageController::class, 'index']);
```

N'oubliez pas d'importer le `HomepageController` !

Enfin, nous allons créer la vue associée, `resources/views/homepage/index.blade.php`, qui affichera la liste des articles :

```php
<h1>Liste des articles</h1>
<ul>
  @foreach(articlesasarticles as articlesasarticle)
  <li>{{ $article->title }}</li>
  @endforeach
</ul>
```

Résultat :



Avec ces modifications, notre application est maintenant capable d'afficher la liste des articles sur la page d'accueil.

Création du layout

La première étape consiste à modifier le fichier `guest.blade.php` dans le dossier `resources/views/layouts/`. Ce fichier servira de layout pour notre frontend et contiendra les éléments communs à toutes les pages, tels que le header et le footer.

```php
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="csrf-token" content="{{ csrf_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Fonts -->
    <link rel="stylesheet" href="https://fonts.bunny.net/css2?
family=Nunito:wght@400;600;700&display=swap">

    <!-- Scripts -->
    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>

<body class="font-sans text-gray-900 antialiased">
    <div class="min-h-screen flex flex-col pt-6 sm:pt-0 bg-gray-100 dark:bg-gray-900">
        <div class="container mx-auto flex flex-col space-y-10">
            <nav class="flex justify-between items-center py-2">
                <div>
                    <a href="/"
                        class="group font-bold text-3xl flex items-center space-x-4 hover:text-
emerald-600 transition ">
                        <x-application-logo
                            class="w-10 h-10 fill-current text-gray-500 group-hover:text-emerald-
500 transition" />
                        <span>Mon blog</span>
                    </a>
                </div>
                <div class="flex items-center space-x-4 justify-end">
                    <a class="font-bold hover:text-emerald-600 transition" href="/">Articles</a>
                </div>
            </nav>

            <main>
                {{ $slot }}
            </main>
        </div>
    </div>
</body>

</html>
```

Une fois ce layout créé, nous pourrons étendre ce layout dans nos autres vues. Nous utiliserons le tag `<x-guest-layout>` pour indiquer que notre vue doit étendre le layout `guest.blade.php` et ajoutons un peu de style avec TailwindCSS :

```php
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4">Liste des articles</h1>
    <ul class="grid sm:grid-cols-2 lg:grid-cols-3 2xl:grid-cols-4 gap-8">
        @foreach (articlesasarticles as articlesasarticle)
            <li>
                <a class="flex bg-white rounded-md  shadow-md p-5 w-full hover:shadow-lg
hover:scale-105 transition"
                    href="#">
                    {{ $article->title }}
                </a>
            </li>
        @endforeach
    </ul>
</x-guest-layout>
```

Pensez à lancer la commande `npm run dev` lorsque vous travaillez sur le frontend de votre site pour voir les changements en temps réel.

Pagination

Nous allons maintenant ajouter la pagination à notre page d'accueil. Pour ce faire, nous allons utiliser la méthode `paginate` de Laravel. Cette méthode va nous permettre de récupérer les articles par page et de générer les liens de pagination. Dans le controller, nous allons modifier la méthode `index` pour récupérer les articles par page :

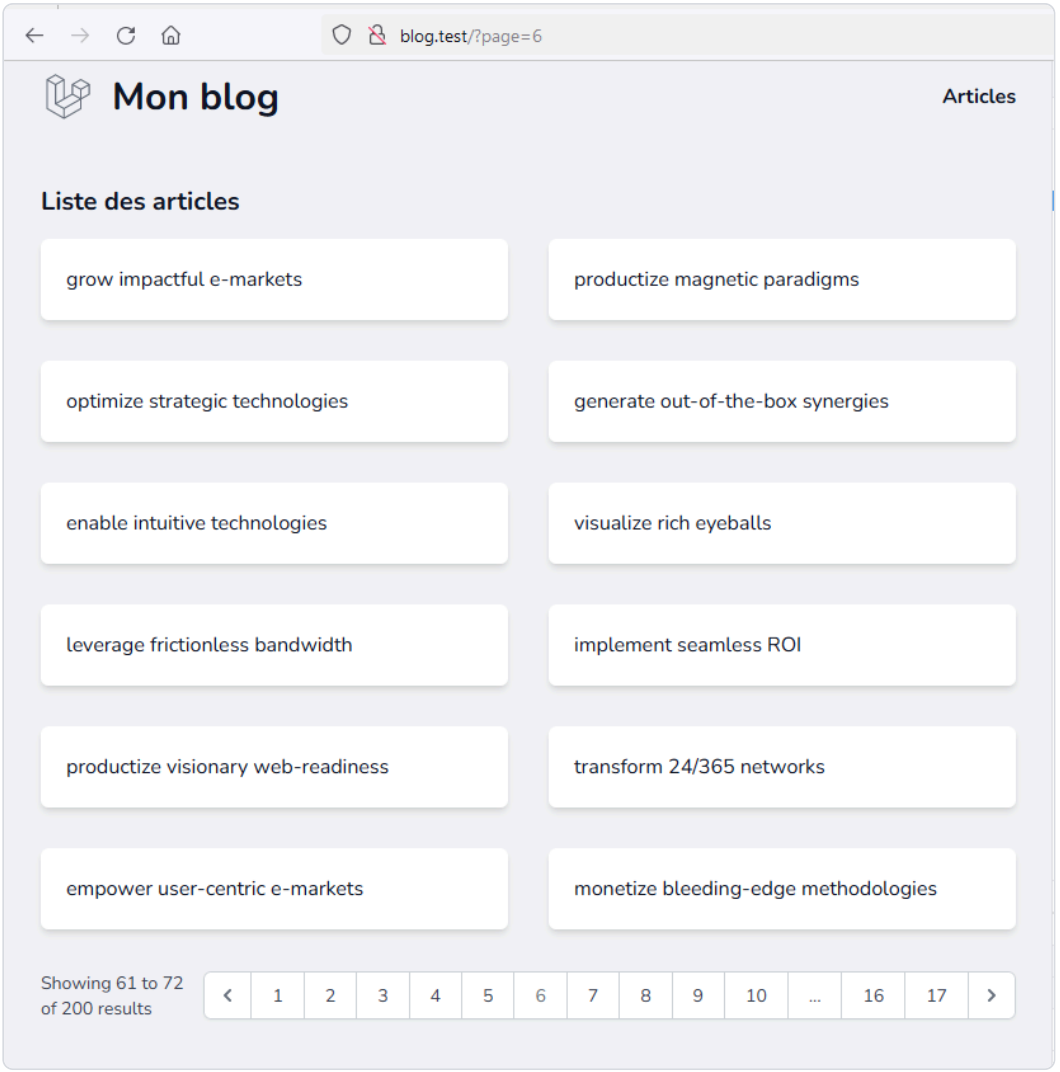
```php
$articles = Article::paginate(12);
```

Nous allons ensuite modifier notre vue pour afficher les liens de pagination :

```php
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4">Liste des articles</h1>
    <ul class="grid sm:grid-cols-2 lg:grid-cols-3 2xl:grid-cols-4 gap-8">
        @foreach (articlesasarticles as articlesasarticle)
            <li>
                <a class="flex bg-white rounded-md  shadow-md p-5 w-full hover:shadow-lg
hover:scale-105 transition"
                    href="#">
                    {{ article->title }}
                </a>
            </li>
        @endforeach
    </ul>

    <div class="mt-8">
        {{ articles->links() }}
    </div>
</x-guest-layout>
```

Résultat :



Les articles

Contrôleur et routes pour les articles

Pour afficher la liste des articles, nous allons créer un `ArticleController` qui aura une méthode `index` pour afficher la liste des articles et une méthode `show` pour afficher le détail d'un article.

```bash
php artisan make:controller ArticleController
```

Nous allons ensuite ajouter les méthodes dans le controller :

```php
namespace App\Http\Controllers;

use App\Models\Article;
use Illuminate\Http\Request;

class ArticleController extends Controller
{
    public function index()
    {
        articles=Article::paginate(12);returnview('front.articles.index',['articles'⇒articles =
Article::paginate(12);

        return view('front.articles.index', [
            'articles' ⇒ articles=Article::paginate(12);returnview('front.articles.index',
['articles'⇒articles,
        ]);
    }

    public function show($id)
    {
        $article = Article::findOrFail($id);

        return view('front.articles.show', [
            'article' ⇒ $article,
        ]);
    }
}
```

Ensuite, nous allons ajouter les routes pour ces méthodes dans le fichier `routes/web.php` :

```php
Route::get('/articles', [ArticleController::class, 'index'])→name('front.articles.index');
Route::get('/articles/{id}', [ArticleController::class, 'show'])→name('front.articles.show');
```

Nous devons modifier notre layout pour modifier le lien vers la liste des articles :

```php
<a class="font-bold hover:text-emerald-600 transition"
                        href="{{ route('front.articles.index') }}">Articles</a>
```

Et nous devons modifier notre vue `homepage.index` pour modifier le lien vers le détail des articles :

```php
<a class="flex bg-white rounded-md shadow-md p-5 w-full hover:shadow-lg hover:scale-105
transition"
    href="{{ route('front.articles.show', article) }}">
    {{ article→title }}
</a>
```

Avant de vérifier si tout fonctionne, nous devons encore créer les vues.

Composant ArticleCard

Pour éviter de répéter le code dans plusieurs vues, nous pouvons créer un composant pour afficher un article. Nous allons donc créer un composant `ArticleCard` en utilisant la commande `make:component` :

```bash
php artisan make:component ArticleCard --view
```

Le composant sera créé dans le répertoire `resources/views/components`. Nous pouvons maintenant éditer ce fichier pour définir la vue de notre composant.

```php
<a class="flex flex-col h-full space-y-4 bg-white rounded-md shadow-md p-5 w-full hover:shadow-lg
hover:scale-105 transition"
    href="{{ route('front.articles.show', article) }}">
    <div class="uppercase font-bold text-gray-800">
        {{ article->title }}
    </div>
    <div class="flex-grow text-gray-700 text-sm text-justify">
        {{ Str::limit(article->body, 120) }}
    </div>
    <div class="text-xs text-gray-500">
        {{ article->published_at }}
    </div>
</a>
```

Ensuite, nous utilisons ce composant dans notre vue qui affiche la liste des articles :

```php
@foreach (articlesasarticles as articlesasarticle)
    <li>
        <x-article-card :article="$article" />
    </li>
@endforeach
```

Nous pouvons voir que nous passons les informations de l'article avec un paramètre nommé `article`. Nous pouvons alors les utiliser dans notre composant grâce à la variable `$article`. C'est ce qu'on appelle un "prop".

Liste des articles

Nous allons maintenant créer une page qui affiche la liste des articles. Générons le fichier de la vue avec cette commande :

```bash
php artisan make:view front/articles/index
```

Nous allons modifier le contenu de ce fichier avec ce code :

```php
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4">Liste des articles</h1>
    <ul class="grid sm:grid-cols-2 lg:grid-cols-3 2xl:grid-cols-4 gap-8">
        @foreach (articlesasarticles as articlesasarticle)
            <li>
                <x-article-card :article="$article" />
            </li>
        @endforeach
    </ul>

    <div class="mt-8">
        {{ $articles->links() }}
    </div>
</x-guest-layout>
```

Détail d'un article

Nous allons maintenant créer une page qui affiche le détail d'un article. Pour ce faire, nous allons créer une nouvelle vue `front.articles.show` :

```php
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4">{{ article->title }}</h1>
    <div class="mb-4 text-xs text-gray-500">
        {{ article->published_at }}
    </div>
    <div>
        {!! \nl2br($article->body) !!}
    </div>
</x-guest-layout>
```

Exercices

- Ajouter un lien de retour vers la liste des articles dans la page de détail d'un article.
- Afficher les images des articles dans la liste des articles et dans la page de détail d'un article.
- Ajouter un footer avec des liens vers les réseaux sociaux.
- Ajouter une page "À propos" qui affiche un texte de présentation du site. Ajouter un lien vers cette page dans le nav.
- Customiser le thème de l'application avec TailwindCSS. documentation

Attention, pour afficher les images, il faut exécuter la commande `php artisan storage:link` pour créer un lien symbolique vers le dossier `storage/app/public` dans le dossier `public/storage`.

Ensuite, pour générer le lien vers une image, vous pouvez utiliser la classe `Storage` :

```php
<img src="{{ Storage::url($article->img_path) }}" alt="illustration de l'article">
```

Administration

Nous allons créer une interface d'administration pour gérer les articles. Nous allons créer :

- une page qui affiche la liste des articles
- une page pour créer un article
- une page pour modifier un article
- un bouton pour supprimer un article

Pour accéder à l'administration, connectez-vous en visitant la page `/login`.

- http://blog.test/login
- **Email :** `test@example.com`
- **MdP :** `password`

Contrôleur et routes pour l'administration

Nous allons créer un dossier Admin dans le dossier des contrôleurs. Nous y ajouterons un contrôleur `ArticleController` qui aura les méthodes `index`, `create`, `store`, `edit`, `update` et `destroy`. Pour ce faire, nous allons utiliser un contrôleur ressource. Ce sont des contrôleurs qui ont des méthodes prédéfinies pour les actions de base d'un CRUD. Nous allons donc utiliser la commande `make:controller` avec l'option `--resource` et l'option `--model` pour spécifier le modèle associé au contrôleur :

```bash
php artisan make:controller Admin/ArticleController --resource --model=Article
```

Nous allons ensuite ajouter une route de type `resource` pour ce contrôleur dans le fichier `routes/web.php`. Nous allons également ajouter un middleware `auth` pour protéger l'accès à ces routes ainsi qu'un préfixe `admin` pour les routes de l'administration.

```php
// autres imports ...
use App\Http\Controllers\Admin\ArticleController as AdminArticleController;

// autres routes ...
Route::middleware(['auth'])->prefix('admin')->group(function () {
    Route::resource('articles', AdminArticleController::class);
});
```

Nous avons déjà un contrôleur `ArticleController` dans le dossier `app/Http/Controllers`. Avec les namespaces PHP, nous pouvons utiliser le même nom de contrôleur pour deux contrôleurs différents. Pour les distinguer dans un même fichier, nous pouvons utiliser un alias. Nous avons donc utilisé l'alias `AdminArticleController` pour le contrôleur de l'administration.

Vue pour la liste des articles

Nous allons maintenant créer une vue pour afficher la liste des articles. Nous allons créer un fichier `resources/views/admin/articles/index.blade.php` :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="py-12">
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg">
                <div class="p-6 sm:px-20 bg-white border-b border-gray-200">
                    <div class="flex justify-between mt-8">
                        <div class=" text-2xl">
                            Liste des articles
                        </div>

                        <div class="flex  items-center justify-center space-x-8">
                            <a href="{{ route('articles.create') }}"
                                class="text-gray-500 font-bold py-2 px-4 rounded hover:bg-gray-200
transition">Ajouter un
                                article</a>
                        </div>
                    </div>

                    <div class="mt-6 text-gray-500">
                        <table class="table-auto w-full">
                            <thead>
                                <tr class="uppercase text-left">
                                    <th class="px-4 py-2 border">Titre</th>
                                    <th class="px-4 py-2 border">Auteur</th>
                                    <th class="px-4 py-2 border">Date de publication</th>
                                    <th class="px-4 py-2 border">Dernière modification</th>
                                    <th class="px-4 py-2 border">Actions</th>
                                </tr>
                            </thead>
                            <tbody>
                                @foreach (articlesasarticles as articlesasarticle)
                                    <tr class="hover:bg-gray-50 odd:bg-gray-100 hover:odd:bg-gray-
200 transition">
                                        <td class="border px-4 py-2">
                                            {{ article->title }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->user->name }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->published_at?->diffForHumans() }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->updated_at->diffForHumans() }}</td>
                                        <td class="border px-4 py-2 space-x-4">
                                            <a href="{{ route('articles.edit', article->id) }}"
                                                class="text-blue-400">Edit</a>
                                            <form action="{{ route('articles.destroy', article
->id) }}" method="POST"
                                                class="inline">
                                                @csrf
                                                @method('DELETE')
                                                <button type="submit" class="text-red-
400">Delete</button>
                                            </form>
```

```
                                    </td>
                                </tr>
                            @endforeach
                        </tbody>
                    </table>

                    <div class="mt-4">
                        {{ $articles->links() }}
                    </div>
                </div>
            </div>
        </div>
    </div>

</x-app-layout>
```

Cette vue utilise le layout `app.blade.php`. La page contient un tableau qui affiche la liste des articles. Nous avons aussi ajouté un lien pour créer un nouvel article. Pour chaque article, nous affichons le titre, l'auteur, la date de publication et la date de dernière modification. Nous avons aussi ajouté un lien pour modifier l'article et un bouton pour supprimer l'article.

Pour calculer le temps écoulé depuis la date de publication, nous devons ajouter cette méthode dans le modèle des Articles, `Article.php` :

PHP

```php
protected function casts(): array
{
    return [
        'published_at' => 'datetime',
    ];
}
```

Contrôleur pour la liste des articles

Nous allons maintenant créer la méthode `index` dans le contrôleur `Admin/ArticleController`. Cette méthode va récupérer la liste des articles et la passer à la vue `admin/articles/index.blade.php`.

PHP

```php
    public function index()
    {
        articles=Article::orderByDesc('updatedat')
->paginate(10);returnview('admin.articles.index',['articles'⇒articles =
Article::orderByDesc('updated_at')
            ->paginate(10)
        ;

        return view(
            'admin.articles.index',
            [
                'articles' => articles=Article::orderByDesc('updatedat')
->paginate(10);returnview('admin.articles.index',['articles'⇒articles,
            ]
        );
    }
```

Nous utilisons la méthode `paginate` pour récupérer les articles par page. Nous avons choisi de récupérer 10 articles par page. Nous utilisons la méthode `orderByDesc` pour trier les articles par date de modification décroissante.

Lien vers la liste des articles dans le menu

Nous allons maintenant ajouter un lien vers la liste des articles dans le menu de l'administration. Nous allons modifier le fichier `resources/views/layouts/navigation.blade.php` :

```php
<!-- Navigation Links -->
<div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
    <x-nav-link :href="route('dashboard')" :active="request()->routeIs('dashboard')">
        {{ __('Dashboard') }}
    </x-nav-link>

    <!--  Lien à ajouter -->
    <x-nav-link :href="route('articles.index')" :active="request()->routeIs('articles.*')">
        {{ __('Articles') }}
    </x-nav-link>
</div>
```

Nous avons ajouté un lien vers la liste des articles. La méthode `routeIs` permet de vérifier si la route actuelle correspond à un pattern. Dans notre cas, nous voulons vérifier si la route actuelle correspond à la route `articles.index` ou à la route `articles.create`, `articles.edit` ou `articles.destroy`. Nous utilisons le caractère `*` pour indiquer que la route peut être suivi d'un identifiant quelconque. Si la route actuelle correspond à l'un de ces patterns, le lien recevra la classe `active`.

Nous devons aussi ajouter un lien pour le menu responsive, toujours dans le fichier `resources/views/layouts/navigation.blade.php` :

```php
    <!-- Responsive Navigation Menu -->
    <div :class="{ 'block': open, 'hidden': !open }" class="hidden sm:hidden">
        <div class="pt-2 pb-3 space-y-1">
            <x-responsive-nav-link :href="route('dashboard')" :active="request()
->routeIs('dashboard')">
                {{ __('Dashboard') }}
            </x-responsive-nav-link>

            <!--  Lien à ajouter -->
            <x-responsive-nav-link :href="route('articles.index')" :active="request()
->routeIs('articles.*')">
                {{ __('Articles') }}
            </x-responsive-nav-link>
        </div>
    ...
```

Création d'un article

Nous allons maintenant créer une page pour créer un article. Nous allons créer un fichier `resources/views/admin/articles/create.blade.php` :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Créer un article
                </div>
            </div>

            <form method="POST" action="{{ route('articles.store') }}" class="flex flex-col space-
y-4 text-gray-500">

                @csrf

                <div>
                    <x-input-label for="title" :value="__('Titre')" />
                    <x-text-input id="title" class="block mt-1 w-full" type="text" name="title"
                        :value="old('title')" autofocus />
                    <x-input-error :messages="errors->get('title')" class="mt-2" />
                </div>

                <div>
                    <x-input-label for="published_at" :value="__('Date de publication')" />
                    <x-text-input id="published_at" class="block mt-1 w-full" type="date"
name="published_at"
                        :value="old('published_at')" />
                    <x-input-error :messages="errors->get('published_at')" class="mt-2" />
                </div>

                <div>
                    <x-input-label for="body" :value="__('Texte de l\'article')" />
                    <textarea id="body"
                        class="block mt-1 w-full border-gray-300 dark:border-gray-700 dark:bg-
gray-900 dark:text-gray-300 focus:border-indigo-500 dark:focus:border-indigo-600 focus:ring-
indigo-500 dark:focus:ring-indigo-600 rounded-md shadow-sm"
                        name="body" rows="10">{{ old('body') }}</textarea>
                    <x-input-error :messages="$errors->get('body')" class="mt-2" />
                </div>

                <div class="flex justify-end">
                    <x-primary-button type="submit">
                        {{ __('Créer') }}
                    </x-primary-button>
                </div>
            </form>
        </div>
    </div>
</x-app-layout>
```

Nous avons créé un formulaire pour créer un article. Nous utilisons les composants `x-input-label`, `x-text-input` et `x-input-error` pour afficher les labels, les champs de saisie et les messages d'erreur. Nous utilisons le composant `x-primary-button` pour afficher le bouton de création.

Dans le contrôleur `app/Http/Controllers/Admin/ArticleController.php`, nous allons modifier la méthode `create` pour afficher la vue `resources/views/admin/articles/create.blade.php` :

```php
    public function create()
    {
        return view('admin.articles.create');
    }
```

Validation des données

Lorsqu'on utilise un formulaire, il est important de valider les données reçues. Nous ne pouvons pas faire confiance aux données envoyées par l'utilisateur. Pour résoudre ce problème, Laravel permet d'utiliser des règles prédéfinies et de générer des messages d'erreur en cas de non-respect de ces règles.

Il y a plusieurs façons de valider les données. Nous allons utiliser les Form Requests. Pour créer un Form Request, nous allons utiliser la commande suivante :

```bash
php artisan make:request ArticleCreateRequest
```

Cette commande va créer un fichier `app/Http/Requests/ArticleCreateRequest.php`. Nous allons modifier ce fichier pour ajouter les règles de validation. On ne doit pas non plus oublier de renvoyer `true` dans la méthode `authorize()` :

```php
class ArticleCreateRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, mixed>
     */
    public function rules()
    {
        return [
            'title' => 'required|unique:articles|max:255',
            'published_at' => 'nullable|date',
            'body' => 'required|max:10000',
        ];
    }
}
```

Pour le titre, nous avons ajouté les règles suivantes :

- `required` : le titre est obligatoire
- `unique:articles` : le titre doit être unique dans la table `articles`
- `max:255` : le titre doit faire au maximum 255 caractères

Pour la date de publication, nous avons ajouté les règles suivantes :

- `nullable` : la date de publication est optionnelle
- `date` : la date de publication doit être une date valide

Pour le texte de l'article, nous avons ajouté les règles suivantes :

- `required` : le texte de l'article est obligatoire
- `max:10000` : le texte de l'article doit faire au maximum 10000 caractères

Vous pouvez trouver la liste des règles disponibles sur la documentation de Laravel.

Enregistrement des données

Dans notre controller, nous allons modifier la méthode `store()` pour utiliser le Form Request et sauvegarder les données reçues.

```php
    public function store(ArticleCreateRequest $request)
    {
        $article = Article::make();
        $article->title = $request->validated()['title'];
        $article->body = $request->validated()['body'];
        $article->published_at = $request->validated()['published_at'];
        $article->user_id = Auth::id();
        $article->save();

        return redirect()->route('articles.index');
    }
```

Tout d'abord, nous remplaçons le type `Request` du paramètre `$request` par celui de la Form Request `ArticleCreateRequest`. Laravel utilisera automatiquement cette classe pour valider les données du formulaire. Ensuite, nous créons un nouvel article en utilisant la méthode `make()` du modèle `Article`. Nous utilisons la méthode `validated()` pour récupérer les données validées. Pour chaque propriété de l'article, nous récupérons la valeur correspondante dans le tableau de données validées. Nous récupérons également l'identifiant de l'utilisateur connecté et nous l'ajoutons à l'article. Nous sauvegardons l'article en base de données grâce à la méthode `save()` du modèle. Enfin, nous redirigeons l'utilisateur vers la liste des articles.

Modification d'un article

Pour modifier un article, nous allons utiliser la méthode `edit()` du contrôleur `ArticleController`. Cette méthode va afficher le formulaire de modification d'un article.

```php
public function edit(Article $article)
{
    return view('admin.articles.edit', [
        'article' => $article,
    ]);
}
```

Nous allons utiliser le formulaire de création d'un article comme base. Nous allons donc créer une vue `resources/views/admin/articles/edit.blade.php` :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Modifier un article
                </div>
            </div>

            <div class="text-gray-500">
                <form method="POST" action="{{ route('articles.update', article) }}" class="flex
flex-col space-y-4">

                    @csrf
                    @method('PUT')

                    <div>
                        <x-input-label for="title" :value="__('Titre')" />
                        <x-text-input id="title" class="block mt-1 w-full" type="text"
name="title"
                            :value="old('title', article)" autofocus />
                        <x-input-error :messages="errors->get('title')" class="mt-2" />
                    </div>

                    <div>
                        <x-input-label for="published_at" :value="__('Date de publication')" />
                        <x-text-input id="published_at" class="block mt-1 w-full" type="date"
name="published_at"
                            :value="old('published_at', article->published_at?->toDateString())"
/>
                        <x-input-error :messages="errors->get('published_at')" class="mt-2" />
                    </div>

                    <div>
                        <x-input-label for="body" :value="__('Texte de l\'article')" />
                        <textarea id="body"
                            class="block mt-1 w-full border-gray-300 dark:border-gray-700 dark:bg-
gray-900 dark:text-gray-300 focus:border-indigo-500 dark:focus:border-indigo-600 focus:ring-
indigo-500 dark:focus:ring-indigo-600 rounded-md shadow-sm"
                            name="body" rows="10">{{ old('body', $article) }}</textarea>
                        <x-input-error :messages="errors->get('body')" class="mt-2" />
                    </div>

                    <div class="flex justify-end">
                        <x-primary-button type="submit">
                            {{ __('Modifier') }}
                        </x-primary-button>
                    </div>
                </form>
            </div>
```

```
        </div>
    </div>
</x-app-layout>
```

Enregistrement des modifications

D'abord, nous allons créer une nouvelle Form Request `ArticleUpdateRequest` pour valider les données reçues avec ces règles :

```php
public function rules()
{
    return [
        'title' => 'required|string|max:255|unique:articles,title,'.$this->route('article')->id,
        'published_at' => 'nullable|date',
        'body' => 'required|string',
    ];
}
```

Nous avons ajouté la règle `unique:articles,title,{$id}` pour vérifier que le titre est unique dans la table `articles` sauf pour l'article en cours de modification. Pour récupérer l'identifiant de l'article en cours de modification, nous utilisons la méthode `route()` de la Form Request. Cette méthode permet de récupérer les paramètres de la route.

N'oubliez pas d'autoriser les requêtes en retournant `true` dans la méthode `authorize()` :

```php
public function authorize(): bool
{
    return true;
}
```

Dans le controller, nous allons modifier la méthode `update()` pour utiliser le Form Request et sauvegarder les données reçues.

```php
public function update(ArticleUpdateRequest request,Articlerequest, Article
request,Articlearticle)
{
    article->title=article->title = article->title=request->validated()['title'];
    article->body=article->body = article->body=request->validated()['body'];
    article->publishedat=article->published_at = article->publishedat=request->validated()
['published_at'];
    $article->save();

    return redirect()->back();
}
```

Nous récupérons l'article à modifier en utilisant le modèle `Article` et le paramètre `Article $article` de la méthode `update()`. Nous utilisons la méthode `validated()` pour récupérer les données validées. Pour chaque propriété de l'article, nous récupérons la valeur correspondante dans le tableau de données validées. Nous sauvegardons l'article en base de données avec la méthode `save()` du modèle. Enfin, nous redirigeons l'utilisateur vers la page précédente.

Suppression d'un article

Pour supprimer un article, nous allons utiliser la méthode `destroy()` du contrôleur `ArticleController`. Cette méthode va supprimer l'article en base de données. Nous allons utiliser un formulaire pour envoyer la requête de suppression. Il est déjà présent dans la vue `resources/views/admin/articles/index.blade.php` :

```php
 <form action="{{ route('articles.destroy', $article->id) }}" method="POST"
    class="inline">
    @csrf
    @method('DELETE')
    <button type="submit" class="text-red-400">Delete</button>
</form>
```

Nous utilisons la méthode `route()` pour générer l'URL de la route `articles.destroy` avec l'identifiant de l'article en paramètre. Nous utilisons la méthode `method()` pour générer le champ `@method('DELETE')` qui permet de simuler une requête de type `DELETE` avec une requête de type `POST`. Nous utilisons la méthode `csrf()` pour générer le champ `@csrf` qui permet de générer un jeton de sécurité pour la requête.

Dans le controller, nous allons modifier la méthode `destroy()` pour supprimer l'article en base de données.

```php
public function destroy(Article $article)
{
    $article->delete();

    return redirect()->back();
}
```

Nous récupérons l'article à supprimer en utilisant le paramètre `Article $article` de la méthode `destroy()`. Nous utilisons la méthode `delete()` du modèle pour supprimer l'article en base de données. Enfin, nous redirigeons l'utilisateur vers la page précédente.

Modal de confirmation

Nous allons utiliser une modal de confirmation pour demander à l'utilisateur de confirmer la suppression de l'article. Nous allons utiliser le composant `x-modal` fournit avec Laravel Breeze. Nous allons modifier la vue `resources/views/admin/articles/index.blade.php` pour utiliser ce composant :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="py-12">
        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
            <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg">
                <div class="p-6 sm:px-20 bg-white border-b border-gray-200">
                    <div class="flex justify-between mt-8">
                        <div class=" text-2xl">
                            Liste des articles
                        </div>

                        <div class="flex  items-center justify-center space-x-8">
                            <a href="{{ route('articles.create') }}"
                                class="text-gray-500 font-bold py-2 px-4 rounded hover:bg-gray-200
transition">Ajouter
                                un
                                article</a>
                        </div>
                    </div>

                    <div class="mt-6 text-gray-500">
                        <table class="table-auto w-full">
                            <thead>
                                <tr class="uppercase text-left">
                                    <th class="px-4 py-2 border">Titre</th>
                                    <th class="px-4 py-2 border">Auteur</th>
                                    <th class="px-4 py-2 border">Date de publication</th>
                                    <th class="px-4 py-2 border">Dernière modification</th>
                                    <th class="px-4 py-2 border">Actions</th>
                                </tr>
                            </thead>
                            <tbody>
                                @foreach (articlesasarticles as articlesasarticle)
                                    <tr class="hover:bg-gray-50 odd:bg-gray-100 hover:odd:bg-gray-
200 transition">
                                        <td class="border px-4 py-2">
                                            {{ article->title }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->user->name }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->published_at?->diffForHumans() ?? 'Pas de
date' }}</td>
                                        <td class="border px-4 py-2">
                                            {{ article->updated_at->diffForHumans() }}</td>
                                        <td class="border px-4 py-2 space-x-4">
                                            <a href="{{ route('articles.edit', article->id) }}"
                                                class="text-blue-400">Edit</a>

                                            <button x-data="{ id: {{ $article->id }} }"
                                                x-on:click.prevent="window.selected = id;
dispatch('open-modal', 'confirm-article-deletion');"
                                                type="submit" class="text-red-400">Delete</button>
                                        </td>
```

```
                                    </tr>
                                @endforeach
                            </tbody>
                        </table>

                        <div class="mt-4">
                            {{ articles->links() }}
                        </div>
                    </div>
                </div>
            </div>
        </div>
        <x-modal name="confirm-article-deletion" focusable>
            <form method="post" onsubmit="event.target.action= '/admin/articles/' +
window.selected" class="p-6">
                @csrf
                @method('DELETE')

                <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
                    Êtes-vous sûr de vouloir supprimer cet article ?
                </h2>

                <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
                    Cette action est irréversible. Toutes les données seront supprimées.
                </p>

                <div class="mt-6 flex justify-end">
                    <x-secondary-button x-on:click="dispatch('close')">
                        Annuler
                    </x-secondary-button>

                    <x-danger-button class="ml-3" type="submit">
                        Supprimer
                    </x-danger-button>
                </div>
            </form>
        </x-modal>
    </div>
</x-app-layout>
```

Le component modal utilise Alpine.js pour gérer l'affichage du modal. Lorsque l'utilisateur clique sur le bouton `Delete`, nous définissons la variable `window.selected` avec l'identifiant de l'article à supprimer. Nous utilisons ensuite cette variable pour générer l'URL de la route `articles.destroy` dans le formulaire du modal.

Upload d'images

Pour uploader des images, nous allons modifier :

- le formulaire de création et d'édition d'article pour ajouter un champ de type `file`
- le controller pour gérer l'upload de l'image (méthode `store()` et `update()`)
- Supprimer l'image lors de la suppression de l'article (méthode `destroy()`)

Formulaire

Nous allons modifier la vue `resources/views/admin/articles/create.blade.php` :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Créer un article
                </div>
            </div>

            <form method="POST" action="{{ route('articles.store') }}" class="flex flex-col space-
y-4 text-gray-500"
                enctype="multipart/form-data">

                @csrf

                <div>
                    <x-input-label for="title" :value="__('Titre')" />
                    <x-text-input id="title" class="block mt-1 w-full" type="text" name="title"
                        :value="old('title')" autofocus />
                    <x-input-error :messages="errors->get('title')" class="mt-2" />
                </div>

                <div>
                    <x-input-label for="published_at" :value="__('Date de publication')" />
                    <x-text-input id="published_at" class="block mt-1 w-full" type="date"
name="published_at"
                        :value="old('published_at')" />
                    <x-input-error :messages="errors->get('published_at')" class="mt-2" />
                </div>

                <div>
                    <x-input-label for="img" :value="__('Image')" />
                    <x-text-input id="img" class="block mt-1 w-full" type="file" name="img" />
                    <x-input-error :messages="errors->get('img')" class="mt-2" />
                </div>

                <div>
                    <x-input-label for="body" :value="__('Texte de l\'article')" />
                    <textarea id="body"
                        class="block mt-1 w-full border-gray-300 dark:border-gray-700 dark:bg-
gray-900 dark:text-gray-300 focus:border-indigo-500 dark:focus:border-indigo-600 focus:ring-
indigo-500 dark:focus:ring-indigo-600 rounded-md shadow-sm"
                        name="body" rows="10">{{ old('body') }}</textarea>
                    <x-input-error :messages="errors->get('body')" class="mt-2" />
                </div>

                <div class="flex justify-end">
                    <x-primary-button type="submit">
                        {{ __('Créer') }}
                    </x-primary-button>
                </div>
            </form>
```

```
        </div>
    </div>
</x-app-layout>
```

Attention, nous avons ajouté l'attribut `enctype="multipart/form-data"` au formulaire pour indiquer que nous allons envoyer des données binaires (l'image). Si vous ne l'ajoutez pas, Laravel ne pourra pas récupérer les données de l'image et vous recevrez une erreur indiquant que l'image n'est pas d'un type autorisé.

Nous allons également modifier la vue `resources/views/admin/articles/edit.blade.php` :

PHP

```
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Articles') }}
        </h2>
    </x-slot>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Modifier un article
                </div>
            </div>

            <div class="text-gray-500">
                <form method="POST" action="{{ route('articles.update', article) }}" class="flex
flex-col space-y-4"
                    enctype="multipart/form-data">

                    @csrf
                    @method('PUT')

                    <div>
                        <x-input-label for="title" :value="__('Titre')" />
                        <x-text-input id="title" class="block mt-1 w-full" type="text"
name="title"
                            :value="old('title', article)" autofocus />
                        <x-input-error :messages="errors->get('title')" class="mt-2" />
                    </div>

                    <div>
                        <x-input-label for="published_at" :value="__('Date de publication')" />
                        <x-text-input id="published_at" class="block mt-1 w-full" type="date"
name="published_at"
                            :value="old('published_at', article->published_at?->toDateString())"
/>
                        <x-input-error :messages="errors->get('published_at')" class="mt-2" />
                    </div>

                    <div>
                        <x-input-label for="img" :value="__('Image')" />
                        @if (article->img_path)
                            <img src="{{ asset('storage/' . article->img_path) }}" alt="Image de
l'article"
                                class="aspect-auto h-64 rounded shadow mt-2 mb-4 object-cover
object-center">
                        @endif
                        <x-text-input id="img" class="block mt-1 w-full" type="file" name="img" />
                        <x-input-error :messages="errors->get('img')" class="mt-2" />
                    </div>

                    <div>
                        <x-input-label for="body" :value="__('Texte de l\'article')" />
                        <textarea id="body"
                            class="block mt-1 w-full border-gray-300 dark:border-gray-700 dark:bg-
gray-900 dark:text-gray-300 focus:border-indigo-500 dark:focus:border-indigo-600 focus:ring-
indigo-500 dark:focus:ring-indigo-600 rounded-md shadow-sm"
```

```
                            name="body" rows="10">{{ old('body', article) }}</textarea>
                        <x-input-error :messages="errors->get('body')" class="mt-2" />
                    </div>

                    <div class="flex justify-end">
                        <x-primary-button type="submit">
                            {{ __('Modifier') }}
                        </x-primary-button>
                    </div>
                </form>
            </div>
        </div>
    </div>
</x-app-layout>
```

Nous avons ajouté un `@if` pour afficher l'image de l'article si elle existe. N'oubliez pas d'ajouter à nouveau le `enctype="multipart/form-data"` au formulaire.

Contrôleur

Nous allons maintenant modifier le contrôleur `app/Http/Controllers/Admin/ArticleController.php` en commençant par la méthode `store` qui pour rappel permet de créer un nouvel article :

PHP

```php
public function store(ArticleCreateRequest $request)
{
    // On crée un nouvel article
    $article = Article::make();

    // On ajoute les propriétés de l'article
    $article->title = $request->validated()['title'];
    $article->body = $request->validated()['body'];
    $article->published_at = $request->validated()['published_at'];
    $article->user_id = Auth::id();


    // Si il y a une image, on la sauvegarde
    if ($request->hasFile('img')) {
        $path = $request->file('img')->store('articles', 'public');
        $article->img_path = $path;
    }

    // On sauvegarde l'article en base de données
    $article->save();

    return redirect()->route('articles.index');
}
```

Nous avons ajouté une condition pour vérifier si l'utilisateur a envoyé une image. Si c'est le cas, on la sauvegarde dans le dossier `public/storage/articles` et on ajoute le chemin de l'image dans la propriété `img_path` de l'article.

Nous allons maintenant modifier la méthode `update` qui permet de modifier un article :

```php
public function update(ArticleUpdateRequest request,Articlerequest, Article
request,Articlearticle)
{
    // On modifies les propriétés de l'article
    article->title=article→title = article->title=request→validated()['title'];
    article->body=article→body = article->body=request→validated()['body'];
    article->publishedat=article→published_at = article->publishedat=request→validated()
['published_at'];

    // Si il y a une image, on la sauvegarde
    if (request→hasFile('img')) {
        $path = $request→file('img')→store('articles', 'public');
        $article→img_path = $path;
    }

    // On sauvegarde les modifications en base de données
    article→save();

    return redirect()→back();
}
```

Nous avons ajouté une condition pour vérifier si l'utilisateur a envoyé une image. Si c'est le cas, on la sauvegarde dans le dossier `public/storage/articles` et on ajoute le chemin de l'image dans la propriété `img_path` de l'article.

Nous devons aussi modifier les deux Form Requests `app/Http/Requests/ArticleCreateRequest.php` et `app/Http/Requests/ArticleUpdateRequest.php` pour ajouter la validation de l'image :

```php
public function rules()
{
    return [
        'title' ⇒ 'required|unique:articles|max:255',
        'published_at' ⇒ 'nullable|date',
        'img' ⇒ 'nullable|image|max:2048',
        'body' ⇒ 'required|max:10000',
    ];
}
```

et

```php
public function rules()
{
    return [
        'title' ⇒ 'required|string|max:255|unique:articles,title,'.$this→route('article')→id,
        'published_at' ⇒ 'nullable|date',
        'img' ⇒ 'nullable|image|max:2048',
        'body' ⇒ 'required|string',
    ];
}
```

Vous pouvez maintenant ajouter une image à vos articles depuis l'interface d'administration.

Erreur de dossier temporaire d'upload avec Laragon

Si cela ne fonctionne pas et que vous utilisez Laragon, vous devez effectuer les étapes suivantes :

1. cliquez sur `Menu` puis `PHP` puis `php.ini`
2. cherchez la ligne `;upload_tmp_dir =`
3. supprimez le `;`
4. ajoutez le chemin vers le dossier `tmp` de Laragon : `upload_tmp_dir = "C:\laragon\tmp"`
5. redémarrez Laragon

Gestion des avatars

Laravel Breeze fournit une page de gestion du profile de l'utilisateur connecté. Nous allons ajouter la gestion des avatars à cette page. Pour cela, nous allons modifier le fichier `resources/views/profile/edit.blade.php` :

```xml
<x-app-layout>
  <x-slot name="header">
    <h2
      class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight"
    >
      {{ __('Profile') }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 space-y-6">
      <div class="p-4 sm:p-8 bg-white dark:bg-gray-800 shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('profile.partials.update-profile-information-form')
        </div>
      </div>

      <div class="p-4 sm:p-8 bg-white dark:bg-gray-800 shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('profile.partials.update-avatar-form')
        </div>
      </div>

      <div class="p-4 sm:p-8 bg-white dark:bg-gray-800 shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('profile.partials.update-password-form')
        </div>
      </div>

      <div class="p-4 sm:p-8 bg-white dark:bg-gray-800 shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('profile.partials.delete-user-form')
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```

Nous avons ajouté un bloc de code dans lequel on inclut le formulaire de modification de l'avatar. Pour cela, nous avons utilisé la directive `@include` de Blade. Elle permet d'inclure un fichier de vue dans une autre vue sans utiliser le système de components de Laravel. Nous allons maintenant créer ce fichier `resources/views/profile/partials/update-avatar-form.blade.php` :

XML

```
<section>
  <header>
    <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
      {{ __('Avatar') }}
    </h2>

    <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
      {{ __('Here you can change your avatar. It will be displayed on your
      profile and on your articles.') }}
    </p>
  </header>

  <form
    method="post"
    action="{{ route('profile.avatar.update') }}"
    class="mt-6 space-y-6"
    enctype="multipart/form-data"
  >
    @csrf @method('patch')

    <div class="flex flex-col space-y-2">
      <x-avatar :user="user" class="h-20 w-20"></x-avatar>

      <div class="">
        <label
          for="avatar"
          class="block text-sm font-medium text-gray-700 dark:text-gray-200"
        >
          {{ __('Avatar') }}
        </label>

        <div class="mt-1">
          <input
            type="file"
            name="avatar"
            id="avatar"
            class="block w-full shadow-sm sm:text-sm dark:bg-gray-700 dark:border-gray-700
dark:text-gray-200 dark:focus:ring-gray-500 dark:focus:border-gray-500 dark:placeholder-gray-400
dark:focus:ring-opacity-50 dark:focus:ring-offset-gray-800 dark:focus:ring-offset-opacity-50
dark:ring-offset-gray-800 dark:ring-offset-opacity-50 dark:ring-gray-500 dark:ring-opacity-50
rounded-md"
          />
        </div>

        <x-input-error :messages="errors->get('avatar')" class="mt-2" />
      </div>
    </div>

    <div class="flex items-center gap-4">
      <x-primary-button>{{ __('Save') }}</x-primary-button>

      @if (session('status') === 'profile-updated')
      <p
        x-data="{ show: true }"
        x-show="show"
        x-transition
        x-init="setTimeout(() => show = false, 2000)"
        class="text-sm text-gray-600 dark:text-gray-400"
      >
```

```
        {{ __('Saved.') }}
      </p>
      @endif
    </div>
  </form>
</section>
```

Nous avons ajouté un composant `x-avatar` qui affiche l'avatar de l'utilisateur que nous pourrons utiliser à plusieurs endroits de notre application. Nous allons maintenant créer ce composant dans le fichier `resources/views/components/avatar.blade.php` :

XML

```
<div {{ attributes->merge(['class' => 'rounded-full overflow-hidden']) }}>
  @if (user->avatar_path)
  <img
    class="w-full h-full aspect-square object-cover object-center"
    src="{{ asset('storage/' . user->avatar_path) }}"
    alt="{{ user->name }}"
  />
  @else
  <div class="w-full h-full aspect-square flex items-center justify-center bg-indigo-100">
    <span class="text-2xl font-medium text-indigo-800">
      {{ $user->name[0] }}
    </span>
  </div>
  @endif
</div>
```

Nous allons maintenant ajouter une route pour la modification de l'avatar dans le fichier `routes/web.php`. Ajoutez la ligne suivante juste après la route de suppression du profil :

PHP

```
// Gestion du profil utilisateur
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
    // On ajoute la route pour la modification de l'avatar
    Route::patch('/profile/avatar', [ProfileController::class, 'updateAvatar'])
->name('profile.avatar.update');
});
```

Nous allons maintenant créer la méthode `updateAvatar` dans le contrôleur `app/Http/Controllers/ProfileController.php` :

```php
public function updateAvatar(Request $request): RedirectResponse
{
    // Validation de l'image sans passer par une form request
    $request->validate([
        'avatar' => ['required', 'image', 'max:2048'],
    ]);

    // Si l'image est valide, on la sauvegarde
    if ($request->hasFile('avatar')) {
        $user = $request->user();
        $path = $request->file('avatar')->store('avatars', 'public');
        $user->avatar_path = $path;
        $user->save();
    }

    return Redirect::route('profile.edit')->with('status', 'avatar-updated');
}
```

Cette méthode est très simple. Elle valide l'image envoyée par l'utilisateur, la sauvegarde dans le dossier `storage/app/public/avatars` et met à jour le chemin de l'avatar dans la base de données.

Nous allons maintenant afficher l'avatar dans la page des détails d'un article. Pour cela, nous allons modifier le fichier `resources/views/front/articles/show.blade.php` :

```xml
<x-guest-layout>
  <h1 class="font-bold text-xl mb-4 capitalize">{{ article->title }}</h1>

  <div class="mb-4 text-xs text-gray-500">
    {{ article->published_at?->diffForHumans() }}
  </div>

  <div class="flex items-center justify-center">
    <img
      src="{{ asset('storage/' . article->img_path) }}"
      alt="illustration de l'article"
      class="rounded shadow aspect-auto object-cover object-center"
    />
  </div>

  <div class="mt-4">{!! \nl2br(e(article->body)) !!}</div>

  <div class="flex mt-8">
    <x-avatar class="h-20 w-20" :user="$article->user" />
    <div class="ml-4 flex flex-col justify-center">
      <div class="text-gray-700">{{ $article->user->name }}</div>
      <div class="text-gray-500">{{ $article->user->email }}</div>
    </div>
  </div>

  <div class="mt-8 flex items-center justify-center">
    <a
      href="{{ route('front.articles.index') }}"
      class="font-bold bg-white text-gray-700 px-4 py-2 rounded shadow"
    >
      Retour à la liste des articles
    </a>
  </div>
</x-guest-layout>
```

Nous avons ajouté le composant `x-avatar` qui affiche l'avatar de l'utilisateur qui a écrit l'article.

Nous devons aussi ajouter le champ `avatar_path` à la table des utilisateurs. Pour cela, nous allons créer une nouvelle migration :

```bash
php artisan make:migration add_avatar_path_to_users_table.php
```

Avec ce code :

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('avatar_path')->nullable();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void {}
};
```

Ensuite, lancer la commande d'exécution des migrations :

```bash
php artisan migrate
```

Utiliser des icônes

Installer le package

Nous allons maintenant ajouter des icônes à notre application. Nous allons utiliser le package blade-ui-kit/blade-icons

Pour installer ce package, lancez la commande suivante :

```bash
composer require blade-ui-kit/blade-icons
```

Plusieurs packs d'icônes sont disponibles. Nous allons utiliser le pack Heroicons. Pour installer ce pack, lancez la commande suivante :

```bash
composer require blade-ui-kit/blade-heroicons
```

En fonction des icônes que vous souhaitez utiliser, vous pouvez installer d'autres packs. Vous trouverez la commande d'installation dans la page de détail de chaque icône.

Utiliser les icônes

Pour voir la liste des icônes disponibles, rendez-vous sur https://blade-ui-kit.com/blade-icons#search

```xml
<x-heroicon-o-heart class="w-6 h-6 text-gray-500" />
```

On peut ajouter des classes CSS pour modifier la taille et la couleur de l'icône.

Nous allons maintenant ajouter une icône pour le bouton d'ajout, de modification et de suppression d'un article dans le fichier `resources/views/admin/articles/index.blade.php` :

XML

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Articles') }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
      <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg">
        <div class="p-6 sm:px-20 bg-white border-b border-gray-200">
          <div class="flex justify-between mt-8">
            <div class="text-2xl">Liste des articles</div>

            <div class="flex  items-center justify-center space-x-8">
              <a
                href="{{ route('articles.create') }}"
                class="text-gray-500 font-bold py-2 px-4 rounded hover:bg-gray-200 transition flex
items-center"
              >
                <x-heroicon-o-plus class="w-4 h-4 mr-2" />
                Ajouter un article
              </a>
            </div>
          </div>

          <div class="mt-6 text-gray-500">
            <table class="table-auto w-full">
              <thead>
                <tr class="uppercase text-left">
                  <th class="px-4 py-2 border">Titre</th>
                  <th class="px-4 py-2 border">Auteur</th>
                  <th class="px-4 py-2 border">Date de publication</th>
                  <th class="px-4 py-2 border">Dernière modification</th>
                  <th class="px-4 py-2 border">Actions</th>
                </tr>
              </thead>
              <tbody>
                @foreach (articlesasarticles as articlesasarticle)
                <tr
                  class="hover:bg-gray-50 odd:bg-gray-100 hover:odd:bg-gray-200 transition"
                >
                  <td class="border px-4 py-2">{{ article->title }}</td>
                  <td class="border px-4 py-2">{{ article->user->name }}</td>
                  <td class="border px-4 py-2">
                    {{ article->published_at?->diffForHumans() ?? 'Pas de date'
                    }}
                  </td>
                  <td class="border px-4 py-2">
                    {{ article->updated_at->diffForHumans() }}
                  </td>
                  <td class="border px-4 py-2 space-x-4">
                    <div class="flex space-x-4">
                      <a
                        href="{{ route('articles.edit', article->id) }}"
                        class="text-blue-400"
                      >
                        <x-heroicon-o-pencil class="w-5 h-5" />
                      </a>
```

```
                    <button
                      x-data="{ id: {{ $article->id }} }"
                      x-on:click.prevent="window.selected = id; dispatch('open-modal', 'confirm-
article-deletion');"
                      type="submit"
                      class="text-red-400"
                    >
                      <x-heroicon-o-trash class="w-5 h-5" />
                    </button>
                  </div>
                </td>
              </tr>
              @endforeach
            </tbody>
          </table>

          <div class="mt-4">{{ articles->links() }}</div>
        </div>
      </div>
    </div>
  </div>
  <x-modal name="confirm-article-deletion" focusable>
    <form
      method="post"
      onsubmit="event.target.action= '/admin/articles/' + window.selected"
      class="p-6"
    >
      @csrf @method('DELETE')

      <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
        Êtes-vous sûr de vouloir supprimer cet article ?
      </h2>

      <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
        Cette action est irréversible. Toutes les données seront supprimées.
      </p>

      <div class="mt-6 flex justify-end">
        <x-secondary-button x-on:click="dispatch('close')">
          Annuler
        </x-secondary-button>

        <x-danger-button class="ml-3" type="submit">
          Supprimer
        </x-danger-button>
      </div>
    </form>
  </x-modal>
</div>
</x-app-layout>
```

On a ajouté les lignes suivantes :

```xml
<x-heroicon-o-plus class="w-4 h-4 mr-2" />
```

```xml
<x-heroicon-o-pencil class="w-5 h-5" />
```

```xml
<x-heroicon-o-trash class="w-5 h-5" />
```

Code source

Le code source de ce tutoriel est disponible sur GitHub : https://github.com/opmvpc/blog-24-laravel11

1.15 Blog (part 2)

Recherche dans la liste des articles

Nous allons ajouter une barre de recherche à la liste des articles de la page `/articles`. Pour cela, nous allons modifier la méthode `index` du controller `ArticleController` et la vue `resources/views/front/articles/index.blade.php`.

Controller

Dans le controller `ArticleController`, nous allons modifier la méthode `index` pour ajouter des clauses `where` à la requête de récupération des articles qui prendront en compte la recherche de l'utilisateur.

```php
public function index(Request $request)
{
    $articles = Article::query()
        ->where('published_at', '<', now())
        ->when($request->query('search'), function ($query) use ($request) {
            $query->where('body', 'LIKE', '%'.$request->query('search').'%')
                ->orWhere('title', 'LIKE', '%'.$request->query('search').'%')
                ->orWhereHas('user', function ($query) use ($request) {
                    $query->where('name', 'LIKE', '%'.$request->query('search').'%');
                })
            ;
        })
        ->orderByDesc('published_at')
        ->paginate(12)
    ;

    return view('front.articles.index', [
        'articles' => $articles,
    ]);
}
```

Nous avons ajouté les lignes suivantes :

```php
 →where('body', 'LIKE', '%'.request->query('search').'->orWhere('title','LIKE','request
→query('search').'%')
 →orWhere('title', 'LIKE', '%'.request->query('search').'->orWhere('title','LIKE','request
→query('search').'%')
 →orWhereHas('user', function (query)use(query) use (query)use(request) {
    query->where('name','LIKE','query→where('name', 'LIKE', '%'.query->where('name','LIKE',
 'request→query('search').'%');
 })
```

La méthode `where` permet de filtrer les résultats de la requête. La méthode `orWhere` permet d'ajouter une clause `OR` à la requête. La méthode `orWhereHas` permet d'ajouter une clause `OR` à la requête en vérifiant si une relation existe.

1. La première clause `where` permet de filtrer les articles dont le texte contient la chaîne de caractères recherchée.
2. La deuxième clause `orWhere` permet de filtrer les articles dont le titre contient la chaîne de caractères recherchée.
3. La troisième clause `orWhereHas` permet de filtrer les articles dont l'auteur contient la chaîne de caractères recherchée.

Vue

Dans la vue `resources/views/front/articles/index.blade.php`, nous allons ajouter un formulaire de recherche à la page. Il utilisera la méthode `GET` et la route `front.articles.index` qui est la route qui pointe vers la méthode `index` du controller `ArticleController`.

```xml
<x-guest-layout>
  <h1 class="font-bold text-xl mb-4">Liste des articles</h1>

  <form action="{{ route('front.articles.index') }}" method="GET" class="mb-4">
    <div class="flex items-center">
      <input
        type="text"
        name="search"
        id="search"
        placeholder="Rechercher un article"
        class="flex-grow border border-gray-300 rounded shadow px-4 py-2 mr-4"
        value="{{ request()->search }}"
        autofocus
      />
      <button
        type="submit"
        class="bg-white text-gray-600 px-4 py-2 rounded-lg shadow"
      >
        <x-heroicon-o-magnifying-glass class="h-5 w-5" />
      </button>
    </div>
  </form>

  <ul class="grid sm:grid-cols-2 lg:grid-cols-3 2xl:grid-cols-4 gap-8">
    @foreach (articlesasarticles as articlesasarticle)
    <li>
      <x-article-card :article="$article" />
    </li>
    @endforeach
  </ul>

  <div class="mt-8">{{ $articles->links() }}</div>
</x-guest-layout>
```

Nous avons aussi ajouté un attribut `value` à l'input de recherche pour que la valeur de la recherche soit toujours présente dans le champ de recherche après avoir soumis le formulaire.

Gestion des rôles des utilisateurs

Nous allons ajouter une gestion des rôles des utilisateurs. Nous allons créer un nouveau modèle `Role` qui va permettre de définir les rôles des utilisateurs.

L'application va utiliser les rôles suivants :

- `admin` : l'utilisateur est administrateur de l'application. Il peut tout faire.
- `author` : l'utilisateur est auteur de l'application. Il peut créer et modifier ses articles. Il peut aussi commenter les articles.
- `user` : l'utilisateur est un utilisateur lambda. Il peut seulement commenter les articles.

Création du modèle

Nous allons créer un nouveau modèle `Role` avec la commande suivante :

```bash
php artisan make:model Role -m
```

On ne demande que la création du modèle et de la migration.

Modèle

Nous allons modifier le modèle `Role` pour y définir les constantes des rôles disponibles ainsi qu'une méthode statique `roles` qui retourne un tableau contenant les rôles disponibles. Nous ajoutons aussi une relation `users` qui permet de récupérer les utilisateurs qui ont ce rôle.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    use HasFactory;

    // Constantes des rôles disponibles
    public const ADMIN = 'Administrateur';
    public const AUTHOR = 'Auteur';
    public const USER = 'Utilisateur';

    public static function roles(): array
    {
        return [
            self::ADMIN,
            self::AUTHOR,
            self::USER,
        ];
    }

    public function users(): \Illuminate\Database\Eloquent\Relations\HasMany
    {
        return $this->hasMany(User::class);
    }
}
```

Nous devons aussi ajouter la méthode `role` dans le modèle `User` pour récupérer le rôle de l'utilisateur après la méthode `articles` :

```php
public function role(): \Illuminate\Database\Eloquent\Relations\BelongsTo
{
    return $this->belongsTo(Role::class);
}
```

Migration

Nous allons modifier la date au début du nom du fichier de migration `[DATETIME]_create_roles_table.php` pour qu'il soit exécuté avant la migration de création de la table `users`.

```bash
0000_01_01_000000_create_roles_table.php
0001_01_01_000000_create_users_table.php
# autres migrations
```

Nous allons modifier la migration des rôles `0000_01_01_000000_create_roles_table.php` comme ceci :

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class() extends Migration {
    /**
     * Run the migrations.
     */
    public function up()
    {
        Schema::create('roles', function (Blueprint $table) {
            $table->id();
            $table->enum('name', \App\Models\Role::roles());
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down()
    {
        Schema::dropIfExists('roles');
    }
};
```

Nous avons ajouté une colonne `name` de type `enum` qui contient les rôles disponibles. Nous avons utilisé la méthode statique `roles` du modèle `Role` pour récupérer les rôles disponibles. Cela veut dire que le champ `name` de la table `roles` ne peut contenir que les valeurs `Administrateur`, `Auteur` et `Utilisateur`.

Nous allons modifier la migration de création de la table `users`, `0001_01_01_000000_create_users_table.php`, comme ceci :

```php
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->foreignId('role_id')->constrained()->default(3);
            $table->rememberToken();
            $table->timestamps();
        });
    }
```

Nous avons ajouté un champ `role_id` qui est une clé étrangère vers la table `roles`. Nous avons aussi ajouté une valeur par défaut de `3` qui correspond au rôle `Utilisateur`.

Seeders

Nous allons commencer par remettre de l'ordre dans les seeders. Nous allons créer des seeders pour les rôles et les utilisateurs. Le seeder pour les articles existe déjà.

Pour créer les seeders, nous allons utiliser la commande suivante :

```bash
php artisan make:seeder RolesTableSeeder
php artisan make:seeder UsersTableSeeder
```

Nous allons modifier le fichier `DatabaseSeeder.php` pour qu'il appelle les seeders de rôles, d'utilisateurs et d'articles :

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Storage;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        // On supprime les anciennes images de test
        images=Storage::disk('public')->files('images');Storage::disk('public')->delete(images =
Storage::disk('public')→files('images');
        Storage::disk('public')→delete(images=Storage::disk('public')
->files('images');Storage::disk('public')->delete(images);

        $this→call([
            RolesTableSeeder::class,
            UsersTableSeeder::class,
            ArticlesTableSeeder::class,
        ]);
    }
}
```

Ensuite, nous allons modifier les seeders pour qu'ils créent les rôles, les utilisateurs et les articles.

Nous allons commencer par le seeder de rôles `RolesTableSeeder.php` :

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;

class RolesTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run()
    {
        // On crée les rôles
        \App\Models\Role::create([
            'name' => \App\Models\Role::ADMIN,
        ]);

        \App\Models\Role::create([
            'name' => \App\Models\Role::AUTHOR,
        ]);

        \App\Models\Role::create([
            'name' => \App\Models\Role::USER,
        ]);
    }
}
```

Nous allons ensuite modifier le seeder d'utilisateurs `UsersTableSeeder.php` :

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run()
    {
        // On crée un admin
        User::factory()->create([
            'name' => 'Admin User',
            'email' => 'admin@example.com',
            'role_id' => 1,
        ]);

        // 10 auteurs
        User::factory(10)->create([
            'role_id' => 2,
        ]);

        // On crée notre utilisateur de test qui sera maintenant un utilisateur lambda
        User::factory()->create([
            'name' => 'User User',
            'email' => 'test@example.com',
            'role_id' => 3,
        ]);

        // 10 utilisateurs lambda
        User::factory(10)->create([
            'role_id' => 3,
        ]);
    }
}
```

Et, on va aussi modifier la factory des articles pour qu'il n'y ait que des articles créés par des auteurs :

```php
<?php

namespace Database\Factories;

use App\Models\Role;
use App\Models\User;
use Illuminate\Contracts\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Article>
 */
class ArticleFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'title' => fake()->bs(),
            'body' => fake()->realTextBetween(minNbChars=500,minNbChars = 500,
minNbChars=500,maxNbChars = 2000),
            'img_path' => function () {
                randomName=Str::uuid();randomName = Str::uuid();
                randomName=Str::uuid();imageUrl = "https://picsum.photos/1024/768.webp?random=
{randomName}";
                path = "images/{randomName}.webp";
                Storage::disk('public')->put(path, file_get_contents(imageUrl));returnimageUrl));

                return imageUrl));returnpath;
            },
            'published_at' => fake()->dateTimeBetween('-2 months', '+ 1 month'),
            'user_id' => User::query()->whereHas('role', function (Builder $query) {
                $query->where('name', Role::AUTHOR);
            })->get()->random()->id,
        ];
    }
}
```

Nous allons maintenant recréer la base de données et la remplir avec les données de test :

```bash
php artisan migrate:fresh --seed
```

Output :

```bash
λ php artisan migrate:fresh --seed

  Dropping all tables ..................................................... 4.75ms DONE

   INFO  Preparing database.

  Creating migration table ................................................ 6.52ms DONE

   INFO  Running migrations.

  0000_01_01_000000_create_roles_table ..................................... 4.67ms DONE
  0001_01_01_000000_create_users_table ..................................... 23.48ms DONE
  0001_01_01_000001_create_cache_table ..................................... 7.59ms DONE
  0001_01_01_000002_create_jobs_table ...................................... 18.68ms DONE
  2024_10_13_101030_create_articles_table .................................. 3.63ms DONE
  2024_10_13_152719_add_avatar_path_to_users_table ......................... 3.70ms DONE


   INFO  Seeding database.

  Database\Seeders\RolesTableSeeder ........................................ RUNNING
  Database\Seeders\RolesTableSeeder ....................................... 14 ms DONE

  Database\Seeders\UsersTableSeeder ........................................ RUNNING
  Database\Seeders\UsersTableSeeder ....................................... 289 ms DONE

  Database\Seeders\ArticlesTableSeeder ..................................... RUNNING
  Database\Seeders\ArticlesTableSeeder ................................. 49,433 ms DONE
```

Si vous avez des erreurs, vérifiez que vous avez bien réalisé les étapes précédentes.

Gestion des utilisateurs

Nous allons maintenant créer un système de gestion des utilisateurs avec les fonctionnalités suivantes :

- Uniquement accessible aux utilisateurs ayant le rôle `Administateur`
- Afficher la liste des utilisateurs
- Afficher les détails et modifier le rôle d'un utilisateur

Création du contrôleur

Nous allons créer un contrôleur de type `resource` pour gérer les utilisateurs :

```bash
php artisan make:controller Admin/UserController --resource --model=User
```

Création des routes

On va ajouter la route `resource` pour le contrôleur `UserController` après celle des articles :

```php
Route::middleware(['auth', 'verified'])->prefix('admin')->group(function () {
    // Gestion des articles (création, modification, suppression)
    Route::resource('/articles', AdminArticleController::class);

    // Gestion des utilisateurs (Détails et changement de rôle)
    Route::resource('/users', UserController::class);
});
```

Création des vues

Nous allons créer les vues pour les actions `index`, `edit` du contrôleur `UserController`.

Vue index

Nous allons d'abord ajouter un dossier `users` dans le dossier `resources/views/admin` et ensuite, créer la vue `index.blade.php` dans le dossier `resources/views/admin/users` :

XML

```xml
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Utilisateurs') }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
      <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg">
        <div class="p-6 sm:px-20 bg-white border-b border-gray-200">
          <div class="flex justify-between mt-8">
            <div class="text-2xl">Liste des utilisateurs</div>

            <div class="flex items-center justify-center space-x-8"></div>
          </div>

          <div class="mt-6 text-gray-500">
            <table class="table-auto w-full">
              <thead>
                <tr class="uppercase text-left">
                  <th class="px-4 py-2 border">Nom</th>
                  <th class="px-4 py-2 border">Email</th>
                  <th class="px-4 py-2 border">Rôle</th>
                  <th class="px-4 py-2 border">Actions</th>
                </tr>
              </thead>
              <tbody>
                @foreach (usersasusers as usersasuser)
                <tr
                  class="hover:bg-gray-50 odd:bg-gray-100 hover:odd:bg-gray-200 transition"
                >
                  <td class="border px-4 py-2">{{ user->name }}</td>
                  <td class="border px-4 py-2">{{ user->email }}</td>
                  <td class="border px-4 py-2">{{ user->role->name }}</td>
                  <td class="border px-4 py-2 space-x-4">
                    <div class="flex space-x-4">
                      <a
                        href="{{ route('users.edit', user) }}"
                        class="text-blue-400"
                      >
                        <x-heroicon-o-pencil class="w-5 h-5" />
                      </a>
                    </div>
                  </td>
                </tr>
                @endforeach
              </tbody>
            </table>

            <div class="mt-4">{{ $users->links() }}</div>
          </div>
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```

Nous affichons la liste des utilisateurs dans un tableau avec les informations suivantes :

- Nom
- Email
- Rôle
- Actions: Modifier

Nous utilisons la pagination de Laravel pour afficher les utilisateurs par page.

Vue edit

Nous allons créer la vue `edit.blade.php` dans le dossier `resources/views/admin/users` :

XML

```xml
<x-app-layout>

    <x-slot name="header">
        <h2 class="font-semibold text-xl text-gray-800 leading-tight">
            {{ __('Utilisateurs') }}
        </h2>
    </x-slot>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Détails de l'utilisateur
                </div>
            </div>



            <div class="flex text-gray-500">
                <div class="flex justify-center items-center">
                    <x-avatar :user="user" class="w-24 h-24" />
                </div>
                <dl class="mt-6 space-y-6">
                    <div class="flex space-x-4">
                        <dt class="text-sm font-medium text-gray-500 w-20 text-right">
                            Nom
                        </dt>
                        <dd class="text-sm text-gray-900">
                            {{ $user->name }}
                        </dd>
                    </div>
                    <div class="flex space-x-4">
                        <dt class="text-sm font-medium text-gray-500 w-20 text-right">
                            Email
                        </dt>
                        <dd class="text-sm text-gray-900">
                            {{ $user->email }}
                        </dd>
                    </div>
                    <div class="flex space-x-4">
                        <dt class="text-sm font-medium text-gray-500 w-20 text-right">
                            Rôle
                        </dt>
                        <dd class="text-sm text-gray-900">
                            {{ $user->role->name }}
                        </dd>
                    </div>
                </dl>
            </div>
        </div>
    </div>

    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 py-12">
        <div class="bg-white overflow-hidden shadow-xl sm:rounded-lg p-6">
            <div class="flex justify-between mt-8">
                <div class=" text-2xl">
                    Modifier le rôle de l'utilisateur
                </div>
            </div>
```

```
            <div class="text-gray-500">
                <form method="POST" action="{{ route('users.update', $user) }}" class="flex flex-
col space-y-4">

                    @csrf
                    @method('PUT')

                    <div>
                        <x-input-label for="role" :value="__('Rôle')" />

                        <select id="role" name="role"
                            class="block mt-1 w-full border-gray-300 rounded-md shadow-sm
focus:border-indigo-300 focus:ring focus:ring-indigo-200 focus:ring-opacity-50"
                            required>
                            @foreach (roles as role)
                                <option value="{{ $role }}" {{ $role == $user->role->name ?
'selected' : '' }}>
                                    {{ $role }}
                                </option>
                            @endforeach
                        </select>

                        <x-input-error :messages="errors->get('role')" class="mt-2" />
                    </div>

                    <div class="flex justify-end">
                        <x-primary-button type="submit">
                            {{ __('Modifier') }}
                        </x-primary-button>
                    </div>
                </form>
            </div>
        </div>
    </div>
</x-app-layout>
```

Nous affichons les informations de l'utilisateur et nous créons un formulaire pour modifier le rôle de l'utilisateur. Nous utilisons un input de type `select` pour afficher la liste des rôles.

Controller

Nous allons modifier le controller `UserController` dans le fichier `app/Http/Controllers/Admin/UserController.php` en commençant par la méthode `index` :

```php
public function index()
{
    // On récupère tous les utilisateurs avec pagination de 10 par page

users=User::paginate(10);//Onpasselesutilisateursa`lavue'admin.users.index'returnview('admin.users
.index',['users'⇒users = User::paginate(10);

    // On passe les utilisateurs à la vue `admin.users.index`
    return view('admin.users.index', [
        'users' ⇒
users=User::paginate(10);//Onpasselesutilisateursa`lavue'admin.users.index'returnview('admin.users
.index',['users'⇒users,
    ]);
}
```

Rien de nouveau ici, nous récupérons tous les utilisateurs avec pagination de 10 par page et nous les passons à la vue `admin.users.index`.

Nous allons maintenant modifier la méthode `edit` :

```php
    public function edit(User $user)
    {
        // On récupère tous les rôles disponibles
        // On utilise la méthode pluck() pour récupérer uniquement le nom des rôles dans un
tableau
        $roles = \App\Models\Role::pluck('name');

        // On passe l'utilisateur et les rôles à la vue `admin.users.edit`
        return view('admin.users.edit', [
            'user' ⇒ $user,
            'roles' ⇒ $roles,
        ]);
    }
```

Pour rappel, la méthode `edit` reçoit en paramètre un objet `User` grâce à la route `users/{user}/edit`. Cette route est responsable de générer la vue comportant le formulaire de modification du rôle de l'utilisateur. Nous récupérons donc tous les rôles disponibles et nous les passons à la vue `admin.users.edit` avec l'utilisateur.

Nous allons maintenant modifier la méthode `update` :

```php
    public function update(Request request,Userrequest, User request,Useruser)
    {
        // Validation du rôle

request->validate([//Onve´rifiequelero^leestbienundesro^lesde´finisdanslemode`leRole'role'⇒'requi
red|in:'.implode(',
',\App\Models\Role::roles()),]);//Misea`jourduro^le//Onre´cupe`relero^lecorrespondantaunomduro^lep
asse´danslareque^terequest→validate([
            // On vérifie que le rôle est bien un des rôles définis dans le modèle Role
            'role' ⇒ 'required|in:'.implode(',', \App\Models\Role::roles()),
        ]);

        // Mise à jour du rôle
        // On récupère le rôle correspondant au nom du rôle passé dans la requête

request->validate([//Onve´rifiequelero^leestbienundesro^lesde´finisdanslemode`leRole'role'⇒'requi
red|in:'.implode(',
',\App\Models\Role::roles()),]);//Misea`jourduro^le//Onre´cupe`relero^lecorrespondantaunomduro^lep
asse´danslareque^terole = \App\Models\Role::where('name', request->role)
->first();//Onassocielero^lea`l'utilisateurenpassantparlarelationrequest→role)→first();
        // On associe le rôle à l'utilisateur en passant par la relation
        request->role)->first();//Onassocielero^lea`l'utilisateurenpassantparlarelationuser-
>role()→associate(role);//Onsauvegardel'utilisateurenbasededonne´esrole);
        // On sauvegarde l'utilisateur en base de données
        role);//Onsauvegardel'utilisateurenbasededonne´esuser→save();

        // Redirection vers la page de modification de l'utilisateur
        return redirect()→back();
    }
```

Nous validons le rôle passé dans la requête. Nous récupérons le rôle correspondant au nom du rôle passé dans la requête et nous l'associons à l'utilisateur en passant par la relation `role`. Nous sauvegardons l'utilisateur en base de données et nous redirigeons vers la page de modification de l'utilisateur.

Autorisations avec les policies

Nous n'avons pas encore protégé l'accès aux pages réservées aux administrateurs. Pour gérer finement les autorisations des différents groupes d'utilisateurs, Laravel nous propose les policies.

Les policies sont des classes qui permettent de définir les autorisations des utilisateurs. Pour chaque Model, nous pouvons définir une policy qui permettra de définir les autorisations pour les actions CRUD (Create, Read, Update, Delete).

Nous allons créer une policy pour le modèle `User` en utilisant la commande Artisan suivante :

```bash
php artisan make:policy UserPolicy --model=User
```

Nous obtenons le fichier `app/Policies/UserPolicy.php` qui contient les méthodes `create`, `view`, `update` et `delete` qui correspondent aux actions CRUD. Nous allons commencer par définir les autorisations pour la méthode `viewAny` qui nous servira à déterminer si l'utilisateur peut accéder à la liste des utilisateurs :

```php
public function viewAny(User $user)
{
    // Seul l'administrateur peut voir la liste des utilisateurs
    return \App\Models\Role::ADMIN === $user->role->name;
}
```

Cette méthode retourne `true` si l'utilisateur est un administrateur et `false` sinon.

Nous allons maintenant définir les autorisations pour la méthode `update` qui nous servira à déterminer si l'utilisateur peut modifier un utilisateur :

```php
public function update(User user,Useruser, User user,Usermodel)
{
    return $user->role->name === \App\Models\Role::ADMIN;
}
```

La méthode `update` reçoit en paramètre l'utilisateur connecté et l'utilisateur à modifier. Cette méthode retourne `true` si l'utilisateur connecté est un administrateur et `false` sinon, comme pour la méthode `viewAny`.

On n'a pas besoin de définir les autorisations pour les méthodes `create`, `view` et `delete` car nous n'utilisons pas ces méthodes dans notre application pour le modèle `User`.

Nous allons maintenant modifier le controller `UserController` pour utiliser les policies. Nous allons commencer par modifier la méthode `index` :

```php
public function index()
{
    // On vérifie que l'utilisateur courant est un administrateur
    Gate::authorize('viewAny', User::class);

    // On récupère tous les utilisateurs avec pagination de 10 par page

users=User::paginate(10);//Onpasselesutilisateursa`lavue'admin.users.index'returnview('admin.users
.index',['users'=>users = User::paginate(10);

    // On passe les utilisateurs à la vue `admin.users.index`
    return view('admin.users.index', [
        'users' =>
users=User::paginate(10);//Onpasselesutilisateursa`lavue'admin.users.index'returnview('admin.users
.index',['users'=>users,
    ]);
}
```

Nous utilisons la classe `Gate` pour appeler la méthode `viewAny` de la policy `UserPolicy` dans laquelle nous vérifions que l'utilisateur courant est un administrateur. Si ce n'est pas le cas, une exception `Illuminate\Auth\Access\AuthorizationException` sera levée.

N'oubliez pas d'importer la classe `Gate` en haut du fichier `app/Http/Controllers/Admin/UserController.php` :

```php
use Illuminate\Support\Facades\Gate;
```

Nous allons maintenant modifier la méthode `edit` :

```php
public function edit(User $user)
{
    // On vérifie que l'utilisateur courant est un administrateur
    Gate::authorize('update', $user);

    // On récupère tous les rôles disponibles
    // On utilise la méthode pluck() pour récupérer uniquement le nom des rôles dans un tableau
    $roles = \App\Models\Role::pluck('name');

    // On passe l'utilisateur et les rôles à la vue `admin.users.edit`
    return view('admin.users.edit', [
        'user' => $user,
        'roles' => $roles,
    ]);
}
```

Nous utilisons la classe `Gate` pour appeler la méthode `update` de la policy `UserPolicy`. Cette méthode reçoit en paramètre l'utilisateur courant et l'utilisateur à modifier. Si l'utilisateur courant n'est pas un administrateur, une exception `Illuminate\Auth\Access\AuthorizationException` sera levée.

Nous allons maintenant ajouter la même ligne de code dans la méthode `update` :

```php
public function update(Request request,Userrequest, User request,Useruser)
{
    // On vérifie que l'utilisateur courant est un administrateur
    Gate::authorize('update', $user);

    // ... code existant ...
}
```

Nous avons maintenant protégé l'accès aux pages réservées aux administrateurs. Vous pouvez tester en vous connectant avec un utilisateur qui n'est pas un administrateur et en essayant d'accéder à la page `/admin/users` ou à la page de modification d'un utilisateur `/admin/users/1/edit`.

Navigation

On doit aussi afficher le lien vers la page de gestion des utilisateurs dans la barre de navigation seulement si l'utilisateur est un administrateur. Pour cela, nous allons modifier le fichier `resources/views/layouts/navigation.blade.php` :

XML

```
<nav
  x-data="{ open: false }"
  class="bg-white dark:bg-gray-800 border-b border-gray-100 dark:border-gray-700"
>
  <!-- Primary Navigation Menu -->
  <div class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
    <div class="flex justify-between h-16">
      <div class="flex">
        <!-- Logo -->
        <div class="shrink-0 flex items-center">
          <a href="{{ route('dashboard') }}">
            <x-application-logo
              class="block h-9 w-auto fill-current text-gray-800 dark:text-gray-200"
            />
          </a>
        </div>

        <!-- Navigation Links -->
        <div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
          <x-nav-link
            :href="route('dashboard')"
            :active="request()->routeIs('dashboard')"
          >
            {{ __('Dashboard') }}
          </x-nav-link>

          <x-nav-link
            :href="route('articles.index')"
            :active="request()->routeIs('articles.*')"
          >
            {{ __('Articles') }}
          </x-nav-link>

          @can('viewAny', App\Models\User::class)
          <x-nav-link
            :href="route('users.index')"
            :active="request()->routeIs('users.*')"
          >
            {{ __('Utilisateurs') }}
          </x-nav-link>
          @endcan
        </div>
      </div>

      <!-- Settings Dropdown -->
      <div class="hidden sm:flex sm:items-center sm:ml-6">
        <x-dropdown align="right" width="48">
          <x-slot name="trigger">
            <button
              class="inline-flex items-center px-3 py-2 border border-transparent text-sm leading-
4 font-medium rounded-md text-gray-500 bg-white dark:bg-gray-800 hover:text-gray-700
dark:hover:text-gray-300 focus:outline-none transition ease-in-out duration-150"
            >
              <div>{{ Auth::user()->name }}</div>

              <div class="ml-1">
                <svg
                  class="fill-current h-4 w-4"
                  xmlns="http://www.w3.org/2000/svg"
```

```
                  viewBox="0 0 20 20"
                >
                  <path
                    fill-rule="evenodd"
                    d="M5.293 7.293a1 1 0 011.414 0L10 10.586l3.293-3.293a1 1 0 111.414 1.414l-4
4a1 1 0 01-1.414 0l-4-4a1 1 0 010-1.414z"
                    clip-rule="evenodd"
                  />
                </svg>
              </div>
            </button>
          </x-slot>

          <x-slot name="content">
            <x-dropdown-link :href="route('profile.edit')">
              {{ __('Profile') }}
            </x-dropdown-link>

            <!-- Authentication -->
            <form method="POST" action="{{ route('logout') }}">
              @csrf

              <x-dropdown-link
                :href="route('logout')"
                onclick="event.preventDefault();
                                            this.closest('form').submit();"
              >
                {{ __('Log Out') }}
              </x-dropdown-link>
            </form>
          </x-slot>
        </x-dropdown>
      </div>

      <!-- Hamburger -->
      <div class="-mr-2 flex items-center sm:hidden">
        <button
          @click="open = ! open"
          class="inline-flex items-center justify-center p-2 rounded-md text-gray-400 dark:text-
gray-500 hover:text-gray-500 dark:hover:text-gray-400 hover:bg-gray-100 dark:hover:bg-gray-900
focus:outline-none focus:bg-gray-100 dark:focus:bg-gray-900 focus:text-gray-500 dark:focus:text-
gray-400 transition duration-150 ease-in-out"
        >
          <svg
            class="h-6 w-6"
            stroke="currentColor"
            fill="none"
            viewBox="0 0 24 24"
          >
            <path
              :class="{ 'hidden': open, 'inline-flex': !open }"
              class="inline-flex"
              stroke-linecap="round"
              stroke-linejoin="round"
              stroke-width="2"
              d="M4 6h16M4 12h16M4 18h16"
            />
            <path
              :class="{ 'hidden': !open, 'inline-flex': open }"
              class="hidden"
```

```
                stroke-linecap="round"
                stroke-linejoin="round"
                stroke-width="2"
                d="M6 18L18 6M6 6l12 12"
              />
            </svg>
          </button>
        </div>
      </div>
    </div>

    <!-- Responsive Navigation Menu -->
    <div :class="{ 'block': open, 'hidden': !open }" class="hidden sm:hidden">
      <div class="pt-2 pb-3 space-y-1">
        <x-responsive-nav-link
          :href="route('dashboard')"
          :active="request()->routeIs('dashboard')"
        >
          {{ __('Dashboard') }}
        </x-responsive-nav-link>

        <x-responsive-nav-link
          :href="route('articles.index')"
          :active="request()->routeIs('articles.*')"
        >
          {{ __('Articles') }}
        </x-responsive-nav-link>

        @can('viewAny', App\Models\User::class)
        <x-responsive-nav-link
          :href="route('users.index')"
          :active="request()->routeIs('users.*')"
        >
          {{ __('Utilisateurs') }}
        </x-responsive-nav-link>
        @endcan
      </div>

      <!-- Responsive Settings Options -->
      <div class="pt-4 pb-1 border-t border-gray-200 dark:border-gray-600">
        <div class="px-4">
          <div class="font-medium text-base text-gray-800 dark:text-gray-200">
            {{ Auth::user()->name }}
          </div>
          <div class="font-medium text-sm text-gray-500">
            {{ Auth::user()->email }}
          </div>
        </div>

        <div class="mt-3 space-y-1">
          <x-responsive-nav-link :href="route('profile.edit')">
            {{ __('Profile') }}
          </x-responsive-nav-link>

          <!-- Authentication -->
          <form method="POST" action="{{ route('logout') }}">
            @csrf

            <x-responsive-nav-link
              :href="route('logout')"
```

```
            onclick="event.preventDefault();
                                    this.closest('form').submit();"
          >
            {{ __('Log Out') }}
          </x-responsive-nav-link>
        </form>
      </div>
    </div>
  </div>
</nav>
```

Nous avons ajouté le lien vers la liste des utilisateurs dans le menu de navigation pour PC et pour téléphone :

```XML
@can('viewAny', App\Models\User::class)
<x-nav-link
  :href="route('users.index')"
  :active="request()->routeIs('users.*')"
>
  {{ __('Utilisateurs') }}
</x-nav-link>
@endcan
```

et

```XML
@can('viewAny', App\Models\User::class)
<x-responsive-nav-link
  :href="route('users.index')"
  :active="request()->routeIs('users.*')"
>
  {{ __('Utilisateurs') }}
</x-responsive-nav-link>
@endcan
```

Nous utilisons la directive `@can` pour vérifier si l'utilisateur connecté a la permission de voir la liste des utilisateurs. Si c'est le cas, nous affichons le lien.

Autorisations pour les articles

Vous pouvez maintenant faire la même chose pour protéger la partie articles. Vous devrez créer une policy pour les articles et définir les autorisations. Ensuite, utilisez la classe `Gate` dans les contrôleurs et la directive `@can` dans les vues pour vérifier les autorisations.

- Les utilisateurs ne peuvent rien faire avec les articles.
- Les auteurs peuvent créer des articles, modifier leurs propres articles et supprimer leurs propres articles.
- Les administrateurs peuvent créer des articles, modifier tous les articles et supprimer tous les articles.

Système de commentaires

Nous allons ajouter un système de commentaires à notre blog. Les utilisateurs connectés pourront commenter les articles. Les administrateurs et le propriétaire d'un article pourront supprimer les commentaires.

Pour cela, nous allons devoir créer un modèle Comment et une table `comments` dans la base de données. La table `comments` aura les champs suivants :

- id
- article_id
- user_id
- body
- created_at
- updated_at

Le modèle Comment aura les relations suivantes:

- article: belongsTo
- user: belongsTo

Nous devrons aussi définir les relations inverse dans les modèles Article et User.

Ensuite, nous devrons modifier l'`ArticleController` pour ajouter la possibilité de commenter un article. Nous allons aussi modifier la vue `show` pour afficher les commentaires et ajouter un formulaire pour ajouter un commentaire.

Nous n'oublierons pas de gérer les autorisations pour les commentaires.

Modèle Comment

Nous allons créer le modèle Comment avec la commande artisan :

```bash
php artisan make:model Comment -mfs
```

Nous avons utilisé les options `-mfs` pour créer le modèle, la migration, la factory et le seeder en une seule commande.

Nous allons d'abord modifier le fichier de migration `create_comments_table` pour ajouter les champs `body`, `article_id` et `user_id` :

```php
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained()->cascadeOnDelete();
        $table->foreignId('article_id')->nullable()->constrained()->nullOnDelete();
        $table->text('body');
        $table->timestamps();
    });
}
```

Nous avons ajouté les champs `user_id` et `article_id` qui seront utilisés pour les relations. Nous avons aussi ajouté le champ `body` qui contiendra le texte du commentaire.

Nous allons ensuite modifier le modèle Comment pour ajouter les relations :

```php
class Comment extends Model
{
    use HasFactory;

    public function article()
    {
        return this->belongsTo(Article::class);
    }

    public function user()
    {
        return this->belongsTo(User::class);
    }
}
```

Nous devons aussi modifier les modèles Article et User pour ajouter les relations inverses:

```php
class Article extends Model
{
    // ... code précédent

    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

et

```php
class User extends Authenticatable
{
    // ... code précédent

    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Données de test

Nous allons ajouter des données de test pour les commentaires. Nous allons modifier la `CommentFactory` pour générer des données aléatoires :

```php
class CommentFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
        return [
            // n'importe quel utilisateur
            'user_id' => \App\Models\User::get()->random()->id,
            // n'importe quel article
            'article_id' => \App\Models\Article::get()->random()->id,
            // un texte aléatoire de 20 à 400 caractères
            'body' => fake()->realTextBetween(minNbChars=20,minNbChars = 20,
minNbChars=20,maxNbChars = 400),
            // une date aléatoire entre -2 mois et maintenant
            'created_at' => fake()->dateTimeBetween('-2 months', 'now'),
        ];
    }
}
```

Nous allons modifier le seeder `CommentSeeder` pour utiliser la factory :

```php
class CommentSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run()
    {
        \App\Models\Comment::factory(300)->create();
    }
}
```

Nous allons ensuite modifier la méthode `run` du seeder `DatabaseSeeder` pour appeler le seeder `CommentSeeder` :

```php
public function run(): void
{
    // On supprime les anciennes images de test
    images=Storage::disk('public')->files('images');Storage::disk('public')->delete(images =
Storage::disk('public')→files('images');
    Storage::disk('public')→delete(images=Storage::disk('public')
->files('images');Storage::disk('public')->delete(images);

    // On appelle les seeders dans l'ordre pour éviter les erreurs de clés étrangères
    $this→call([
        RolesTableSeeder::class,
        UsersTableSeeder::class,
        ArticlesTableSeeder::class,
        CommentSeeder::class,
    ]);
}
```

Vous pouvez maintenant recréer la base de données et exécuter les seeders:

```bash
php artisan migrate:fresh --seed
```

Contrôleur ArticleController

Nous allons modifier le contrôleur `ArticleController` de la partie front-office de notre blog pour afficher les commentaires.

Nous allons modifier la méthode `show` pour récupérer les commentaires de l'article et les passer à la vue :

```php
public function show($id)
{
    // On récupère l'article et on renvoie une erreur 404 si l'article n'existe pas
    $article = Article::findOrFail($id);
    // On récupère les commentaires de l'article, avec les utilisateurs associés (via la relation)
    // On les trie par date de création (le plus ancien en premier)
    $comments = $article
        →comments()
        →with('user')
        →orderBy('created_at')
        →get()
    ;

    // On renvoie la vue avec les données
    return view('front.articles.show', [
        'article' ⇒ $article,
        'comments' ⇒ $comments,
    ]);
}
```

Nous allons ensuite modifier la vue `front.articles.show` pour afficher les commentaires :

PHP

```
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4 capitalize">{{ article->title }}</h1>

    <div class="mb-4 text-xs text-gray-500">
        {{ article->published_at?->diffForHumans() }}
    </div>

    <div class="flex items-center justify-center">
        <img src="{{ asset('storage/' . article->img_path) }}" alt="illustration de l'article"
            class="rounded shadow aspect-auto object-cover object-center">
    </div>

    <div class="mt-4">
        {!! \nl2br(e(article->body)) !!}
    </div>

    <div class="flex mt-8">
        <x-avatar class="h-20 w-20" :user="article->user" />
        <div class="ml-4 flex flex-col justify-center">
            <div class="text-gray-700">
                {{ $article->user->name }}
            </div>
            <div class="text-gray-500">
                {{ $article->user->email }}
            </div>
        </div>
    </div>

    <div class="mt-8 flex items-center justify-center">
        <a href="{{ route('front.articles.index') }}" class="font-bold bg-white text-gray-700 px-4
py-2 rounded shadow">
            Retour à la liste des articles
        </a>
    </div>

    <div class="mt-8">
        <h2 class="font-bold text-xl mb-4">Commentaires</h2>

        <div class="flex-col space-y-4">
            @forelse (article->comments as comment)
<divclass="flexbg-whiterounded-mdshadowp-4space-x-4">
<divclass="flexjustify-startitems-starth-full"><x-avatarclass="h-10w-10":user="comment)
                <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                    <div class="flex justify-start items-start h-full">
                        <x-avatar class="h-10 w-10" :user="comment)
<divclass="flexbg-whiterounded-mdshadowp-4space-x-4">
<divclass="flexjustify-startitems-starth-full"><x-avatarclass="h-10w-10":user="comment->user" />
                    </div>
                    <div class="flex flex-col justify-center">
                        <div class="text-gray-700">
                            {{ comment->user->name }}
                        </div>
                        <div class="text-gray-500">
                            {{ comment->created_at->diffForHumans() }}
                        </div>
                        <div class="text-gray-700">
                            {{ $comment->body }}
                        </div>
                    </div>
```

```
                </div>
            @empty
                <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                    Aucun commentaire pour l'instant
                </div>
            @endforelse
        </div>
</x-guest-layout>
```

Si vous visitez la page d'un article, vous devriez voir les commentaires en bas de la page.

Routes pour les commentaires

Nous allons maintenant ajouter les routes pour les commentaires, après la route `front.articles.show` :

```php
// ... autres routes

// Détail d'un article
Route::get('/articles/{id}', [ArticleController::class, 'show'])->name('front.articles.show');

// Gestion des commentaires, uniquement pour les utilisateurs authentifiés
Route::middleware('auth')->group(function () {
    // Ajout d'un commentaire
    Route::post('/articles/{article}/comments', [ArticleController::class, 'addComment'])
->name('front.articles.comments.add');
    // Suppression d'un commentaire
    Route::delete('/articles/{article}/comments/{comment}', [ArticleController::class,
'deleteComment'])->name('front.articles.comments.delete');
});

// ... autres routes
```

Nous avons donc ajouté deux routes :

- `front.articles.comments.add` pour ajouter un commentaire
- `front.articles.comments.delete` pour supprimer un commentaire

Méthodes addComment et deleteComment

Nous allons maintenant ajouter les méthodes `addComment` et `deleteComment` au contrôleur `ArticleController`.

Nous allons commencer par la méthode `addComment` :

```php
public function addComment(Request request,Articlerequest, Article request,Articlearticle)
{
    // On vérifie que l'utilisateur est authentifié
    request->validate(['body'⇒'required|string|max:2000',]);//Oncre´elecommentairerequest->validate([
        'body' ⇒ 'required|string|max:2000',
    ]);

    // On crée le commentaire
    request->validate(['body'⇒'required|string|max:2000',]);//Oncre´elecommentairecomment = article->comments()->make();//Onremplitlesdonne´esarticle→comments()→make();
    // On remplit les données
    article->comments()->make();//Onremplitlesdonne´escomment→body = request->input('body');request→input('body');
    request->input('body');comment→user_id = Auth::user()→id;
    // On sauvegarde le commentaire
    $comment→save();

    // On redirige vers la page de l'article
    return redirect()→back();
}
```

Nous allons ensuite ajouter la méthode `deleteComment` :

```php
public function deleteComment(Article article,Commentarticle, Comment article,Commentcomment)
{
    // On vérifie que l'utilisateur à le droit de supprimer le commentaire
    Gate::authorize('delete', comment);//Onsupprimelecommentairecomment);

    // On supprime le commentaire
    comment);//Onsupprimelecommentairecomment→delete();

    // On redirige vers la page de l'article
    return redirect()→back();
}
```

Autorisation

Nous allons maintenant ajouter une autorisation pour la suppression d'un commentaire.

Nous allons créer un fichier `CommentPolicy.php` à l'aide de la commande suivante :

```bash
php artisan make:policy CommentPolicy
```

Nous allons ensuite modifier le fichier `app/Policies/CommentPolicy.php` pour ajouter la méthode `delete` :

```php
<?php

namespace App\Policies;

use App\Models\Comment;
use App\Models\User;
use Illuminate\Auth\Access\HandlesAuthorization;

class CommentPolicy
{
    use HandlesAuthorization;

    /**
     * Create a new policy instance.
     */
    public function __construct()
    {
    }

    public function delete(User user,Commentuser, Comment user,Commentcomment)
    {
        // Seul l'administateur ou le créateur du commentaire peut supprimer un commentaire
        return \App\Models\Role::ADMIN === user->role->name||user→role→name ||
user->role->name||user→id === $comment→user_id;
    }
}
```

On vérifie que l'utilisateur est soit l'administrateur, soit qu'il est le créateur du commentaire en comparant les identifiants.

Formulaire de création de commentaire

Nous allons maintenant ajouter un formulaire pour créer un commentaire et ajouter le bouton de suppression d'un commentaire visible uniquement par l'administrateur et le créateur du commentaire.

Nous allons modifier la vue `front.articles.show` pour ajouter le formulaire et le bouton de suppression :

XML

```
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4 capitalize">{{ article->title }}</h1>

    <div class="mb-4 text-xs text-gray-500">
        {{ article->published_at?->diffForHumans() }}
    </div>

    <div class="flex items-center justify-center">
        <img src="{{ asset('storage/' . article->img_path) }}" alt="illustration de l'article"
            class="rounded shadow aspect-auto object-cover object-center" />
    </div>

    <div class="mt-4">{!! \nl2br(e(article->body)) !!}</div>

    <div class="flex mt-8">
        <x-avatar class="h-20 w-20" :user="article->user" />
        <div class="ml-4 flex flex-col justify-center">
            <div class="text-gray-700">{{ $article->user->name }}</div>
            <div class="text-gray-500">{{ $article->user->email }}</div>
        </div>
    </div>

    <div class="mt-8 flex items-center justify-center">
        <a href="{{ route('front.articles.index') }}" class="font-bold bg-white text-gray-700 px-4
py-2 rounded shadow">
            Retour à la liste des articles
        </a>
    </div>

    <div class="mt-8">
        <h2 class="font-bold text-xl mb-4">Commentaires</h2>

        <div class="flex-col space-y-4">
            @forelse (article->comments as comment)
<divclass="flexbg-whiterounded-mdshadowp-4space-x-4">
<divclass="flexjustify-startitems-starth-full"><x-avatarclass="h-10w-10":user="comment)
                <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                    <div class="flex justify-start items-start h-full">
                        <x-avatar class="h-10 w-10" :user="comment)
<divclass="flexbg-whiterounded-mdshadowp-4space-x-4">
<divclass="flexjustify-startitems-starth-full"><x-avatarclass="h-10w-10":user="comment->user" />
                    </div>
                    <div class="flex flex-col justify-center w-full space-y-4">
                        <div class="flex justify-between">
                            <div class="flex space-x-4 items-center justify-center">
                                <div class="flex flex-col justify-center">
                                    <div class="text-gray-700">{{ comment->user->name }}</div>
                                    <div class="text-gray-500 text-sm">
                                        {{ comment->created_at->diffForHumans() }}
                                    </div>
                                </div>
                            </div>
                            <div class="flex justify-center">
                                @can('delete', comment)
                                    <button x-data="{ id: {{ $comment->id }} }"
                                        x-on:click.prevent="window.selected = id; dispatch('open-
modal', 'confirm-comment-deletion');"
                                        type="submit" class="font-bold bg-white text-gray-700 px-4
py-2 rounded shadow">
```

```
                                    <x-heroicon-o-trash class="h-5 w-5" />
                                </button>
                            @endcan
                        </div>
                    </div>
                    <div class="flex flex-col justify-center w-full text-gray-700">
                        <p class="border bg-gray-100 rounded-md p-4">
                            {{ comment->body }}
                        </p>
                    </div>
                </div>
            </div>
        @empty
            <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                Aucun commentaire pour l'instant
            </div>
        @endforelse @auth
        <form action="{{ route('front.articles.comments.add', article) }}" method="POST"
            class="flex bg-white rounded-md shadow p-4">
            @csrf
            <div class="flex justify-start items-start h-full mr-4">
                <x-avatar class="h-10 w-10" :user="auth()->user()" />
            </div>
            <div class="flex flex-col justify-center w-full">
                <div class="text-gray-700">{{ auth()->user()->name }}</div>
                <div class="text-gray-500 text-sm">{{ auth()->user()->email }}</div>
                <div class="text-gray-700">
                    <textarea name="body" id="body" rows="4" placeholder="Votre commentaire"
                        class="w-full rounded-md shadow-sm border-gray-300 bg-gray-100
focus:border-indigo-300 focus:ring focus:ring-indigo-200 focus:ring-opacity-50 mt-4"></textarea>
                </div>
                <div class="text-gray-700 mt-2 flex justify-end">
                    <button type="submit" class="font-bold bg-white text-gray-700 px-4 py-2
rounded shadow">
                        Ajouter un commentaire
                    </button>
                </div>
            </div>
        </form>
    @else
        <div class="flex bg-white rounded-md shadow p-4 text-gray-700 justify-between items-
center">
            <span> Vous devez être connecté pour ajouter un commentaire </span>
            <a href="{{ route('login') }}" class="font-bold bg-white text-gray-700 px-4 py-2
rounded shadow-md">Se
                connecter</a>
        </div>
    @endauth
</div>
<x-modal name="confirm-comment-deletion" focusable>
    <form method="post"
        onsubmit="event.target.action= '/articles/{{ article->id }}/comments/' +
window.selected"
        class="p-6">
        @csrf @method('DELETE')

        <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
            Êtes-vous sûr de vouloir supprimer ce commentaire ?
        </h2>
```

```
            <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
                Cette action est irréversible. Toutes les données seront supprimées.
            </p>

            <div class="mt-6 flex justify-end">
                <x-secondary-button x-on:click="dispatch('close')">
                    Annuler
                </x-secondary-button>

                <x-danger-button class="ml-3" type="submit">
                    Supprimer
                </x-danger-button>
            </div>
        </form>
    </x-modal>
</div>
</x-guest-layout>
```

Nous avons maintenant un formulaire pour ajouter un commentaire, et un bouton pour supprimer un commentaire. Nous avons également ajouté un modal pour confirmer la suppression d'un commentaire.

Vous pouvez essayer d'écrire un commentaire, puis de le supprimer pour voir si tout fonctionne correctement. Si vous avez des erreurs, vérifiez que vous avez bien suivi les étapes précédentes.

Failles XSS

Les failles XSS sont des failles de sécurité qui permettent à un attaquant d'exécuter du code JavaScript dans le navigateur d'un utilisateur. Cela peut être utilisé pour voler des informations confidentielles, des cookies, ou pour envoyer des données à un serveur malveillant.

Pour faire cela, un attaquant peut envoyer un commentaire avec du code JavaScript. Par exemple, si un utilisateur peut ajouter un commentaire, il peut envoyer le commentaire suivant :

```XML
<script>
  alert("Hello world");
</script>
```

Lorsque le navigateur de l'utilisateur affichera le commentaire, il exécutera le code JavaScript. Cela peut être utilisé pour voler des informations confidentielles, des cookies, ou pour envoyer des données à un serveur malveillant.

Pour éviter cela, nous utilisons depuis le début du cours les doubles moustaches ou la fonction helper `e()` de Laravel pour afficher des données. Cela permet d'échapper automatiquement les données pour éviter les failles XSS.

Vous pouvez aussi supprimer les balises HTML avec la fonction `strip_tags` de PHP :

```PHP
body=striptags(body = strip_tags(body=striptags(request->body);
```

Faites toujours attention aux données que vous pouvez recevoir de l'utilisateur. Si vous utilisez une librairie tierce, vérifiez que celle-ci n'est pas vulnérable aux failles XSS.

Afficher le nombre de commentaires

Nous allons maintenant afficher le nombre de commentaires sur l'article. Pour cela, nous allons devoir modifier les méthodes `index` de `HomepageController` et `ArticlesController` pour récupérer le nombre de commentaires.

Commençons par modifier la méthode `index` de `HomepageController` :

```php
public function index()
{
    articles=Article::query()->where('publishedat','<',now())->withCount('comments')
->orderByDesc('publishedat')->take(4)->get();returnview('homepage.index',['articles'⇒articles =
Article::query()
        →where('published_at', '<', now())
        →withCount('comments')
        →orderByDesc('published_at')
        →take(4)
        →get()
    ;

    return view('homepage.index', [
        'articles' ⇒ articles=Article::query()->where('publishedat','<',now())
->withCount('comments')->orderByDesc('publishedat')->take(4)->get();returnview('homepage.index',
['articles'⇒articles,
    ]);
}
```

Il faut aussi retirer les liens de paginations de la vue `homepage.index` et ajouter un lien vers tous les articles :

```php
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4">Liste des articles</h1>
    <ul class="grid sm:grid-cols-2 lg:grid-cols-3 2xl:grid-cols-4 gap-8">
        @foreach (articlesasarticles as articlesasarticle)
            <li>
                <x-article-card :article="$article" />
            </li>
        @endforeach
    </ul>

    <div class="mt-8 flex items-center justify-center">
        <a href="{{ route('front.articles.index') }}" class="font-bold bg-white text-gray-700 px-4
py-2 rounded shadow">
            Tous les articles
        </a>
    </div>
</x-guest-layout>
```

Nous avons ajouté la méthode `withCount` pour récupérer le nombre de commentaires. Cela va ajouter une colonne `comments_count` à chaque article.

Nous allons maintenant modifier la méthode `index` de `ArticlesController` :

```php
public function index(Request $request)
{
    $articles = Article::query()
        ->where('published_at', '<', now())
        ->when($request->query('search'), function ($query) use ($request) {
            $query->where('body', 'LIKE', '%'.$request->query('search').'%')
                ->orWhere('title', 'LIKE', '%'.$request->query('search').'%')
                ->orWhereHas('user', function ($query) use ($request) {
                    $query->where('name', 'LIKE', '%'.$request->query('search').'%');
                })
            ;
        })
        ->withCount('comments')
        ->orderByDesc('published_at')
        ->paginate(12)
    ;

    return view('front.articles.index', [
        'articles' => $articles,
    ]);
}
```

Même chose ici, nous avons ajouté la méthode `withCount` pour récupérer le nombre de commentaires.

Nous allons maintenant modifier le component `article-card` pour afficher le nombre de commentaires :

```php
<a class="flex flex-col h-full space-y-4 bg-white rounded-md shadow-md p-5 w-full hover:shadow-lg
hover:scale-105 transition"
    href="{{ route('front.articles.show', article) }}">
    <img src="{{ Storage::url(article->img_path) }}" alt="illustration de l'article" />
    <div class="uppercase font-bold text-gray-800">{{ article->title }}</div>
    <div class="flex-grow text-gray-700 text-sm text-justify">
        {{ Str::limit(article->body, 120) }}
    </div>
    <div class="flex justify-between items-center">
        <div class="text-sm text-gray-500">
            {{ article->published_at->diffForHumans() }}
        </div>
        <div class="flex items-center space-x-2">
            <x-heroicon-o-chat-bubble-bottom-center-text class="h-5 w-5 text-gray-500" />
            <div class="text-sm text-gray-500">{{ article->comments_count }}</div>
        </div>
    </div>
</a>
```

Nous avons ajouté une icône et le nombre de commentaires en bas à droite de la carte.

Vous pouvez essayer d'ajouter un commentaire, puis de voir si le nombre de commentaires est bien actualisé.

Profil public des utilisateurs

Nous allons à présent ajouter un profil public pour les utilisateurs. Nous allons afficher les articles publiés par l'utilisateur, et les commentaires qu'il a écrits.

Créer la route

Nous allons commencer par créer la route pour afficher le profil public d'un utilisateur. Placez cette route dans le fichier `routes/web.php`, juste en dessous des routes de gestion du profil :

```php
// Gestion du profil utilisateur
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
    Route::patch('/profile/avatar', [ProfileController::class, 'updateAvatar'])
->name('profile.avatar.update');
});

// Détail d'un profil utilisateur
Route::get('/profile/{user}', [ProfileController::class, 'show'])->name('profile.show');
```

Controller

Nous allons ajouter une méthode `show` au début du controller `ProfileController` :

```php
public function show(User $user): View
{
    // Les articles publiés par l'utilisateur
    $articles = $user
        ->articles()
        ->where('published_at', '<', now())
        ->withCount('comments')
        ->orderByDesc('published_at')
        ->get()
    ;

    // Les commentaires de l'utilisateur triés par date de création
    $comments = $user
        ->comments()
        ->orderByDesc('created_at')
        ->get()
    ;

    // On renvoie la vue avec les données
    return view('profile.show', [
        'user' => $user,
        'articles' => $articles,
        'comments' => $comments,
    ]);
}
```

Nous récupérons les articles publiés par l'utilisateur, et les commentaires qu'il a écrits.

Vue

Nous allons maintenant créer la vue `profile.show` :

XML

```
<x-guest-layout>
  <div class="flex w-full">
    <x-avatar class="h-20 w-20" :user="user" />
    <div class="ml-4 flex flex-col">
      <div class="text-gray-800 font-bold">{{ $user->name }}</div>
      <div class="text-gray-700 text-sm">{{ $user->email }}</div>
      <div class="text-gray-500 text-xs">{{ $user->role->name }}</div>
      <div class="text-gray-500 text-xs">
        Membre depuis {{ $user->created_at->diffForHumans() }}
      </div>
    </div>
  </div>
  <div class="mt-8">
    <h2 class="font-bold text-xl mb-4">Articles</h2>
    <ul class="grid sm:grid-cols-2 gap-8">
      @forelse (articles as article)<li><x-article-card:article="article)
      <li>
        <x-article-card :article="article)<li><x-article-card:article="article" />
      </li>
      @empty
      <div class="text-gray-700">Aucun article</div>
      @endforelse
    </ul>
  </div>

  <div class="mt-8">
    <h2 class="font-bold text-xl mb-4">Commentaires</h2>

    <div class="flex-col space-y-4">
      @forelse (commentsascomments as commentsascomment)
      <div class="flex bg-white rounded-md shadow p-4 space-x-4">
        <div class="flex justify-start items-start h-full">
          <x-avatar class="h-10 w-10" :user="comment->user" />
        </div>
        <div class="flex flex-col justify-center w-full space-y-4">
          <div class="flex justify-between">
            <div class="flex space-x-4 items-center justify-center">
              <div class="flex flex-col justify-center">
                <div class="text-gray-700">{{ $comment->user->name }}</div>
                <div class="text-gray-500 text-sm">
                  {{ $comment->created_at->diffForHumans() }}
                </div>
              </div>
            </div>
            <div class="flex justify-center">
              @can('delete', comment)
              <button
                x-data="{ id: {{ comment->id }} }"
                x-on:click.prevent="window.selected = id; dispatch('open-modal', 'confirm-comment-
deletion');"
                type="submit"
                class="font-bold bg-white text-gray-700 px-4 py-2 rounded shadow"
              >
                <x-heroicon-o-trash class="h-5 w-5" />
              </button>
              @endcan
            </div>
          </div>
          <div class="flex flex-col justify-center w-full text-gray-700">
```

```
            <p class="border bg-gray-100 rounded-md p-4">
              {{ comment->body }}
            </p>
          </div>
        </div>
      </div>
      @empty
      <div class="flex bg-white rounded-md shadow p-4 space-x-4">
        Aucun commentaire pour l'instant
      </div>
      @endforelse
    </div>
    <x-modal name="confirm-comment-deletion" focusable>
      <form
        method="post"
        onsubmit="event.target.action= '/articles/{{ article->id ?? 1 }}/comments/' +
window.selected"
        class="p-6"
      >
        @csrf @method('DELETE')

        <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
          Êtes-vous sûr de vouloir supprimer ce commentaire ?
        </h2>

        <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
          Cette action est irréversible. Toutes les données seront supprimées.
        </p>

        <div class="mt-6 flex justify-end">
          <x-secondary-button x-on:click="$dispatch('close')">
            Annuler
          </x-secondary-button>

          <x-danger-button class="ml-3" type="submit">
            Supprimer
          </x-danger-button>
        </div>
      </form>
    </x-modal>
  </div>
</x-guest-layout>
```

Nous pouvons maintenant ajouter des liens vers le profil de l'utilisateur dans les articles et les commentaires dans la vue `article.show` :

XML

```
<x-guest-layout>
    <h1 class="font-bold text-xl mb-4 capitalize">{{ article->title }}</h1>

    <div class="mb-4 text-xs text-gray-500">
        {{ article->published_at?->diffForHumans() }}
    </div>

    <div class="flex items-center justify-center">
        <img src="{{ asset('storage/' . article->img_path) }}" alt="illustration de l'article"
            class="rounded shadow aspect-auto object-cover object-center" />
    </div>

    <div class="mt-4">{!! \nl2br(e(article->body)) !!}</div>

    <a class="flex mt-8 hover:-translate-y-1 transition
    " href="{{ route('profile.show', article->user) }}">
        <x-avatar class="h-20 w-20" :user="article->user" />
        <div class="ml-4 flex flex-col justify-center">
            <div class="text-gray-700">{{ article->user->name }}</div>
            <div class="text-gray-500">{{ article->user->email }}</div>
        </div>
    </a>

    <div class="mt-8 flex items-center justify-center">
        <a href="{{ route('front.articles.index') }}" class="font-bold bg-white text-gray-700 px-4
py-2 rounded shadow">
            Retour à la liste des articles
        </a>
    </div>

    <div class="mt-8">
        <h2 class="font-bold text-xl mb-4">Commentaires</h2>

        <div class="flex-col space-y-4">
            @forelse (article->commentsasarticle->comments as article->commentsascomment)
                <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                    <a class="flex justify-start items-start h-full" href="{{
route('profile.show', comment->user) }}">
                        <x-avatar class="h-10 w-10" :user="comment->user" />
                    </a>
                    <div class="flex flex-col justify-center w-full space-y-4">
                        <div class="flex justify-between">
                            <div class="flex space-x-4 items-center justify-center">
                                <div class="flex flex-col justify-center">
                                    <a class="text-gray-700" href="{{ route('profile.show',
comment->user) }}">
                                        {{ comment->user->name }}
                                    </a>
                                    <div class="text-gray-500 text-sm">
                                        {{ comment->created_at->diffForHumans() }}
                                    </div>
                                </div>
                            </div>
                            <div class="flex justify-center">
                                @can('delete', comment)
                                    <button x-data="{ id: {{ comment->id }} }"
                                        x-on:click.prevent="window.selected = id; dispatch('open-
modal', 'confirm-comment-deletion');"
                                        type="submit" class="font-bold bg-white text-gray-700 px-4
```

```
py-2 rounded shadow">
                                        <x-heroicon-o-trash class="h-5 w-5" />
                                    </button>
                                @endcan
                            </div>
                        </div>
                        <div class="flex flex-col justify-center w-full text-gray-700">
                            <p class="border bg-gray-100 rounded-md p-4">
                                {{ comment->body }}
                            </p>
                        </div>
                    </div>
                </div>
            @empty
                <div class="flex bg-white rounded-md shadow p-4 space-x-4">
                    Aucun commentaire pour l'instant
                </div>
            @endforelse @auth
            <form action="{{ route('front.articles.comments.add', article) }}" method="POST"
                class="flex bg-white rounded-md shadow p-4">
                @csrf
                <div class="flex justify-start items-start h-full mr-4">
                    <x-avatar class="h-10 w-10" :user="auth()->user()" />
                </div>
                <div class="flex flex-col justify-center w-full">
                    <div class="text-gray-700">{{ auth()->user()->name }}</div>
                    <div class="text-gray-500 text-sm">{{ auth()->user()->email }}</div>
                    <div class="text-gray-700">
                        <textarea name="body" id="body" rows="4" placeholder="Votre commentaire"
                            class="w-full rounded-md shadow-sm border-gray-300 bg-gray-100
focus:border-indigo-300 focus:ring focus:ring-indigo-200 focus:ring-opacity-50 mt-4"></textarea>
                    </div>
                    <div class="text-gray-700 mt-2 flex justify-end">
                        <button type="submit" class="font-bold bg-white text-gray-700 px-4 py-2
rounded shadow">
                            Ajouter un commentaire
                        </button>
                    </div>
                </div>
            </form>
        @else
            <div class="flex bg-white rounded-md shadow p-4 text-gray-700 justify-between items-
center">
                <span> Vous devez être connecté pour ajouter un commentaire </span>
                <a href="{{ route('login') }}" class="font-bold bg-white text-gray-700 px-4 py-2
rounded shadow-md">Se
                    connecter</a>
            </div>
        @endauth
    </div>
    <x-modal name="confirm-comment-deletion" focusable>
        <form method="post"
            onsubmit="event.target.action= '/articles/{{ article->id }}/comments/' +
window.selected"
            class="p-6">
            @csrf @method('DELETE')

            <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
                Êtes-vous sûr de vouloir supprimer ce commentaire ?
            </h2>
```

```
            <p class="mt-1 text-sm text-gray-600 dark:text-gray-400">
                Cette action est irréversible. Toutes les données seront supprimées.
            </p>

            <div class="mt-6 flex justify-end">
                <x-secondary-button x-on:click="dispatch('close')">
                    Annuler
                </x-secondary-button>

                <x-danger-button class="ml-3" type="submit">
                    Supprimer
                </x-danger-button>
            </div>
        </form>
    </x-modal>
</div>
</x-guest-layout>
```

Les visiteurs peuvent maintenant accéder aux profils des utilisateurs en cliquant sur leurs noms ou avatars.

Exercices complémentaires

- Ajoutez la possibilité de liker un article pour un utilisateur connecté
- Ajoutez la possibilité de suivre un utilisateur (Follow)
- Ajouter une gestion des personnes qu'on follow et la liste des personnes qui nous follow

Code source

Le code source de ce tutoriel est disponible sur GitHub : https://github.com/opmvpc/blog-24-laravel11

1.16 Comment installer un projet Laravel existant sur un nouvel ordinateur

Ce chapitre vous guidera à travers les étapes nécessaires pour installer un projet Laravel existant sur un nouvel ordinateur. Nous supposerons que vous avez déjà installé les outils requis et que vous utilisez un serveur de développement local comme Laragon ou Laravel Valet/Herd.

Prérequis

Avant de commencer, assurez-vous que les éléments suivants sont installés sur votre ordinateur :

- **PHP** : Le langage de programmation sur lequel Laravel est construit, avec les extensions requises.
- **Composer** : Un gestionnaire de dépendances pour PHP, utilisé pour installer les packages nécessaires.
- **Node.js et npm** : Node.js est un environnement d'exécution JavaScript, et npm est son gestionnaire de packages. Ils sont utilisés pour compiler les assets front-end de Laravel.
- **Un serveur de développement local** : Outils comme **Laragon** ou **Laravel Valet/Herd** facilitent le développement local en gérant les configurations de serveur.

Étapes d'installation

1. Cloner le dépôt du projet Laravel

La première étape consiste à obtenir une copie du code source du projet. Vous pouvez le faire en clonant le dépôt Git du projet :

```bash
git clone https://github.com/nom-utilisateur/nom-du-projet.git
```

Cette action télécharge le code source du projet sur votre machine locale, vous permettant de travailler avec le même code que celui du dépôt distant.

2. Naviguer vers le répertoire du projet

Après avoir cloné le dépôt, déplacez-vous dans le répertoire du projet :

```bash
cd nom-du-projet
```

Cela vous positionne dans le dossier du projet où vous exécuterez les commandes suivantes.

3. Installer les dépendances PHP avec Composer

Installez les packages PHP nécessaires en exécutant :

```bash
composer install
```

Cette commande lit le fichier `composer.json` et installe toutes les bibliothèques et dépendances PHP dont le projet a besoin pour fonctionner.

4. Copier le fichier `.env.example` en `.env`

Le fichier `.env` contient les variables d'environnement spécifiques à votre application, comme les informations de base de données. Créez-en une copie à partir de l'exemple fourni :

```bash
cp .env.example .env
```

Cette action crée un fichier `.env` que vous pourrez personnaliser pour votre environnement local.

5. Générer une nouvelle clé d'application

Générez une clé unique pour votre application Laravel :

```bash
php artisan key:generate
```

La clé d'application est utilisée pour sécuriser les données chiffrées et les sessions. Cette commande met à jour la variable `APP_KEY` dans votre fichier `.env`.

6. Configurer les paramètres de base de données

Ouvrez le fichier `.env` avec un éditeur de texte et configurez les informations de connexion à votre base de données :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=nom_de_votre_base_de_donnees
DB_USERNAME=votre_nom_d_utilisateur
DB_PASSWORD=votre_mot_de_passe
```

Ces paramètres permettent à Laravel de se connecter à votre base de données pour effectuer des opérations comme les migrations et les requêtes.

7. Migrer la base de données

Créez les tables nécessaires dans votre base de données en exécutant :

```bash
php artisan migrate
```

Les migrations sont des scripts qui créent ou modifient la structure de la base de données. Si la base de données spécifiée n'existe pas, la commande vous proposera de la créer.

8. Peupler la base de données avec des seeders (si disponibles)

Si le projet contient des seeders pour insérer des données initiales, exécutez :

```bash
php artisan db:seed
```

Les seeders remplissent la base de données avec des données fictives ou initiales, utiles pour les tests ou pour démarrer l'application avec des données par défaut.

9. Créer un lien symbolique pour le stockage (si nécessaire)

Si le projet utilise le stockage de fichiers (par exemple, pour les uploads), créez un lien symbolique vers le répertoire public :

```bash
php artisan storage:link
```

Cette commande crée un lien symbolique entre `storage/app/public` et `public/storage`, permettant un accès web aux fichiers stockés.

10. Installer les dépendances Node.js

Installez les packages front-end nécessaires avec npm :

```bash
npm install
```

Cette commande lit le fichier `package.json` et installe toutes les dépendances JavaScript requises pour le front-end de l'application.

11. Compiler les assets front-end

Compilez les fichiers CSS et JavaScript du projet :

- Pour le développement (avec source maps et sans minification) :

```bash
npm run dev
```

- Pour la production (optimisé et minifié) :

```bash
npm run build
```

La compilation des assets génère les fichiers optimisés nécessaires pour le front-end de l'application, en combinant et en minifiant le code pour de meilleures performances.

12. Démarrer le serveur local

Avec votre serveur de développement local (Laragon, Laravel Valet/Herd), votre application devrait être automatiquement reconnue et servie. Assurez-vous que votre serveur est en cours d'exécution et accédez à l'application via votre navigateur web à l'adresse appropriée, par exemple :

```bash
http://nom-du-projet.test
```

Cette étape vous permet de visualiser et de tester votre application Laravel dans un environnement local.

Conclusion

Vous avez maintenant installé avec succès un projet Laravel existant sur votre nouvel ordinateur. Ce processus vous permet de configurer rapidement un environnement de développement local pour travailler sur le projet. N'oubliez pas de consulter la documentation du projet pour d'éventuelles instructions supplémentaires spécifiques à celui-ci.

CHAPITRE 2

Vue.js



Le chapitre "Vue.js" de ce cours de Projet Web Multimédia explore un framework JavaScript moderne et populaire utilisé pour la création d'interfaces utilisateur interactives et réactives. Vue.js est conçu pour faciliter le développement de projets web complexes en offrant une approche déclarative et basée sur les composants.

Dans ce chapitre, vous découvrirez les principes fondamentaux de Vue.js, y compris son installation et sa configuration de base. Vous explorerez également les concepts clés tels que les composants, les directives, les liaisons de données et les transitions. Vous apprendrez comment utiliser Vue.js pour créer des applications web dynamiques et réactives, en exploitant ses fonctionnalités puissantes pour gérer l'état de l'application, manipuler la DOM et interagir avec les données.

Que vous soyez un développeur débutant ou expérimenté, ce chapitre vous guidera à travers les étapes nécessaires pour maîtriser Vue.js et créer des applications web modernes et performantes. Préparez-vous à explorer les fonctionnalités avancées de Vue.js, telles que Vue Router pour la gestion des routes, Vuex pour la gestion de l'état de l'application, et bien plus encore.



2.1 Introduction à Vue.js



https://vuejs.org/

Vue.js : Qu'est-ce que c'est ?

Vue.js est un framework JavaScript progressif utilisé pour construire des interfaces utilisateur interactives et réactives. Il est souvent comparé à d'autres frameworks tels que React et Angular, mais se distingue par sa simplicité et sa facilité d'apprentissage. Vue.js adopte une approche fondée sur les composants, ce qui permet de diviser l'interface utilisateur en petites parties réutilisables.

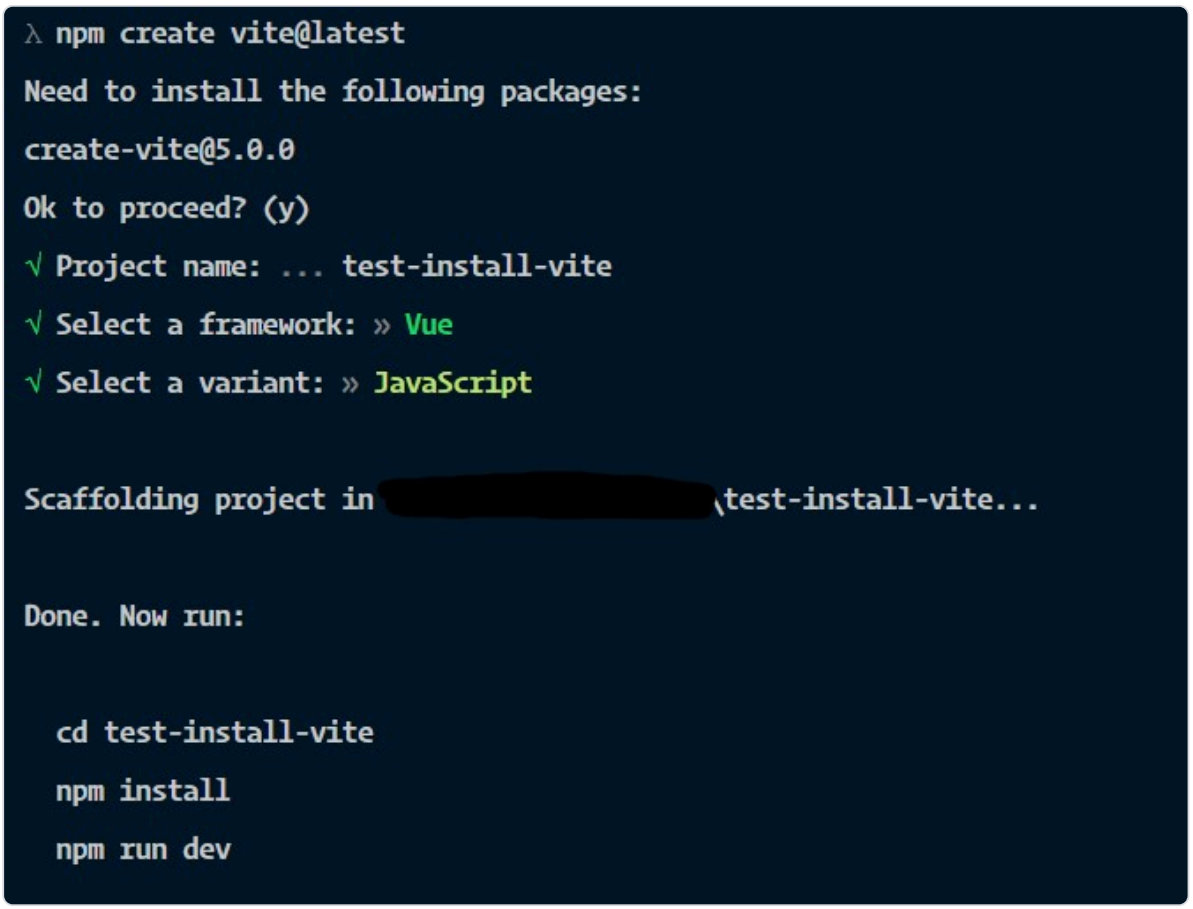
Principales caractéristiques de Vue.js

- **Réactivité** : Vue.js utilise un système de réactivité qui permet de lier automatiquement les données à l'interface utilisateur. Lorsque les données changent, l'interface utilisateur est mise à jour de manière réactive.
- **Composants** : Vue.js encourage l'utilisation de composants réutilisables, ce qui facilite la création d'interfaces utilisateur modulaires et maintenables.
- **Directives** : Les directives de Vue.js aident à manipuler le DOM de manière déclarative. Par exemple, la directive `v-if` permet de conditionner l'affichage d'un élément en fonction d'une expression.

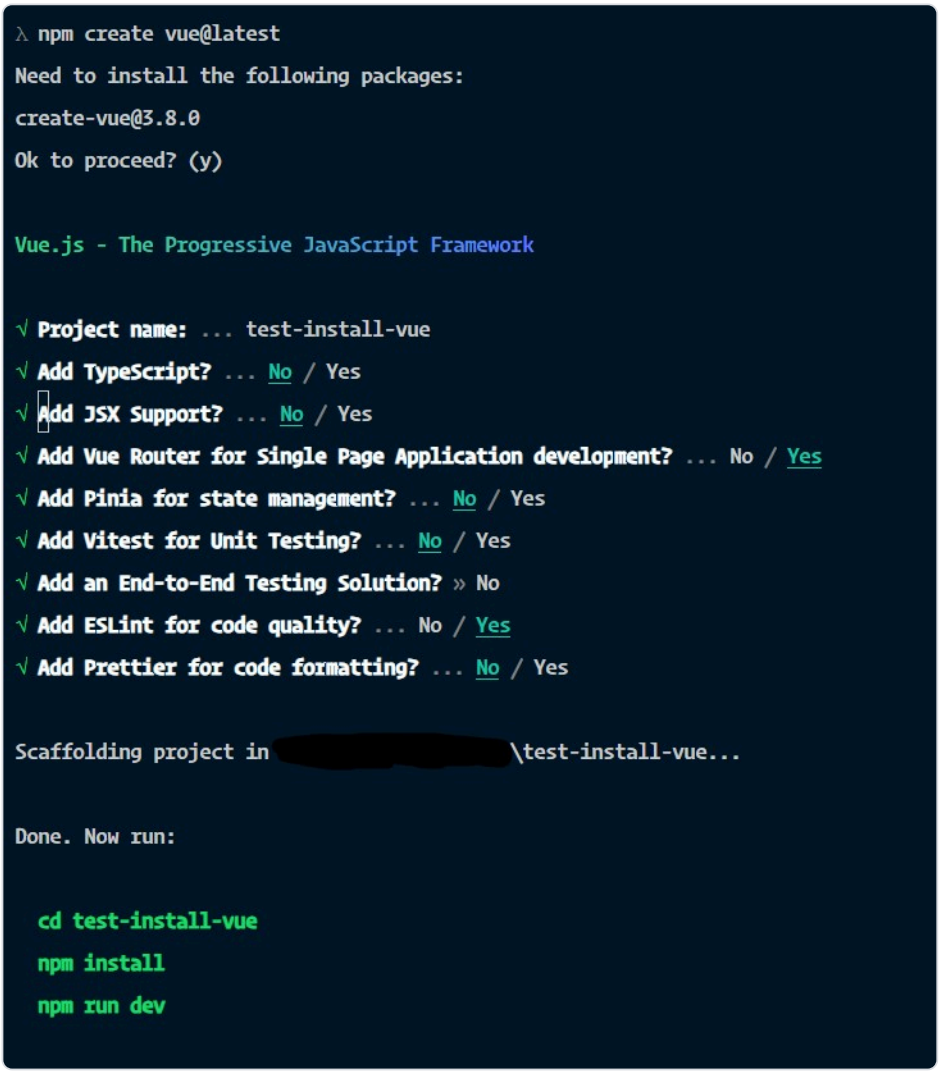
Avantages de l'utilisation de Vue.js

- **Facilité d'apprentissage** : Vue.js est réputé pour sa courbe d'apprentissage douce, ce qui en fait un choix populaire pour les débutants en développement web.
- **Performances** : Vue.js est conçu pour être rapide et efficace. Son système de réactivité optimisé permet d'obtenir des performances élevées.
- **Flexibilité** : Vue.js est très flexible et peut être utilisé pour créer des applications de toutes tailles. Il peut être intégré à d'autres bibliothèques ou frameworks existants.
- **Documentation complète** : Vue.js dispose d'une documentation complète et de nombreux exemples qui facilitent l'apprentissage et le développement avec le framework.

Installation avec Vite



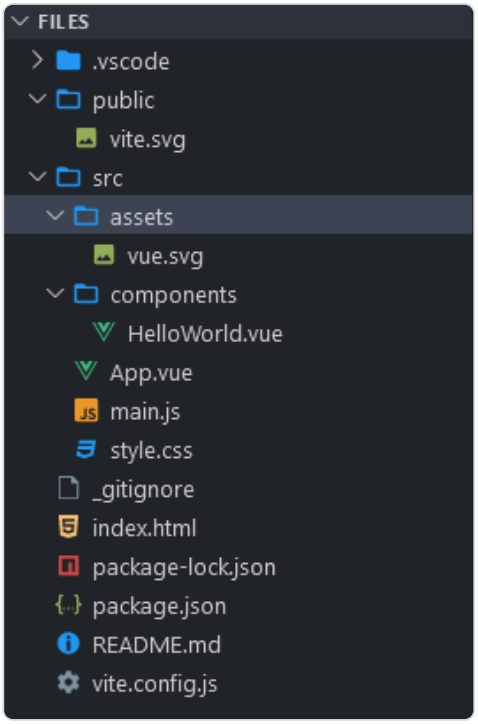
Installation avec Vue



Structure du projet

En explorant la structure du projet, vous verrez plusieurs fichiers et dossiers importants, tels que :

- `index.html` : Le point d'entrée de l'application.
- `src` : Le dossier contenant les fichiers source de l'application.
- `package.json` : Configuration du projet.



index.html

```xml
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + Vue</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

C'est un fichier HTML semblable à ce que vous connaissez. Il y a deux éléments essentiels à noter :

- La balise `<div id="app"></div>` est l'élément racine de l'application Vue.js. C'est là que l'application sera rendue.
- La balise `<script type="module" src="/src/main.js"></script>` charge le fichier JavaScript principal de l'application, `main.js`, en tant que module. Ce fichier est situé dans le dossier `/src` du projet. Le fichier `main.js` est responsable de l'initialisation de l'application Vue.js et de l'importation des composants nécessaires.

package.json

Le fichier `package.json` est un fichier de configuration utilisé dans les projets JavaScript, y compris les projets Vue.js et Vite.js. Il est généralement situé à la racine du projet et contient des informations sur le projet lui-même, ainsi que sur ses dépendances, scripts et autres métadonnées.

```JSON
{
  "name": "vite-vue-starter",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "vue": "^3.3.4"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.3",
    "vite": "^4.4.9"
  }
}
```

Voici à quoi sert le fichier `package.json` :

1. **Gestion des dépendances** : Le fichier `package.json` répertorie toutes les dépendances du projet, c'est-à-dire les bibliothèques et les modules externes utilisés dans le projet. Ces dépendances peuvent être des packages npm (Node Package Manager) téléchargés à partir du registre npm public ou des packages locaux spécifiques au projet. Le fichier `package.json` spécifie également les versions des dépendances utilisées, ce qui permet de garantir la reproductibilité du projet.

2. **Scripts de commande** : Le fichier `package.json` permet de définir des scripts de commande personnalisés qui peuvent être exécutés à partir de la ligne de commande. Ces scripts peuvent effectuer différentes tâches, telles que le lancement de l'application, la compilation des fichiers source, les tests unitaires, la création de builds de production, etc. Les scripts de commande peuvent être exécutés en utilisant l'outil `npm` ou `yarn`, en fonction de celui que vous utilisez dans votre projet.

3. **Informations sur le projet** : Le fichier `package.json` contient des informations sur le projet lui-même, telles que le nom du projet, la version, la description, l'auteur, la licence, etc. Ces informations fournissent une vue d'ensemble du projet et sont souvent utilisées pour générer automatiquement la documentation, les métadonnées des packages et d'autres informations.

4. **Configuration du projet** : Le fichier `package.json` permet également de spécifier des configurations spécifiques au projet. Par exemple, dans un projet Vue.js, vous pouvez définir des options de configuration pour le bundler, le préprocesseur CSS, le transpileur JavaScript, etc. Ces configurations peuvent être spécifiées dans la section `vue` du fichier `package.json` ou dans des fichiers de configuration séparés tels que `vite.config.js`.

Le dossier `src`

Le dossier `src` contient les fichiers sources de l'application, tels que le point d'entrée JavaScript, les composants, les styles et les ressources.

main.js

Le fichier `main.js` est le point d'entrée de l'application. C'est ici que l'application Vue.js est initialisée et montée sur l'élément racine de l'application. Vous pouvez également y importer des bibliothèques externes, configurer des plugins et définir des directives globales.

```javascript
import { createApp } from 'vue'
import './style.css'
import App from './App.vue'

createApp(App).mount('#app')
```

1. La première ligne `import { createApp } from 'vue'` importe la fonction `createApp` du package `vue`. Cela permet d'utiliser les fonctionnalités de Vue.js dans notre application.
2. La deuxième ligne `import './style.css'` importe le fichier `style.css`. Cela permet d'appliquer les styles définis dans ce fichier à notre application.
3. La troisième ligne `import App from './App.vue'` importe le composant `App` à partir du fichier `App.vue`. Ce composant représente la vue principale de notre application.
4. La dernière ligne `createApp(App).mount('#app')` crée une instance de l'application Vue en utilisant le composant `App` importé précédemment. Ensuite, cette instance est montée sur l'élément avec l'ID `#app` dans le fichier `index.html`. Cela signifie que notre application Vue sera rendue à l'intérieur de cet élément dans la page HTML.

App.vue

Le fichier `App.vue` est le composant principal de l'application et contient la structure de base de l'interface utilisateur.

```vue
<script setup>
import HelloWorld from './components/HelloWorld.vue'
</script>

<template>
  <div>
    <a href="https://vitejs.dev" target="_blank">
      <img src="/vite.svg" class="logo" alt="Vite logo" />
    </a>
    <a href="https://vuejs.org/" target="_blank">
      <img src="./assets/vue.svg" class="logo vue" alt="Vue logo" />
    </a>
  </div>
  <HelloWorld msg="Vite + Vue" />
</template>

<style scoped>
.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.vue:hover {
  filter: drop-shadow(0 0 2em #42b883aa);
}
</style>
```

Un fichier Vue se compose généralement de trois parties principales : `<template>`, `<script>`, et `<style>`. Chacune de ces parties joue un rôle spécifique dans la construction d'un composant Vue.

1. La section `<template>` : La section `<template>` est l'endroit où vous définissez la structure HTML de votre composant. Vous pouvez utiliser des balises HTML standard ainsi que des directives spécifiques à Vue pour lier des données et définir la logique de rendu. C'est ici que vous créez l'interface utilisateur de votre composant en utilisant des éléments HTML et des composants Vue.

2. La section `<script>` : La section `<script>` est l'endroit où vous définissez la logique JavaScript de votre composant. Vous pouvez y importer des dépendances, déclarer des variables, définir des méthodes, des propriétés et des hooks de cycle de vie, et interagir avec les données. Cette section est également utilisée pour importer et exporter des composants, configurer des plugins ou définir des mixins.

3. La section `<style>` : La section `<style>` est l'endroit où vous définissez les styles CSS de votre composant. Vous pouvez utiliser des sélecteurs CSS standard pour cibler les éléments de votre template et leur appliquer des styles spécifiques. La section `<style>` peut être déclarée de différentes manières, notamment avec l'attribut `scoped` pour limiter la portée des styles uniquement au composant en cours.

HelloWorld.vue

Dans le code du composant `HelloWorld.vue`, nous pouvons voir l'utilisation des composants et des props, ainsi que la notion de ref.

```vue
<script setup>
import { ref } from 'vue'

defineProps({
  msg: String,
})

const count = ref(0)
</script>

<template>
  <h1>{{ msg }}</h1>

  <div class="card">
    <button type="button" @click="count++">count is {{ count }}</button>
    <p>
      Edit
      <code>components/HelloWorld.vue</code> to test HMR
    </p>
  </div>

  <p>
    Check out
    <a href="https://vuejs.org/guide/quick-start.html#local" target="_blank"
      >create-vue</a
    >, the official Vue + Vite starter
  </p>
  <p>
    Install
    <a href="https://github.com/vuejs/language-tools" target="_blank">Volar</a>
    in your IDE for a better DX
  </p>
  <p class="read-the-docs">Click on the Vite and Vue logos to learn more</p>
</template>

<style scoped>
.read-the-docs {
  color: #888;
}
</style>
```

Les composants sont des éléments réutilisables dans une application Vue.js. Ils permettent de découper l'interface utilisateur en morceaux plus petits et autonomes, ce qui facilite la gestion et la réutilisation du code. Chaque composant peut avoir son propre template, sa logique et ses styles.

Les props sont des propriétés qui permettent de passer des données d'un composant parent à un composant enfant. Dans cet exemple, le composant `HelloWorld` reçoit une prop nommée `msg` de type `String`. Cette prop est utilisée dans le template pour afficher le contenu de la prop dans un élément `<h1>`. Les props permettent de rendre les composants plus flexibles et réutilisables en leur permettant de recevoir des données dynamiques de l'extérieur.

La notion de `ref` est utilisée pour créer une référence réactive à une valeur dans un composant Vue. Dans ce cas, la variable `count` est déclarée en tant que `ref(0)`, ce qui signifie qu'elle est réactive et peut être utilisée dans le template. Lorsque le bouton est cliqué, la valeur de `count` est incrémentée. Les refs sont utiles pour suivre les valeurs qui peuvent être modifiées et réagir aux changements de ces valeurs.

En résumé, les composants sont des éléments réutilisables qui permettent de découper l'interface utilisateur en morceaux plus petits et autonomes. Les props sont utilisées pour passer des données d'un composant parent à un composant enfant. La notion de `ref` permet de créer des références réactives à des valeurs dans un composant Vue. Ces concepts sont essentiels pour la construction d'applications Vue.js flexibles et réactives.

2.2 TailwindCSS

Introduction à TailwindCSS

TailwindCSS est un framework CSS qui a pris de l'ampleur grâce à son approche "utility-first". Contrairement aux frameworks traditionnels qui proposent des composants prédéfinis, Tailwind permet aux développeurs de construire rapidement des interfaces personnalisées sans quitter leur HTML.

Historique et philosophie

Tailwind a été créé par Adam Wathan et a été officiellement lancé en novembre 2017. L'idée était de fournir un ensemble de classes utilitaires atomiques qui peuvent être combinées pour créer n'importe quel design directement dans le balisage HTML. Cette approche vise à réduire le temps passé à passer d'un fichier CSS à un fichier HTML et à augmenter la cohérence du design sur toute l'interface.

Comparaison avec Bootstrap

Bootstrap est l'un des frameworks CSS les plus populaires et utilise une approche basée sur des composants prédéfinis. Voici quelques différences clés entre Bootstrap et TailwindCSS :

- **Approche de conception** : Bootstrap utilise des composants prédéfinis, tandis que Tailwind adopte une approche "utility-first" qui encourage la composition de classes utilitaires pour créer des styles uniques.
- **Personnalisation** : Avec Bootstrap, personnaliser des composants peut nécessiter de surcharger des styles, tandis que Tailwind est conçu pour être entièrement personnalisé via le fichier de configuration.

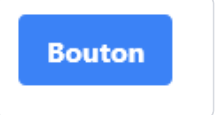
Exemples pratiques

Dans cette section, nous allons voir comment utiliser TailwindCSS pour créer des éléments d'interface courants.

Bouton

Voici un exemple de comment créer un bouton simple avec TailwindCSS :

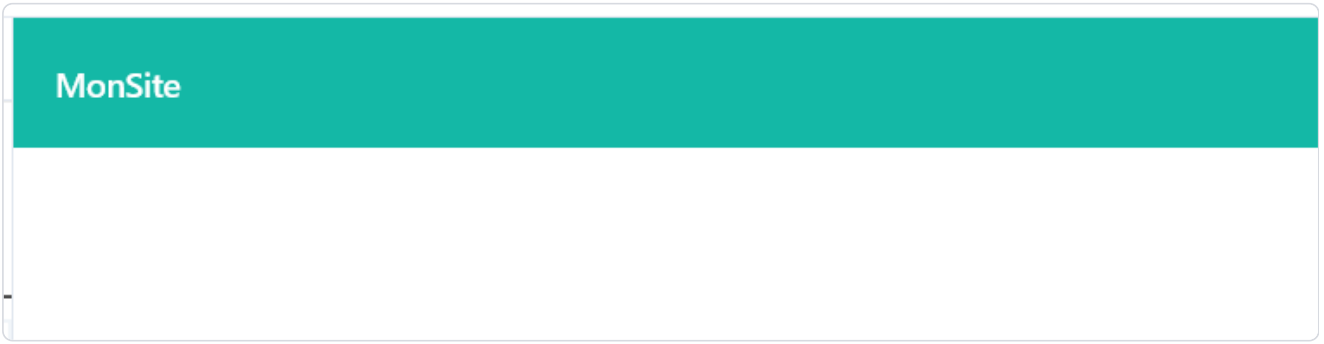
```xml
<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
  Bouton
</button>
```



Navbar

Pour une barre de navigation (navbar), TailwindCSS permet de construire quelque chose comme ceci :

```xml
<nav class="flex items-center justify-between flex-wrap bg-teal-500 p-6">
  <div class="flex items-center flex-shrink-0 text-white mr-6">
    <span class="font-semibold text-xl tracking-tight">MonSite</span>
  </div>
  <!-- Reste de la navbar -->
</nav>
```



Formulaire

Créer un formulaire simple avec TailwindCSS pourrait ressembler à cela :

Installation de TailwindCSS avec Vite

Pour intégrer TailwindCSS dans un projet Vite, suivez les étapes suivantes :

Installation des dépendances

Ouvrez votre terminal et exécutez :

```bash
npm install tailwindcss @tailwindcss/vite
```

- **Documentation d'installation :** https://tailwindcss.com/docs/installation/using-vite

Ajout de l'import de Tailwind dans le CSS

Dans votre fichier `./src/assets/main.css`, remplacez le contenu du fichier par les directives de Tailwind suivantes :

```css
@import "tailwindcss";
```

Avec ces étapes, vous aurez intégré TailwindCSS dans votre projet Vite et serez prêt à construire votre interface avec une flexibilité et une personnalisation accrues.

Intégration de Tailwind CSS dans l'application

Modification du fichier vite.config.js

1. Ouvrez le fichier `vite.config.js` situé à la racine du projet.
2. Importez le plugin Tailwind CSS en modifiant le fichier comme ceci :

```javascript
import { defineConfig } from 'vite'
// ...
import tailwindcss from '@tailwindcss/vite' // importez le plugin

export default defineConfig({
  // ...
  plugins: [
    tailwindcss(), // ajouté le plugin à la liste
  ],
})
```

Modification du fichier App.vue

1. Ouvrez le fichier `App.vue` situé dans le répertoire `src`.
2. Remplacez le contenu existant du fichier par le code suivant :

VUE

```
<script setup>
import { ref } from 'vue';

// État pour le menu de navigation
const isNavOpen = ref(false);
const toggleNav = () => {
  isNavOpen.value = !isNavOpen.value;
};
</script>

<template>
  <div class="flex flex-col h-full">
    <header>
      <!-- Barre de navigation fixe -->
      <nav class="bg-gray-800 text-white p-4 fixed w-full top-0 z-50">
        <div class="container mx-auto flex justify-between items-center">
          <a class="text-lg font-bold" href="#">Fixed navbar</a>
          <button
            class="md:hidden p-2"
            @click="toggleNav"
            aria-label="Toggle navigation"
          >
            <span class="material-icons">menu</span>
          </button>
          <div :class="{'hidden': !isNavOpen, 'flex': isNavOpen}" class="w-full md:flex md:w-
auto">
            <ul class="md:flex md:space-x-4">
              <li>
                <a class="block p-2 hover:bg-gray-700" href="#">Home</a>
              </li>
              <li>
                <a class="block p-2 hover:bg-gray-700" href="#">Link</a>
              </li>
              <li>
                <a class="block p-2 text-gray-500 cursor-not-allowed" aria-
disabled="true">Disabled</a>
              </li>
            </ul>
            <form class="flex mt-4 md:mt-0">
              <input
                class="form-input px-4 py-2 border rounded-lg md:mr-2"
                type="search"
                placeholder="Search"
                aria-label="Search"
              />
              <button class="bg-transparent text-white border border-white px-4 py-2 ml-2 rounded-
lg hover:bg-white hover:text-gray-800" type="submit">
                Search
              </button>
            </form>
          </div>
        </div>
      </nav>
    </header>

    <!-- Contenu principal de la page -->
    <main class="flex-shrink-0 pt-24 pb-5">
      <div class="container mx-auto px-4">
        <h1 class="text-3xl font-bold mt-5">Sticky footer with fixed navbar</h1>
```

```
        <p class="text-lg mt-4">
          Fixez un pied de page au bas de la fenêtre dans les navigateurs de bureau avec
          ce code HTML et CSS personnalisé. Une barre de navigation fixe a été ajoutée avec un
<code class="font-mono bg-gray-200 px-1 py-0.5 rounded">padding-top: 60px;</code>
          sur le <code class="font-mono bg-gray-200 px-1 py-0.5 rounded">main &gt;
.container</code>.
        </p>
      </div>
    </main>

    <!-- Pied de page -->
    <footer class="bg-gray-100 mt-auto py-3">
      <div class="container mx-auto px-4">
        <span class="text-gray-600">Place sticky footer content here.</span>
      </div>
    </footer>
  </div>
</template>

<style>
#app {
  height: 100vh;
}
</style>
```

Ce code crée une structure de base pour notre application en utilisant les classes CSS de Tailwind. Il comprend une barre de navigation (`nav`), une section principale (`main`) et un pied de page (`footer`).
Ressources :

https://tailwindcss.com/
https://play.tailwindcss.com/

2.3 Les Composants

Introduction aux Composants

Les composants constituent l'un des concepts fondamentaux de Vue.js. Ils permettent de découper l'interface utilisateur en morceaux réutilisables et modulaires. Un composant est essentiellement une instance de Vue avec son propre cycle de vie, ses données, ses méthodes et son rendu.

Définition des composants dans Vue.js

Dans Vue.js, un composant est une instance réutilisable qui encapsule une partie de l'interface utilisateur. Il peut être utilisé plusieurs fois dans une application, ce qui facilite la création d'interfaces complexes et la réutilisation de code. Les composants peuvent être imbriqués les uns dans les autres pour former une structure hiérarchique.

Avantages de l'utilisation des composants

L'utilisation de composants présente de nombreux avantages :

- **Réutilisabilité** : Les composants permettent de réutiliser du code, ce qui facilite le développement et la maintenance de l'application.

- **Modularité** : Les composants permettent de découper l'interface utilisateur en morceaux indépendants, ce qui facilite la compréhension et la gestion du code.
- **Encapsulation** : Les composants encapsulent leur propre état et comportement, ce qui évite les conflits et facilite la gestion de l'application.
- **Composabilité** : Les composants peuvent être combinés et imbriqués pour former des interfaces complexes, ce qui facilite la création d'interfaces riches et interactives.
- **Testabilité** : Les composants peuvent être testés de manière isolée, ce qui facilite la mise en place de tests unitaires.

Structure d'un composant Vue.js

Un composant Vue.js est généralement composé des éléments suivants :

- **Template** : Le template définit la structure et le rendu du composant. Il utilise la syntaxe HTML avec des ajouts spécifiques à Vue.js, tels que les directives et les expressions.
- **Script** : Le script contient la logique du composant, y compris les données, les méthodes et les hooks du cycle de vie. Il est écrit en JavaScript ou en TypeScript.
- **Style** : Le style contient les règles CSS qui s'appliquent au composant. Il peut être écrit en CSS classique, en préprocesseur CSS (comme Sass ou Less) ou en CSS-in-JS.

Un exemple de structure de composant Vue.js :

```vue
<script setup>
// ... code javascript
</script>

<template>
  <!-- Structure du composant (html) -->
</template>

<style>
/* Styles du composant (css) */
</style>
```

Vous trouverez aussi cette structure dans des exemples sur internet :

```vue
<template>
  <!-- Structure du composant -->
</template>

<script>
export default {
  // Données du composant
  data() {
    return {
      // ...
    };
  },
  // Méthodes du composant
  methods: {
    // ...
  },
  // Hooks du cycle de vie du composant
  created() {
    // ...
  },
  // ...
};
</script>

<style>
/* Styles du composant */
</style>
```

C'est l'ancienne façon d'écrire les composants Vue.js. Nous utiliserons la première notation dans ce cours.

Création du Composant Nav.vue

Maintenant que nous avons une compréhension de base des composants dans Vue.js, nous allons passer à la pratique en créant notre premier composant : Nav.vue. Ce composant sera responsable de l'affichage du code du navbar.

Voici les étapes à suivre pour créer le composant Nav.vue :

1. Créez un nouveau fichier `Nav.vue` dans le répertoire des composants de votre projet avec ce code:

   ```vue
   <script setup></script>

   <template></template>
   ```

2. Copiez le code du navbar de votre fichier `App.vue` et collez-le dans le template du composant `Nav.vue`. Le code ressemblera à ceci :

```vue
<template>
  <nav class="w-full bg-gray-900 fixed top-0">
    <!-- Code du navbar -->
  </nav>
</template>
```

1. Utilisez le composant `Nav.vue` dans le fichier `App.vue` en important le composant et en l'ajoutant à la structure du template. Remplacez le code du navbar par la balise `<Nav />` pour utiliser le composant.

```vue
<template>
  <div class="flex flex-col h-full">
    <header>
      <!-- Utilisation du composant Nav -->
      <Nav />
    </header>

    <!-- ... Le reste du code ... -->
  </div>
</template>
```

1. Importez le composant avec ce code dans la balise script du fichier `App.vue` :

```vue
<script setup>
import Nav from './components/Nav.vue'
</script>
```

Maintenant, vous avez créé avec succès le composant `Nav.vue` et l'avez utilisé dans le fichier `App.vue`. Vous pouvez personnaliser le composant `Nav.vue` selon vos besoins en ajoutant des classes, des styles ou des fonctionnalités supplémentaires.

Création du Composant Footer.vue

Maintenant que nous avons créé le composant `Nav.vue`, nous allons procéder de la même manière pour créer le composant `Footer.vue`. Ce composant sera responsable de l'affichage du code du footer.

Voici les étapes à suivre pour créer le composant `Footer.vue` :

1. Créez un nouveau fichier `Footer.vue` dans le répertoire des composants de votre projet.
2. Copiez le code du footer de votre fichier `App.vue` et collez-le dans le template du composant `Footer.vue`. Le code ressemblera à ceci :

```vue
<template>
  <footer class="mt-auto py-3 bg-gray-100">
    <!-- Code du footer -->
  </footer>
</template>
```

La balise script, comme la balise style, ne sont pas obligatoires.

1. Utilisez le composant `Footer.vue` dans le fichier `App.vue` en important le composant et en l'ajoutant à la structure du template. Remplacez le code du footer par la balise `<Footer />` pour utiliser le composant.

```vue
<template>
  <div class="flex flex-col h-full">
    <header>
      <!-- Utilisation du composant Nav -->
      <Nav />
    </header>

    <!-- ... Le reste du code ... -->

    <!-- Utilisation du composant Footer -->
    <Footer />
  </div>
</template>
```

1. Importez le composant avec ce code dans la balise script du fichier `App.vue` :

```vue
<script setup>
import Footer from './components/Footer.vue'
</script>
```

Maintenant, vous avez créé avec succès le composant `Footer.vue` et l'avez utilisé dans le fichier `App.vue`. Tout comme pour le composant `Nav.vue`, vous pouvez personnaliser le composant `Footer.vue` selon vos besoins en ajoutant des classes, des styles ou des fonctionnalités supplémentaires.

Les Slots

Dans Vue.js, les slots sont une fonctionnalité puissante qui permet de créer des composants flexibles et réutilisables. Les slots permettent d'insérer du contenu dynamique à l'intérieur d'un composant, ce qui le rend plus flexible et adaptable à différents cas d'utilisation.

Introduction à la notion de slot dans Vue.js

Un slot est une zone réservée à l'intérieur d'un composant où du contenu peut être inséré. Il permet de définir des points d'entrée pour le contenu dynamique afin de personnaliser l'apparence ou le comportement d'un composant. Les slots peuvent être utilisés pour insérer du contenu texte, des éléments HTML ou d'autres composants à l'intérieur d'un composant parent.

Voici un exemple simple d'utilisation d'un composant `PrimaryButton.vue` avec un slot :

PrimaryButton.vue :

```vue
<template>
  <button class="bg-blue-500 hover:bg-blue-600 text-white font-bold py-2 px-4 rounded">
    <slot></slot>
  </button>
</template>
```

Dans ce composant `PrimaryButton.vue`, nous avons un bouton stylisé avec une classe `primary-button`. À l'intérieur du bouton, nous avons utilisé un slot anonyme `<slot></slot>`. Cela signifie que tout contenu inséré entre les balises `<PrimaryButton>` lors de l'utilisation du composant sera affiché à l'intérieur du slot.

App.vue :

```vue
<script setup>
import PrimaryButton from './PrimaryButton.vue';
</script>

<template>
  <div>
    <PrimaryButton>
      Cliquez ici <!-- slot -->
    </PrimaryButton>
  </div>
</template>
```

Dans ce fichier `App.vue`, nous utilisons le composant `PrimaryButton` en l'entourant de balises `<PrimaryButton>`. À l'intérieur des balises, nous avons inséré le texte "Cliquez ici". Ce contenu sera inséré dans le slot anonyme du composant `PrimaryButton`, qui sera affiché à l'intérieur du bouton stylisé.

Lorsque le composant `App.vue` est rendu, il affiche le bouton avec le texte "Cliquez ici" à l'intérieur, en utilisant le style défini dans le composant `PrimaryButton.vue`.

Création du Composant Layout.vue avec un slot

Pour illustrer l'utilisation des slots, nous allons créer un composant `Layout.vue` qui servira de mise en page de base pour notre application. Ce composant aura un slot nommé `content` où le contenu spécifique à chaque page pourra être inséré.

Voici les étapes pour créer le composant `Layout.vue` :

1. Créez un nouveau fichier `Layout.vue` dans le répertoire des composants de votre projet.
2. Dans le template du composant `Layout.vue`, définissez une zone réservée pour le contenu en utilisant la balise `<slot>`. Vous pouvez également ajouter une structure HTML de base pour la mise en page.

```vue
<script setup>
import Nav from './Nav.vue';
import Footer from './Footer.vue';
</script>

<template>
  <div class="flex flex-col min-h-screen">
    <header>
      <Nav />
    </header>

    <main class="flex-grow py-5">
      <div class="container mx-auto">
        <slot name="title" />
        <slot name="content" />
      </div>
    </main>

    <Footer />
  </div>
</template>
```

1. Utilisez le composant `Layout.vue` dans le fichier `App.vue`. Entourez le contenu spécifique à chaque page avec la balise `<template v-slot:content>` et insérez le contenu à l'intérieur.

```vue
<script setup>
import Layout from './components/Layout.vue';
</script>

<template>
  <Layout>
    <template v-slot:title>
      <h1 class="mt-5 text-3xl font-bold">Pied de page fixe avec barre de navigation</h1>
    </template>

    <template v-slot:content>
      <p class="text-lg">
        Utilisez Tailwind pour fixer un pied de page au bas de la fenêtre avec des classes
utilitaires comme
        <code class="bg-gray-200 text-sm p-1 rounded">min-h-screen</code> et
        <code class="bg-gray-200 text-sm p-1 rounded">flex flex-col</code>.
      </p>
      <p>
        Découvrez plus d'exemples en visitant
        <a href="https://tailwindcss.com/docs"
           class="text-blue-500 hover:underline"
          >la documentation officielle de Tailwind CSS</a>.
      </p>
    </template>
  </Layout>
</template>
```

Maintenant, vous avez créé le composant `Layout.vue` qui utilise un slot pour insérer du contenu dynamique. Vous pouvez réutiliser ce composant dans différentes pages de votre application en insérant du contenu spécifique à chaque page à l'intérieur du slot. Nous verrons plus tard comment avoir plusieurs pages sur notre application.

https://vuejs.org/guide/components/slots.html#slots

Les Props

Dans Vue.js, les props sont une fonctionnalité qui permet de passer des données de composant parent à composant enfant. Les props permettent de rendre les composants plus flexibles et réutilisables en leur permettant de recevoir des données externes.

Introduction à la notion de props dans Vue.js

Les props sont des attributs personnalisés que vous pouvez définir sur un composant et qui peuvent être passés de son composant parent. Les props permettent au composant parent de communiquer avec le composant enfant en lui transmettant des valeurs.

```vue
<UserCard
  name="Eli Kopter"
  job="CEO"
  img="/img/users/elikopter.jpg"
/>
```

Création du Composant NavLink.vue avec des props

Pour illustrer l'utilisation des props, nous allons créer un composant `NavLink.vue` qui affichera des liens de navigation personnalisés. Ce composant prendra une prop `url` pour spécifier l'URL du lien et une prop `text` pour spécifier le texte du lien.

Voici les étapes pour créer le composant `NavLink.vue` :

1. Créez un nouveau fichier `NavLink.vue` dans le répertoire des composants de votre projet.
2. Dans le template du composant `NavLink.vue`, utilisez les props pour afficher le lien de navigation personnalisé.

```vue
<template>
  <li class="list-none">
    <a
      class="block px-4 py-2"
      :class="{
        'text-blue-600 font-semibold': isActive,
        'text-gray-500 pointer-events-none opacity-50': isDisabled
      }"
      aria-current="page"
      :href="url"
      :disabled="isDisabled"
      :aria-disabled="isDisabled"
    >
      {{ text }}
    </a>
  </li>
</template>
```

1. Définissez les props `url` et `text` dans la section `<script>` du fichier `NavLink.vue`.

```vue
<script setup>
defineProps({
  url: {
    type: String,
    required: false,
    default: '',
  },
  text: {
    type: String,
    required: true,
  },
  isActive: {
    type: Boolean,
    required: false,
    default: false,
  },
  isDisabled: {
    type: Boolean,
    required: false,
    default: false,
  },
});
</script>
```

Dans cet exemple, nous avons défini quatre props : `url` de type `String`, `text` de type `String`, `isActive` de type `Boolean` (vrai ou faux) et `isDisabled` de type `Boolean` aussi. La prop `text` est requise, ce qui signifie qu'ils doivent être fournis lors de l'utilisation du composant `NavLink`.

1. Utilisez le composant `NavLink.vue` dans le composant `Nav.vue` en passant les valeurs des props.

VUE

```vue
<script setup>
import NavLink from './NavLink.vue';
import { ref } from 'vue';

const isMenuOpen = ref(false);
</script>

<template>
  <nav class="bg-gray-900 fixed top-0 w-full z-50">
    <div class="container mx-auto flex items-center justify-between p-4">
      <a class="text-white text-lg font-bold" href="#">Navbar Fixe</a>
      <button
        class="text-white block md:hidden focus:outline-none"
        type="button"
        @click="isMenuOpen = !isMenuOpen"
        aria-controls="navbarCollapse"
        aria-expanded="false"
        aria-label="Toggle navigation"
      >
        <svg
          class="h-6 w-6"
          fill="none"
          stroke="currentColor"
          viewBox="0 0 24 24"
          xmlns="http://www.w3.org/2000/svg"
        >
          <path
            stroke-linecap="round"
            stroke-linejoin="round"
            stroke-width="2"
            d="M4 6h16M4 12h16m-7 6h7"
          ></path>
        </svg>
      </button>
      <div :class="{'hidden md:flex': !isMenuOpen}" class="w-full md:w-auto md:flex-grow">
        <ul class="flex flex-col md:flex-row items-center space-y-2 md:space-y-0 md:space-x-4">
          <NavLink url="#" text="Home" :isActive="true" />
          <NavLink url="#" text="Link" />
          <NavLink text="Disabled" :isDisabled="true" />
        </ul>
        <form class="flex items-center mt-4 md:mt-0">
          <input
            class="form-input rounded-lg border border-gray-300 py-1 px-2 focus:ring focus:border-
blue-300"
            type="search"
            placeholder="Recherche"
            aria-label="Search"
          />
          <button
            class="ml-2 bg-green-500 text-white px-4 py-1 rounded-lg hover:bg-green-600"
            type="submit"
          >
            Recherche
          </button>
        </form>
      </div>
```

```
    </div>
  </nav>
</template>
```

Dans cet exemple, nous utilisons le composant `NavLink` en lui passant les valeurs des props `url` et `text`. Chaque utilisation du composant `NavLink` crée un lien de navigation avec l'URL et le texte spécifiés.

https://vuejs.org/guide/components/props.html

Nettoyage des fichiers

Nous allons supprimer les fichiers qui ne sont plus utiles:

- Dans le dossier `src`
  - `style.css`
- dans le dossier `components`
  - `HelloWorld.vue`
- dans le dossier `assets`
  - `vue.svg`

Conclusion

Dans ce chapitre, nous avons exploré la notion de composants dans Vue.js. Nous avons commencé par une introduction générale sur les composants, en expliquant leur utilité et leur structure de base. Ensuite, nous avons mis en pratique nos connaissances en créant les composants `Nav.vue` et `Footer.vue` pour encapsuler le code du navbar et du footer respectivement.

Nous avons ensuite abordé la notion de slot dans Vue.js, en créant le composant `Layout.vue` qui utilise un slot pour insérer du contenu dynamique. Les slots nous permettent de personnaliser l'apparence et le comportement d'un composant en insérant du contenu spécifique à chaque utilisation.

Enfin, nous avons introduit les props dans Vue.js en créant le composant `NavLink.vue` qui utilise des props pour afficher des liens de navigation personnalisés. Les props permettent de rendre les composants plus flexibles et réutilisables en leur permettant de recevoir des données externes.

En récapitulant, voici les principaux points abordés dans ce chapitre :

- Les composants dans Vue.js permettent de découper l'interface utilisateur en morceaux réutilisables et modulaires.
- Les slots permettent d'insérer du contenu dynamique à l'intérieur d'un composant.
- Les props permettent de passer des données de composant parent à composant enfant.

Les composants, les slots et les props sont des concepts essentiels dans le développement Vue.js. Ils favorisent la réutilisabilité, la modularité et la flexibilité du code, ce qui facilite la création d'applications robustes et évolutives.

N'hésitez pas à pratiquer davantage et à explorer d'autres fonctionnalités avancées de Vue.js pour approfondir vos connaissances.

2.4 Vue Router

Introduction

Dans ce chapitre, nous allons explorer Vue Router, une bibliothèque de routage pour les applications Vue.js. Vue Router permet de gérer la navigation au sein d'une application Vue en utilisant des routes. Cela signifie que vous pouvez créer des URL spécifiques pour chaque vue de votre application et naviguer entre elles de manière fluide et sans rechargement de la page. La navigation sera entièrement régie par Vue Router en JavaScript.

L'installation de Vue Router se fait avec la commande `npm install vue-router@4` s'il n'est pas déjà installé dans le projet.

Une fois que Vue Router est installé, nous allons adapter notre projet existant pour l'utiliser.

https://router.vuejs.org/

Utilisation de Vue Router

Nous allons d'abord voir un exemple simple d'application qui utilise Vue Router. Ensuite, nous verrons comment adapter notre projet pour l'utiliser étape par étape.

Configuration de Vue Router

Voici un exemple de configuration de routes pour un projet simple :

JAVASCRIPT

```javascript
import { createRouter, createWebHistory } from 'vue-router';
import Home from './pages/Home.vue';
import About from './pages/About.vue';

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home,
  },
  {
    path: '/about',
    name: 'About',
    component: About,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});
```

Dans cet exemple, nous avons défini deux routes : une pour la page d'accueil ("/") et une pour la page "À Propos" ("/about"). Chaque route est associée à un composant Vue correspondant.

RouterView

Le composant `RouterView` est un composant fourni par Vue Router qui sert de point d'entrée pour afficher les composants correspondants aux routes de votre application.

Lorsque vous définissez des routes avec Vue Router, chaque route est associée à un composant spécifique. Lorsque l'URL de votre application correspond à une route spécifique, le composant associé à cette route sera rendu dans le `RouterView`.

En d'autres termes, le `RouterView` est l'endroit où le contenu dynamique de votre application sera affiché en fonction de la route actuelle. Lorsque l'URL change, le `RouterView` est mis à jour automatiquement pour afficher le composant correspondant à la nouvelle route.

Par exemple, si vous avez une route pour la page d'accueil et une autre route pour la page "À Propos", vous pouvez définir votre `RouterView` dans le fichier `App.vue` comme suit :

```vue
<script setup>
import { RouterView, RouterLink } from 'vue-router';
</script>

<template>
 <div id="app">
    <nav>
      <ul>
        <li>
          <RouterLink to="/">Accueil</RouterLink>
        </li>
        <li>
          <RouterLink to="/about">À Propos</RouterLink>
        </li>
      </ul>
    </nav>

    <RouterView>
  </div>
</template>
```

Lorsque l'URL de votre application est `/`, le composant associé à la route de la page d'accueil sera rendu dans le `RouterView`. Si l'URL est `/about`, le composant associé à la route de la page "À Propos" sera rendu à la place.

En utilisant le `RouterView`, vous pouvez facilement gérer la navigation entre les différentes vues de votre application en fonction des routes définies avec Vue Router.

RouterLink

Les `RouterLink` sont des composants fournis par Vue Router qui permettent de créer des liens de navigation dans une application Vue utilisant Vue Router.

Les `RouterLink` servent à remplacer les balises `<a>` traditionnelles utilisées pour créer des liens HTML. Ils sont spécialement conçus pour fonctionner avec Vue Router et offrent des fonctionnalités supplémentaires. Attention, vous ne devez pas les utiliser pour les liens vers d'autres sites que votre application (exemple : les liens vers les réseaux sociaux)

Lorsque vous utilisez un `RouterLink`, vous spécifiez l'URL cible du lien en utilisant la prop `to`. Lorsque le lien est cliqué, Vue Router se charge de naviguer vers l'URL spécifiée et de rendre le composant correspondant dans le `RouterView`.

Adaptation du projet

La première étape consiste à créer un dossier "pages" dans lequel nous placerons deux fichiers : `Home.vue` et `About.vue`. Ces fichiers représenteront les vues que nous afficherons dans notre application.

Pages

Home.vue

```vue
<script setup>
import { RouterLink } from 'vue-router';
</script>

<template>
  <div class="bg-gray-100 py-12">
    <div class="container mx-auto text-center">
      <h1 class="text-4xl font-bold mb-4">Bienvenue sur Mon Site Web</h1>
      <p class="text-lg mb-6">
        Votre source fiable d'informations et de ressources en ligne.
      </p>
      <RouterLink
        to="/about"
        class="bg-blue-500 hover:bg-blue-600 text-white font-bold py-3 px-6 rounded-lg"
      >
        En savoir plus sur nous
      </RouterLink>
    </div>
  </div>
</template>
```

Le fichier `Home.vue` représente la vue d'accueil de notre application. Il affiche un titre accrocheur ainsi qu'une brève description de notre site. De plus, il contient un lien vers la vue "À Propos" de notre application.

About.vue

```vue
<template>
  <div class="container mx-auto py-12 px-4">
    <h2 class="text-3xl font-bold mb-6">À Propos de Nous</h2>
    <p class="text-lg mb-4">
      Nous sommes une équipe dévouée de professionnels passionnés par la création
      de contenu informatif et utile en ligne. Notre objectif est de vous fournir
      des informations précises, des conseils pratiques et des ressources de haute
      qualité pour vous aider dans votre vie quotidienne et professionnelle.
    </p>
    <p class="text-lg mb-4">
      Que vous recherchiez des tutoriels, des actualités, des analyses
      approfondies ou des astuces pratiques, notre site est conçu pour répondre à
      vos besoins. Nous mettons l'accent sur la clarté, la fiabilité et la
      pertinence de nos contenus pour vous offrir une expérience exceptionnelle.
    </p>
    <p class="text-lg">
      Nous sommes fiers de notre engagement envers l'excellence éditoriale et
      notre désir de vous fournir des informations qui vous aident à prendre des
      décisions éclairées. Explorez notre site et découvrez tout ce que nous avons
      à offrir.
    </p>
  </div>
</template>
```

Le fichier `About.vue` représente la vue "À Propos" de notre application. Il affiche des informations sur notre équipe et notre engagement à fournir un contenu de qualité.

Layout

Nous allons maintenant modifier le layout de notre application pour améliorer la structure et l'apparence générale. Pour cela, nous allons créer un dossier "Layout" et y placer le fichier `Layout.vue`. Assurez-vous de supprimer l'ancien fichier du dossier "components".

Layout.vue

```vue
<script setup>
import Nav from '../components/Nav.vue';
import Footer from '../components/Footer.vue';
</script>

<template>
  <div class="flex flex-col min-h-screen">
    <header>
      <Nav />
    </header>

    <main class="flex-grow pb-5 pt-24">
      <div class="container mx-auto px-4">
        <slot />
      </div>
    </main>

    <Footer />
  </div>
</template>
```

Le fichier `Layout.vue` représente le layout de notre application. Il utilise les composants `Nav` et `Footer` pour afficher une barre de navigation en haut et un pied de page en bas. Le contenu principal de chaque vue sera inséré à l'emplacement du `<slot />`.

Nous avons modifié les chemins d'importation des composants Nav et Footer puisque le Layout se trouve maintenant dans le dossier layouts.

Nous actualiserons l'importation du layout dans le fichier `App.vue` par la suite.

App.vue

Voici le code modifié du fichier `App.vue` :

```vue
<script setup>
import Layout from './layouts/Layout.vue';
import { RouterView } from 'vue-router';
</script>

<template>
  <Layout>
    <RouterView />
  </Layout>
</template>
```

Dans ce code, nous avons importé le composant `Layout` à partir du fichier `Layout.vue` situé dans le dossier "layouts". Nous avons également importé le composant `RouterView` de Vue Router.

Ensuite, nous avons utilisé le composant `Layout` comme conteneur principal de notre application. À l'intérieur du `Layout`, nous avons ajouté le composant `RouterView` qui sera utilisé pour afficher le contenu de chaque vue en fonction de la route actuelle.

Cette structure nous permet d'avoir un layout global pour notre application, avec le contenu spécifique de chaque vue rendu dans le `RouterView`.

Navigation

Nous allons devoir modifier le composant `Nav.vue` et le composant `NavLink.vue`.

Voici le code modifié du composant `Nav.vue` utilisant les `RouterLink` :

```vue
<script setup>
import NavLink from './NavLink.vue';
import { RouterLink } from 'vue-router';
import { ref } from 'vue';

const isMenuOpen = ref(false);
</script>

<template>
  <nav class="bg-gray-900 fixed top-0 w-full z-50">
    <div class="container mx-auto flex items-center justify-between p-4">
      <RouterLink class="text-white text-lg font-bold" to="/">Mon site</RouterLink>
      <button
        class="text-white md:hidden focus:outline-none"
        @click="isMenuOpen = !isMenuOpen"
        aria-controls="navbarCollapse"
        aria-expanded="false"
        aria-label="Toggle navigation"
      >
        <svg
          class="w-6 h-6"
          fill="none"
          stroke="currentColor"
          viewBox="0 0 24 24"
          xmlns="http://www.w3.org/2000/svg"
        >
          <path
            stroke-linecap="round"
            stroke-linejoin="round"
            stroke-width="2"
            d="M4 6h16M4 12h16m-7 6h7"
          ></path>
        </svg>
      </button>
      <div :class="{'block': isMenuOpen, 'hidden': !isMenuOpen}" class="w-full md:w-auto md:flex
md:items-center">
        <ul class="flex flex-col md:flex-row items-center space-y-4 md:space-y-0 md:space-x-6">
          <NavLink url="/" text="Accueil" />
          <NavLink url="/about" text="À propos" />
        </ul>
      </div>
    </div>
  </nav>
</template>
```

Et voici le code modifié du composant `NavLink.vue` :

```vue
<script setup>
import { RouterLink } from 'vue-router';

defineProps({
  url: {
    type: String,
    required: false,
    default: '',
  },
  text: {
    type: String,
    required: true,
  },
  isDisabled: {
    type: Boolean,
    required: false,
    default: false,
  },
});
</script>

<template>
  <li class="list-none">
    <RouterLink
      class="block px-4 py-2 text-base"
      :class="{
        'text-gray-500 pointer-events-none opacity-50': isDisabled,
        'text-blue-500 hover:text-blue-600': !isDisabled
      }"
      activeClass="font-semibold text-blue-600"
      aria-current="page"
      :to="url"
      :aria-disabled="isDisabled"
    >
      {{ text }}
    </RouterLink>
  </li>
</template>
```

Dans ces modifications, nous avons utilisé les balises `RouterLink` à la place des balises `<a>`. Les `RouterLink` sont fournis par Vue Router et sont spécialement conçus pour fonctionner avec les routes de l'application.

Nous avons également supprimé la prop `isActive` dans le composant `NavLink`, car Vue Router ajoute automatiquement la classe spécifiée par la prop `activeClass` au `RouterLink` correspondant à la page que l'utilisateur visite actuellement. On passe donc les classes `font-semibold text-blue-600`.

Avec ces modifications, les liens de navigation dans le composant `Nav` utiliseront les `RouterLink` pour naviguer vers la page d'accueil et la page "À propos" de l'application.

index.html

Voici le code modifié du fichier `index.html` :

```xml
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Mon site</title>
    <meta
      name="description"
      content="Nous sommes une équipe dévouée de professionnels passionnés par la création
    de contenu informatif et utile en ligne."
    />
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

Dans ce code, nous avons ajouté la balise `<meta>` avec l'attribut `name="description"` pour définir la description de votre site. Vous pouvez la remplacer par la description souhaitée pour votre application.

Nous avons modifié la balise `<html>` pour définir la langue de la page sur "fr" pour le français.

main.js

Maintenant, nous allons modifier le fichier `main.js` pour ajouter Vue Router à notre application, déclarer nos routes et enregistrer le routeur.

Voici le code modifié du fichier `main.js` :

```javascript
import { createRouter, createWebHistory } from 'vue-router';
import { createApp } from 'vue';

// Importation de Tailwind CSS
import './assets/main.css'; // Assurez-vous que le fichier Tailwind est correctement configuré

import App from './App.vue';
import Home from './pages/Home.vue';
import About from './pages/About.vue';

// Définition des routes
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
];

// Création du routeur
const router = createRouter({
  history: createWebHistory(),
  routes,
});

// Création de l'application Vue
const app = createApp(App);

// Utilisation du routeur dans l'application
app.use(router);

// Montage de l'application sur l'élément #app de index.html
app.mount('#app');
```

Dans ce code, nous avons importé les modules nécessaires de Vue Router et de Vue.js. Nous avons également importé les styles de Tailwind de notre fichier `style.css` pour appliquer les styles personnalisés à notre application.

Ensuite, nous avons défini les routes de notre application en utilisant un tableau d'objets. Chaque objet dans le tableau représente une route avec une propriété `path` spécifiant l'URL de la route et une propriété `component` spécifiant le composant Vue associé à la route.

Ensuite, nous avons créé une instance du routeur en utilisant `createRouter` de Vue Router. Nous avons passé l'option `history` avec `createWebHistory()` pour utiliser le mode de l'historique HTML5 pour la gestion des URL.

Ensuite, nous avons créé l'application Vue en utilisant `createApp` de Vue.js. Nous avons utilisé `app.use(router)` pour enregistrer le routeur dans l'application et permettre à Vue Router de gérer la navigation.

Enfin, nous avons monté l'application Vue sur l'élément `#app` de `index.html` en utilisant `app.mount('#app')`.

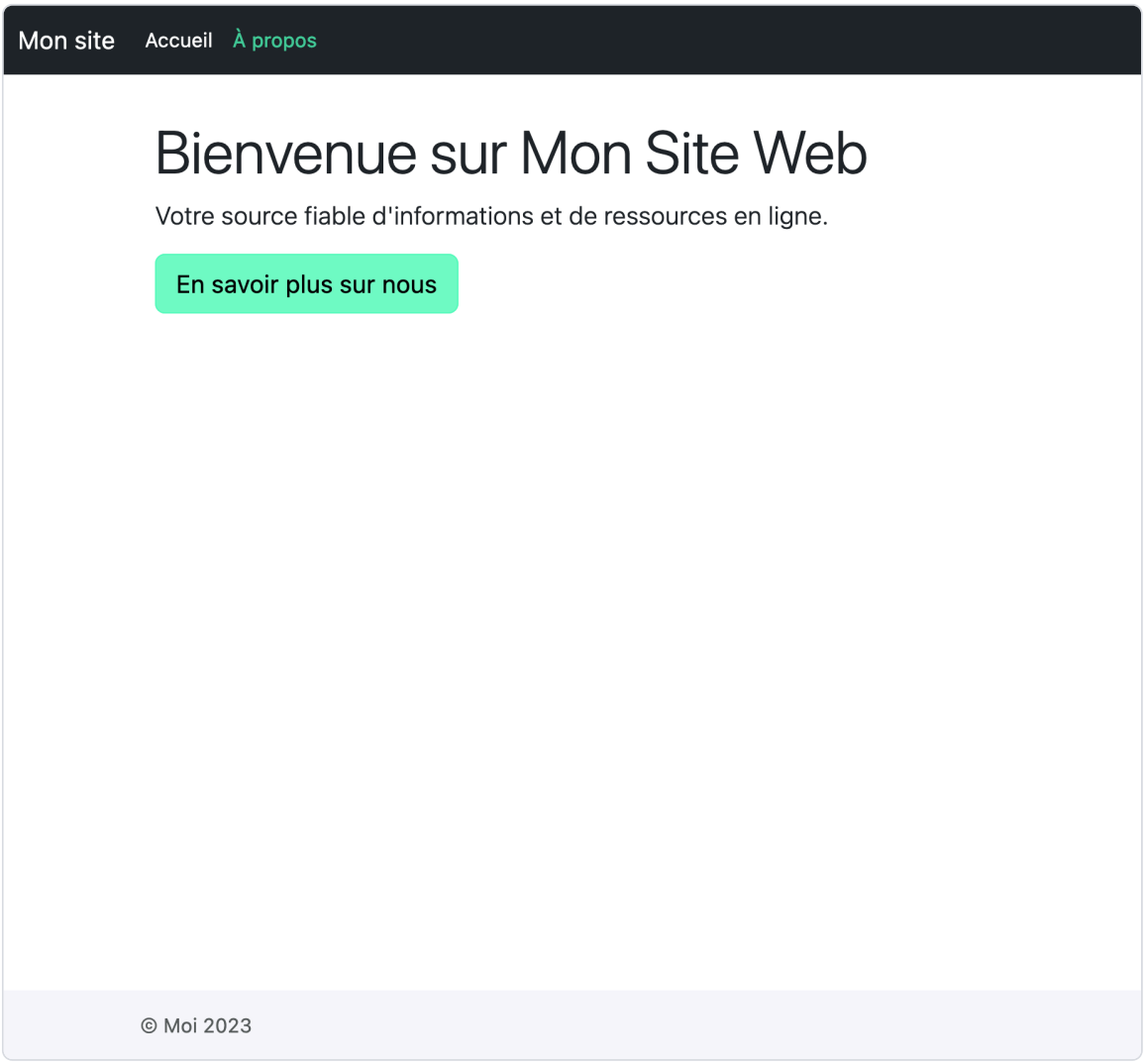
Avec ces modifications, nous avons configuré Vue Router dans notre application Vue et défini les routes pour la navigation entre les différentes vues.

Vérification du fonctionnement

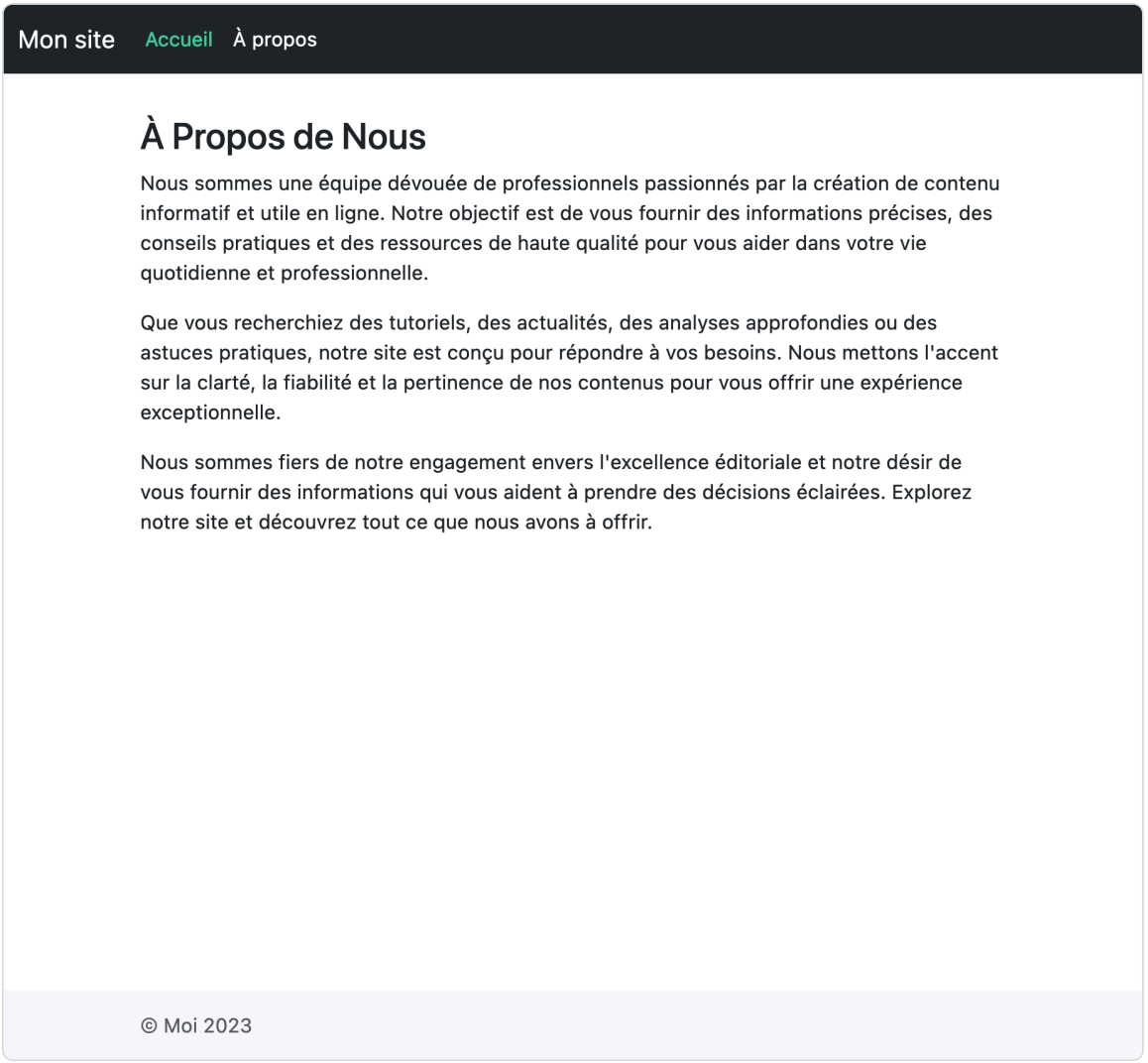
Si vous avez suivi toutes les étapes précédentes correctement, votre site devrait maintenant fonctionner avec Vue Router. Les liens de navigation devraient rediriger vers les deux pages différentes que nous avons créées : la page d'accueil et la page "À Propos".

Pour vérifier le fonctionnement, vous pouvez cliquer sur les liens de navigation dans votre application. Par exemple, en cliquant sur le lien "Accueil", vous devriez être redirigé vers la page d'accueil où vous verrez le contenu correspondant. De même, en cliquant sur le lien "À Propos", vous devriez être redirigé vers la page "À Propos" avec le contenu correspondant.

Page d'accueil :



Page à propos :



Si vous rencontrez des problèmes et que votre site ne fonctionne pas comme prévu, je vous recommande de revoir attentivement les étapes précédentes pour vous assurer que vous avez suivi toutes les modifications nécessaires. Vérifiez également que vous avez correctement importé les composants, défini les routes et enregistré le routeur dans votre application.

Félicitations ! Vous avez réussi à adapter votre projet pour utiliser Vue Router et à créer une navigation fluide entre les différentes vues de votre application. Vous pouvez maintenant continuer à développer et améliorer votre site en utilisant les fonctionnalités puissantes de Vue Router.

2.5 Les Refs et les v-models dans Vue.js

Liens utiles

- https://stackblitz.com/\~/github.com/opmvpc/synth-se-vue
- https://synth-se-vue.vercel.app/

Introduction

Dans ce chapitre, nous allons explorer les concepts des refs et des v-models dans Vue.js. Les refs sont utilisés pour déclarer des états réactifs dans une application Vue.js, tandis que les v-models permettent de lier facilement des données à des éléments de formulaire. Nous allons découvrir comment utiliser ces fonctionnalités pour améliorer la réactivité et l'interactivité de nos applications.

Les Refs

Les refs sont utilisés pour déclarer et accéder à des états réactifs dans une application Vue.js. Nous allons commencer par apprendre comment déclarer un état réactif en utilisant la fonction `ref()` de la Composition API.

Déclaration de l'état réactif avec la fonction `ref()`

La fonction `ref()` est recommandée pour déclarer un état réactif dans la Composition API. Elle enveloppe la valeur passée en argument dans un objet ref avec une propriété `.value`. Par exemple :

```javascript
import { ref } from 'vue'

const count = ref(0)

console.log(count) // { value: 0 }
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

Nous pouvons également accéder aux refs dans le template d'un composant en les déclarant dans la balise script. Par exemple :

```javascript
<script setup>
import { ref } from 'vue';

// Déclaration du ref
const count = ref(0);
</script>
```

Dans le template, nous pouvons utiliser directement le ref sans avoir besoin d'ajouter `.value`. Par exemple :

```vue
<template>
  <div>{{ count }}</div>
</template>
```

Mutation d'un ref dans les gestionnaires d'événements

Les refs peuvent être mutés directement dans les gestionnaires d'événements. Par exemple, nous pouvons incrémenter la valeur d'un ref dans un gestionnaire d'événements de cette façon :

```vue
<template>
  <div>
    <p>Le compteur est à : {{ count }}</p>
    <button @click="count++">Incrémenter</button>
  </div>
</template>
```

Déclaration de fonctions pour modifier les refs

Pour des logiques plus complexes, nous pouvons déclarer des fonctions qui modifient les refs dans la même portée et les exposer comme méthodes du composant. Par exemple :

```javascript
<script setup>
import { ref } from 'vue';

// Déclaration du ref
const count = ref(0);

// Définition de la fonction d'incrémentation
function increment() {
  count.value++;
}
</script>
```

Les méthodes exposées peuvent ensuite être utilisées comme gestionnaires d'événements. Par exemple :

```vue
<template>
  <div>
    <p>Le compteur est à : {{ count }}</p>
    <button @click="increment">Incrémenter</button>
  </div>
</template>
```

Les v-models

Les v-models sont utilisés pour lier facilement des données à des éléments de formulaire dans une application Vue.js. Ils simplifient la gestion des données des formulaires et permettent une mise à jour automatique des valeurs liées.

Utilisation du v-model pour lier des données à des éléments de formulaire

Le v-model peut être utilisé avec les éléments de formulaire courants tels que les champs de saisie, les sélecteurs et les zones de texte. Par exemple, nous pouvons utiliser le v-model pour lier une variable à un champ de saisie :

```vue
<template>
  <input v-model="name" />
  <div>Valeur : {{ name }}</div>
</template>
```

Personnalisation du comportement du v-model avec des composants personnalisés

Le v-model peut également être utilisé avec des composants personnalisés pour personnaliser le comportement de la liaison des données. Pour cela, nous devons créer un composant avec une prop `value` et un événement `input`. Par exemple :

```vue
<script setup>
import { ref } from 'vue';
import CustomInput from './CustomInput.vue';

// Déclaration du ref pour le message
const message = ref('');
</script>

<template>
  <custom-input v-model="message"></custom-input>
</template>
```

Dans le composant `CustomInput`, nous devons émettre l'événement `input` pour mettre à jour la valeur liée. Par exemple :

```vue
<script setup>
import { defineProps, defineEmits } from 'vue';

// Définition des props
const props = defineProps({
  value: String,
});

// Définition des événements
const emit = defineEmits(['input']);
</script>

<template>
  <input :value="props.value" @input="emit('input',event.target.value)" />
</template>
```

Utilisation du v-model avec des composants tiers

Le v-model peut également être utilisé avec des composants tiers qui prennent en charge cette fonctionnalité. Par exemple, certains composants de bibliothèques populaires comme Vuetify et Element Plus prennent en charge le v-model pour faciliter la liaison des données.

Exemple pratique

Pour illustrer l'utilisation des refs et des v-models, nous allons créer un exemple pratique. Voici le code de l'exemple :

```vue
<script setup>
import { ref } from 'vue';

const count = ref(0);
const name = ref('');
</script>

<template>
  <h2>Compteur</h2>
  <button class="btn btn-primary" @click="count++">{{ count }}</button>

  <h2 class="mt-3">Nom</h2>
  <div>Valeur : {{ name.length > 0 ? name : 'pas de nom' }}</div>
  <input v-model="name" />
</template>
```

2.6 V-if, V-show et V-for

Introduction

Les directives V-if, V-show et V-for sont des fonctionnalités essentielles de Vue.js qui permettent de manipuler la structure et l'affichage des éléments dans un template. Dans ce chapitre, nous allons explorer en détail ces directives et comprendre comment les utiliser efficacement dans le développement d'applications Vue.js.

V-if

La directive V-if permet de conditionner l'affichage d'éléments dans un template en fonction d'une expression booléenne. Lorsque l'expression est évaluée à `true`, les éléments sont rendus dans le DOM. En revanche, lorsque l'expression est évaluée à `false`, les éléments sont retirés du DOM.

Voici un exemple de code utilisant la directive V-if :

```vue
<script setup>
import { ref } from 'vue';

const isOpen = ref(false);
</script>

<template>
  <button class="btn btn-primary" @click="isOpen = !isOpen">
    {{ isOpen ? 'Fermer' : 'Ouvrir' }}
  </button>

  <ul v-if="isOpen" class="list-group mt-3">
    <li class="list-group-item">Pomme</li>
    <li class="list-group-item">Banane</li>
    <li class="list-group-item">Poire</li>
    <li class="list-group-item">Raisin</li>
  </ul>

  <ul v-else class="list-group mt-3">
    <li class="list-group-item">Listes cachées</li>
  </ul>
</template>
```

Dans cet exemple, un bouton est utilisé pour basculer la valeur de `isOpen` entre `true` et `false`. Lorsque `isOpen` est `true`, la liste de fruits est affichée, sinon un message "Listes cachées" est affiché à la place.

V-show

La directive V-show est similaire à la directive V-if, mais au lieu de retirer les éléments du DOM lorsque l'expression est évaluée à `false`, elle utilise les styles CSS pour les rendre invisibles. Cela signifie que les éléments sont toujours présents dans le DOM, mais ils ne sont pas visibles à l'écran.

Voici un exemple de code utilisant la directive V-show :

```vue
<script setup>
import { ref } from 'vue';

const isOpen = ref(false);
</script>

<template>
  <button class="btn btn-primary" @click="isOpen = !isOpen">
    {{ isOpen ? 'Fermer' : 'Ouvrir' }}
  </button>

  <ul v-show="isOpen" class="list-group mt-3">
    <li class="list-group-item">Pomme</li>
    <li class="list-group-item">Banane</li>
    <li class="list-group-item">Poire</li>
    <li class="list-group-item">Raisin</li>
  </ul>
</template>
```

Dans cet exemple, la liste de fruits est toujours présente dans le DOM, mais lorsque `isOpen` est `false`, elle est rendue invisible à l'aide de styles CSS.

Il est important de noter que la directive V-show est plus performante que la directive V-if dans les cas où les éléments sont fréquemment affichés et masqués.

V-for

La directive V-for permet d'itérer sur une liste d'éléments et de générer du contenu dynamique dans un template. Elle est similaire à une boucle `for` en programmation.

Voici un exemple de code utilisant la directive V-for :

VUE

```
<script setup>
import { ref } from 'vue';

const fruits = ref([
  {
    name: 'Pomme',
  },
  {
    name: 'Banane',
  },
  {
    name: 'Poire',
  },
  {
    name: 'Raisin',
  },
]);
</script>

<template>
  <h1>Liste de fruits</h1>
  <ul class="list-group mt-3">
    <li
      v-for="(fruit, index) in fruits"
      :key="`fruit-${index}`"
      class="list-group-item"
    >
      {{ fruit.name }}
    </li>
  </ul>
</template>
```

Dans cet exemple, la liste de fruits est itérée à l'aide de la directive V-for. Chaque élément de la liste est représenté par la variable `fruit`, et l'index de l'élément est représenté par la variable `index`. Le contenu de chaque élément est affiché à l'aide de l'expression “double moustaches” :

::: v-pre

`{{ fruit.name }}`

:::

La directive V-for est extrêmement utile pour générer du contenu dynamique à partir de données provenant d'une source externe ou d'un état interne de l'application.

2.7 Liaison de classes et de styles

Introduction

La liaison de classes et de styles est un aspect essentiel de la manipulation de l'interface utilisateur dans Vue.js. Elle permet de modifier dynamiquement les classes et les styles des éléments HTML en fonction de l'état des données. Dans ce chapitre, nous explorerons les différentes techniques pour effectuer cette liaison et nous verrons comment les utiliser de manière efficace dans vos projets Vue.js.

Liaison de classes HTML

La liaison de classes HTML se fait à l'aide de la directive `v-bind:class`. Cette directive permet de lier des classes à un élément de manière dynamique en fonction de l'état des données. Voici un exemple :

```vue
<script setup>
import { ref } from 'vue';

// Déclaration des états réactifs
const isActive = ref(true);
const hasError = ref(false);
</script>

<template>
  <div :class="{ active: isActive, 'text-danger': hasError }"></div>
</template>
```

Dans cet exemple, la classe `active` sera appliquée si la propriété `isActive` est vraie, et la classe `text-danger` sera appliquée si la propriété `hasError` est vraie.

Il est également possible de lier plusieurs classes en utilisant la syntaxe d'objet :

```vue
<template>
  <div :class="{ active: isActive, 'text-danger': hasError, 'font-bold': isBold }"></div>
</template>
```

Dans cet exemple, les classes `active`, `text-danger` et `font-bold` seront appliquées en fonction de l'état des propriétés correspondantes.

Liaison de styles HTML

La liaison de styles HTML se fait à l'aide de la directive `v-bind:style`. Cette directive permet de lier des styles à un élément de manière dynamique en utilisant des objets JavaScript. Voici un exemple :

```vue
<script setup>
import { ref } from 'vue';

// Déclaration des refs réactifs
const textColor = ref('red');
const fontSize = ref(16);
</script>

<template>
  <div :style="{ color: textColor, 'font-size': fontSize + 'px' }"></div>
</template>
```

Dans cet exemple, la couleur du texte sera déterminée par la valeur de la propriété `textColor`, et la taille de la police sera déterminée par la valeur de la propriété `fontSize`.

Il est également possible de lier plusieurs styles en utilisant un objet JavaScript :

```vue
<template>
  <div :style="{
    color: textColor,
    'font-size': fontSize + 'px',
    'font-weight': fontWeight
  }"></div>
</template>
```

Dans cet exemple, les styles `color`, `font-size` et `font-weight` seront appliqués en fonction des valeurs des propriétés correspondantes.

Liaison de classes et de styles avec des tableaux

Il est possible de lier des classes et des styles en utilisant des tableaux. Cela permet d'appliquer une liste de classes ou de styles à un élément. Voici un exemple :

```vue
<script setup>
import { ref } from 'vue';

// Déclaration des refs réactifs
const activeClass = ref('active');
const errorClass = ref('text-danger');
</script>

<template>
  <div :class="[activeClass, errorClass]"></div>
</template>
```

Dans cet exemple, les classes `active` et `text-danger` seront appliquées à l'élément.

Il est également possible de conditionner l'application des classes ou des styles à l'aide d'expressions ternaires :

```vue
<template>
  <div :class="[isActive ? activeClass : '', errorClass]"></div>
</template>
```

Dans cet exemple, la classe `activeClass` sera appliquée uniquement si la propriété `isActive` est vraie.

Liaison de classes et de styles avec les composants

La liaison de classes et de styles peut également être utilisée avec les composants. Lorsque vous utilisez l'attribut de classe ou la directive `:class` sur un composant avec un élément racine unique, les classes seront ajoutées à l'élément racine du composant et fusionnées avec les classes existantes. Voici un exemple :

```vue
<template>
  <MyComponent class="extra-class"></MyComponent>
</template>
```

Dans cet exemple, la classe `extra-class` sera ajoutée à l'élément racine du composant `MyComponent`.

Il est également possible de lier des classes à un composant en utilisant la directive `:class` avec un objet :

```vue
<template>
  <MyComponent :class="{ active: isActive }"></MyComponent>
</template>
```

Dans cet exemple, la classe `active` sera appliquée à l'élément racine du composant lorsque la propriété `isActive` est vraie.

Bonnes pratiques et astuces

- Utilisez des propriétés calculées pour générer des objets de classes et de styles de manière dynamique.
- Utilisez des objets de styles directement pour rendre votre code plus propre et plus lisible.
- Utilisez des tableaux pour appliquer plusieurs classes ou styles à un élément.

Exemple pratique

Voici un exemple concret d'utilisation de la liaison de classes et de styles dans un formulaire :

```vue
<script setup>
import { ref } from 'vue';

// Déclaration des refs réactifs pour les champs et la validation
const name = ref('');
const email = ref('');
const isNameValid = ref(true);
const isEmailValid = ref(false);
</script>

<template>
  <form>
    <div :class="{ 'is-valid': isNameValid, 'is-invalid': !isNameValid }">
      <input type="text" v-model="name" required />
    </div>

    <div :class="{ 'is-valid': isEmailValid, 'is-invalid': !isEmailValid }">
      <input type="email" v-model="email" required />
    </div>

    <button type="submit">Submit</button>
  </form>
</template>
```

Dans cet exemple, les classes `is-valid` et `is-invalid` seront appliquées aux div qui entourent les champs de saisie en fonction de la validité des données.

2.8 Écoute des événements dans Vue.js

Introduction

Dans Vue.js, il est possible d'écouter les événements du DOM et d'exécuter du code JavaScript en réponse à ces événements. Cela permet d'interagir avec l'utilisateur et de rendre les applications plus dynamiques. Dans ce chapitre, nous allons explorer les différentes méthodes pour écouter les événements dans Vue.js.

Handlers inline

Les handlers inline sont utilisés dans les cas simples où la logique de l'événement est courte et directe. Ils peuvent être utilisés pour exécuter du code JavaScript directement dans le template. Par exemple, pour incrémenter une variable `count` lorsqu'un bouton est cliqué, nous pouvons utiliser le code suivant :

```vue
<script setup>
import { ref } from 'vue';

// Déclaration du ref réactif
const count = ref(0);
</script>

<template>
  <div>
    <button @click="count++">Incrémenter</button>
    <p>{{ count }}</p>
  </div>
</template>
```

Dans cet exemple, nous utilisons la directive `@click` pour écouter l'événement de clic sur le bouton et exécuter l'expression `count++` pour incrémenter la variable `count`. La valeur de `count` est ensuite affichée dans le paragraphe.

Handlers de méthode

Les handlers de méthode sont utilisés lorsque la logique de l'événement est plus complexe ou nécessite un traitement supplémentaire. Plutôt que d'exécuter du code directement dans le template, nous pouvons définir une méthode dans le composant et l'appeler en tant que handler d'événement. Voici un exemple :

```vue
<script setup>
// Définition de la fonction greet
function greet() {
  alert('Bonjour !');
}
</script>

<template>
  <div>
    <button @click="greet">Saluer</button>
  </div>
</template>
```

Dans cet exemple, nous définissons une méthode `greet` qui affiche une boîte de dialogue avec le message "Bonjour !". Lorsque le bouton est cliqué, la méthode `greet` est appelée en tant que handler d'événement.

Passer des arguments aux handlers de méthode

Il est possible de passer des arguments personnalisés aux handlers de méthode. Cela permet d'effectuer des actions spécifiques en fonction du contexte de l'événement. Voici un exemple :

```vue
<script setup>
// Définition de la fonction say
function say(message) {
  alert(message);
}
</script>

<template>
  <div>
    <button @click="say('hello')">Dire bonjour</button>
    <button @click="say('bye')">Dire au revoir</button>
  </div>
</template>
```

Dans cet exemple, nous utilisons la méthode `say` pour afficher un message personnalisé en fonction du bouton cliqué.

Accéder à l'objet d'événement

Parfois, il est nécessaire d'accéder à l'objet d'événement d'origine dans un handler. Cela peut être utile pour effectuer des opérations supplémentaires ou accéder à des informations spécifiques sur l'événement. Voici un exemple :

```vue
<script setup>
// Définition de la fonction warn
function warn(message, event) {
  if (event) {
    event.preventDefault();
    console.log(event.target);
  }
  alert(message);
}
</script>

<template>
  <div>
    <button @click="warn('Le formulaire ne peut pas encore être soumis.',
$event)">Soumettre</button>
  </div>
</template>
```

Dans cet exemple, nous utilisons la méthode `warn` pour afficher un message d'avertissement et empêcher le comportement par défaut du formulaire lorsqu'il est soumis. Nous accédons à l'objet d'événement en utilisant le paramètre `$event`.

Modificateurs d'événement

Les modificateurs d'événement permettent de gérer facilement les détails de l'événement sans mélanger la logique des données. Ils sont utilisés avec la directive `v-on` ou son raccourci `@`. Voici quelques modificateurs d'événement couramment utilisés :

- `.stop` : arrête la propagation de l'événement.

- `.prevent` : empêche le comportement par défaut de l'événement.
- `.self` : déclenche le handler uniquement si l'élément lui-même est cliqué.
- `.capture` : utilise le mode de capture lors de l'ajout de l'écouteur d'événement.
- `.once` : déclenche le handler une seule fois.
- `.passive` : indique au navigateur que l'événement ne sera pas annulé.

Voici un exemple d'utilisation des modificateurs d'événement :

```vue
<script setup>
// Définition des fonctions
function doThis() {
  // Code à exécuter pour "doThis"
}

function onSubmit() {
  // Code à exécuter pour "onSubmit"
}

function doThat() {
  // Code à exécuter pour "doThat"
}
</script>

<template>
  <div>
    <a @click.stop="doThis">Lien</a>
    <form @submit.prevent="onSubmit">Formulaire</form>
    <a @click.stop.prevent="doThat">Lien avec plusieurs modificateurs</a>
  </div>
</template>
```

Dans cet exemple, nous utilisons différents modificateurs d'événement pour contrôler le comportement des handlers.

2.9 Cycle de vie des composants Vue.js

Introduction au cycle de vie des composants

Le cycle de vie des composants est une étape essentielle dans le développement d'applications Vue.js. Chaque instance de composant passe par une série d'étapes d'initialisation lors de sa création. Ces étapes comprennent la configuration de l'observation des données, la compilation du template, le montage de l'instance dans le DOM, et la mise à jour du DOM lorsque les données changent. Pendant ce processus, des fonctions appelées "hooks du cycle de vie" sont exécutées, offrant aux développeurs la possibilité d'ajouter leur propre code à des moments spécifiques.

Les hooks du cycle de vie

Les hooks du cycle de vie sont des fonctions fournies par Vue.js qui sont automatiquement appelées à différentes étapes du cycle de vie d'un composant. Les hooks les plus couramment utilisés sont `onMounted`, `onUpdated` et `onUnmounted`. Le hook `onMounted` permet d'exécuter du code après que le composant a terminé son rendu initial et créé les nœuds DOM. Le hook `onUpdated` est appelé chaque fois que le composant est mis à jour, tandis que le hook `onUnmounted` est exécuté lorsque le composant est supprimé du DOM.

```vue
<script setup>
import { onMounted } from 'vue'

onMounted(() => {
  console.log(`the component is now mounted.`)
})
</script>
```

Le hook `onMounted`

Le hook `onMounted` est particulièrement utile pour effectuer des actions après le rendu initial du composant et la création des nœuds DOM. Par exemple, il peut être utilisé pour effectuer des requêtes API, initialiser des plugins externes ou configurer des écouteurs d'événements. Lorsque le hook `onMounted` est appelé, Vue associe automatiquement la fonction de rappel enregistrée à l'instance du composant active. Dans notre exemple, le code utilise le hook `onMounted` pour effectuer une requête API et récupérer des données sur des personnes à partir d'une URL.

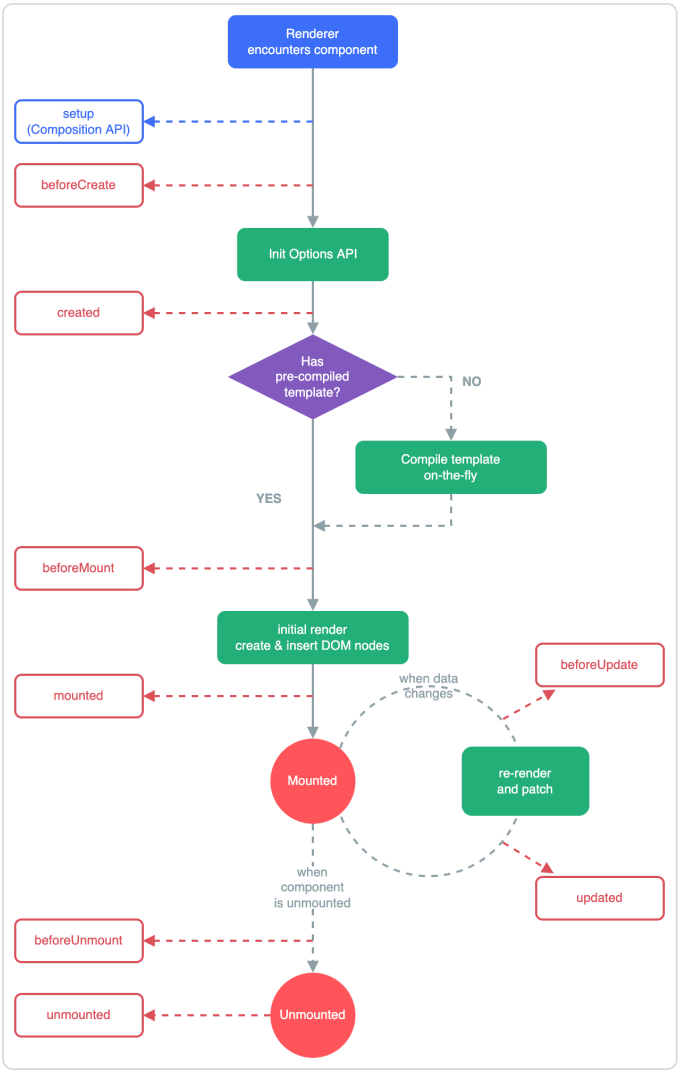
Les bonnes pratiques pour l'utilisation des hooks

Il est important de suivre certaines bonnes pratiques lors de l'utilisation des hooks du cycle de vie. Tout d'abord, il est essentiel d'enregistrer les hooks de manière synchrone pendant la configuration du composant. Il ne faut pas retarder l'enregistrement des hooks en utilisant des fonctions asynchrones ou des délais. De plus, il est recommandé de respecter l'ordre d'appel des hooks et d'éviter les appels conditionnels ou les boucles qui peuvent entraîner des comportements inattendus.

```javascript
setTimeout(() => {
  onMounted(() => {
    // this won't work.
  })
}, 100)
```

Utilisation des hooks en dehors de `<script setup>`

Bien que l'utilisation des hooks soit généralement recommandée à l'intérieur de la section `<script setup>`, il est également possible de les appeler en dehors de cette section. Cependant, il est important de respecter certaines conditions pour garantir un appel synchrone des hooks. Par exemple, l'appel doit être effectué depuis une fonction externe avec une pile d'appels synchrone, et l'appel doit être effectué à partir de la fonction `setup()` ou de la section `<script setup>`. Des exemples d'utilisation des hooks en dehors de `<script setup>` peuvent être donnés pour illustrer cette possibilité.



https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram

2.10 Les Références de Template

Introduction

Dans le développement d'applications Vue.js, il peut y avoir des cas où nous avons besoin d'accéder directement aux éléments DOM sous-jacents malgré le modèle de rendu déclaratif de Vue qui abstrait la plupart des opérations DOM. C'est là que les références de template entrent en jeu. Dans ce chapitre, nous allons explorer les références de template et comment les utiliser pour accéder aux éléments DOM de manière directe.

Références de Template

Les références de template sont définies à l'aide de l'attribut spécial `ref`. Par exemple, pour obtenir une référence directe à un champ de saisie, nous pouvons utiliser la syntaxe suivante :

```vue
<template>
  <input ref="myInput">
</template>
```

Dans cet exemple, `myInput` est la référence de template que nous pouvons utiliser pour accéder à l'élément DOM correspondant.

Accès aux Références

Pour accéder à une référence de template, nous devons utiliser la Composition API de Vue. Nous devons d'abord déclarer une référence avec le même nom dans notre script :

```vue
<script setup>
import { ref, onMounted } from 'vue'

const myInput = ref(null)

onMounted(() => {
  myInput.value.focus()
})
</script>

<template>
  <input ref="myInput" />
</template>
```

Dans cet exemple, nous utilisons la fonction `ref` pour déclarer une référence nommée `myInput`. Nous pouvons ensuite utiliser cette référence dans le bloc `onMounted` pour accéder à l'élément DOM correspondant et lui donner le focus.

2.11 Exercices avec Vue.js et TailwindCSS

Dans cette série d'exercices, les étudiants vont mettre en pratique leurs connaissances de Vue.js et TailwindCSS en créant des composants élémentaires. Chaque exercice augmentera progressivement en difficulté, permettant ainsi aux étudiants de renforcer leur compréhension et leur maîtrise de ces outils. Vous pouvez utiliser TypeScript si vous le souhaitez.

Exercice 1 : Création de Titres

Objectif : Créer un composant de titre qui peut être utilisé pour différents niveaux de titres (h1, h2, h3, etc.).

Instructions :

1. Créez un composant `TitleComponent`.
2. Ajoutez une prop pour passer le niveau de titre souhaité.
3. Stylez le titre en fonction de son niveau avec TailwindCSS.

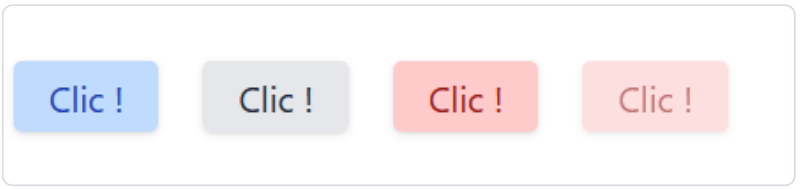


Exercice 2 : Création d'un Bouton

Objectif : Créer un composant bouton réutilisable avec plusieurs variantes de style avec TailwindCSS.

Instructions :

1. Créez un nouveau composant Vue.js nommé `ButtonComponent`.
2. Utilisez les classes TailwindCSS pour appliquer différents styles de boutons (par exemple, primaire, secondaire, danger, etc.).
3. Ajoutez les props nécessaires pour personnaliser le texte du bouton et le type de bouton (par exemple, `type="primary"`).



Exercice 3 : Création d'un Formulaire de Contact

Objectif : Créer un formulaire de contact avec validation des champs.

Instructions :

1. Créez une nouvelle page `Contact.vue`.
2. Le formulaire doit inclure des champs pour le nom, l'email, le sujet et le message.
3. Utilisez `v-model` pour la liaison bidirectionnelle des champs du formulaire.
4. Affichez les valeurs des inputs en dessous du formulaire dans une section dédiée quand l'utilisateur clique sur le bouton “Envoyer”.



Exercice 4 : Création d'un Select avec Recherche Autocomplete

Objectif : Construire un composant de sélection personnalisé avec une fonctionnalité de recherche autocomplete.

Instructions :

1. Créez un composant `AutocompleteSelectComponent`.
2. Le composant doit permettre à l'utilisateur de taper une recherche et de filtrer les options de sélection disponibles.
3. Utilisez des slots ou des props pour passer les options au composant.
4. Assurez-vous que le composant est accessible et navigable au clavier.



Exercice 5 : Création d'un Tableau

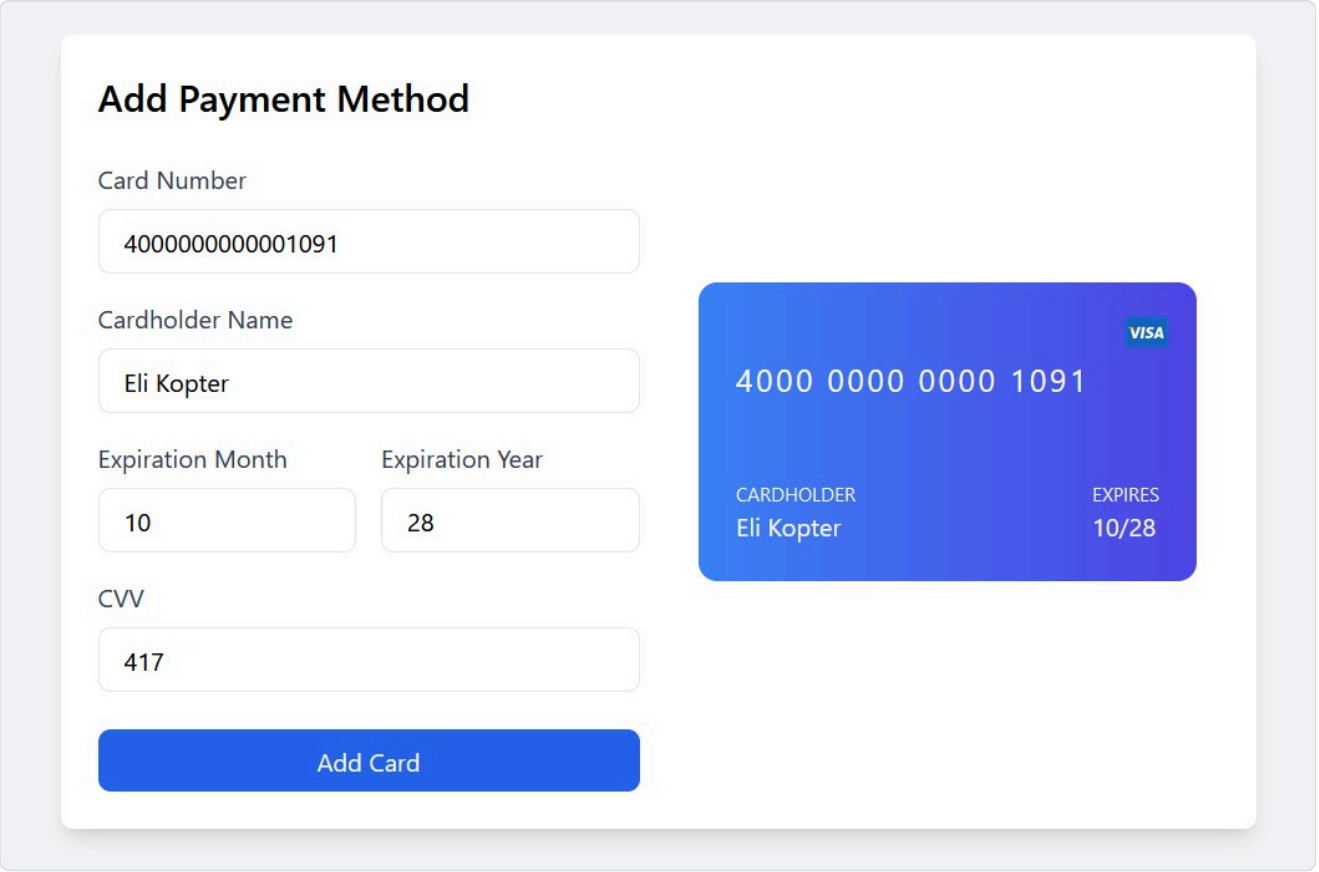
Objectif : Créer un composant de tableau qui affiche des données sous forme de lignes et de colonnes.

Instructions :

1. Créez un composant `TableComponent`.
2. Passez les données du tableau et la configuration des colonnes via des props.
3. Utilisez TailwindCSS pour le style du tableau.
4. Implémentez des fonctionnalités supplémentaires telles que le tri, la recherche et la pagination si le temps le permet.

2.12 Création d'un Formulaire de Paiement

Dans ce chapitre, vous allez créer une page web interactive qui permet aux utilisateurs d'ajouter un moyen de paiement. Vous utiliserez **Vue.js 3** pour la logique applicative et **Tailwind CSS** pour le style. L'objectif est de développer un formulaire avec une validation en temps réel, ainsi qu'une visualisation dynamique de la carte de crédit.



Objectifs d'Apprentissage

- Maîtriser l'utilisation de Vue.js 3 pour créer des interfaces réactives.
- Appliquer Tailwind CSS pour styliser rapidement une application.
- Implémenter une validation en temps réel des champs du formulaire.
- Rechercher et appliquer des méthodes de validation des numéros de carte de crédit.
- Dynamiser l'affichage en fonction des données saisies par l'utilisateur.

Étapes de l'Exercice

1. Création du Formulaire de Paiement

- Dans le composant principal, créez un formulaire contenant les champs suivants :
  - **Numéro de carte de crédit**
  - **Nom du titulaire**
  - **Date d'expiration** (mois et année)
  - **Code CVV**

2. Validation en Temps Réel

- Implémentez une validation en temps réel pour chaque champ :
  - **Numéro de carte** : Vérifiez que le numéro est valide en utilisant l'algorithme de Luhn. Vous devrez vous documenter sur cet algorithme et l'implémenter vous-même.

- **Date d'expiration** : Assurez-vous que la date n'est pas dépassée.
- **Code CVV** : Vérifiez qu'il contient le bon nombre de chiffres (3 pour Visa et Mastercard).

3. Détection du Type de Carte

- En fonction des premiers chiffres du numéro de carte, déterminez s'il s'agit d'une **Visa** ou d'une **Mastercard**. Vous pouvez vous baser sur les règles de numérotation des cartes disponibles en ligne.

4. Visualisation Dynamique de la Carte

- Créez un composant qui affiche une représentation visuelle de la carte de crédit :
  - Affichez le logo Visa ou Mastercard selon le type détecté.
  - Changez les couleurs ou le design en fonction du type de carte.
  - Mettez à jour les informations affichées (numéro, nom, date) en temps réel à mesure que l'utilisateur remplit le formulaire.

5. Stylisation avec Tailwind CSS

- Utilisez Tailwind CSS pour styliser le formulaire et la visualisation de la carte :
  - Assurez-vous que le design est responsive et agréable à utiliser.
  - Appliquez des styles pour indiquer visuellement les erreurs de validation.

6. Extension de l'Exercice (Optionnel)

- **Autres Types de Cartes** : Si le temps le permet, intégrez la prise en charge d'autres types de cartes comme American Express ou Maestro.
- **Animations** : Ajoutez des animations pour améliorer l'interactivité, comme une transition lors du changement de type de carte.
- **LocalStorage** : Enregistrez les informations saisies dans le navigateur pour les préremplir lors de la prochaine visite (en respectant les règles de sécurité concernant les données sensibles).

Conclusion

Cet exercice vous permettra de combiner vos compétences en JavaScript, Vue.js et CSS pour créer une application web interactive et professionnelle. Il met l'accent sur la validation des données, l'interactivité en temps réel et le design adaptatif.

2.13 API Pokémon

Introduction à l'API RESTful

Qu'est-ce qu'une API RESTful et à quoi sert-elle ?

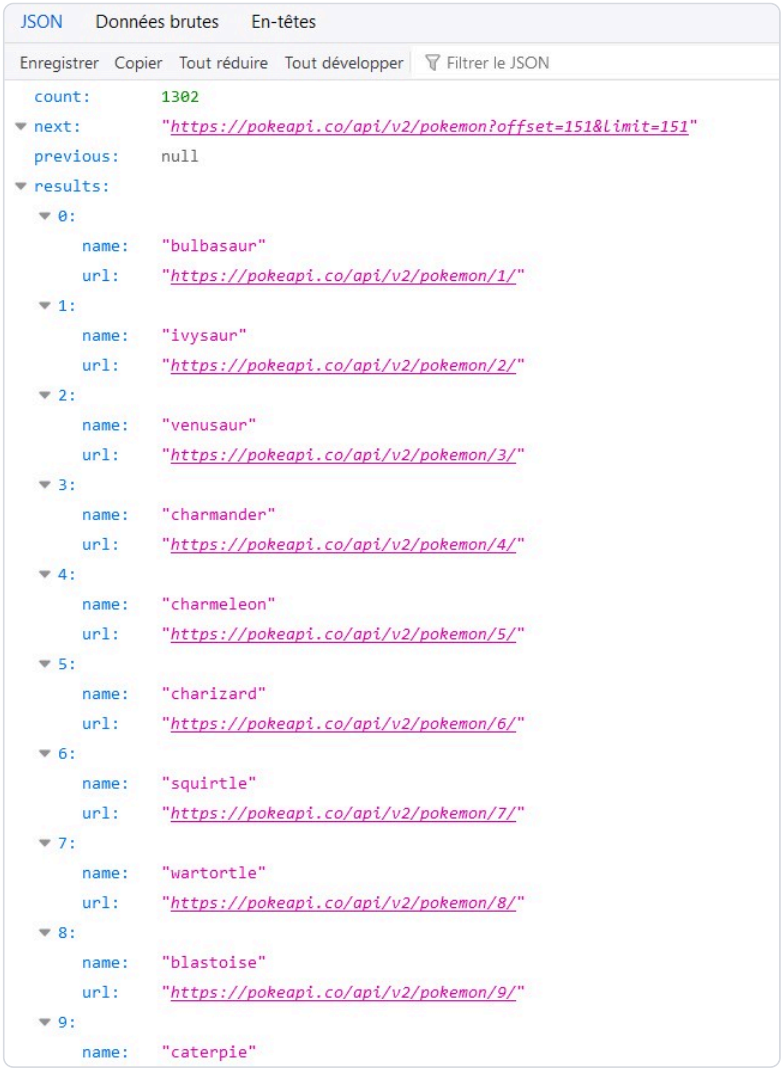
- **Définition :** Une API (Application Programming Interface) RESTful est un service web qui permet la communication entre différentes applications via le protocole HTTP en respectant les principes de l'architecture REST (Representational State Transfer).

- **Utilité :** Les API RESTful sont utilisées pour échanger des données entre un client et un serveur. Elles permettent aux développeurs d'accéder à des ressources web (comme les données des Pokémon) et de les manipuler en effectuant des requêtes HTTP standardisées (GET, POST, PUT, DELETE).

Exemple de requête à l'API Pokémon

Vous pouvez tester l'API en entrant cette URL dans votre navigateur :

- https://pokeapi.co/api/v2/pokemon?limit=151



Utilisation de l'API avec Fetch, Async/Await et Vue.js 3

Pour interagir avec une API RESTful en JavaScript moderne, on utilise souvent `fetch` avec `async/await`. Dans le contexte de Vue.js 3, voici comment intégrer ces concepts dans un composant.

Exemple de composant Vue.js 3 pour récupérer et afficher la liste des Pokémon avec un indicateur de chargement

```vue
<script setup>
import { ref, onMounted } from 'vue';

const pokemonList = ref([]);
const isLoading = ref(true);
const error = ref(null);

async function fetchPokemonList() {
  try {
    const response = await fetch('https://pokeapi.co/api/v2/pokemon?limit=151');
    const data = await response.json();
    pokemonList.value = data.results;
  } catch (err) {
    error.value = err;
  } finally {
    isLoading.value = false;
  }
}

onMounted(() => {
  fetchPokemonList();
});
</script>

<template>
  <div>
    <div v-if="isLoading">
      Chargement en cours...
    </div>
    <div v-else-if="error">
      Une erreur est survenue : {{ error.message }}
    </div>
    <ul v-else>
      <li v-for="pokemon in pokemonList" :key="pokemon.name">
        {{ pokemon.name }}
      </li>
    </ul>
  </div>
</template>
```

Explications

- **Imports et références :**
  - `ref` : Permet de créer des variables réactives.
  - `onMounted` : Un hook du cycle de vie qui exécute une fonction lorsque le composant est monté.
- **Variables réactives :**
  - `pokemonList` : Stocke la liste des Pokémon récupérés.
  - `isLoading` : Indique si la requête est en cours.
  - `error` : Stocke une éventuelle erreur survenue lors de la requête.

- **Fonction asynchrone** `fetchPokemonList` :
  - Utilise `await` pour attendre la résolution de `fetch` et de `response.json()`.
  - En cas de succès, stocke les résultats dans `pokemonList`.
  - En cas d'erreur, stocke l'erreur dans `error`.
  - Le bloc `finally` est exécuté dans tous les cas pour mettre `isLoading` à `false`.
- **Cycle de vie :**
  - `onMounted` appelle `fetchPokemonList` lorsque le composant est monté.
- **Template :**
  - Affiche "Chargement en cours..." tant que `isLoading` est `true`.
  - Affiche un message d'erreur si `error` n'est pas `null`.
  - Affiche la liste des Pokémon une fois les données chargées avec succès.

Description de l'exercice

Objectif

Créer un Pokédex interactif en utilisant Vue.js 3, Tailwind CSS pour le design, et l'API de PokéAPI. L'application doit être responsive pour une utilisation optimale sur différents appareils.

Étapes à réaliser

Initialisation du projet

- Créer une nouvelle application Vue.js 3.
- Installer Tailwind CSS et le configurer avec Vue.js.
- S'assurer que l'application est responsive dès le départ.

Récupérer la liste des Pokémon

- Utiliser l'endpoint suivant pour obtenir la liste des Pokémon :

```bash
https://pokeapi.co/api/v2/pokemon?limit=151
```

  (Cela vous donnera les 151 premiers Pokémon.)

- Afficher les noms et les images de chaque Pokémon dans une grille ou une liste.

Afficher les images des Pokémon

- Chaque Pokémon possède des images accessibles via les données renvoyées.
- Vous pouvez également utiliser les sprites officiels ou les artworks en haute résolution disponibles dans l'API.

Fonctionnalité de recherche (obligatoire)

- Implémenter une barre de recherche permettant aux utilisateurs de trouver un Pokémon par son nom ou son numéro.
- La recherche doit être dynamique et filtrer les résultats en temps réel.

Ajouter des filtres (optionnel mais encouragé)

- Permettre aux utilisateurs de filtrer la liste des Pokémon par type (Feu, Eau, Plante, etc.).
- Utiliser l'endpoint des types pour récupérer les informations nécessaires :

```bash
https://pokeapi.co/api/v2/type/
```

Créer une fiche détaillée pour chaque Pokémon

- Lorsque l'utilisateur clique sur un Pokémon, afficher une fiche complète avec :
  - **Plusieurs images :** Inclure différents sprites ou formes du Pokémon.
  - **Taille et poids**
  - **Type(s)**
  - **Zones où il apparaît**
  - **Évolutions**
  - **Statistiques de base** (attaque, défense, vitesse, etc.)
- Utiliser les endpoints appropriés de l'API pour récupérer ces informations supplémentaires.

Navigation et interactivité

- Permettre à l'utilisateur de naviguer facilement entre la liste principale et les fiches détaillées.
- Ajouter des animations ou des transitions pour améliorer l'expérience utilisateur.

Responsive Design

- Utiliser Tailwind CSS pour créer un design attrayant et responsive.
- Tester l'application sur différents appareils (mobile, tablette, desktop).

Améliorations optionnelles

- **Pagination :** Implémenter une pagination pour charger plus de Pokémon au-delà des 151 premiers.
- **Gestion des erreurs :** Afficher des messages d'erreur conviviaux en cas de problème avec les requêtes API.
- **Optimisation des performances :** Mettre en cache les requêtes ou utiliser le lazy loading pour les images.

Consignes techniques

- **Framework :** Utiliser Vue.js 3 pour développer l'application.
- **Design :** Utiliser Tailwind CSS pour le style et s'assurer que le site est responsive.

- **Composants :** Structurer l'application en composants Vue pour une meilleure organisation (par exemple, un composant pour la liste, un pour la fiche détaillée, un pour la barre de recherche, un pour les filtres).
- **Appels API :** Utiliser `fetch` avec `async/await` pour effectuer les requêtes HTTP.
- **Fonctionnalité de recherche :** Implémenter une recherche efficace qui filtre les Pokémon en temps réel.
- **Filtres (optionnel) :** Ajouter des filtres par type de Pokémon pour améliorer l'expérience utilisateur.
- **Images multiples :** Inclure plusieurs images dans les fiches individuelles pour enrichir le contenu.

Ressources utiles

- **Documentation de PokéAPI :** https://pokeapi.co/docs/v2
- **Guide Vue.js 3 :** https://vuejs.org/guide/introduction.html
- **Utilisation de fetch avec async/await :** MDN Web Docs - Fetch API
- **Tailwind CSS :** https://tailwindcss.com/docs

CHAPITRE 3

Inertia.js et les Nouveaux Laravel Starter Kits : Introduction

- **Concept** : Inertia.js demeure cette bibliothèque géniale qui fait le lien entre ton backend Laravel chéri et les frameworks frontend (Vue.js, React, Svelte).
- **Fonctionnement** : Toujours pas besoin d'API REST séparée, tu gardes tes routes et contrôleurs Laravel comme un vrai artisan. ✨
- **Avantages** :
  - **Approche Hybride** : Le meilleur des deux mondes - la puissance de Laravel + la réactivité des interfaces modernes.
  - **Développement Simplifié** : Zéro API à gérer = moins de prises de tête (même pour toi 😉).
  - **Expérience Utilisateur Premium** : Tes applis réagissent comme des SPAs de compétition.



Laravel Starter Kits : La Nouvelle Ère (Bye Bye Jetstream 👋)

Mars 2025 a marqué un tournant : Laravel a **viré Jetstream ET Breeze** au profit de **3 starter kits dédiés**, chacun avec son propre repo standalone. Fini le package qu'on installe après coup, maintenant t'as **tout le code dans ton projet dès le début**.

Les 3 Nouveaux Starter Kits :

1. **Vue Starter Kit** 💚 *(ton chouchou visiblement)*
   - Construit avec **Vue 3** + **Inertia.js**

- Utilise **shadcn-vue** pour les composants UI
- **Tailwind CSS v4** intégré
- Parfait si tu kiffes la syntaxe Vue et ses Single File Components

2. **React Starter Kit** ⚛️
   - Basé sur **React** + **Inertia.js**
   - Composants **shadcn/ui** (Tailwind v4)
   - Pour ceux qui veulent l'écosystème React (les traîtres quoi 🤡)
3. **Livewire Starter Kit** ⚡
   - **Livewire** + **Blade** (full PHP, zéro JS)
   - Composants **Livewire Flux** (gratuits)
   - Pour les puristes qui veulent rester côté serveur

Fonctionnalités Incluses (Tous les Kits) :

- ✅ Authentification complète (login, register, reset password)
- ✅ Gestion du profil utilisateur
- ✅ Dark mode support
- ✅ Multiples layouts
- ✅ Tailwind CSS v4

Options Supplémentaires :

- Vérification email (`MustVerifyEmail`)
- Confirmation de mot de passe (`password.confirm`)
- Intégration WorkOS (authentification tierce)

Pourquoi C'est Mieux que Jetstream/Breeze ? 🎯

1. **Application complète out-of-the-box** : Plus de package à installer, tout est déjà là.
2. **Simplicité** : Un starter kit = un stack. Fini le "choose your own adventure" qui te faisait perdre 3h.
3. **Personnalisation facile** : Tout le code est dans **ton** projet (`app/` , `resources/`), tu modifies ce que tu veux sans te soucier des mises à jour de package.
4. **Bibliothèques modernes** : shadcn, Flux... du matos pro dès le départ.

Installation Éclair ⚡

```bash
laravel new mon-projet-de-ouf
```

Et tu choisis ton starter kit. C'est tout. Même toi tu devrais y arriver ([redacted]'-')[redacted]

Conclusion

Les nouveaux **Laravel Starter Kits** ont enterré Jetstream en beauté 🪦. Inertia.js reste au cœur des stacks **Vue** et **React**, offrant cette magie d'une SPA sans API. Que tu choisisses Vue (le bon choix 💅), React (mouais...) ou Livewire (le côté obscur), t'as maintenant des bases solides, modernes et **100% customisables**.

Développer avec Laravel en BIG 2025, c'est enfin aussi smooth qu'on te l'avait promis.

Ressources Officielles 📚

- Documentation Laravel Starter Kits
- Documentation Inertia.js
- Article de lancement des Starter Kits
- Vue Starter Kit - Repo GitHub

3.1 Pourquoi utiliser Inertia ?

Introduction Générale

Dans le développement web, combiner **Laravel** (backend) et **Vue.js** (frontend) est courant pour créer des applications dynamiques. Traditionnellement, ces deux parties communiquent via des requêtes AJAX, en utilisant des outils comme **Axios** ou **Fetch**.

Problèmes de la méthode traditionnelle :

- Duplication de logique (validation, gestion des données)
- Gestion complexe des états côté client et serveur
- Code répétitif pour gérer erreurs, chargement, succès...

Inertia.js propose une alternative moderne, agissant comme un "pont" entre Laravel et Vue.js. Il élimine le besoin d'API REST ou GraphQL et permet de créer des applications fluides et réactives sans sacrifier la simplicité d'un monolithe.

Dans ce chapitre, nous verrons :

1. La méthode traditionnelle avec Axios/Fetch, illustrée par des exemples
2. Les avantages d'Inertia.js et comment il simplifie ces interactions
3. Le nouveau composant `<Form>` d'Inertia (l'approche recommandée)
4. Le helper `useForm` pour un contrôle programmatique avancé

Méthode Traditionnelle

Dans une architecture classique, le frontend (Vue.js) communique avec le backend (Laravel) via des appels API. Ces interactions se font souvent à l'aide de bibliothèques comme **Axios** ou l'API native **Fetch**.



Récupérer les données d'une page

Prenons l'exemple d'une page qui affiche une liste d'articles. Avec la méthode traditionnelle :

1. Une requête est envoyée depuis le frontend pour récupérer les données
2. Les données reçues sont stockées dans une variable locale pour être affichées

Code exemple avec Axios :

```vue
<script setup>
import { ref, onMounted } from 'vue';
import axios from 'axios';

const articles = ref([]);

onMounted(async () => {
  const response = await axios.get('/api/articles'); // Requête API
  articles.value = response.data; // Stockage des données
});
</script>

<template>
  <div>
    <h1>Articles</h1>
    <ul>
      <li v-for="article in articles" :key="article.id">{{ article.title }}</li>
    </ul>
  </div>
</template>
```

Soumettre un formulaire

Pour envoyer un formulaire (par exemple, pour créer un nouvel article) :

1. Les données sont collectées et envoyées au backend via une requête POST
2. Le frontend doit gérer manuellement les erreurs de validation renvoyées par le backend

Code exemple avec Axios :

```vue
<script setup>
import { ref } from 'vue';
import axios from 'axios';

const formData = ref({ title: '', content: '' });
const errors = ref({});

const submitForm = async () => {
  try {
    const response = await axios.post('/api/articles', formData.value); // Requête POST
    console.log(response.data); // Succès
  } catch (err) {
    errors.value = err.response.data.errors || {}; // Gestion des erreurs
  }
};
</script>

<template>
  <form @submit.prevent="submitForm">
    <input v-model="formData.title" placeholder="Title" />
    <textarea v-model="formData.content" placeholder="Content"></textarea>
    <button type="submit">Submit</button>
    <div v-if="errors.title">{{ errors.title }}</div>
    <div v-if="errors.content">{{ errors.content }}</div>
  </form>
</template>
```

Limites de cette méthode

- **Gestion manuelle des états** : Il faut écrire du code pour gérer les transitions entre états (chargement, succès, erreurs)
- **Duplication de logique** : La validation des données doit être implémentée à la fois côté backend (Laravel) et côté frontend (Vue.js)
- **Complexité croissante** : Avec plusieurs pages ou formulaires, la gestion devient plus difficile à maintenir

C'est pour répondre à ces défis qu'**Inertia.js** propose une approche différente, simplifiant considérablement ces interactions.

Introduction à Inertia.js

Inertia.js simplifie la gestion des interactions entre Laravel (backend) et Vue.js (frontend). Plutôt que de communiquer via des appels API explicites, Inertia agit comme un "pont" qui transmet directement les données et affiche les composants Vue.js sans recharger la page.

Fonctionnement d'Inertia

Avec Inertia :

1. Le backend génère des réponses qui incluent les données nécessaires et le composant Vue.js correspondant
2. Les transitions entre les pages se font sans rechargement complet, offrant une expérience utilisateur fluide
3. La logique est centralisée : la gestion des données, des erreurs et des transitions est simplifiée

Exemple : Récupérer les données d'une page

Inertia permet de transmettre les données directement du backend au frontend sans besoin de requêtes API explicites.

Backend Laravel :

```php
use Inertia\Inertia;

Route::get('/articles', function () {
    articles=Article::all();returnInertia::render('Articles',['articles'⇒articles =
Article::all();
    return Inertia::render('Articles', ['articles' ⇒
articles=Article::all();returnInertia::render('Articles',['articles'⇒articles]);
});
```

Frontend Vue.js :

```vue
<script setup>
defineProps(['articles']);
</script>

<template>
  <div>
    <h1>Articles</h1>
    <ul>
      <li v-for="article in articles" :key="article.id">{{ article.title }}</li>
    </ul>
  </div>
</template>
```

Dans cet exemple, Inertia transmet les données `$articles` directement au composant Vue, éliminant le besoin d'un appel API côté client.

Le Composant `<Form>` : L'Approche Moderne (2025+)

Depuis 2025, Inertia propose un **composant** `<Form>` qui simplifie drastiquement la création de formulaires. C'est désormais **l'approche recommandée** pour la majorité des cas d'usage.

Fonctionnement

Le composant `<Form>` se comporte comme un formulaire HTML classique, mais utilise Inertia en arrière-plan pour éviter les rechargements complets de page. **Pas besoin de** `v-model` ni de gestion manuelle des données : il suffit de donner un attribut `name` à chaque input.

Backend Laravel :

```php
use Illuminate\Http\Request;

Route::post('/articles', function (Request $request) {
    $validated = $request->validate([
        'title' => 'required|string|max:255',
        'content' => 'required',
    ]);

    Article::create($validated);

    return redirect()->route('articles'); // Redirection après succès
});
```

Frontend Vue.js :

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
</script>

<template>
  <Form action="/articles" method="post">
    <input type="text" name="title" placeholder="Title" />
    <textarea name="content" placeholder="Content"></textarea>
    <button type="submit">Create Article</button>
  </Form>
</template>
```

✨ **C'est tout !** Pas de `v-model`, pas de gestion manuelle de state. Le composant gère automatiquement la soumission, les erreurs et les états.

Afficher les Erreurs et États

Le composant `<Form>` expose des **slot props** pour accéder aux erreurs, états de chargement, etc.

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
</script>

<template>
  <Form
    action="/articles"
    method="post"
    #default="{ errors, processing, wasSuccessful }"
  >
    <input type="text" name="title" placeholder="Title" />
    <div v-if="errors.title">{{ errors.title }}</div>

    <textarea name="content" placeholder="Content"></textarea>
    <div v-if="errors.content">{{ errors.content }}</div>

    <button type="submit" :disabled="processing">
      {{ processing ? 'Creating...' : 'Create Article' }}
    </button>

    <div v-if="wasSuccessful" class="success">Article created! 🎉</div>
  </Form>
</template>
```



Fonctionnalités Avancées

Le composant `<Form>` supporte :

- **Structures de données imbriquées** : `name="report[description]"` ou `name="user.email"`
- **Upload de fichiers** : `<input type="file" name="documents" multiple />`
- **Transformation des données** : `:transform="data => ({ ...data, user_id: 123 })"`
- **Réinitialisation automatique** : `resetOnSuccess` ou `resetOnError`

- **Dotted key notation** : `name="user.profile.bio"` crée automatiquement `{ user: { profile: { bio: "..." } } }`

Le Helper `useForm` : Pour un Contrôle Programmatique

Si tu as besoin d'un contrôle plus fin sur tes formulaires (modifications dynamiques, logique complexe), Inertia fournit toujours le **helper** `useForm`.

Frontend Vue.js :

```vue
<script setup>
import { useForm } from '@inertiajs/vue3';

const form = useForm({
  title: '',
  content: '',
});

const submitForm = () => {
  form.post('/articles', {
    onSuccess: () => form.reset(), // Réinitialise après succès
  });
};
</script>

<template>
  <form @submit.prevent="submitForm">
    <input v-model="form.title" placeholder="Title" />
    <div v-if="form.errors.title">{{ form.errors.title }}</div>

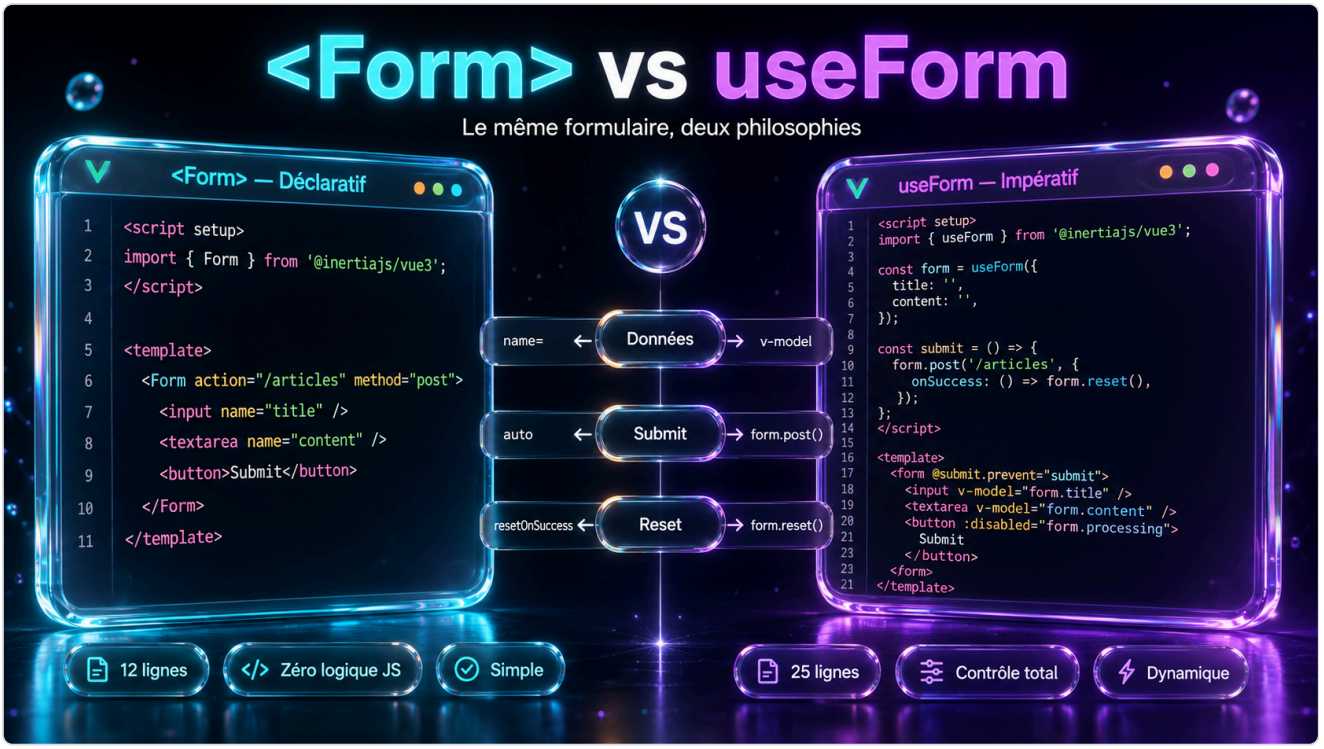
    <textarea v-model="form.content" placeholder="Content"></textarea>
    <div v-if="form.errors.content">{{ form.errors.content }}</div>

    <button :disabled="form.processing" type="submit">
      {{ form.processing ? 'Submitting...' : 'Submit' }}
    </button>
  </form>
</template>
```

Quand utiliser `useForm` ?

- Tu as besoin de manipuler les données dynamiquement avant soumission
- Tu veux un contrôle programmatique total (transformations, validations client, etc.)
- Ton formulaire a une logique métier complexe

Pour les cas simples, préfère le composant `<Form>` ! 🎯



Wayfinder : Le Type-Safety pour tes Routes (Bonus)

Wayfinder est une fonctionnalité (relativement récente) qui génère des fonctions TypeScript type-safe pour tes routes Laravel. Au lieu d'écrire des URLs en dur, tu peux faire :

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
import { store } from 'App/Http/Controllers/ArticleController'; // Import auto-généré
</script>

<template>
  <Form :action="store()">
    <input type="text" name="title" />
    <button type="submit">Create</button>
  </Form>
</template>
```

Avantages :

- Type-safety : ton IDE t'avertit si une route n'existe pas
- Refactoring sécurisé : renommer une route met à jour tous les appels
- Pas de magic strings : finies les erreurs de typo sur `/artcles` au lieu de `/articles`

Wayfinder fonctionne avec le composant `<Form>`, le helper `useForm`, et les appels `router` manuels. On lui consacrera un chapitre dédié plus tard. ☺

Avantages d'Inertia.js

- **Transitions fluides** : Les changements de page ne nécessitent pas de rechargement complet

- **Gestion simplifiée des formulaires** : Le composant `<Form>` réduit drastiquement le code nécessaire
- **Moins de duplication** : La logique de validation reste côté backend, sans réplication frontend
- **Architecture monolithique** : Pas besoin d'API REST séparée, tout reste dans Laravel
- **Developer Experience** : Avec Wayfinder, tu as même du type-safety sur tes routes

Avec ces fonctionnalités, Inertia facilite la création d'applications modernes tout en conservant la simplicité d'une architecture monolithique. En 2025, c'est **LE** standard pour du Laravel + Vue.js. 🚀

3.2 Installation de Laravel avec un Starter Kit (Vue + Inertia.js)

Pour intégrer Inertia.js dans un nouveau projet Laravel, Laravel propose désormais des **Starter Kits** clés en main qui simplifient drastiquement l'installation. Fini les étapes interminables, tout se fait en quelques commandes ! 🚀

Prérequis

Avant de commencer, assurez-vous d'avoir installé :

- **PHP** (version 8.2 ou supérieure)
- **Composer**
- **Node.js** et **npm**

Étape 1 : Installation du Laravel Installer

Le Laravel Installer est un outil CLI qui facilite la création de projets Laravel. Si vous ne l'avez pas encore, installez-le globalement via Composer :

```bash
composer global require laravel/installer
```

> ***Note** : Assurez-vous que le répertoire des binaires globaux de Composer est dans votre PATH système pour pouvoir utiliser la commande `laravel`.*

Étape 2 : Création du Projet

Créez un nouveau projet Laravel avec le Starter Kit interactif. Dans votre terminal, exécutez :

```bash
laravel new mon-projet
```

Remplacez `mon-projet` par le nom de votre application.

Étape 3 : Sélection Interactive des Options

Le Laravel Installer va vous poser plusieurs questions pour configurer votre projet. Voici ce que vous allez rencontrer :

1. Choix du Starter Kit

```
Which starter kit would you like to install? [None]:
  [none     ] None
  [react    ] React
  [vue      ] Vue
  [livewire ] Livewire
```

Sélectionnez `vue` pour utiliser Vue.js avec Inertia.js. ✅

2. Choix du Provider d'Authentification

```
Which authentication provider do you prefer? [Laravel's built-in authentication]:
  [laravel] Laravel's built-in authentication
  [workos ] WorkOS (Requires WorkOS account)
  [none   ] No authentication scaffolding
```

Sélectionnez `laravel` pour utiliser le système d'authentification natif de Laravel (recommandé pour la plupart des projets).

3. Choix du Framework de Tests

```
Which testing framework do you prefer? [Pest]:
  [0] Pest
  [1] PHPUnit
```

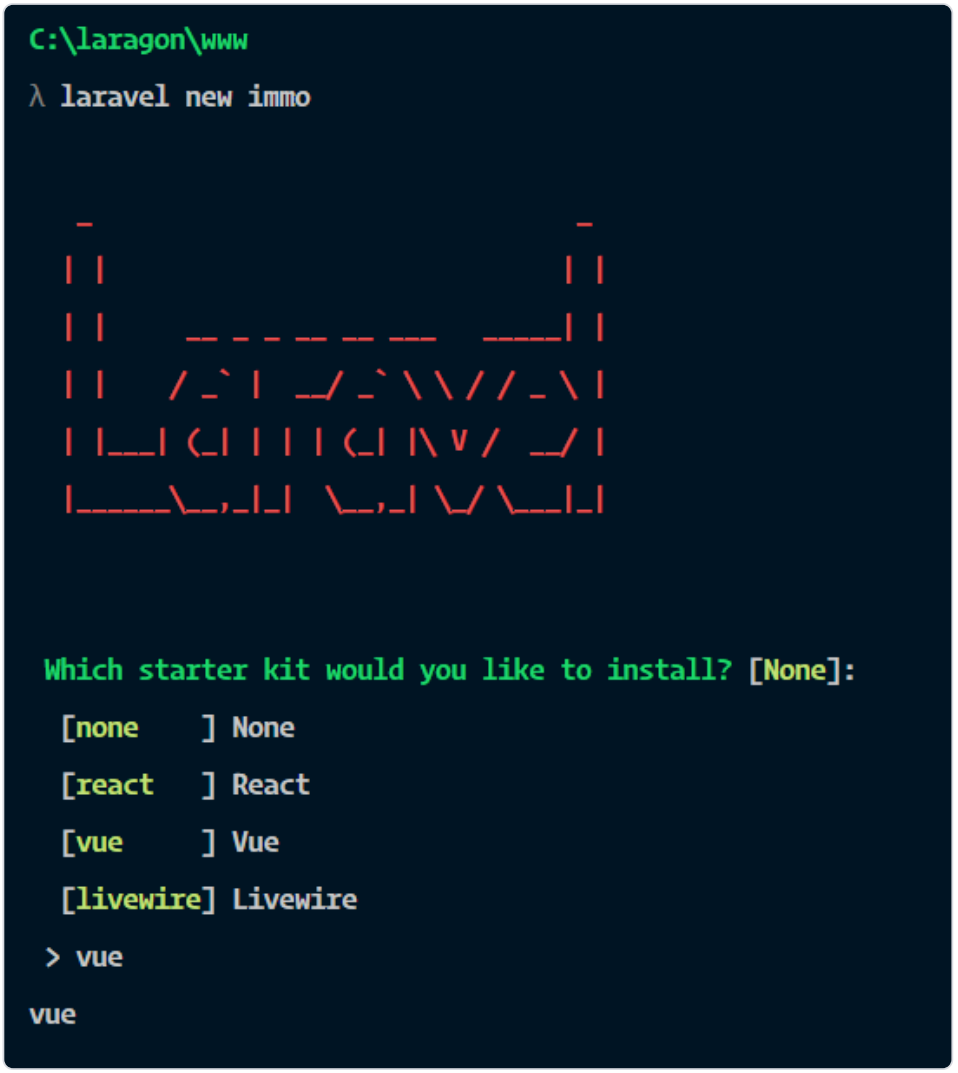
Sélectionnez `Pest` (c'est la recommandation moderne de Laravel). Si tu préfères PHPUnit, choisis-le, mais bon... Pest c'est mieux 😏

4. Laravel Boost (Optionnel)

BASH

```
Do you want to install Laravel Boost to improve AI assisted coding? (yes/no) [yes]:
```

Sélectionnez `no` pour un projet standard (sauf si tu veux expérimenter avec les fonctionnalités d'assistance IA).


C:\laragon\www
λ laravel new immo
Which starter kit would you like to install? [None]:
[none] None
[react] React
[vue] Vue
[livewire] Livewire
> vue
vue

```
 Which authentication provider do you prefer? [Laravel's built-in authentication]:
  [laravel] Laravel's built-in authentication
  [workos ] WorkOS (Requires WorkOS account)
  [none   ] No authentication scaffolding
 > laravel
laravel



 Which testing framework do you prefer? [Pest]:
  [0] Pest
  [1] PHPUnit
 >




 Do you want to install Laravel Boost to improve AI assisted coding? (yes/no) [yes]:
 > no

    Creating a "laravel/vue-starter-kit" project at "./side-projects-graveyard"
    Installing laravel/vue-starter-kit (dev-main 6510b268c8b65a9b9ffa79dbd6eb78d00a1ffd76)
  - Downloading laravel/vue-starter-kit (dev-main 6510b26)
      - Installing laravel/vue-starter-kit (dev-main 6510b26): Extracting archive
    Created project in C:\laragon\www/side-projects-graveyard
```

Étape 4 : Installation et Compilation

Une fois les options sélectionnées, le Laravel Installer va :

1. Télécharger et installer le starter kit Vue
2. Configurer Inertia.js
3. Installer toutes les dépendances nécessaires

Étape 5 : Finaliser l'Installation

Une fois le projet créé, naviguez dans le dossier :

```bash
cd mon-projet
```

Installez les dépendances frontend et lancez le build :

```bash
npm install && npm run build
```

Enfin, démarrez le serveur de développement Laravel :

```bash
composer run dev
```

Configuration Optionnelle

Désactiver la Double Authentification pour l'Utilisateur de Test

Les nouveaux starter kits activent la double authentification (2FA) **par défaut** pour les utilisateurs créés via les seeders. Si vous voulez désactiver cette fonctionnalité pour votre utilisateur de test, modifiez le fichier `database/seeders/DatabaseSeeder.php` :

```php
<?php

namespace Database\Seeders;

use App\Models\User;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    public function run(): void
    {
        User::factory()->withoutTwoFactor()->create([
            'name' => 'Test User',
            'email' => 'test@example.com',
        ]);
    }
}
```

Note : La méthode `withoutTwoFactor()` désactive explicitement la 2FA pour cet utilisateur. 🔓

Ensuite, exécutez le seeder :

```bash
php artisan db:seed
```

Structure du Projet

Après l'installation, votre projet contiendra :

- **Backend Laravel** : Controllers, routes, modèles dans `app/`, `routes/`, etc.
- **Frontend Vue.js** : Composants, pages et layouts dans `resources/js/`
- **Inertia.js** : Configuré et prêt à l'emploi pour lier Laravel et Vue
- **Tailwind CSS v4** : Préconfiguré avec shadcn-vue pour des composants UI modernes
- **Authentification complète** : Login, register, password reset, profil utilisateur, 2FA

Arborescence Frontend (Vue)

```bash
resources/js/
├── components/    # Composants Vue réutilisables
├── composables/   # Composables / hooks Vue
├── layouts/       # Layouts d'application (sidebar, header, auth)
├── lib/           # Fonctions utilitaires et configuration
├── pages/         # Pages de l'application (composants Inertia)
└── types/         # Définitions TypeScript
```

Layouts Disponibles

Le Vue Starter Kit propose **plusieurs layouts prédéfinis** que tu peux personnaliser :

Layouts Principaux

- **Sidebar Layout** (par défaut) : Barre latérale pour la navigation
- **Header Layout** : Navigation en en-tête horizontal

Pour changer de layout, modifie `resources/js/layouts/AppLayout.vue` :

```javascript
// Par défaut (sidebar)
import AppLayout from '@/layouts/app/AppSidebarLayout.vue';

// Alternative (header)
import AppLayout from '@/layouts/app/AppHeaderLayout.vue';
```

Variantes de Sidebar

- `variant="sidebar"` (par défaut)
- `variant="inset"`
- `variant="floating"`

Modifie `resources/js/components/AppSidebar.vue` pour changer la variante :

```vue
<Sidebar collapsible="icon" variant="sidebar">
```

Layouts d'Authentification

- **Simple** (par défaut)
- **Card**
- **Split**

Pour changer le layout d'authentification, modifie `resources/js/layouts/AuthLayout.vue` :

```javascript
// Par défaut (simple)
import AuthLayout from '@/layouts/auth/AuthSimpleLayout.vue';

// Alternatives
import AuthLayout from '@/layouts/auth/AuthSplitLayout.vue';
```

Conclusion

Avec cette nouvelle approche, installer Laravel + Vue.js + Inertia.js est devenu **ridiculement simple** 🎯 :

1. `composer global require laravel/installer`
2. `laravel new mon-projet`
3. Sélectionner les options interactivement
4. `npm install && npm run build`
5. `composer run dev`

C'est tout ! Plus besoin de jongler avec Jetstream, Breeze, ou des commandes Artisan obscures. L'installation est interactive, moderne, et te donne un projet **production-ready** dès le départ.

Tu peux maintenant te concentrer sur **construire ton application** plutôt que de configurer ton environnement pendant des heures. Bienvenue en 2025 🚀

Ressources

- Documentation Laravel Starter Kits
- Documentation Inertia.js
- Article de lancement des Starter Kits
- Repo GitHub Vue Starter Kit

3.3 shadcn/ui et shadcn-vue : Une Nouvelle Approche des Composants UI

Qu'est-ce que shadcn-vue ?

shadcn-vue est une collection de composants UI réutilisables, conçus avec Vue.js et Tailwind CSS. Contrairement aux bibliothèques de composants traditionnelles, shadcn-vue adopte une approche radicalement différente : les composants ne sont **pas installés comme une dépendance npm**, mais **copiés directement dans votre projet**.

Cette approche vous donne un contrôle total sur vos composants, car ils deviennent une partie intégrante de votre code source.

La Philosophie "Copy, Paste, Own"

Différence avec les Bibliothèques Classiques

Avec des bibliothèques comme Bootstrap, Vuetify ou Element Plus, vous installez un package npm qui contient tous les composants. Ces composants restent dans `node_modules` et vous les importez comme des dépendances externes.

Avec shadcn-vue, le processus est différent :

1. Vous sélectionnez les composants dont vous avez besoin
2. Vous les installez via CLI, qui les copie dans votre projet
3. Les composants sont placés dans `resources/js/components/ui/`
4. Vous pouvez les modifier librement, car ils font partie de votre code

Avantages de Cette Approche

Aspect	Bibliothèques Classiques	shadcn-vue
Customisation	Limitée (via props ou CSS override)	Totale (modification directe du code)
Taille du bundle	Toute la bibliothèque	Uniquement les composants utilisés
Dépendances	Package externe à maintenir	Code source sous votre contrôle
Apprentissage	API spécifique à apprendre	Code Vue.js standard à lire et comprendre

Pourquoi shadcn-vue dans les Starter Kits Laravel ?

Les nouveaux Laravel Starter Kits (React et Vue) incluent shadcn/shadcn-vue pour plusieurs raisons :

✅ **Design moderne et professionnel** : Les composants suivent les standards d'accessibilité (ARIA) et les bonnes pratiques de design
✅ **Intégration Tailwind CSS** : Tous les composants utilisent Tailwind, assurant une cohérence visuelle avec votre projet
✅ **Facilité de customisation** : Modifier l'apparence d'un composant se fait en changeant les classes Tailwind directement dans le fichier
✅ **Pas de surcharge** : Vous n'installez que les composants dont vous avez besoin
✅ **TypeScript natif** : Types inclus pour une meilleure autocomplétion et détection d'erreurs

Installation et Utilisation

Ajouter un Nouveau Composant

Le Vue Starter Kit est préconfigurée avec shadcn-vue. Pour ajouter un composant, utilisez la CLI :

```bash
npx shadcn-vue@latest add button
```

Cette commande va :

- Télécharger le composant `Button.vue`
- Le placer dans `resources/js/components/ui/button/`
- Installer les dépendances nécessaires si besoin

Utiliser le Composant

Une fois installé, importez et utilisez le composant comme n'importe quel composant Vue :

```vue
<script setup lang="ts">
import { Button } from '@/components/ui/button';
</script>

<template>
  <div>
    <Button>Cliquez ici</Button>
    <Button variant="destructive">Supprimer</Button>
    <Button variant="outline" size="sm">Petit bouton</Button>
  </div>
</template>
```

Personnaliser un Composant

Le fichier du composant se trouve maintenant dans votre projet. Pour le personnaliser, ouvrez-le et modifiez-le :

Chemin du fichier : `resources/js/components/ui/button/Button.vue`

```vue
<template>
  <button
    :class="cn(
      'inline-flex items-center justify-center rounded-md',
      'bg-blue-600 hover:bg-blue-700', // Modifiez ces classes
      // ... autres classes
    )"
  >
    <slot />
  </button>
</template>
```

Toutes les modifications s'appliqueront immédiatement à tous les usages du composant dans votre application.

Composants Disponibles

shadcn-vue propose une large gamme de composants :

Composants de Base :

- Button, Input, Textarea, Checkbox, Radio, Select
- Label, Badge, Avatar, Separator

Navigation :

- Tabs, Breadcrumb, Pagination, Command, NavigationMenu

Overlays & Modales :

- Dialog (Modal), Sheet (Drawer), Popover, Tooltip, AlertDialog

Affichage de Données :

- Table, Card, Alert, Toast, Progress

Formulaires :

- Form, Calendar, Date Picker, Combobox

Layout :

- Accordion, Collapsible, Sidebar

La liste complète est disponible sur shadcn-vue.com.

Exemple Pratique : Ajouter un Switch

Imaginons que vous souhaitiez ajouter un composant Switch (interrupteur) pour activer/désactiver le mode sombre.

Étape 1 : Installer le composant

```bash
npx shadcn-vue@latest add switch
```

Étape 2 : Utiliser le composant

```vue
<script setup lang="ts">
import { Switch } from '@/components/ui/switch';
import { ref } from 'vue';

const darkMode = ref(false);
</script>

<template>
  <div class="flex items-center space-x-2">
    <Switch v-model:checked="darkMode" id="dark-mode" />
    <label for="dark-mode" class="cursor-pointer">Mode sombre</label>
  </div>
</template>
```

Le composant est maintenant intégré à votre projet et fonctionne immédiatement.

Les Layouts du Starter Kit

Le Vue Starter Kit utilise shadcn-vue de manière intensive. Les différents layouts (sidebar, header, auth) sont construits avec des composants shadcn-vue :

- **AppSidebar.vue** → Utilise le composant `Sidebar` de shadcn-vue
- **Formulaires d'authentification** → Utilisent `Form`, `Input`, `Button`

- **Navigation** → Utilise `NavigationMenu`, `DropdownMenu`

Ces composants sont déjà installés et configurés. Vous pouvez les explorer dans le dossier `resources/js/components/` pour comprendre leur fonctionnement.

Bonnes Pratiques

Explorer les Composants Installés

Prenez le temps d'ouvrir les fichiers des composants dans `resources/js/components/ui/`. C'est une excellente opportunité d'apprendre :

- Comment structurer des composants Vue.js réutilisables
- Comment utiliser Tailwind CSS efficacement
- Comment implémenter l'accessibilité (attributs ARIA)

Modifier avec Prudence

Bien que vous puissiez modifier les composants librement, réfléchissez aux conséquences :

- Une modification affecte **tous** les usages du composant dans votre projet
- Documentez vos changements si vous travaillez en équipe
- Considérez créer une variante plutôt que de modifier le composant de base si vous avez besoin d'un comportement spécifique

Installer uniquement ce dont vous avez besoin

N'installez pas tous les composants "au cas où". Installez-les au fur et à mesure de vos besoins pour garder votre projet léger.

Comparaison : Avant et Après

Avant shadcn-vue

Vous aviez deux options pour créer des interfaces :

1. **Utiliser une bibliothèque complète** (Vuetify, Element Plus)
   - ❌ Personnalisation difficile
   - ❌ Surcharge du bundle
   - ❌ Dépendance à une API externe
2. **Tout coder vous-même**
   - ❌ Long et répétitif
   - ❌ Risque de bugs
   - ❌ Accessibilité souvent négligée

Avec shadcn-vue

Vous bénéficiez des deux approches :

- ✅ Composants de qualité professionnelle, immédiatement disponibles
- ✅ Contrôle total pour les adapter à vos besoins
- ✅ Code source lisible et modifiable
- ✅ Pas de "boîte noire" externe

Conclusion

shadcn-vue représente une évolution dans la manière de concevoir les bibliothèques de composants. En plaçant le code directement dans votre projet, cette approche vous donne :

✅ Un contrôle total sur vos composants
✅ Une meilleure compréhension du code que vous utilisez
✅ Une personnalisation sans friction
✅ Une maintenance simplifiée (pas de version externe à suivre)

Avec les Laravel Starter Kits, shadcn-vue est déjà configuré et prêt à l'emploi. Vous pouvez commencer à construire votre interface immédiatement en installant les composants dont vous avez besoin.

Ressources

- Documentation shadcn-vue
- Documentation shadcn/ui (version React)
- GitHub shadcn-vue
- Radix Vue (primitives sous-jacents)
- Tailwind CSS

3.4 Pages avec Inertia

Inertia.js est un outil qui facilite la création d'applications mono-page (SPA) avec des méthodologies de serveur traditionnelles comme Laravel, sans nécessiter d'API séparée. Il sert de pont entre le backend et le frontend, permettant une expérience utilisateur fluide et dynamique.

Concept de base

Fonctionnement général

Dans une application utilisant Inertia, chaque page correspond à une route et un contrôleur dans le backend, ainsi qu'à un composant JavaScript dans le frontend, spécifiquement Vue.js pour ce cours. Ce modèle permet de gérer le routage et la logique côté serveur tout en offrant une expérience utilisateur comparable à une SPA.



Avantages clés

- **Pas d'API nécessaire** : Inertia transmet directement les données aux composants Vue.js, éliminant le besoin d'une API dédiée.
- **Chargement préalable des données** : Les données nécessaires sont récupérées avant le rendu par le navigateur, supprimant les états de chargement et améliorant l'expérience utilisateur.

Création de pages

Les pages Inertia sont des composants JavaScript, ici spécifiquement des composants Vue.js.

Exemple pratique

Prenons le fichier `User/Show.vue` dans une application Laravel, qui est un composant Vue.js utilisant Inertia.js.

Composant Vue.js

```vue
<script setup>
import Layout from './Layout'
import { Head } from '@inertiajs/vue3'

defineProps({ user: Object })
</script>

<template>
  <Layout>
    <Head title="Welcome" />
    <h1>Welcome</h1>
    <p>Hello {{ user.name }}, welcome to your first Inertia app!</p>
  </Layout>
</template>
```

Ce composant affiche un message de bienvenue à l'utilisateur, avec les données passées comme props.

Contrôleur Laravel

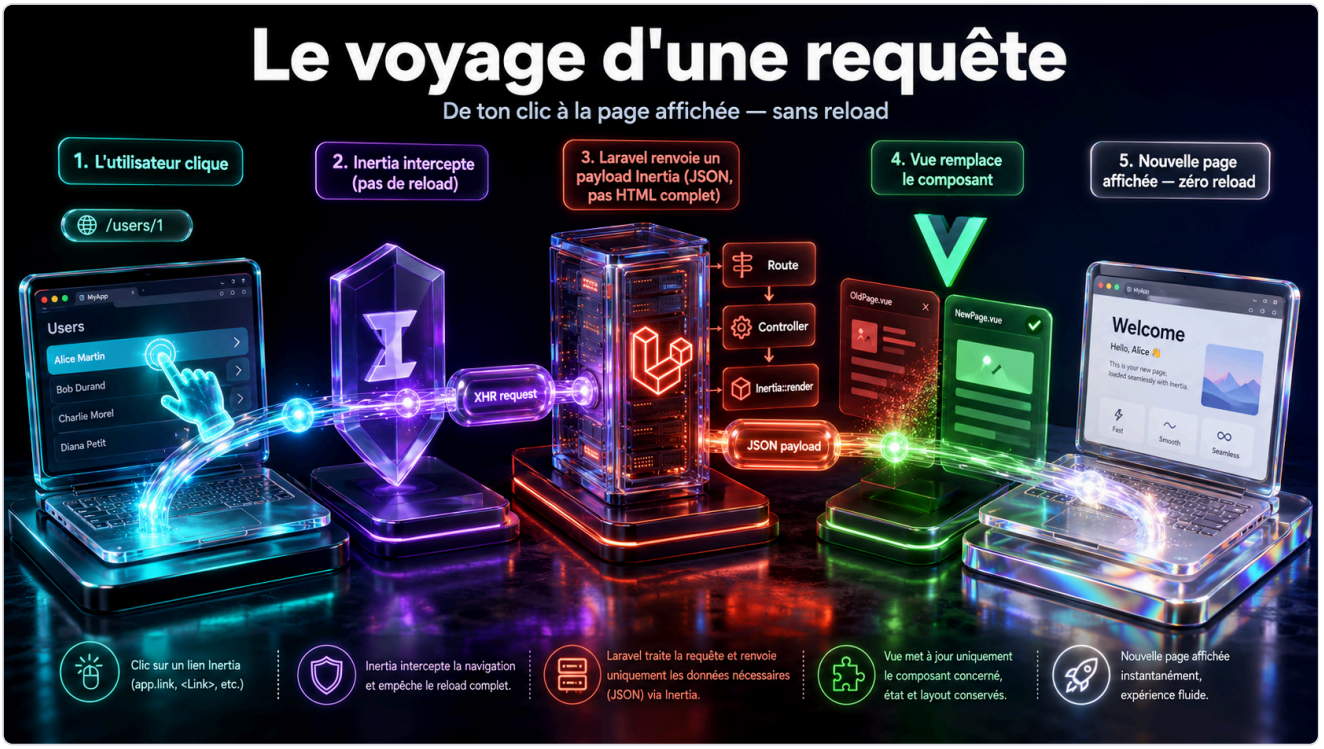
```php
use Inertia\Inertia;

class UserController extends Controller
{
    public function show(User $user)
    {
        return Inertia::render('User/Show', [
          'user' => $user
        ]);
    }
}
```

La méthode `show` renvoie une réponse Inertia qui rend le composant `User/Show.vue`, en passant l'objet `user`.

But de cette approche

Inertia simplifie le développement des SPAs en utilisant des pratiques de développement traditionnelles, réduisant la complexité et augmentant l'efficacité. Elle permet de se concentrer sur la logique métier tout en offrant une interface utilisateur réactive et moderne.



Optimisation des réponses Inertia

Pourquoi optimiser les réponses?

Optimiser les réponses permet de minimiser la latence et d'améliorer la réactivité de l'application. En ne transmettant que les données nécessaires, vous évitez les surcharges inutiles et assurez une expérience utilisateur plus fluide, surtout sur des réseaux lents ou des appareils à faible performance.

Comment optimiser les réponses

Pour illustrer l'optimisation des réponses, utilisons l'exemple d'un contrôleur qui gère l'affichage d'un utilisateur (`User`) dans une application Laravel utilisant Inertia.js.

Exemple de contrôleur optimisé

```php
use Inertia\Inertia;

class UserController extends Controller
{
    public function show(User $user)
    {
        // Création d'une réponse Inertia en sélectionnant uniquement les attributs nécessaires
        return Inertia::render('User/Show', [
            'user' => $user->only('id', 'name', 'email', 'created_at')
        ]);
    }
}
```

Dans cet exemple, la méthode `only` est utilisée pour sélectionner uniquement les attributs spécifiques de l'objet `User` qui sont nécessaires pour la page. Cela évite de transmettre des données superflues comme les mots de passe ou les informations sensibles qui ne sont pas utilisées par la vue.

Avantages de cette approche

- **Réduction de la bande passante** : Moins de données sont transférées entre le serveur et le client.
- **Amélioration des performances** : Le serveur traite moins de données, et le client traite moins de JSON, ce qui accélère le rendu.
- **Sécurité accrue** : En limitant les données exposées aux clients, vous réduisez le risque de fuite d'informations sensibles.



3.5 Routage avec Laravel et Inertia

Le routage dans une application utilisant Laravel et Inertia présente quelques spécificités importantes, notamment en ce qui concerne la gestion des redirections et l'utilisation de helpers pour la génération des URLs. Nous allons explorer ces aspects à travers des explications et des exemples concrets.

Définition des routes

Les routes dans une application Laravel avec Inertia sont définies dans le fichier `web.php`, de manière similaire à une application Laravel traditionnelle. Cependant, l'utilisation d'Inertia implique souvent des interactions spécifiques, surtout pour les requêtes autres que GET.

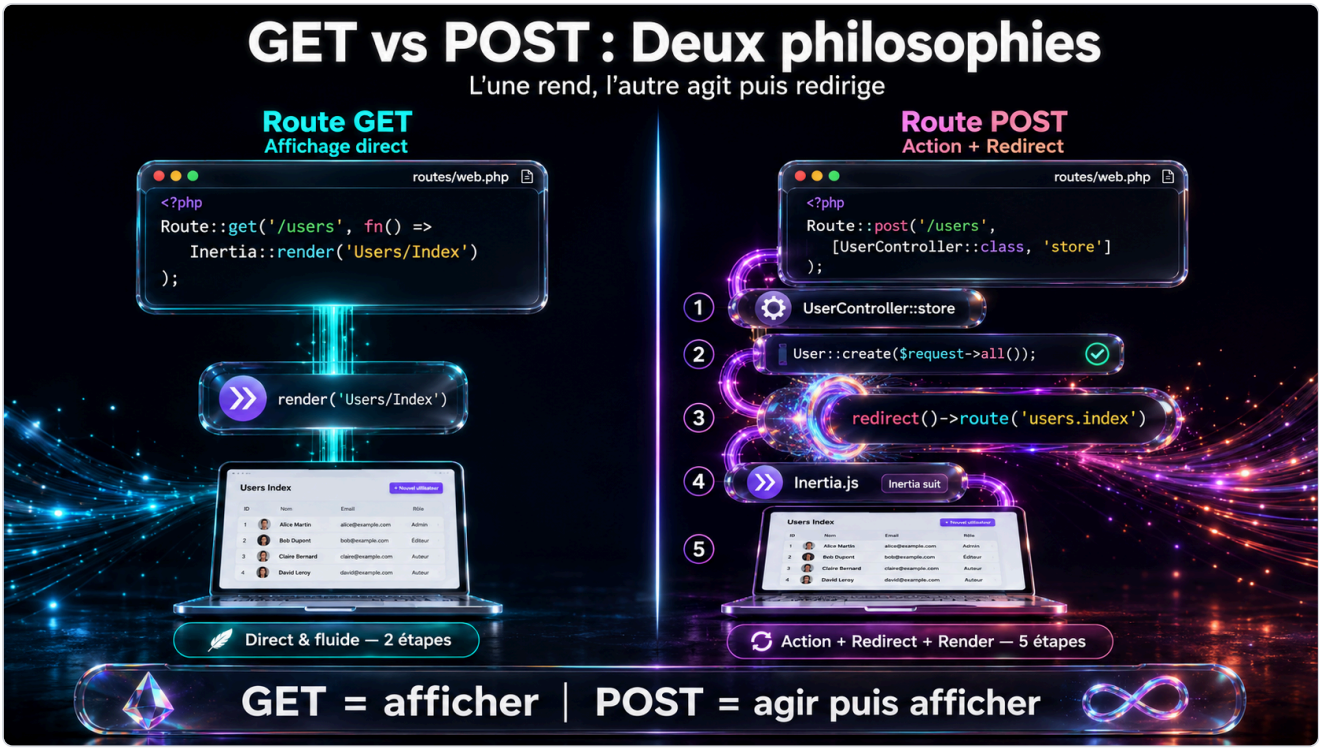
Exemple de définition de route

Voici un exemple de fichier `routes/web.php` dans une application Laravel utilisant Inertia :

```php
use Inertia\Inertia;
use Illuminate\Support\Facades\Route;

Route::get('/users', function () {
    return Inertia::render('Users/Index');
});

Route::post('/users', 'UserController@store');
```



Gestion des redirections

Lors de la création ou de la modification de ressources via des requêtes non-GET, il est crucial de répondre par une redirection appropriée. Inertia s'attend à ce que ces actions soient suivies d'une réponse de redirection pour pouvoir mettre à jour la page de manière appropriée.

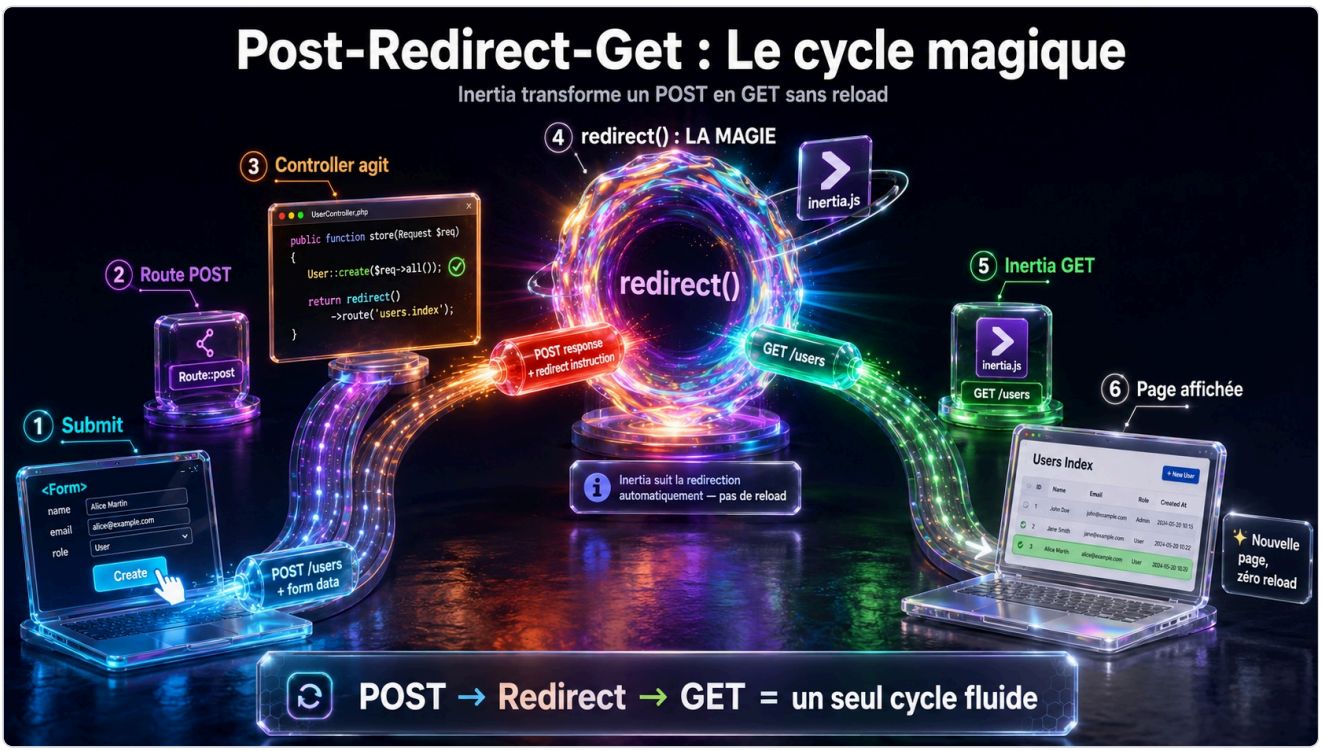
Exemple de redirection après une action

Supposons que vous ayez un contrôleur `UserController` avec une méthode `store` pour créer un nouvel utilisateur. Voici comment vous pourriez gérer la redirection :

```php
public function store(Request $request)
{
    // Logique pour créer un utilisateur
    User::create($request->all());

    // Redirection vers la page d'index des utilisateurs
    return redirect()->route('users.index');
}
```

Dans cet exemple, après la création de l'utilisateur, le contrôleur redirige l'utilisateur vers la page d'index des utilisateurs. Inertia interceptera cette redirection et chargera la page d'index des utilisateurs sans recharger complètement le navigateur.



Utilisation du helper `route()` dans VueJS

Inertia permet également d'utiliser un helper `route()` dans les fichiers VueJS, similaire à celui utilisé dans les vues Blade de Laravel. Ce helper facilite la génération d'URLs pour les liens ou les actions dans les composants Vue.

Exemple d'utilisation du helper `route()`

Dans un composant Vue, vous pourriez utiliser le helper `route()` pour créer un lien vers la page d'index des utilisateurs :

```vue
<script setup>
import { Link } from '@inertiajs/vue3';
</script>

<template>
  <Link :href="route('users.index')">Voir tous les utilisateurs</Link>
</template>
```

Ce code crée un lien qui, lorsqu'il est cliqué, dirigera Inertia pour charger la page d'index des utilisateurs en utilisant la définition de route côté serveur, sans rechargement complet de la page.



3.6 Laravel Wayfinder : Le Type-Safety pour Vos Routes

Qu'est-ce que Wayfinder ?

Laravel Wayfinder est un outil qui génère automatiquement des **fonctions TypeScript typées** pour vos routes et contrôleurs Laravel. Au lieu d'écrire des URLs en dur dans votre code frontend, vous importez des fonctions qui représentent vos routes backend.

Le Problème

Traditionnellement, pour appeler une route Laravel depuis Vue.js, vous écrivez :

```typescript
// ❌ Approche traditionnelle : URLs en dur
router.post('/posts/1', formData);
```

Problèmes avec cette approche :

- Risque de **typos** : `/psots/1` au lieu de `/posts/1`
- Aucune **autocomplétion** : votre IDE ne vous aide pas
- Pas de **vérification des paramètres** : oublier un paramètre obligatoire passe inaperçu
- **Refactoring impossible** : renommer une route côté backend casse le frontend silencieusement



La Solution : Wayfinder

Avec Wayfinder, vous importez des fonctions générées automatiquement :

```typescript
// ✅ Approche Wayfinder : fonctions typées
import { update } from '@/actions/App/Http/Controllers/PostController';

form.submit(update(1)); // Type-safe, autocomplétion, erreurs détectées
```

Avantages :

- ✅ **Type-safety** : TypeScript vérifie les paramètres à la compilation
- ✅ **Autocomplétion** : votre IDE suggère les routes disponibles
- ✅ **Refactoring sécurisé** : renommer une méthode met à jour tous les appels
- ✅ **Zéro duplication** : une seule source de vérité (vos contrôleurs Laravel)

Génération Automatique

Important : Avec le Vue Starter Kit, la génération des définitions TypeScript se fait **automatiquement** lorsque vous lancez :

```bash
composer run dev
```

Cette commande démarre à la fois le serveur Laravel et Vite, qui surveille vos fichiers et régénère les définitions Wayfinder à chaque modification de routes ou contrôleurs.

Fichiers générés :

Wayfinder crée trois dossiers dans `resources/js/` :

- `wayfinder/` : Fichiers de configuration internes
- `actions/` : Fonctions correspondant à vos **méthodes de contrôleurs**
- `routes/` : Fonctions correspondant à vos **routes nommées**

> ***Note** : Ces dossiers sont régénérés automatiquement. Vous pouvez les ajouter à `.gitignore` car ils ne doivent pas être versionnés.*



Utilisation de Base

Importer une Action de Contrôleur

Imaginons que vous avez un contrôleur `PostController` avec une méthode `show` :

```php
// app/Http/Controllers/PostController.php
class PostController extends Controller
{
    public function show(Post $post)
    {
        return Inertia::render('Posts/Show', [
            'post' => $post,
        ]);
    }
}
```

Avec Wayfinder, vous pouvez importer cette action :

```typescript
import { show } from '@/actions/App/Http/Controllers/PostController';

show(1); // Retourne : { url: "/posts/1", method: "get" }
```

Utilisation avec Inertia

Wayfinder s'intègre parfaitement avec Inertia.js :

Avec le composant `<Link>` :

```vue
<script setup lang="ts">
import { Link } from '@inertiajs/vue3';
import { show } from '@/actions/App/Http/Controllers/PostController';
</script>

<template>
  <Link :href="show(1)">Voir l'article #1</Link>
</template>
```

Avec le Form Helper :

```vue
<script setup lang="ts">
import { useForm } from '@inertiajs/vue3';
import { store } from '@/actions/App/Http/Controllers/PostController';

const form = useForm({
  title: 'Mon article',
  content: 'Contenu de l\'article',
});

const submitForm = () => {
  form.submit(store()); // Wayfinder résout automatiquement l'URL et la méthode
};
</script>

<template>
  <form @submit.prevent="submitForm">
    <!-- Vos champs de formulaire -->
  </form>
</template>
```

Avec le composant `<Form>` :

```vue
<script setup lang="ts">
import { Form } from '@inertiajs/vue3';
import { store } from '@/actions/App/Http/Controllers/PostController';
</script>

<template>
  <Form :action="store()">
    <input type="text" name="title" />
    <button type="submit">Créer</button>
  </Form>
</template>
```

Cas d'Usage Avancés

Méthodes Multiples

Si vous souhaitez spécifier une méthode HTTP particulière :

```typescript
import { update } from '@/actions/App/Http/Controllers/PostController';

update.put(1);   // { url: "/posts/1", method: "put" }
update.patch(1); // { url: "/posts/1", method: "patch" }
```

Obtenir Uniquement l'URL

Si vous avez seulement besoin de l'URL (sans l'objet complet) :

```typescript
import { show } from '@/actions/App/Http/Controllers/PostController';

show.url(1); // "/posts/1"
```

Paramètres Multiples

Wayfinder accepte différents formats de paramètres :

```typescript
import { update } from '@/actions/App/Http/Controllers/PostController';

// Tableau de paramètres
update([1, 2]); // Route : /posts/{post}/authors/{author}

// Objet avec noms de paramètres
update({ post: 1, author: 2 });

// Objets imbriqués
update({ post: { id: 1 }, author: { id: 2 } });
```

Route Bindings Personnalisés

Si votre route utilise un champ personnalisé (comme `slug`) :

```php
Route::get('/posts/{post:slug}', [PostController::class, 'show']);
```

Wayfinder détecte cela et accepte le slug directement :

```typescript
import { show } from '@/actions/App/Http/Controllers/PostController';

show('mon-article'); // { url: "/posts/mon-article", method: "get" }
show({ slug: 'mon-article' }); // Alternative avec objet
```

Query Parameters

Vous pouvez ajouter des paramètres de requête :

```typescript
import { show } from '@/actions/App/Http/Controllers/PostController';

const options = {
  query: {
    page: 1,
    sort_by: 'name',
  },
};

show.url(1, options); // "/posts/1?page=1&sort_by=name"
```

Routes Nommées

Wayfinder génère également des fonctions pour vos routes nommées :

```php
Route::get('/posts/{post}', [PostController::class, 'show'])
    ->name('posts.show');
```

Vous pouvez les importer depuis le dossier `routes` :

```typescript
import { show } from '@/routes/posts';

show(1); // { url: "/posts/1", method: "get" }
```

Contrôleurs Invocables

Si vous utilisez des contrôleurs invocables (avec méthode `__invoke`) :

```php
class StorePostController extends Controller
{
    public function __invoke(Request $request)
    {
        // ...
    }
}
```

Importez directement le contrôleur et invoquez-le :

```typescript
import StorePostController from '@/actions/App/Http/Controllers/StorePostController';

StorePostController(); // { url: "/posts", method: "post" }
```

Mots Réservés JavaScript

Si une méthode de votre contrôleur porte un nom qui est un mot réservé JavaScript (comme `delete`, `import`), Wayfinder la renomme automatiquement en ajoutant `Method` :

```php
public function delete(Post $post) { ... }
```

Devient :

```typescript
import { deleteMethod } from '@/actions/App/Http/Controllers/PostController';

deleteMethod(1); // { url: "/posts/1", method: "delete" }
```

Avantages de Wayfinder

Pour le Développement

- **Productivité accrue** : autocomplétion et suggestions de l'IDE
- **Moins d'erreurs** : détection des problèmes à la compilation, pas à l'exécution
- **Refactoring sûr** : renommer une route met à jour automatiquement tous les appels

Pour la Maintenance

- **Source unique de vérité** : les routes sont définies côté Laravel uniquement
- **Synchronisation automatique** : toute modification backend se propage au frontend
- **Documentation vivante** : les types TypeScript documentent les routes disponibles

Pour l'Équipe

- **Moins de communication nécessaire** : les nouveaux développeurs découvrent les routes via l'autocomplétion

- **Standards cohérents** : tout le monde utilise la même approche
- **Onboarding simplifié** : pas besoin de mémoriser les URLs



Bonnes Pratiques

Nommez Vos Routes

Donnez des noms explicites à vos routes pour faciliter leur utilisation :

```php
Route::post('/posts', [PostController::class, 'store'])
    ->name('posts.store');
```

Utilisez le Route Model Binding

Laissez Laravel résoudre automatiquement les modèles :

```php
public function show(Post $post) // Wayfinder génère : show(1)
```

Au lieu de :

```php
public function show($id) // Moins clair
```

Explorez les Fichiers Générés

Prenez le temps d'ouvrir les fichiers dans `resources/js/actions/` pour comprendre ce qui est généré. C'est une excellente façon d'apprendre.

Conclusion

Laravel Wayfinder transforme la manière dont vous appelez vos routes backend depuis votre frontend. En générant automatiquement des fonctions TypeScript typées, il élimine les erreurs de typos, facilite le refactoring, et améliore significativement l'expérience de développement.

Avec les Laravel Starter Kits, Wayfinder est déjà configuré et s'exécute automatiquement. Vous n'avez qu'à utiliser les imports pour profiter d'une intégration backend/frontend sans friction.

Note importante : Wayfinder est actuellement en **Beta**. L'API peut encore évoluer avant la version 1.0.0. Consultez régulièrement le changelog pour les changements.

Ressources

- Documentation officielle Wayfinder
- Wayfinder et Inertia.js
- TypeScript Documentation

3.7 Formulaires avec Inertia.js

Dans ce chapitre, nous allons explorer les deux approches pour gérer les formulaires avec Inertia.js : le **composant** `<Form>` (recommandé pour la majorité des cas) et le **helper** `useForm` (pour un contrôle programmatique avancé). Nous verrons également comment gérer l'upload de fichiers et l'affichage de la progression.

Le Composant `<Form>` : L'Approche Moderne

Le composant `<Form>` est l'approche **recommandée** pour créer des formulaires avec Inertia.js. Il se comporte comme un formulaire HTML classique, sans nécessiter de `v-model` ni de gestion manuelle de l'état.

Exemple : Créer une Maison

Voici un formulaire complet pour créer un bien immobilier dans une application d'agence :

VUE

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
import { store, index } from '@/actions/App/Http/Controllers/HouseController';
import AppLayout from '@/layouts/AppLayout.vue';
import { Button } from '@/components/ui/button';
import { Input } from '@/components/ui/input';
import { Textarea } from '@/components/ui/textarea';
import { Label } from '@/components/ui/label';
import {
    Card,
    CardContent,
    CardDescription,
    CardHeader,
    CardTitle,
} from '@/components/ui/card';

const props = defineProps({
    houseTypes: Array,
});
</script>

<template>
    <AppLayout>
        <div class="container mx-auto max-w-3xl px-3 py-8">
            <Card>
                <CardHeader>
                    <CardTitle>Ajouter une maison</CardTitle>
                    <CardDescription>
                        Remplissez les informations ci-dessous pour ajouter un
                        nouveau bien immobilier
                    </CardDescription>
                </CardHeader>
                <CardContent>
                    <Form
                        :action="store()"
                        #default="{ errors, processing }"
                        class="space-y-6"
                    >
                        <!-- Titre -->
                        <div class="space-y-2">
                            <Label for="title">
                                Titre
                                <span class="text-destructive">*</span>
                            </Label>
                            <Input
                                id="title"
                                type="text"
                                name="title"
                                placeholder="Ex: Belle maison à Paris"
                            />
                            <p
                                v-if="errors.title"
                                class="text-sm text-destructive"
                            >
                                {{ errors.title }}
                            </p>
                        </div>

                        <!-- Type de bien -->
```

```
                        <div class="space-y-2">
                            <Label for="house_type_id">
                                Type de bien
                                <span class="text-destructive">*</span>
                            </Label>
                            <select
                                id="house_type_id"
                                name="house_type_id"
                                class="flex h-9 w-full rounded-md border border-input bg-
transparent px-3 py-1 text-sm shadow-xs transition-colors placeholder:text-muted-foreground focus-
visible:ring-1 focus-visible:ring-ring focus-visible:outline-none"
                            >
                                <option value="">Sélectionnez un type</option>
                                <option
                                    v-for="type in houseTypes"
                                    :key="type.id"
                                    :value="type.id"
                                >
                                    {{ type.name }}
                                </option>
                            </select>
                            <p
                                v-if="errors.house_type_id"
                                class="text-sm text-destructive"
                            >
                                {{ errors.house_type_id }}
                            </p>
                        </div>

                        <!-- Prix -->
                        <div class="space-y-2">
                            <Label for="price">
                                Prix (€)
                                <span class="text-destructive">*</span>
                            </Label>
                            <Input
                                id="price"
                                type="number"
                                name="price"
                                step="0.01"
                                placeholder="250000"
                            />
                            <p
                                v-if="errors.price"
                                class="text-sm text-destructive"
                            >
                                {{ errors.price }}
                            </p>
                        </div>

                        <!-- Adresse -->
                        <div class="space-y-2">
                            <Label for="address">
                                Adresse
                                <span class="text-destructive">*</span>
                            </Label>
                            <Input
                                id="address"
                                type="text"
                                name="address"
```

```
                placeholder="123 rue de la Paix, 75001 Paris"
            />
            <p
                v-if="errors.address"
                class="text-sm text-destructive"
            >
                {{ errors.address }}
            </p>
        </div>

        <!-- Chambres et Taille -->
        <div class="grid gap-4 md:grid-cols-2">
            <div class="space-y-2">
                <Label for="bedrooms">
                    Nombre de chambres
                    <span class="text-destructive">*</span>
                </Label>
                <Input
                    id="bedrooms"
                    type="number"
                    name="bedrooms"
                    placeholder="3"
                    min="0"
                />
                <p
                    v-if="errors.bedrooms"
                    class="text-sm text-destructive"
                >
                    {{ errors.bedrooms }}
                </p>
            </div>

            <div class="space-y-2">
                <Label for="size">
                    Taille (m²)
                    <span class="text-destructive">*</span>
                </Label>
                <Input
                    id="size"
                    type="number"
                    name="size"
                    step="0.01"
                    placeholder="120.50"
                    min="0"
                />
                <p
                    v-if="errors.size"
                    class="text-sm text-destructive"
                >
                    {{ errors.size }}
                </p>
            </div>
        </div>

        <!-- Description -->
        <div class="space-y-2">
            <Label for="description">Description</Label>
            <Textarea
                id="description"
                name="description"
```

```
                                rows="5"
                                placeholder="Décrivez le bien..."
                            />
                            <p
                                v-if="errors.description"
                                class="text-sm text-destructive"
                            >
                                {{ errors.description }}
                            </p>
                        </div>

                        <!-- Boutons -->
                        <div class="flex gap-4">
                            <Button type="submit" :disabled="processing">
                                {{
                                    processing
                                        ? 'Enregistrement...'
                                        : 'Créer la maison'
                                }}
                            </Button>

                            <Button
                                type="button"
                                variant="outline"
                                @click="$inertia.visit(index())"
                            >
                                Annuler
                            </Button>
                        </div>
                    </Form>
                </CardContent>
            </Card>
        </div>
    </AppLayout>
</template>
```

Points Clés du Composant `<Form>`

1. **Pas de** `v-model` : Chaque input a simplement un attribut `name`. Le composant gère automatiquement la collecte des données.
2. **Intégration Wayfinder** : L'action `:action="store()"` utilise Wayfinder pour générer automatiquement l'URL et la méthode HTTP.
3. **Slot Props** : `#default="{ errors, processing }"` expose les erreurs de validation et l'état de traitement du formulaire.
4. **Affichage des Erreurs** : Les erreurs sont accessibles via `errors.nomDuChamp` et s'affichent automatiquement après validation côté serveur.
5. **État de Chargement** : La propriété `processing` permet de désactiver le bouton pendant la soumission et d'afficher un spinner.

Modification avec Valeurs Par Défaut

Pour modifier une ressource existante, utilisez l'attribut `defaultValue` (ou `selected` pour les `<select>`) :

VUE

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
import { update, index } from '@/actions/App/Http/Controllers/HouseController';

const props = defineProps({
    house: Object,
    houseTypes: Array,
});
</script>

<template>
    <Form
        :action="update(house.id)"
        #default="{ errors, processing }"
        class="space-y-6"
    >
        <!-- Titre avec valeur par défaut -->
        <div class="space-y-2">
            <Label for="title">Titre</Label>
            <Input
                id="title"
                type="text"
                name="title"
                :defaultValue="house.title"
                placeholder="Ex: Belle maison à Paris"
            />
            <p v-if="errors.title" class="text-sm text-destructive">
                {{ errors.title }}
            </p>
        </div>

        <!-- Select avec valeur par défaut -->
        <div class="space-y-2">
            <Label for="house_type_id">Type de bien</Label>
            <select id="house_type_id" name="house_type_id">
                <option
                    v-for="type in houseTypes"
                    :key="type.id"
                    :value="type.id"
                    :selected="type.id === house.house_type_id"
                >
                    {{ type.name }}
                </option>
            </select>
        </div>

        <!-- Prix avec valeur par défaut -->
        <div class="space-y-2">
            <Label for="price">Prix (€)</Label>
            <Input
                id="price"
                type="number"
                name="price"
                step="0.01"
                :defaultValue="house.price"
            />
        </div>

        <!-- Autres champs... -->
```

```
        <Button type="submit" :disabled="processing">
            Mettre à jour
        </Button>
    </Form>
</template>
```

Note importante : Utilisez `:defaultValue` (avec deux-points) pour les valeurs dynamiques, et `defaultValue` (sans deux-points) pour les valeurs statiques.

Gestion Côté Serveur avec Laravel

Création d'une Ressource

```php
use Illuminate\Http\Request;
use App\Models\House;

public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|string|max:255',
        'house_type_id' => 'required|exists:house_types,id',
        'price' => 'required|numeric|min:0',
        'address' => 'required|string|max:255',
        'bedrooms' => 'required|integer|min:0',
        'size' => 'required|numeric|min:0',
        'description' => 'nullable|string',
    ]);

    House::create($validated);

    return redirect()->route('houses.index')
        ->with('success', 'Maison créée avec succès.');
}
```

Mise à Jour d'une Ressource

```php
public function update(Request request,Houserequest, House request,Househouse)
{
    validated=validated = validated=request→validate([
        'title' ⇒ 'required|string|max:255',
        'house_type_id' ⇒ 'required|exists:house_types,id',
        'price' ⇒ 'required|numeric|min:0',
        'address' ⇒ 'required|string|max:255',
        'bedrooms' ⇒ 'required|integer|min:0',
        'size' ⇒ 'required|numeric|min:0',
        'description' ⇒ 'nullable|string',
    ]);

    house->update(house→update(house->update(validated);

    return redirect()→route('houses.index')
        →with('success', 'Maison mise à jour avec succès.');
}
```

Note : Inertia gère automatiquement les erreurs de validation. Si `$request→validate()` échoue, les erreurs sont renvoyées au frontend et accessibles via `errors` dans le slot props.

Le Helper `useForm` : Pour un Contrôle Avancé

Le helper `useForm` est utile lorsque vous avez besoin d'un **contrôle programmatique** sur vos formulaires. C'est particulièrement adapté pour :

- Les uploads de fichiers avec gestion manuelle
- Les transformations de données complexes avant soumission
- Les formulaires avec logique métier sophistiquée

Exemple : Upload d'Images

Dans notre application d'agence immobilière, l'upload d'images pour une maison est géré via `useForm` :

VUE

```vue
<script setup>
import { useForm } from '@inertiajs/vue3';

const props = defineProps({
    house: Object,
});

// Form pour l'upload d'images
const imageForm = useForm({
    images: null,
});

const handleImageChange = (event) => {
    const files = event.target.files;
    if (files && files.length > 0) {
        // Convertir FileList en Array
        imageForm.images = Array.from(files);
    }
};

const uploadImages = () => {
    imageForm.post(`/houses/{props.house.id}/images`, {
        preserveScroll: true,
        onSuccess: () => {
            imageForm.reset();
            // Reset l'input file
            const input = document.getElementById('images');
            if (input) {
                input.value = '';
            }
        },
    });
};
</script>

<template>
    <form @submit.prevent="uploadImages" class="space-y-4">
        <div class="space-y-2">
            <Label for="images">Ajouter de nouvelles images</Label>
            <input
                id="images"
                type="file"
                multiple
                accept="image/*"
                @input="handleImageChange"
                class="flex h-10 w-full rounded-md border border-input bg-background px-3 py-2
text-sm"
            />
            <p class="text-sm text-muted-foreground">
                Vous pouvez sélectionner plusieurs images (max 2MB par image)
            </p>
            <p
                v-if="imageForm.errors['images.0']"
                class="text-sm text-destructive"
            >
                {{ imageForm.errors['images.0'] }}
            </p>
        </div>
```

```
        <!-- Progression upload -->
        <div v-if="imageForm.progress" class="space-y-2">
            <div class="flex justify-between text-sm">
                <span>Upload en cours...</span>
                <span class="font-medium">
                    {{ imageForm.progress.percentage }}%
                </span>
            </div>
            <div class="h-2 w-full rounded-full bg-secondary">
                <div
                    class="h-2 rounded-full bg-primary transition-all"
                    :style="{
                        width: `{imageForm.progress.percentage}%`,
                    }"
                />
            </div>
        </div>

        <Button
            type="submit"
            :disabled="imageForm.processing || !imageForm.images"
        >
            {{
                imageForm.processing
                    ? 'Upload en cours...'
                    : 'Ajouter les images'
            }}
        </Button>
    </form>
</template>
```

Points Clés du `useForm`

1. **Initialisation** : `const imageForm = useForm({ images: null })`
2. **Liaison Manuelle** : Avec `useForm`, vous gérez manuellement les changements via `v-model` ou des handlers d'événements.
3. **Méthodes de Soumission** :
   - `imageForm.post(url, options)`
   - `imageForm.put(url, options)`
   - `imageForm.patch(url, options)`
   - `imageForm.delete(url, options)`
4. **Options de Soumission** :
   - `preserveScroll: true` : Maintient la position du scroll
   - `onSuccess: () => {}` : Callback après succès
   - `onError: () => {}` : Callback en cas d'erreur
5. **Propriétés Disponibles** :
   - `imageForm.processing` : Indique si le formulaire est en cours de soumission
   - `imageForm.progress` : Objet contenant `percentage` pour l'upload de fichiers
   - `imageForm.errors` : Objet contenant les erreurs de validation

- `imageForm.wasSuccessful` : `true` après une soumission réussie

6. **Méthodes Utiles** :
   - `imageForm.reset()` : Réinitialise le formulaire
   - `imageForm.clearErrors()` : Efface toutes les erreurs
   - `imageForm.setError('field', 'message')` : Définit une erreur manuellement

Upload de Fichiers : Gestion Côté Serveur

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
use App\Models\House;
use App\Models\HouseImage;

public function uploadImages(Request request,Houserequest, House request,Househouse)
{
    request->validate(['images'⇒'required|array',
'images.*'⇒'image|mimes:jpeg,png,jpg,gif|max:2048',]);foreach(request→validate([
        'images' ⇒ 'required|array',
        'images.*' ⇒ 'image|mimes:jpeg,png,jpg,gif|max:2048',
    ]);

    foreach (request->validate(['images'⇒'required|array',
'images.*'⇒'image|mimes:jpeg,png,jpg,gif|max:2048',]);foreach(request→file('images') as $image)
{
        $path = $image→store('houses', 'public');

        HouseImage::create([
            'house_id' ⇒ $house→id,
            'path' ⇒ $path,
        ]);
    }

    return back()→with('success', 'Images ajoutées avec succès.');
}
```

Note : Inertia convertit automatiquement les requêtes contenant des fichiers en `FormData`. Aucune configuration supplémentaire n'est nécessaire.

Suppression d'une Image

Pour supprimer une image, utilisez directement le `router` d'Inertia :

```vue
<script setup>
import { router } from '@inertiajs/vue3';

const props = defineProps({
    house: Object,
});

const deleteImage = (imageId) => {
    if (confirm('Êtes-vous sûr de vouloir supprimer cette image ?')) {
        router.delete(`/houses/${props.house.id}/images/${imageId}`, {
            preserveScroll: true,
        });
    }
};
</script>

<template>
    <Button
        size="sm"
        variant="destructive"
        @click="deleteImage(image.id)"
    >
        Supprimer
    </Button>
</template>
```

Côté Serveur :

```php
use App\Models\HouseImage;
use Illuminate\Support\Facades\Storage;

public function deleteImage(House house,HouseImagehouse, HouseImage house,HouseImageimage)
{
    // Vérifier que l'image appartient bien à cette maison
    if (image->houseid≠=image→house_id ≠= image->houseid≠=house→id) {
        abort(403);
    }

    // Supprimer le fichier physique
    Storage::disk('public')→delete(image->path);//Supprimerl'enregistrementenbaseimage→path);

    // Supprimer l'enregistrement en base
    image->path);//Supprimerl'enregistrementenbaseimage→delete();

    return back()→with('success', 'Image supprimée avec succès.');
}
```

Quand Utiliser `<Form>` vs `useForm` ?

Critère	`<Form>` Component	`useForm` Helper
Simplicité	✅ Très simple, pas de `v-model`	⚠️ Plus verbeux
Cas d'usage	CRUD standard, formulaires classiques	Logique complexe, transformations

Critère	`<Form>` Component	`useForm` Helper
Upload de fichiers	✅ Supporte nativement	✅ Avec gestion manuelle
Valeurs par défaut	`defaultValue` / `selected`	Pré-remplir avec `useForm(data)`
Contrôle programmatique	⚠️ Limité (via refs)	✅ Total
Recommandation	**Privilégier par défaut**	Pour les cas avancés

Règle générale : Utilisez `<Form>` sauf si vous avez une raison spécifique d'utiliser `useForm`.

Options Avancées

Avec `<Form>` Component

VUE

```
<Form
    :action="update(house.id)"
    :options="{
        preserveScroll: true,
        preserveState: true,
        only: ['house', 'flash'],
    }"
    resetOnSuccess
    #default="{ errors, processing, wasSuccessful }"
>
    <!— Vos champs —>

    <div v-if="wasSuccessful" class="text-green-600">
        Modifications enregistrées !
    </div>
</Form>
```

Avec `useForm` Helper

JAVASCRIPT

```
form.post('/houses', {
    preserveScroll: true,
    onSuccess: () => {
        form.reset();
        alert('Maison créée avec succès !');
    },
    onError: (errors) => {
        console.error('Erreurs:', errors);
    },
});
```

Conclusion

Inertia.js propose deux approches puissantes pour gérer les formulaires :

1. **Le composant** `<Form>` : Simple, déclaratif, idéal pour la majorité des cas

2. **Le helper** `useForm` : Contrôle total, parfait pour les besoins avancés

Dans les deux cas, Inertia gère automatiquement :

- La conversion en `FormData` pour les fichiers
- La gestion des erreurs de validation Laravel
- Le tracking de progression pour les uploads
- Les états de chargement et de succès

En combinant ces outils avec **Wayfinder** et **shadcn-vue**, vous obtenez une expérience de développement moderne, type-safe, et productive. 🚀

Ressources

- Documentation Inertia.js - Forms
- Documentation Inertia.js - File Uploads
- Documentation Laravel - Validation
- Documentation Laravel - File Storage
- Projet Agence immobilière sur GitHub : https://github.com/opmvpc/immo

3.8 Pagination avec Laravel et Inertia.js

La pagination est essentielle pour afficher une grande quantité de données sur plusieurs pages tout en réduisant la charge sur le client et le serveur. Avec Laravel, la méthode `paginate()` génère des données paginées. Utilisée avec Inertia.js, elle devient encore plus simple et fluide.

Comment fonctionne la pagination avec `paginate()` ?

Laravel fournit une méthode puissante appelée `paginate()` qui renvoie :

- Une collection de données (par page).
- Des métadonnées utiles comme le nombre total d'éléments, la page actuelle, les URLs pour la navigation, etc.

Exemple de réponse JSON générée par `paginate()` :

```json
{
  "data": [
    { "id": 1, "name": "John Doe" },
    { "id": 2, "name": "Jane Smith" }
  ],
  "current_page": 1,
  "last_page": 5,
  "per_page": 10,
  "total": 50,
  "prev_page_url": null,
  "next_page_url": "http://example.com?page=2",
  "links": [
    { "url": null, "label": "&laquo; Previous", "active": false },
    { "url": "http://example.com?page=1", "label": "1", "active": true },
    { "url": "http://example.com?page=2", "label": "2", "active": false },
    { "url": "http://example.com?page=3", "label": "3", "active": false },
    { "url": "http://example.com?page=4", "label": "4", "active": false },
    { "url": "http://example.com?page=5", "label": "5", "active": false },
    { "url": "http://example.com?page=2", "label": "Next &raquo;", "active": false }
  ]
}
```

Différences entre Axios/Fetch et Inertia.js pour la pagination

Axios/Fetch	Inertia.js
Nécessite des appels API séparés pour chaque page.	Les données sont directement envoyées depuis Laravel.
Gestion manuelle des états de chargement et des erreurs.	Gestion implicite des transitions et des erreurs.
Réponse paginée doit être formatée côté client.	Les données sont directement utilisables avec les props.
Risque de duplication de logique côté frontend et backend.	La logique reste centralisée côté backend.

Avec Inertia.js, la navigation entre les pages est fluide et ne nécessite aucun rechargement, ce qui améliore l'expérience utilisateur.

Le backend avec Laravel

Laravel propose la méthode `paginate()` pour fournir une réponse paginée contenant :

- Les données pour la page en cours.
- Les métadonnées utiles (total des éléments, page actuelle, liens vers les pages précédentes et suivantes, etc.).

Exemple d'implémentation backend

```php
use App\Models\User;
use Inertia\Inertia;

Route::get('/users', function () {
    users=User::paginate(10);//Renvoieunecollectionpagine´ereturnInertia::render('Users',
['users'⇒users = User::paginate(10); // Renvoie une collection paginée
    return Inertia::render('Users', ['users' ⇒
users=User::paginate(10);//Renvoieunecollectionpagine´ereturnInertia::render('Users',
['users'⇒users]);
});
```

La méthode `paginate()` fournit une réponse paginée structurée, qui sera envoyée au frontend via Inertia.

Le composant Vue.js pour la page des utilisateurs

Voici le fichier complet de la page `Users.vue`, qui affiche une liste d'utilisateurs et gère la pagination.

```vue
<script setup>
import Pagination from './Pagination.vue'; // Importation du composant Pagination

// Récupération des données paginées transmises par le backend via les props d'Inertia
defineProps(['users']);
</script>

<template>
  <div>
    <h1 class="text-xl font-bold mb-4">Liste des Utilisateurs</h1>
    <!-- Liste des utilisateurs -->
    <ul>
      <li
        v-for="user in users.data"
        :key="user.id"
        class="mb-2 p-2 border rounded bg-gray-50"
      >
        {{ user.name }}
      </li>
    </ul>
    <!-- Composant de Pagination -->
    <Pagination :pagination="users" />
  </div>
</template>
```

Le composant réutilisable de pagination

Voici un composant Vue.js complet pour gérer la pagination, basé sur les données renvoyées par `paginate()`.

VUE

```
<script setup>
import { Link } from "@inertiajs/vue3";
import { computed } from "vue";

const { pagination } = defineProps({
    pagination: {
        type: Object,
        required: true,
        validator: (obj) => {
            return (
                obj.hasOwnProperty("current_page") &&
                obj.hasOwnProperty("total") &&
                obj.hasOwnProperty("per_page")
            );
        },
    },
});

const showingFrom = computed(
    () => (pagination.current_page - 1) * pagination.per_page + 1
);
const showingTo = computed(() =>
    Math.min(pagination.current_page * pagination.per_page, pagination.total)
);
</script>

<template>
    <nav
        v-if="pagination.last_page > 1"
        role="navigation"
        aria-label="Pagination Navigation"
        class="flex items-center justify-between"
    >
        <!-- Version mobile -->
        <div class="flex justify-between flex-1 sm:hidden">
            <Link
                v-if="pagination.prev_page_url"
                :href="pagination.prev_page_url"
                preserve-scroll
                class="relative inline-flex items-center px-4 py-2 text-sm font-medium text-gray-
700 bg-white border border-gray-300 leading-5 rounded-md hover:text-gray-500 focus:outline-none
focus:ring ring-gray-300 focus:border-blue-300 active:bg-gray-100 active:text-gray-700 transition
ease-in-out duration-150 dark:bg-gray-800 dark:border-gray-600 dark:text-gray-300"
            >
                Previous
            </Link>
            <span
                v-else
                class="relative inline-flex items-center px-4 py-2 text-sm font-medium text-gray-
500 bg-white border border-gray-300 cursor-default leading-5 rounded-md dark:text-gray-600
dark:bg-gray-800 dark:border-gray-600"
            >
                Previous
            </span>

            <Link
                v-if="pagination.next_page_url"
                :href="pagination.next_page_url"
                preserve-scroll
```

```
                class="relative inline-flex items-center px-4 py-2 ml-3 text-sm font-medium text-
gray-700 bg-white border border-gray-300 leading-5 rounded-md hover:text-gray-500 focus:outline-
none focus:ring ring-gray-300 focus:border-blue-300 active:bg-gray-100 active:text-gray-700
transition ease-in-out duration-150 dark:bg-gray-800 dark:border-gray-600 dark:text-gray-300"
            >
                Next
            </Link>
            <span
                v-else
                class="relative inline-flex items-center px-4 py-2 ml-3 text-sm font-medium text-
gray-500 bg-white border border-gray-300 cursor-default leading-5 rounded-md dark:text-gray-600
dark:bg-gray-800 dark:border-gray-600"
            >
                Next
            </span>
        </div>

        <!-- Version desktop -->
        <div
            class="hidden sm:flex-1 sm:flex sm:items-center sm:justify-between"
        >
            <div>
                <p class="text-sm text-gray-700 leading-5 dark:text-gray-400">
                    Showing
                    <span class="font-medium">{{ showingFrom }}</span>
                    to
                    <span class="font-medium">{{ showingTo }}</span>
                    of
                    <span class="font-medium">{{ pagination.total }}</span>
                    results
                </p>
            </div>

            <div>
                <span
                    class="relative z-0 inline-flex rtl:flex-row-reverse shadow-sm rounded-md"
                >
                    <!-- Bouton Previous -->
                    <Link
                        v-if="pagination.prev_page_url"
                        :href="pagination.prev_page_url"
                        preserve-scroll
                        class="relative inline-flex items-center px-2 py-2 text-sm font-medium
text-gray-500 bg-white border border-gray-300 rounded-l-md leading-5 hover:text-gray-400 focus:z-
10 focus:outline-none focus:ring ring-gray-300 focus:border-blue-300 active:bg-gray-100
active:text-gray-500 transition ease-in-out duration-150 dark:bg-gray-800 dark:border-gray-600"
                    >
                        <svg
                            class="w-5 h-5"
                            fill="currentColor"
                            viewBox="0 0 20 20"
                        >
                            <path
                                fill-rule="evenodd"
                                d="M12.707 5.293a1 1 0 010 1.414L9.414 10l3.293 3.293a1 1 0 01-
1.414 1.414l-4-4a1 1 0 010-1.414l4-4a1 1 0 011.414 0z"
                                clip-rule="evenodd"
                            />
                        </svg>
                    </Link>
```

```
                    <span
                        v-else
                        class="relative inline-flex items-center px-2 py-2 text-sm font-medium
text-gray-500 bg-white border border-gray-300 cursor-default rounded-l-md leading-5 dark:bg-gray-
800 dark:border-gray-600"
                    >
                        <svg
                            class="w-5 h-5"
                            fill="currentColor"
                            viewBox="0 0 20 20"
                        >
                            <path
                                fill-rule="evenodd"
                                d="M12.707 5.293a1 1 0 010 1.414L9.414 10l3.293 3.293a1 1 0 01-
1.414 1.414l-4-4a1 1 0 010-1.414l4-4a1 1 0 011.414 0z"
                                clip-rule="evenodd"
                            />
                        </svg>
                    </span>

                    <!-- Numéros de page -->
                    <template
                        v-for="link in pagination.links.slice(1, -1)"
                        :key="link.label"
                    >
                        <span
                            v-if="link.active"
                            class="relative inline-flex items-center px-4 py-2 -ml-px text-sm
font-medium text-blue-600 bg-blue-50 border border-gray-300 cursor-default leading-5 dark:bg-gray-
800 dark:border-gray-600"
                            >{{ link.label }}</span
                        >
                        <Link
                            v-else
                            :href="link.url"
                            preserve-scroll
                            class="relative inline-flex items-center px-4 py-2 -ml-px text-sm
font-medium text-gray-700 bg-white border border-gray-300 leading-5 hover:text-gray-500 focus:z-10
focus:outline-none focus:ring ring-gray-300 focus:border-blue-300 active:bg-gray-100 active:text-
gray-700 transition ease-in-out duration-150 dark:bg-gray-800 dark:border-gray-600 dark:text-gray-
400"
                            >{{ link.label }}</Link
                        >
                    </template>

                    <!-- Bouton Next -->
                    <Link
                        v-if="pagination.next_page_url"
                        :href="pagination.next_page_url"
                        preserve-scroll
                        class="relative inline-flex items-center px-2 py-2 -ml-px text-sm font-
medium text-gray-500 bg-white border border-gray-300 rounded-r-md leading-5 hover:text-gray-400
focus:z-10 focus:outline-none focus:ring ring-gray-300 focus:border-blue-300 active:bg-gray-100
active:text-gray-500 transition ease-in-out duration-150 dark:bg-gray-800 dark:border-gray-600"
                    >
                        <svg
                            class="w-5 h-5"
                            fill="currentColor"
                            viewBox="0 0 20 20"
                        >
```

```
                            <path
                                fill-rule="evenodd"
                                d="M7.293 14.707a1 1 0 010-1.414L10.586 10 7.293 6.707a1 1 0
011.414-1.414l4 4a1 1 0 010 1.414l-4 4a1 1 0 01-1.414 0z"
                                clip-rule="evenodd"
                            />
                        </svg>
                    </Link>
                    <span
                        v-else
                        class="relative inline-flex items-center px-2 py-2 -ml-px text-sm font-
medium text-gray-500 bg-white border border-gray-300 cursor-default rounded-r-md leading-5
dark:bg-gray-800 dark:border-gray-600"
                    >
                        <svg
                            class="w-5 h-5"
                            fill="currentColor"
                            viewBox="0 0 20 20"
                        >
                            <path
                                fill-rule="evenodd"
                                d="M7.293 14.707a1 1 0 010-1.414L10.586 10 7.293 6.707a1 1 0
011.414-1.414l4 4a1 1 0 010 1.414l-4 4a1 1 0 01-1.414 0z"
                                clip-rule="evenodd"
                            />
                        </svg>
                    </span>
                </span>
            </div>
        </div>
    </nav>
</template>
```

Pourquoi cette structure ?

1. **Propre et réutilisable** : Le composant Pagination est autonome et peut être utilisé partout où la pagination est nécessaire.
2. **Centralisation de la logique** : Toute la logique de pagination est gérée côté Laravel.
3. **Transparence d'Inertia.js** : Avec Inertia, les transitions entre les pages sont fluides et sans rechargement complet.

Cette structure permet de créer une expérience utilisateur agréable tout en maintenant une architecture propre et modulaire.

3.9 Gestion des policies avec Laravel et Inertia.js

Laravel fournit un puissant système de **policies** pour gérer les autorisations d'accès aux fonctionnalités d'une application. Cependant, avec **Inertia.js**, les helpers Blade comme `@can` ou `@cannot` ne sont pas disponibles. La solution consiste à transmettre explicitement les permissions au frontend sous forme de **props**.

Comment fonctionnent les policies dans Laravel ?

Une **policy** dans Laravel définit les règles d'autorisation pour un modèle. Par exemple, pour un modèle `Post`, une policy pourrait définir si un utilisateur peut éditer ou supprimer un article :

```php
public function edit(User user,Postuser, Post user,Postpost)
{
    return user->id===user→id === user->id===post→user_id;
}
```

Avec Blade, on utiliserait cette policy ainsi :

```php
@can('edit', $post)
    <a href="/posts/{{ $post→id }}/edit">Edit</a>
@endcan
```

Avec **Inertia.js**, il faut transmettre ces autorisations au frontend via les props.

Transmettre les permissions au frontend

Dans le backend, les autorisations sont vérifiées avec les méthodes des policies et envoyées au composant **Vue.js** via les props :

```php
use App\Models\Post;
use Illuminate\Support\Facades\Gate;
use Inertia\Inertia;

Route::get('/posts/{post}', function (Post $post) {
    return Inertia::render('PostDetails', [
        'post' => $post,
        'can' => [
            'edit' => Gate::allows('edit', $post),
            'delete' => Gate::allows('delete', $post),
        ],
    ]);
});
```

Ici, `Gate::allows` vérifie si l'utilisateur peut exécuter une action spécifique sur le modèle, et les résultats sont transmis comme un objet `can`.

Utilisation dans le composant Vue.js

Dans le composant **Vue.js**, ces autorisations sont utilisées pour conditionner l'affichage :

```vue
<script setup>
defineProps(['post', 'can']);

function deletePost() {
  if (confirm('Are you sure you want to delete this post?')) {
    // Appel pour supprimer le post
  }
}
</script>

<template>
  <div>
    <h1 class="text-xl font-bold">{{ post.title }}</h1>
    <p class="mb-4">{{ post.content }}</p>

    <div class="flex space-x-2">
      <inertia-link
        v-if="can.edit"
        :href="`/posts/${post.id}/edit`"
        class="px-4 py-2 bg-blue-500 text-white rounded"
      >
        Edit
      </inertia-link>

      <button
        v-if="can.delete"
        @click="deletePost"
        class="px-4 py-2 bg-red-500 text-white rounded"
      >
        Delete
      </button>
    </div>
  </div>
</template>
```

Avantages

- **Centralisation des autorisations** : Toute la logique d'autorisation reste gérée côté Laravel.
- **Clarté et modularité** : Les permissions sont transmises explicitement, rendant le code plus lisible et maintenable.
- **Flexibilité** : Cette approche permet de gérer des autorisations dynamiques basées sur des rôles, des statuts ou d'autres critères.

En transmettant les permissions comme des props, on maintient une séparation claire entre la logique métier et la présentation tout en respectant les principes d'Inertia.js.

3.10 Exercice : Application Portfolio Personnel

Objectif

Développer une application web complète permettant de gérer et d'afficher un portfolio professionnel. L'application comprendra un **backoffice** pour administrer le contenu et un **frontoffice** créatif pour présenter vos réalisations.

Partie 1 : Backoffice - Gestion du Contenu

1.1 Gestion des Projets

Chaque projet contient :

Informations principales :

- Titre, slug (auto-généré), description courte, description détaillée (WYSIWYG)
- Type (Site web, App mobile, API, Design, etc.), Statut (Brouillon/Publié/Archivé)
- Date de réalisation, Client, Contexte (Scolaire/Stage/Freelance/Personnel), Durée, Rôle
- Lien projet, Lien GitHub, Ordre d'affichage

Relations :

- Technologies utilisées (relation many-to-many)
- Images : 1 image principale + galerie (plusieurs images)

1.2 Gestion des Technologies

- Nom, Catégorie (Backend/Frontend/BDD/DevOps/Design), Niveau (Débutant/Intermédiaire/Avancé/Expert)
- Logo (upload), Couleur (hex), Ordre d'affichage

1.3 Gestion des Expériences

- Poste, Entreprise, Lieu, Date début/fin, Description (WYSIWYG)
- Type (Stage/CDI/CDD/Freelance/Alternance)

1.4 Gestion de la Formation

- Diplôme, Établissement, Lieu, Date début/fin, Description

1.5 Informations Personnelles

- Nom, Titre professionnel, Bio (WYSIWYG), Photo de profil
- Email, Téléphone, Localisation, Disponibilité
- CV (upload PDF)
- Liens sociaux (LinkedIn, GitHub, Twitter, etc.)

Partie 2 : Fonctionnalités Techniques

2.1 Éditeur WYSIWYG

Intégrer un éditeur de texte riche pour les descriptions. Options suggérées :

- **Tiptap** (moderne, headless, flexible mais plus de config)
- **Quill** (simple, léger)
- **CKEditor 5** (complet)
- **TinyMCE** (populaire)

Fonctionnalités minimales : formatage texte, titres, listes, liens, blocs de code.

2.2 Upload d'Images

- Upload multiple avec progression
- Validation (format, max 5MB)
- Suppression individuelle avec confirmation
- Réorganisation des images (bonus)

2.3 Pagination et Recherche

Pour la liste des projets :

- Pagination (10-15 par page)
- Recherche (titre, description, technologies)
- Filtres (Statut, Type, Année)
- Tri (date, ordre d'affichage, alphabétique)

2.4 Interface

Utiliser shadcn-vue pour créer une interface professionnelle (tableaux, formulaires, cards, dialogs, sidebar).

Partie 3 : Frontoffice - Affichage Public

3.1 Page d'Accueil

Présentation attractive : nom, titre, photo, phrase d'accroche, technologies principales, CTA vers projets/contact.

3.2 Page Projets

Grille de projets avec image, titre, description courte, technologies. Filtres par type/technologie. Lien vers détail.

3.3 Page Détail Projet

Affichage complet : galerie d'images, description détaillée, technologies, infos (client, durée, contexte), liens externes, navigation précédent/suivant.

3.4 Page À Propos

Bio, parcours académique (timeline), expériences (timeline), technologies avec niveaux visualisés, lien téléchargement CV.

3.5 Page Contact

Formulaire simple (nom, email, sujet, message) avec envoi par email Laravel Mail. Protection anti-spam (honeypot).

Partie 4 : Fonctionnalités Optionnelles

Ces fonctionnalités augmenteront la qualité du projet mais ne sont pas obligatoires :

4.1 Blog Technique

Section blog : articles avec titre, contenu (WYSIWYG), image, catégories/tags. Liste paginée + page détail.

4.2 Google Analytics

Intégration Google Analytics pour tracking des visites. Affichage de statistiques basiques dans le backoffice.

4.3 Optimisation SEO

- Meta tags personnalisables (title, description)
- Open Graph pour partage réseaux sociaux
- URLs propres (slugs)
- Images optimisées (compression, lazy loading)

Stack Technique

- Backend : Laravel 12
- Frontend : Vue.js 3 (Composition API) + Inertia.js
- Starter Kit : Laravel Vue Starter Kit
- UI : shadcn-vue + Tailwind CSS v4
- Base de données : MySQL
- Auth : Laravel built-in

Bonnes pratiques :

- Utiliser Wayfinder pour les routes
- Utiliser le composant `<Form>` d'Inertia
- Code organisé, commits Git réguliers

Design du Frontoffice

Le design est **totalement libre**. C'est l'occasion de montrer votre créativité et votre savoir-faire. Le frontoffice doit être :

- **Original** : Créez quelque chose d'unique qui vous représente
- **Responsive** : Fonctionnel sur mobile, tablette, desktop
- **Professionnel** : Soigné, moderne, sans fautes
- **Performant** : Images optimisées, navigation fluide

Inspirez-vous de portfolios existants mais ne copiez pas. Ce portfolio vous servira pour vos candidatures !

3.11 Du modèle à l'accueil : le CRUD projets

L'énoncé est costaud, alors plutôt que de te jeter dans le grand bain, ce chapitre te tient la main pour un premier tour complet : on crée le modèle `Project`, on génère des données factices, on construit les pages d'admin (liste, création, édition, suppression), et on termine sur une homepage publique. Une fois ce squelette en place, le reste de l'énoncé (technos, expériences, blog, etc.) c'est la même recette à appliquer.

Générer les fichiers d'un coup

On commence par tout générer en une commande grâce aux flags d'Artisan :

```bash
php artisan make:model Project -mfsr
```

Ce qui crée le modèle, la **m**igration, la **f**actory, le **s**eeder et le contrôleur **r**esource. Cinq fichiers, une ligne.

Migration et modèle

Dans la migration, on ajoute les deux champs qui décrivent un projet :

```php
Schema::create('projects', function (Blueprint $table) {
    $table->id();
    $table->string('title');
    $table->text('description');
    $table->timestamps();
});
```

Puis on autorise l'assignation de masse dans le modèle, sinon Laravel refusera d'enregistrer quoi que ce soit :

```php
class Project extends Model
{
    use HasFactory;

    protected $fillable = ['title', 'description'];
}
```

Factory et seeder

La factory définit à quoi ressemble un projet bidon :

```php
public function definition()
{
    return [
        'title' => fake()->bs(),
        'description' => fake()->paragraphs(3, true),
    ];
}
```

> 💡 `fake()->bs()` *génère du jargon corporate (« synergize scalable solutions »). Parfait pour des titres de projet placeholder.*

Le seeder appelle la factory pour générer dix projets :

```php
public function run()
{
    Project::factory(10)->create();
}
```

Et on enregistre le `ProjectSeeder` dans le `DatabaseSeeder` pour qu'il tourne avec les autres :

```php
public function run()
{
    // ... autres seeders
    $this->call([
        ProjectSeeder::class,
    ]);
}
```

Pour repartir d'une base propre avec les données factices :

```bash
php artisan migrate:fresh --seed
```

> ⚠️ `migrate:fresh` ***drop*** *toutes les tables. Ne lance jamais ça en production. (Si tu te poses la question, c'est probablement déjà trop tard.)*

Routes

Une seule ligne pour câbler les sept routes CRUD :

```php
// ... autres routes
Route::prefix("admin")->middleware(['auth', 'verified'])->group(function () {
    Route::resource('projects', ProjectController::class);
});
```

Toutes les routes sont derrière `auth` + `verified`, donc inaccessibles aux invités. Le `DatabaseSeeder` crée un user de test (`test@example.com` / `password`) — utilise-le pour te connecter.

> 💡 *Ces imports magiques comme `@/routes/projects` et `@/actions/App/Http/Controllers/ProjectController` qu'on va utiliser plus bas, c'est **Wayfinder** qui les génère automatiquement à partir de tes routes Laravel. Plus besoin de hardcoder des URL côté front.*

Le contrôleur : méthode `index`

On commence par la liste des projets :

```php
public function index()
{
    projects=Project::all();returnInertia::render('Projects/Index',['projects'⇒projects =
Project::all();

    return Inertia::render('Projects/Index', [
        'projects' ⇒ projects=Project::all();returnInertia::render('Projects/Index',
['projects'⇒projects,
    ]);
}
```

Inertia transmet le tableau `$projects` comme prop à la vue Vue. Pas d'API REST à écrire, pas de fetch à faire côté front — c'est ça la magie d'Inertia.

La vue Index

On crée `resources/js/Pages/Projects/Index.vue` en s'inspirant du Dashboard du starter kit, et on utilise le composant `Card` de shadcn-vue (doc) :

VUE

```vue
<script setup>
import { Head } from '@inertiajs/vue3';
import { create, edit, index } from '@/routes/projects';
import { Card, CardContent, CardFooter, CardHeader, CardTitle } from '@/components/ui/card';
import Button from '@/components/ui/button/Button.vue';
import TextLink from '@/components/TextLink.vue';

defineOptions({
    layout: {
        breadcrumbs: [
            {
                title: 'Projets',
                href: index(),
            },
        ],
    },
});

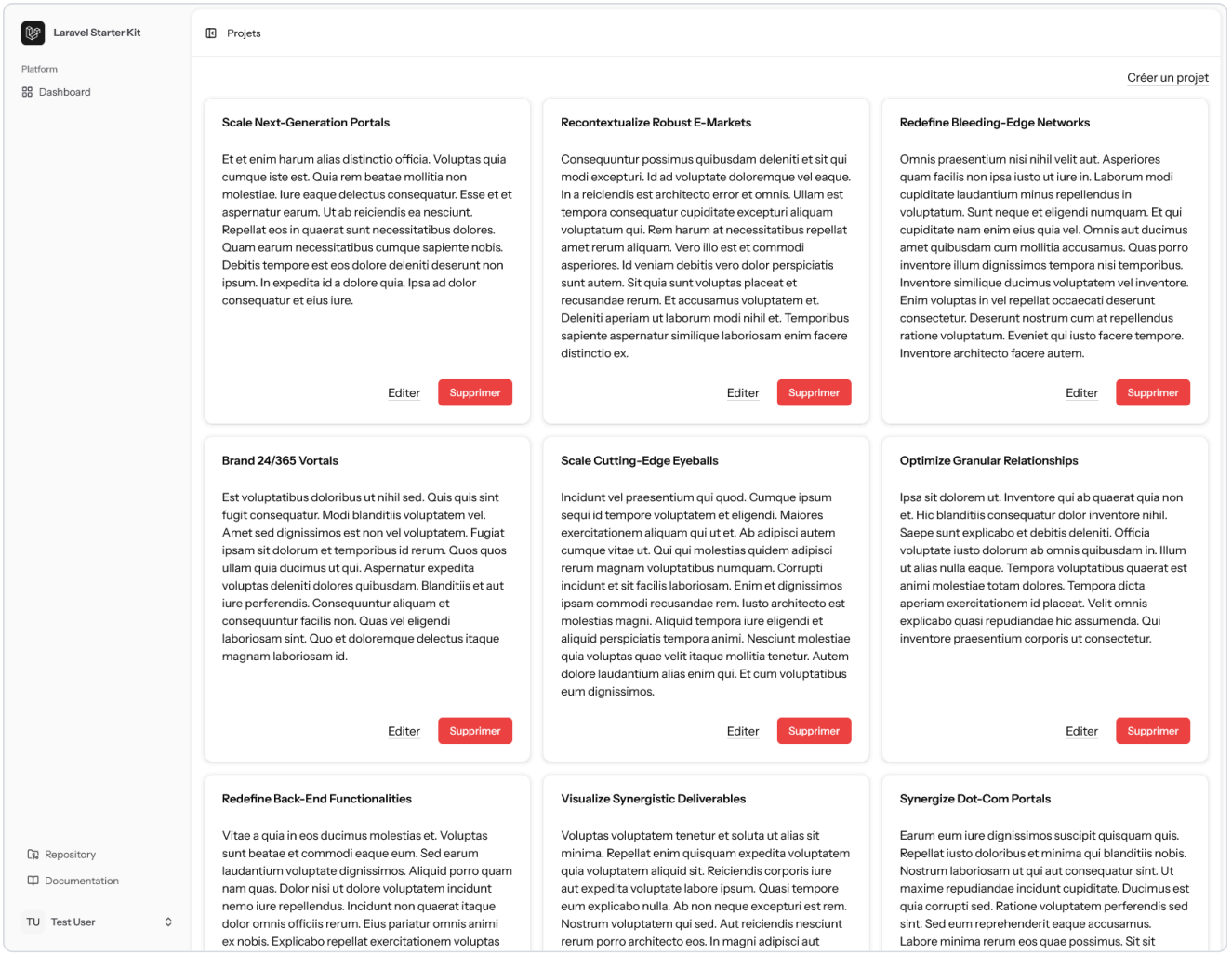
defineProps({
    projects: Array,
});
</script>

<template>

    <Head title="Projets" />

    <div class="flex h-full flex-1 flex-col gap-4 overflow-x-auto rounded-xl p-4">
        <div class="flex items-center justify-end">
            <TextLink :href="create()">
                Créer un projet
            </TextLink>
        </div>
        <div class="grid auto-rows-min gap-4 md:grid-cols-3">
            <Card v-for="project in projects" :key="project.id">
                <CardHeader>
                    <CardTitle class="capitalize">{{ project.title }}</CardTitle>
                </CardHeader>
                <CardContent class="flex-1">
                    <p>{{ project.description }}</p>
                </CardContent>
                <CardFooter class="flex justify-end space-x-4">
                    <TextLink :href="edit(project.id)">
                        Editer
                    </TextLink>
                    <Button variant="destructive" class="ml-2">
                        Supprimer
                    </Button>
                </CardFooter>
            </Card>
        </div>
    </div>
</template>
```

Une grille de cartes, chacune avec son titre, sa description, et deux actions. Le lien « Créer un projet » est en haut à droite.



Lien dans la sidebar

Pour atteindre cette page en deux clics, on ajoute une entrée dans la barre latérale du dashboard. Pense aux imports manquants :

```vue
import { usePage } from '@inertiajs/vue3';
import { LucideFolder } from 'lucide-vue-next';
import projects from '@/routes/projects';
import type { NavItem } from '@/types';

// pour matcher la page courante et appliquer le style "actif" sur toutes les routes CRUD (index,
create, edit)
const page = usePage();

const mainNavItems: NavItem[] = [
    {
        title: 'Dashboard',
        href: dashboard(),
        icon: LayoutGrid,
    },
    {
        title: 'Projets',
        href: projects.index(),
        icon: LucideFolder,
        isActive: page.component.startsWith('Projects')
    }
];
```

> 💡 *Toutes les icônes Lucide sont déjà installées dans le starter kit. Catalogue complet sur lucide.dev.*

Méthode `create` et vue Create

Le contrôleur retourne juste la vue, sans données particulières à transmettre :

```php
public function create()
{
    return inertia('Projects/Create');
}
```

Avant d'écrire le formulaire, on installe le composant `Textarea` de shadcn :

```bash
npx shadcn-vue@latest add textarea
```

Puis on crée `resources/js/Pages/Projects/Create.vue` en s'inspirant du formulaire de profil fourni par le starter kit :

VUE

```
<script setup>
import { Form, Head } from '@inertiajs/vue3';
import Heading from '@/components/Heading.vue';
import InputError from '@/components/InputError.vue';
import { Button } from '@/components/ui/button';
import { Input } from '@/components/ui/input';
import { Label } from '@/components/ui/label';
import projects from '@/routes/projects';
import ProjectController from '@/actions/App/Http/Controllers/ProjectController';
import Textarea from '@/components/ui/textarea/Textarea.vue';
import Separator from '@/components/ui/separator/Separator.vue';

defineOptions({
    layout: {
        breadcrumbs: [
            {
                title: 'Projets',
                href: projects.index(),
            },
            {
                title: 'Créer',
                href: projects.create(),
            },
        ],
    },
});
</script>

<template>

    <Head title="Créer un projet" />

    <div class="px-4 py-6">
        <Heading title="Projet" description="Créer un nouveau projet" />

        <div class="flex flex-col lg:flex-row lg:space-x-12">

            <Separator class="my-6 lg:hidden" />

            <div class="flex-1 md:max-w-2xl">
                <section class="max-w-xl space-y-12">

                    <div class="flex flex-col space-y-6">
                        <Heading variant="small" title="Informations sur le projet"
                            description="Mettez à jour le titre et la description du projet" />

                        <Form v-bind="ProjectController.store.form()" class="space-y-6" v-slot="{
errors, processing }">
                            <div class="grid gap-2">
                                <Label for="title">Titre</Label>
                                <Input id="title" class="mt-1 block w-full" name="title"
                                    placeholder="Titre du projet" />
                                <InputError class="mt-2" :message="errors.title" />
                            </div>

                            <div class="grid gap-2">
                                <Label for="description">Description</Label>
                                <Textarea id="description" class="mt-1 block w-full"
name="description" rows="10"
```

```
                                placeholder="Description du projet" />
                            <InputError class="mt-2" :message="errors.description" />
                        </div>

                        <div class="flex items-center gap-4">
                            <Button :disabled="processing">Enregistrer</Button>
                        </div>
                    </Form>
                </div>
            </section>
        </div>
    </div>
</div>
</template>
```

Le `v-bind="ProjectController.store.form()"` câble automatiquement la méthode HTTP, l'URL et le CSRF. Les erreurs de validation arrivent dans le slot `errors` et s'affichent grâce à `<InputError>`. Pas une ligne d'axios écrite.

Méthode `store`

Validation côté serveur, création, redirection, message flash :

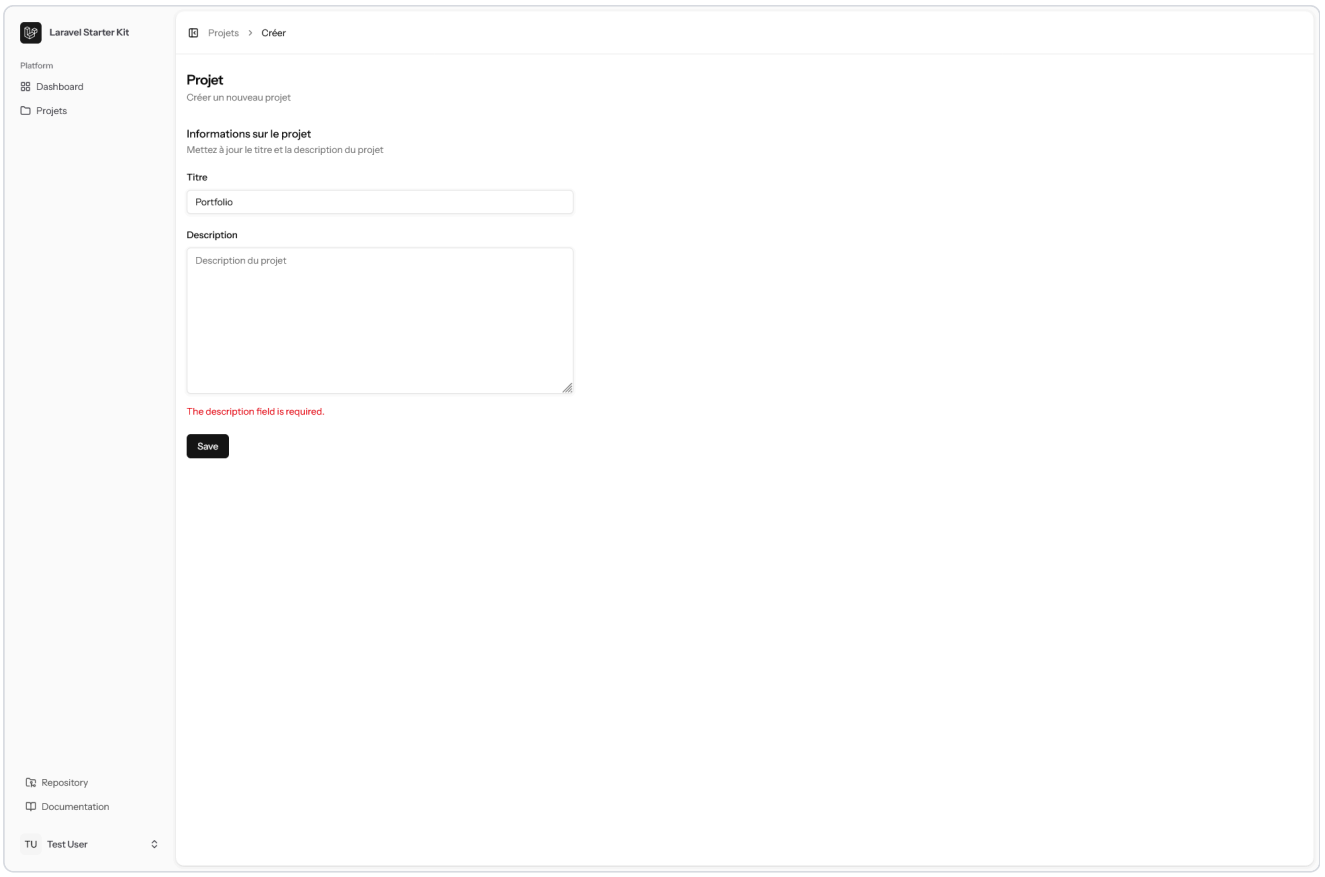
```php
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|string|max:255',
        'description' => 'required|string|max:65535',
    ]);

    Project::create($validated);

    Inertia::flash('toast', ['type' => 'success', 'message' => __('Projet créé avec succès.')]);

    return redirect()->route('projects.index');
}
```

> 💡 *Le composant `Toast` est déjà branché sur cette prop `toast` dans le layout du starter kit. Tant que tu envoies un objet `{ type, message }` via `Inertia::flash`, ça s'affiche automatiquement. Pas de magie noire à comprendre, juste à savoir que c'est là.*



Méthode `edit` et vue Edit

Le contrôleur transmet cette fois le projet à éditer :

```php
public function edit(Project $project)
{
    return inertia('Projects/Edit', [
        'project' => $project,
    ]);
}
```

> 💡 *Laravel utilise l'**injection de modèle** : tu déclares `Project $project` en paramètre, et il va le chercher automatiquement à partir de l'ID dans l'URL. Si le projet n'existe pas, c'est un 404 direct.*

La vue `resources/js/Pages/Projects/Edit.vue` est quasi identique à `Create.vue`, avec deux différences : on pré-remplit les champs via `:default-value`, et on bind `ProjectController.update.form({ project: project.id })` pour cibler la bonne route.

VUE

```
<script setup>
import { Form, Head } from '@inertiajs/vue3';
import Heading from '@/components/Heading.vue';
import InputError from '@/components/InputError.vue';
import { Button } from '@/components/ui/button';
import { Input } from '@/components/ui/input';
import { Label } from '@/components/ui/label';
import projects from '@/routes/projects';
import ProjectController from '@/actions/App/Http/Controllers/ProjectController';
import Textarea from '@/components/ui/textarea/Textarea.vue';
import Separator from '@/components/ui/separator/Separator.vue';

defineOptions({
    layout: {
        breadcrumbs: [
            {
                title: 'Projets',
                href: projects.index(),
            },
            {
                title: 'Éditer',
            },
        ],
    },
});

const props = defineProps({
    project: Object,
});
</script>

<template>

    <Head title="Éditer un projet" />

    <div class="px-4 py-6">
        <Heading title="Projet" description="Éditer un projet existant" />

        <div class="flex flex-col lg:flex-row lg:space-x-12">

            <Separator class="my-6 lg:hidden" />

            <div class="flex-1 md:max-w-2xl">
                <section class="max-w-xl space-y-12">

                    <div class="flex flex-col space-y-6">
                        <Heading variant="small" title="Informations sur le projet"
                            description="Mettez à jour le titre et la description du projet" />

                        <Form v-bind="ProjectController.update.form({ project: project.id })"
class="space-y-6"
                            v-slot="{ errors, processing }">
                            <div class="grid gap-2">
                                <Label for="title">Titre</Label>
                                <Input id="title" class="mt-1 block w-full" name="title" :default-
value="project.title"
                                    placeholder="Titre du projet" />
                                <InputError class="mt-2" :message="errors.title" />
                            </div>
```

```
                            <div class="grid gap-2">
                                <Label for="description">Description</Label>
                                <Textarea id="description" class="mt-1 block w-full"
name="description" rows="10"
                                    placeholder="Description du projet" :default-
value="project.description" />
                                <InputError class="mt-2" :message="errors.description" />
                            </div>

                            <div class="flex items-center gap-4">
                                <Button :disabled="processing">Enregistrer</Button>
                            </div>
                        </Form>
                    </div>
                </section>
            </div>
        </div>
    </div>
</template>
```

Méthode `update`

Même logique que `store`, mais on appelle `update()` sur l'instance existante :

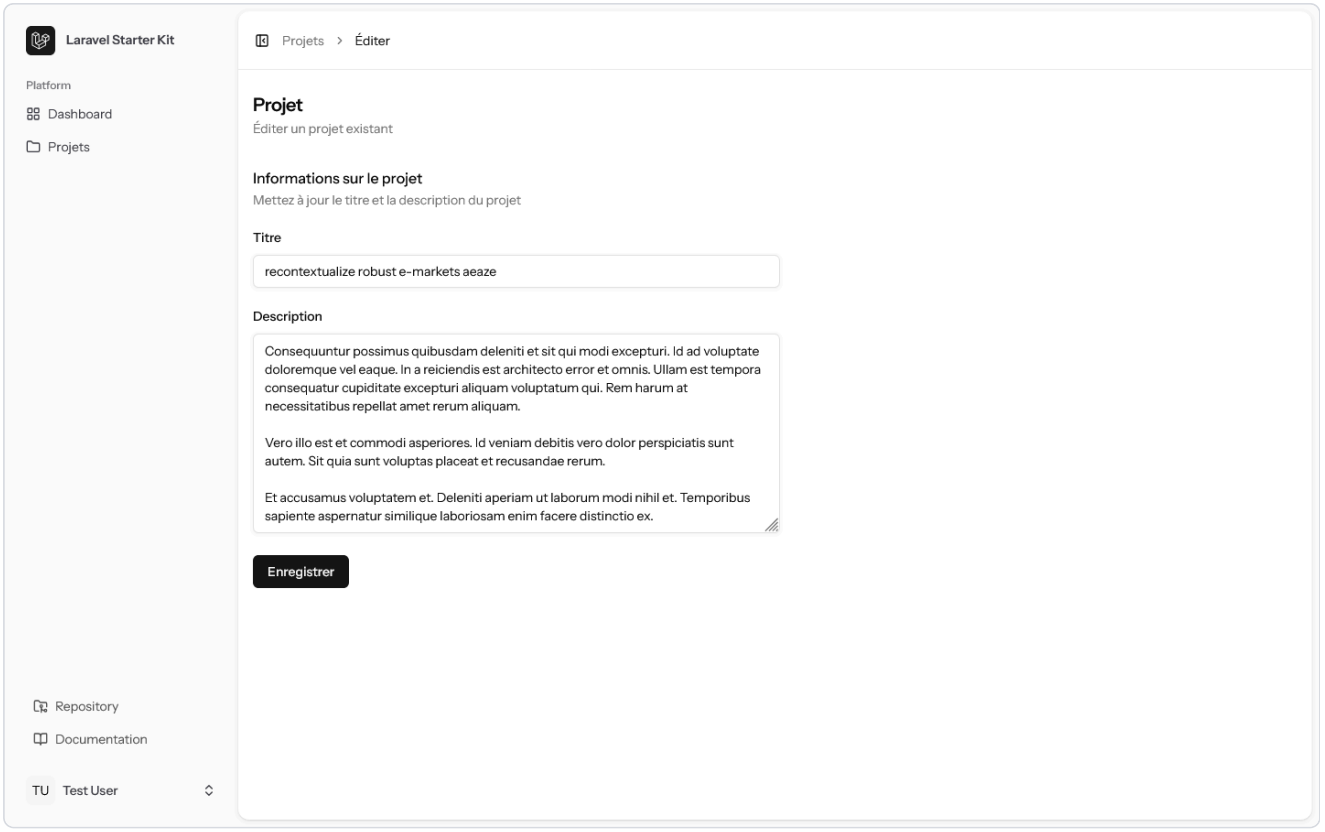
PHP

```php
public function update(Request request,Projectrequest, Project request,Projectproject)
{
    validated=validated = validated=request->validate([
        'title' => 'required|string|max:255',
        'description' => 'required|string|max:65535',
    ]);

    project->update(project->update(project->update(validated);

    Inertia::flash('toast', ['type' => 'success', 'message' => __('Projet mis à jour avec
succès.')]);

    return redirect()->route('projects.index');
}
```



Méthode `destroy`

Avant de coder, une règle d'or : **jamais de suppression sans confirmation**. Cliquer sur « Supprimer » par erreur sur 6 mois de travail, c'est le genre d'expérience qu'on évite à ses utilisateurs.

Côté contrôleur, rien de sorcier :

```php
public function destroy(Project $project)
{
    $project->delete();

    Inertia::flash('toast', ['type' => 'success', 'message' => __('Projet supprimé avec succès.')]);

    return redirect()->route('projects.index');
}
```

Côté front, on utilise un `Dialog` shadcn (doc). Le starter kit en contient déjà un exemple dans `DeleteUser.vue` — calque-toi dessus. On crée donc un composant dédié `DeleteProject.vue` :

VUE

```vue
<script setup>
import { Form } from '@inertiajs/vue3';
import ProjectController from '@/actions/App/Http/Controllers/ProjectController';
import { Button } from '@/components/ui/button';
import {
    Dialog,
    DialogClose,
    DialogContent,
    DialogDescription,
    DialogFooter,
    DialogHeader,
    DialogTitle,
    DialogTrigger,
} from '@/components/ui/dialog';

const props = defineProps({
    project: Object,
});
</script>

<template>
    <Dialog>
        <DialogTrigger as-child>
            <Button variant="destructive" class="ml-2">Supprimer</Button>
        </DialogTrigger>
        <DialogContent>
            <Form v-bind="ProjectController.destroy.form({ id: project.id })" :options="{
                preserveScroll: true,
            }" class="space-y-6" v-slot="{ processing }">
                <DialogHeader class="space-y-3">
                    <DialogTitle>Êtes-vous sûr de vouloir supprimer ce projet ?</DialogTitle>
                    <DialogDescription>
                        Cette action est irréversible, toutes les données liées à ce projet seront
                        définitivement supprimées.
                    </DialogDescription>
                </DialogHeader>

                <DialogFooter class="gap-2">
                    <DialogClose as-child>
                        <Button variant="secondary">
                            Annuler
                        </Button>
                    </DialogClose>

                    <Button type="submit" variant="destructive" :disabled="processing">
                        Supprimer
                    </Button>
                </DialogFooter>
            </Form>
        </DialogContent>
    </Dialog>
</template>
```

On remplace ensuite le bouton « Supprimer » provisoire de `Index.vue` par ce composant, et on en profite pour tronquer les descriptions trop longues :

VUE

```vue
<script setup>
import { Head } from '@inertiajs/vue3';
import { create, edit, index } from '@/routes/projects';
import { Card, CardContent, CardFooter, CardHeader, CardTitle } from '@/components/ui/card';
import TextLink from '@/components/TextLink.vue';
import DeleteProject from '@/components/DeleteProject.vue';

defineOptions({
    layout: {
        breadcrumbs: [
            {
                title: 'Projets',
                href: index(),
            },
        ],
    },
});

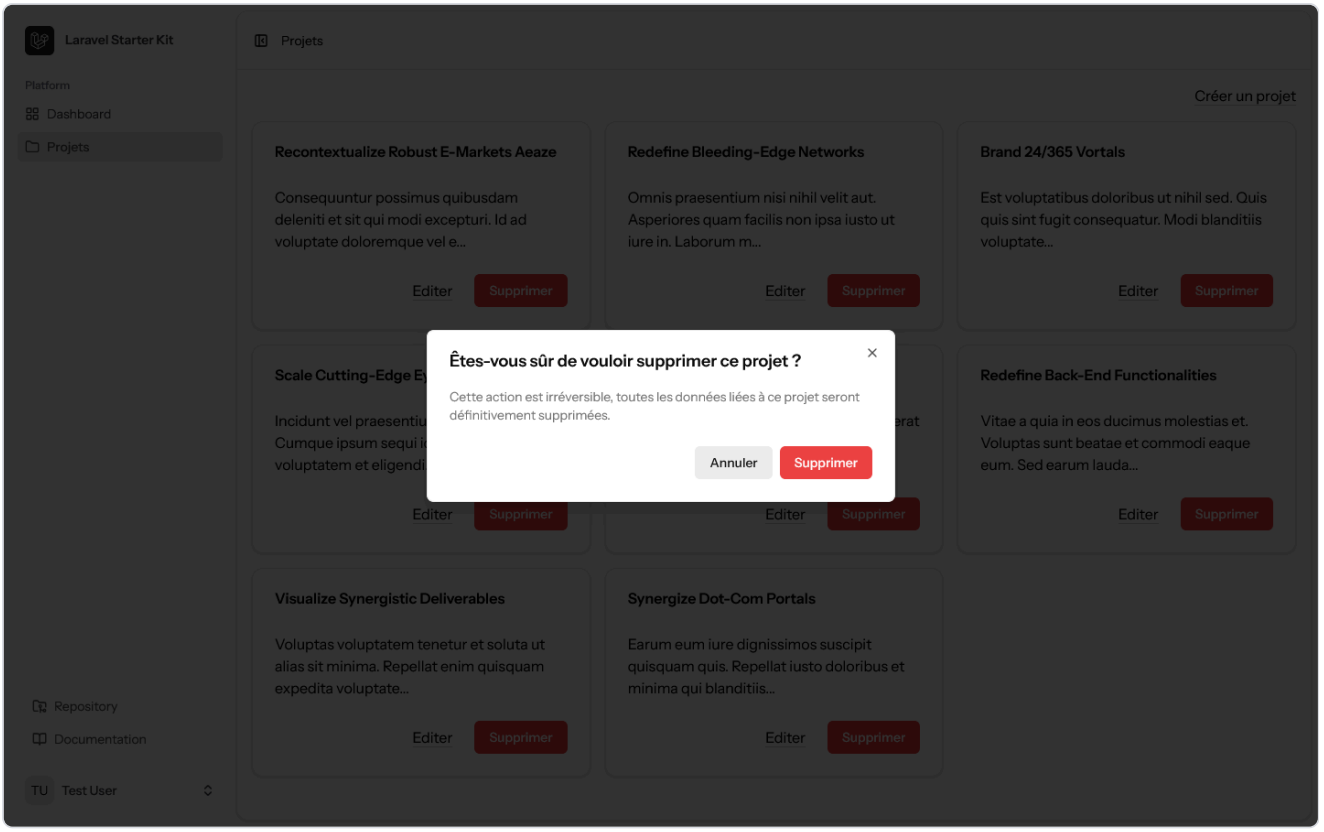
defineProps({
    projects: Array,
});

const truncate = (text, length) => {
    if (text.length <= length) {
        return text;
    }
    return text.substring(0, length) + '...';
};
</script>

<template>

    <Head title="Projets" />

    <div class="flex h-full flex-1 flex-col gap-4 overflow-x-auto rounded-xl p-4">
        <div class="flex items-center justify-end">
            <TextLink :href="create()">
                Créer un projet
            </TextLink>
        </div>
        <div class="grid auto-rows-min gap-4 md:grid-cols-3">
            <Card v-for="project in projects" :key="project.id">
                <CardHeader>
                    <CardTitle class="capitalize">{{ project.title }}</CardTitle>
                </CardHeader>
                <CardContent class="flex-1">
                    <p>{{ truncate(project.description, 100) }}</p>
                </CardContent>
                <CardFooter class="flex justify-end space-x-4">
                    <TextLink :href="edit(project.id)">
                        Editer
                    </TextLink>
                    <DeleteProject :project="project" />
                </CardFooter>
            </Card>
        </div>
    </div>
</template>
```



> 💡 *On n'a pas implémenté la méthode `show()` du contrôleur : la route existe (créée par `Route::resource`), mais on s'en servira plus tard pour la page de détail publique d'un projet.*

La homepage publique

L'admin est fini. Place au frontoffice — la partie que verront tes visiteurs. On veut une page d'accueil qui liste les projets, accessible sans connexion.

On génère un contrôleur dédié dans un sous-namespace `Front` pour bien séparer ce qui est public de ce qui est admin :

```bash
php artisan make:controller Front/HomepageController
```

Dans `routes/web.php`, on remplace la route d'accueil par défaut :

```php
Route::get('/', HomepageController::class)->name('home');
```

Pas de méthode dans l'URL ? C'est parce que `HomepageController` est **invokable** — il n'a qu'une seule méthode publique, `__invoke()`, qu'on peut donc invoquer directement :

```php
class HomepageController extends Controller
{
    public function __invoke()
    {
        projects=Project::all();returninertia('Homepage',['projects'⇒projects = Project::all();

        return inertia('Homepage', [
            'projects' ⇒ projects=Project::all();returninertia('Homepage',['projects'⇒projects,
        ]);
    }
}
```

> 💡 *Les contrôleurs invokables sont parfaits pour les actions uniques (une page statique, un endpoint isolé). Au-delà d'une méthode, repasse sur un contrôleur classique.*

Un layout pour le frontoffice

L'admin utilise `AppLayout` (sidebar, breadcrumbs, etc.). Pour le public, on veut une mise en page différente — barre de navigation simple, footer. On crée `resources/js/layouts/FrontLayout.vue` :

```vue
<script setup lang="ts">
import { Link } from '@inertiajs/vue3';

const { title = '', description = '' } = defineProps<{
    title?: string;
    description?: string;
}>();
</script>

<template>
    <div class="min-h-screen flex flex-col bg-gray-100">
        <nav class="flex items-center justify-between p-4 bg-lime-200 border-b border-lime-300">
            <Link href="/" class="text-xl font-bold">
                Portfolio
            </Link>
            <div class="space-x-4">
                <Link href="/">Accueil</Link>
                <Link href="#">Projets</Link>
                <Link href="#">Contact</Link>
            </div>
        </nav>

        <div class="pt-4 pb-8 px-4">
            <h1 class="text-2xl font-bold p-4">{{ title }}</h1>
            <p class="text-gray-600 px-4">{{ description }}</p>
        </div>

        <slot />

        <footer class="p-4 bg-gray-100 flex items-center justify-center mt-auto">
            <p>&copy; {{ new Date().getFullYear() }} Portfolio. Tous droits réservés.</p>
        </footer>
    </div>
</template>
```

> ⚠️ *Ce design est délibérément moche. C'est la base à laquelle l'énoncé te demande de mettre ta patte créative — tu n'es pas censé livrer ça en l'état.*

Router les pages vers le bon layout

On veut que les pages dans `Pages/Front/` utilisent `FrontLayout`, les autres restent sur `AppLayout` ou `AuthLayout`. Le second argument de `createInertiaApp.layout` reçoit le nom complet du composant (ex. `Front/Homepage`, `auth/Login`), donc un `switch` suffit :

```typescript
// ...
createInertiaApp({
    title: (title) => (title ? `title-{title} - title-{appName}` : appName),
    layout: (name) => {
        switch (true) {
            case name.startsWith('Front/'):
                return FrontLayout;
            case name.startsWith('auth/'):
                return AuthLayout;
            case name.startsWith('settings/'):
                return [AppLayout, SettingsLayout];
            default:
                return AppLayout;
        }
    },
    // ... autres options
```

La vue Homepage

Enfin, `resources/js/Pages/Front/Homepage.vue` affiche les projets :

VUE

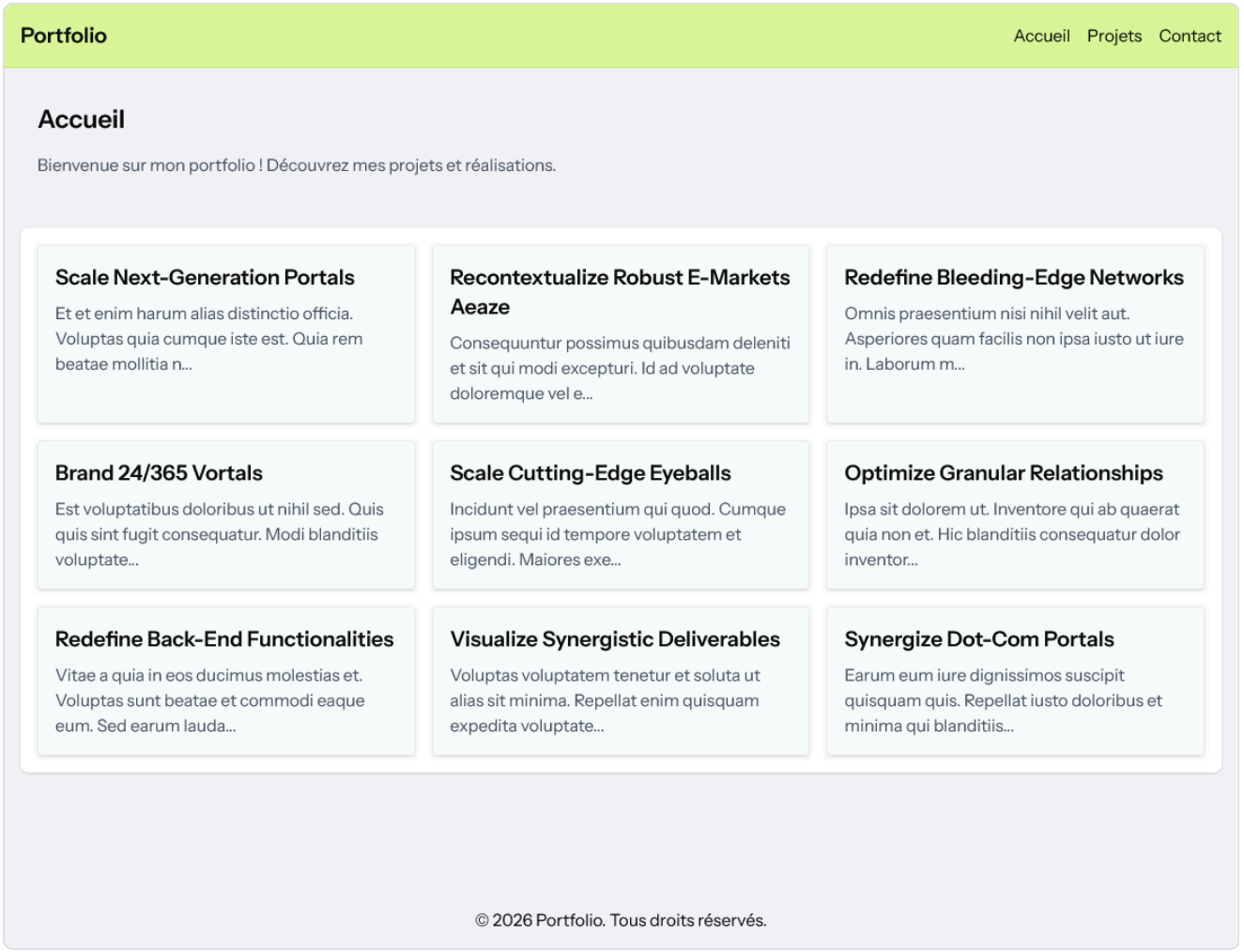
```
<script setup>
import { Link } from '@inertiajs/vue3';

defineProps({
    projects: Array,
});

defineOptions({
    layout: {
        title: 'Accueil',
        description: 'Bienvenue sur mon portfolio ! Découvrez mes projets et réalisations.',
    },
});

const truncate = (text, length) => {
    if (text.length <= length) {
        return text;
    }
    return text.substring(0, length) + '...';
};
</script>

<template>
    <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-4 m-4 p-4 rounded-lg bg-white
shadow">
        <Link href="#" v-for="project in projects" :key="project.id" class="border rounded p-4
shadow bg-gray-50">
            <h2 class="text-xl font-bold mb-2 capitalize">{{ project.title }}</h2>
            <p class="text-gray-600">{{ truncate(project.description, 100) }}</p>
        </Link>
    </div>
</template>
```



Récap

À ce stade tu as :

- ✅ Un modèle `Project` complet (migration, factory, seeder)
- ✅ Un CRUD admin protégé par authentification
- ✅ Des formulaires Inertia avec validation serveur et messages flash
- ✅ Une confirmation de suppression via Dialog
- ✅ Une homepage publique avec son layout dédié

Le squelette est posé. Tout ce qui reste à faire dans l'énoncé suit le même schéma.

Et après ?

L'énoncé liste plusieurs entités à gérer en plus. Trois prochaines étapes recommandées dans l'ordre :

1. **Les technologies** — refais le même CRUD que pour les projets, puis branche la relation many-to-many entre `Project` et `Technology`.
2. **L'upload d'images** (image principale + galerie) pour les projets. Garde les `5MB` max et la validation de format en tête.

3. **La page de détail publique** d'un projet (`Front/ProjectDetail.vue`) — c'est là qu'on implémentera enfin la méthode `show()` du contrôleur.

> 💡 ***Bonne pratique*** *: commit après chaque feature qui marche. Un projet sans historique git, c'est un projet sans filet de sécurité.*

Lien GitHub

https://github.com/opmvpc/bac2-portfolio-26

3.12 SEO On-Page – Séduire Google (et les Humains)

Le SEO On-Page, c'est l'art de dire à Google : *"Eh, regarde ! Ma page est super bien rangée et elle parle exactement de ce que cherche cet utilisateur."*

Si votre code est une jungle, Google passera son chemin. Voici les règles d'or pour structurer votre contenu sans vous prendre la tête.

I. Le Squelette : Structurez votre HTML (Hn)

Imaginez votre page comme un livre. Google a besoin d'un sommaire pour comprendre de quoi vous parlez. C'est le rôle des balises `H1` , `H2` , `H3` ...

- **H1 (Le Titre du Livre) :** Il n'y en a **qu'un seul** par page. Il doit contenir votre mot-clé principal.
- **H2 (Les Chapitres) :** Ils séparent les grandes idées.
- **H3 (Les Sous-parties) :** Pour aller dans le détail.

> ***La Règle d'Or :*** *Ne sautez jamais de niveau (pas de H4 direct après un H1) et n'utilisez pas les titres juste pour "mettre en gros". C'est du sens, pas de la déco.*

Comparatif Express :

✅ Le bon élève	❌ Le mauvais élève
`<h1>` Recette du Gâteau au Chocolat `</h1>`	`<h1>` Bienvenue `</h1>`
`<h2>` Les Ingrédients `</h2>`	`<h2>` Menu `</h2>`
`<h3>` Choix du chocolat `</h3>`	`<h4>` Mentions Légales `</h4>`
Hiérarchie logique et descriptive.	*Aucun sens, titres génériques.*

II. La Vitrine : Meta Title & Description

C'est ce qui apparaît dans les résultats de recherche (la SERP). C'est votre **seule chance** de convaincre l'internaute de cliquer sur votre lien plutôt que sur celui du voisin.

1. Le Meta Title (L'enseigne du magasin)

C'est le facteur le plus important pour le classement.

- **Longueur :** 50-60 caractères (sinon Google le coupe...).
- **Contenu :** Mot-clé principal au début + une accroche.

2. La Meta Description (Le vendeur)

Elle ne joue pas directement sur le classement, mais elle booste votre taux de clic (CTR). Donnez envie !

L'exemple qui tue :

- **Mauvais (L'ennui total) :**
  - Title: `Page Recette 01`
  - Desc: `Voici une recette de gâteau.`
  - *Résultat : Personne ne clique.*
- **Excellent (Sexy & Précis) :**
  - Title: `Gâteau au Chocolat Moelleux : Recette Facile en 15 min`
  - Desc: `Envie de chocolat ? Testez notre recette inratable ! Croustillant dehors, fondant dedans. Pas besoin d'être un chef pour régaler vos amis.`
  - *Résultat : On salive, on clique.*

III. Les Yeux de Google : Balises Alt (Images)

Google est aveugle. Si vous ne lui dites pas ce qu'il y a sur l'image, il ne voit qu'un fichier nommé `IMG_5422.jpg`.

La balise `alt` sert à deux choses : le référencement (Google Images) et l'accessibilité (lecteurs d'écran pour malvoyants).

- ❌ `<img src="choco.jpg" alt=""/>` (Le vide intersidéral)
- ✅ `<img src="choco.jpg" alt="Part de gâteau au chocolat fondant avec une framboise">` (Descriptif et précis)

IV. Le Carburant : Les Mots-Clés

Le temps où l'on répétait 50 fois le même mot est révolu (c'était en 2005, passez à autre chose). Aujourd'hui, on vise **l'intention de recherche**.

Comment les choisir ?

Mettez-vous à la place de l'utilisateur. Que tape-t-il dans la barre de recherche ?

- **Trop générique :** "Chocolat" (Trop de concurrence, intention floue).

- **La pépite (Longue traîne) :** "Recette gâteau chocolat sans gluten rapide". Moins de volume, mais 100% pertinent.

Où les placer ?

Saupoudrez-les naturellement, comme du sel :

1. Dans le **H1** (Indispensable).
2. Dans l'introduction (Le premier paragraphe).
3. Dans un ou deux **H2**.
4. Dans la balise **Title** et l'**URL**.

> ⚠️ *Attention au "Keyword Stuffing" :*
>
> *"Ce gâteau au chocolat est le meilleur gâteau au chocolat pour manger du chocolat."*
>
> *→ Google vous punira pour spam. Écrivez pour les humains d'abord !*

📝 Exercice : "Pimp My Page"

À vous de jouer ! Prenez une page de votre projet actuel et faites-lui un relooking extrême :

1. **Check-up Structure :** Vérifiez que vous avez un seul H1 unique et pertinent.
2. **Opération Séduction :** Réécrivez votre `<title>` et votre `meta description`. Soyez vendeur, soyez précis.
3. **Chasse aux fantômes :** Ajoutez des balises `alt` descriptives à toutes vos images.

3.13 Open Graph – Ne soyez pas moche sur les réseaux

Vous avez soigné votre SEO pour plaire à Google. Super. Mais que se passe-t-il quand quelqu'un partage votre lien sur Facebook, LinkedIn, WhatsApp ou Slack ?

Sans Open Graph (OG), c'est la roulette russe : une image coupée, un titre bizarre ou pire... rien du tout.

Avec les balises OG, vous prenez le contrôle de votre apparence. C'est la différence entre tendre une carte de visite dorée et gribouiller son nom sur une serviette en papier.

I. C'est quoi ce truc ?

Le protocole "Open Graph" a été inventé par Facebook, mais aujourd'hui **tous** les réseaux sociaux l'utilisent (LinkedIn, Twitter/X, Discord, etc.).

C'est simple : ce sont des balises cachées dans le `<head>` de votre code qui disent aux réseaux sociaux :

> *"Eh ! Quand on partage cette page, affiche CETTE image et CE titre, pas n'importe quoi."*



II. Les 4 Fantastiques (Les Balises Indispensables)

Vous n'avez besoin que de 4 balises pour transformer l'apparence de vos liens.

1. `og:image` (La Star)

C'est l'élément le plus important. Une image vaut 1000 mots, surtout sur un fil d'actualité où l'on scrolle à toute vitesse.

- **Le conseil :** Utilisez une image spécifique, aux bonnes dimensions (idéalement **1200 × 630 pixels**).
- **L'erreur :** Laisser le réseau social choisir une image au hasard (souvent votre logo tout petit ou une icône de pied de page).

2. `og:title` (L'Accroche)

Similaire à votre balise `<title>` SEO, mais vous pouvez être plus "clicky" ou conversationnel ici.

- **Exemple :** "Découvrez notre recette !" (Un peu mou) vs "Le Gâteau au Chocolat qui a cassé Internet 🍫" (Mieux).

3. `og:description` (Le Teasing)

Un court résumé (2-3 lignes max). Contrairement à la meta description Google, celle-ci est souvent affichée en gros sous le titre sur Facebook/LinkedIn.

4. `og:url` (L'Adresse Officielle)

L'URL canonique de la page (sans les paramètres de tracking bizarres).

III. Le Choc des Photos : Avant / Après

Visualisez le partage de votre lien sur un fil LinkedIn ou dans une conversation WhatsApp :

🤢 Sans Open Graph (Le mode "Loto")	😎 Avec Open Graph (Le mode "Pro")
Image : Aucune, ou une icône pixélisée coupée en deux.	**Image :** Une grande bannière visuelle qui claque.
Titre : "Accueil" ou "index.html".	**Titre :** "Boostez votre Productivité : Le Guide".
Desc : "Copyright 2024... Mentions légales..." (Il a pris le premier texte qu'il a trouvé).	**Desc :** "5 méthodes simples pour gagner 2h par jour."
Résultat : On dirait un site cassé ou un spam.	*Résultat : On a confiance, on clique.*

IV. Code : À quoi ça ressemble ?

C'est très simple à ajouter dans votre HTML (`<head>`) :

HTML

```xml
<meta property="og:title" content="Le Guide Ultime du SEO" />
<meta property="og:type" content="article" />
<meta property="og:url" content="https://monsite.com/guide-seo" />
<meta property="og:image" content="https://monsite.com/images/banner-social.jpg" />
<meta property="og:description" content="Tout comprendre au référencement en 5 minutes chrono." />
```

> 💡 *Le cousin Twitter :*
>
> *Twitter (X) utilise l'Open Graph, mais préfère ses propres balises pour un rendu optimal. Ajoutez simplement :*
>
> *<meta name="twitter:card" content="summary_large_image">*
>
> *Cela force l'affichage de la grande image.*

V. L'Outil Magique : Le Debugger

Vous avez ajouté vos balises mais l'image ne s'affiche pas ? C'est souvent une histoire de **cache**. Les réseaux sociaux gardent en mémoire l'ancienne version de votre page.

Pour forcer la mise à jour et prévisualiser le résultat, utilisez les outils officiels (gratuits) :

1. Pour Facebook & Messenger :

   👉 Facebook Sharing Debugger (La référence absolue)

2. Pour LinkedIn :

   👉 LinkedIn Post Inspector (Indispensable pour le B2B)

3. Pour Twitter / X :

👉 Twitter Card Validator

La méthode : Copiez votre URL dedans, cliquez sur "Debug" ou "Inspect", et hop ! Vous voyez exactement ce que verront vos utilisateurs. Si l'image n'apparaît pas, cliquez sur le bouton "Scrape Again" pour forcer le robot à relire votre page.

❗ Attention, il faut que le site soit en ligne ! Pas de localhost.

📝 Exercice Flash

1. Allez sur le **Facebook Sharing Debugger**.
2. Entrez l'URL de votre site préféré (ou le vôtre).
3. Regardez la "Card" générée.
   - L'image est-elle coupée ?
   - Le titre donne-t-il envie ?
4. Si c'était votre site, quelle image auriez-vous choisie pour obtenir plus de clics ?

Et voilà pour la technique !

On a couvert la **structure** (SEO On-Page) et **l'apparence sociale** (Open Graph).

3.14 Intégration des Outils Google

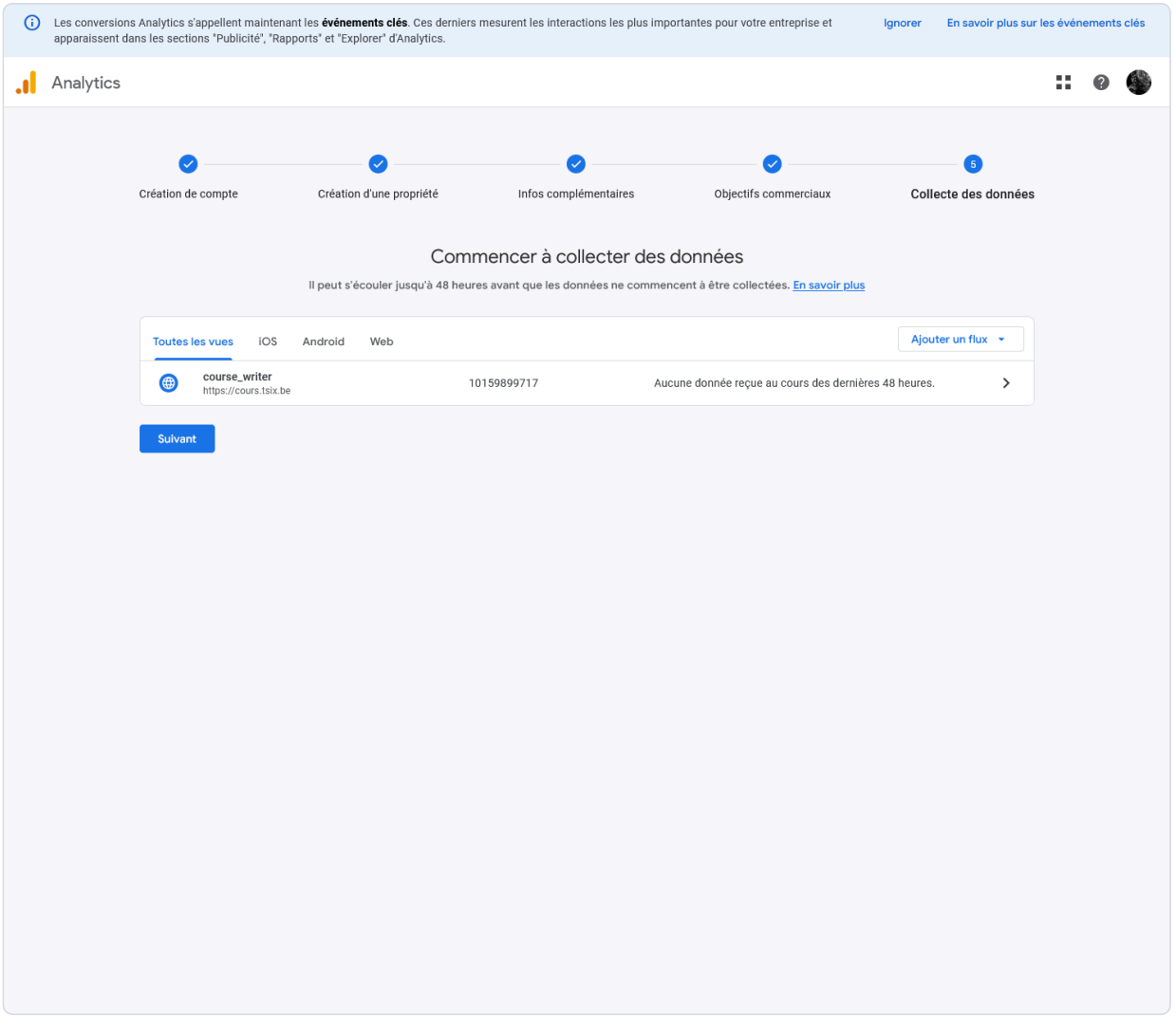
Pourquoi Google Analytics et Search Console ?

Si vous comptez avancer à l'aveugle, libre à vous de **ne rien installer**. Mais si, par chance, vous voulez un minimum de visibilité sur vos performances (et vos échecs), vous allez avoir besoin de **Google Analytics (GA)** et **Google Search Console (GSC)**.

- **Google Analytics** : Mesure le trafic de votre site (nombre de visiteurs, pages vues, taux de rebond, etc.).
- **Google Search Console** : Surveille l'indexation par Google, vos positions, votre CTR (taux de clic), et détecte les erreurs (erreurs 404, pages bloquées, etc.).

Mise en Place de Google Analytics

1. **Création du Compte**
   - Rendez-vous sur Google Analytics et suivez les instructions pour créer un compte.
   - Choisissez **GA4** (la nouvelle version) ou configurez la "propriété" en Universal Analytics si vous avez des besoins spécifiques (mais GA4 est la norme actuelle).



2. Installation du Script sur Votre Site

- **Bon exemple** :

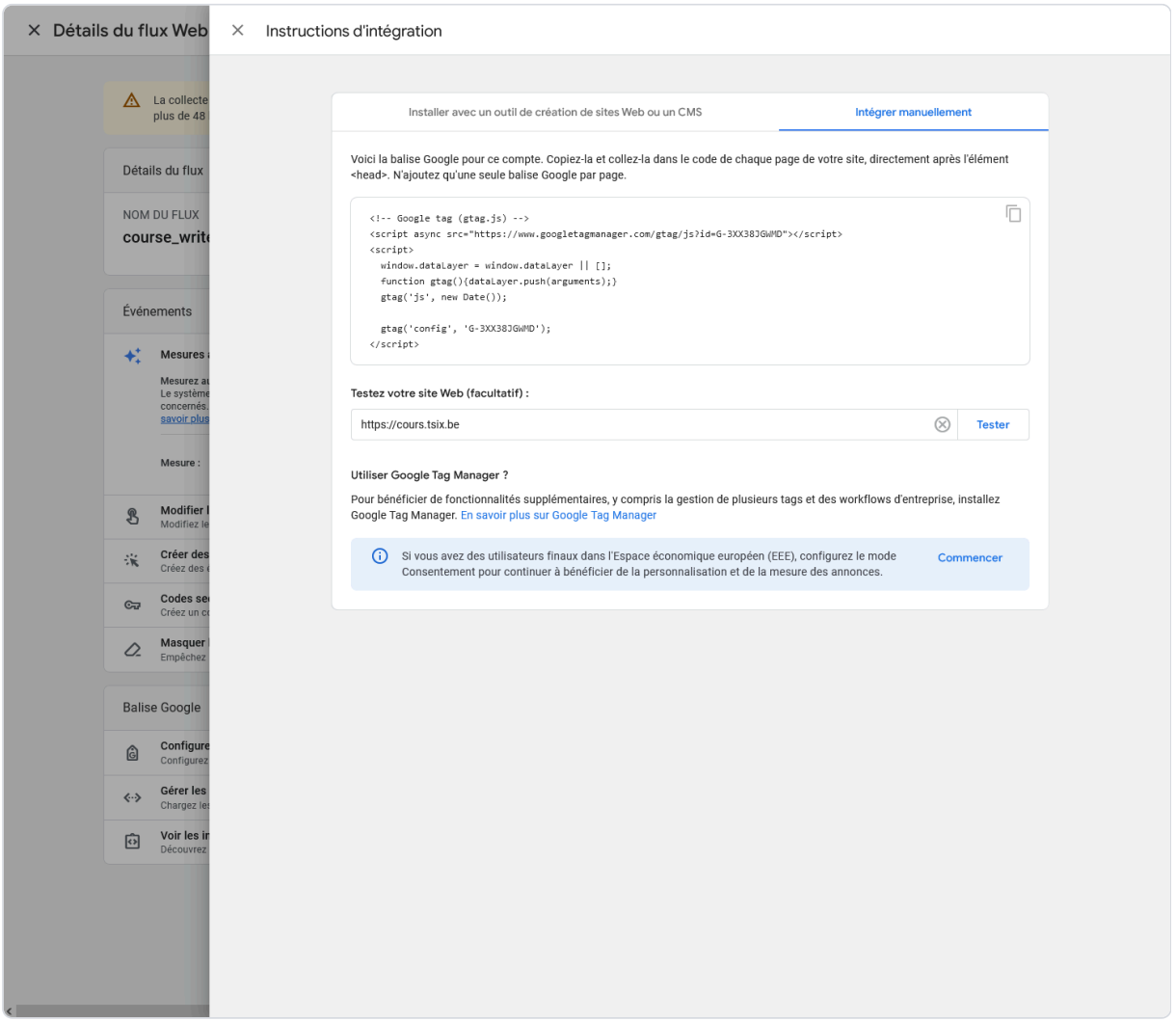
```XML
<!-- Google tag (gtag.js) -->
<script async src="https://www.googletagmanager.com/gtag/js?id=G-XXXXXXXXXX"></script>
<script>
  window.dataLayer = window.dataLayer || [];
  function gtag(){dataLayer.push(arguments);}
  gtag('js', new Date());
  gtag('config', 'G-XXXXXXXXXX');
</script>
```

  - Script placé dans le `<head>` (avant la balise `</head>`) pour charger rapidement et ne pas perturber vos stats.

- **Mauvais exemple** :

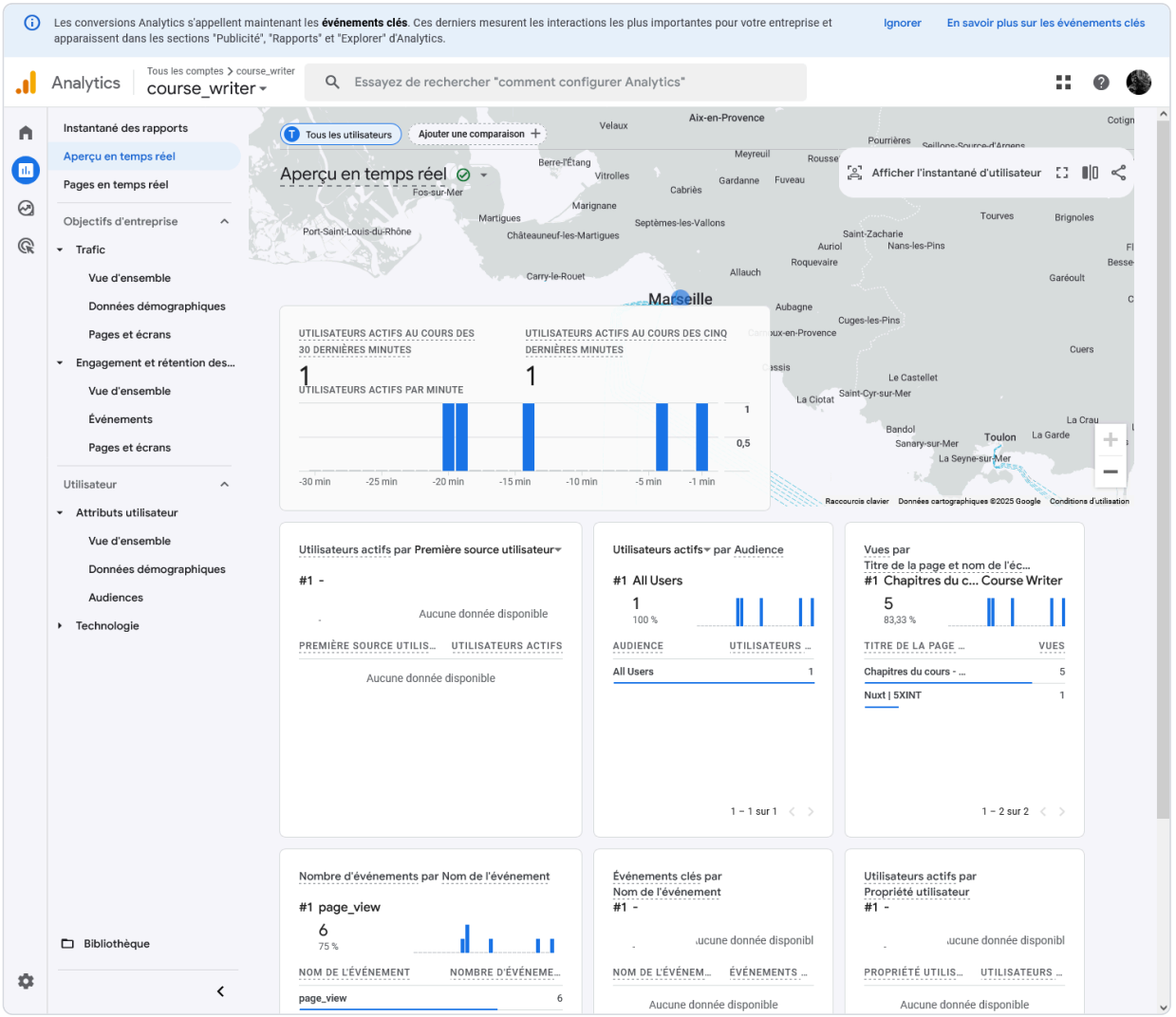
```XML
<!-- Code balancé n'importe où, ou manquant -->
<script>
  gtag('config', 'G-XXXXXXXXXX'); // sans avoir chargé gtag.js
</script>
```

- Si vous oubliez d’inclure la bibliothèque, Analytics ne pourra pas collecter vos données.



3. Vérification

- Accédez à votre interface GA et vérifiez si vous voyez de la “vraie” activité en temps réel (ou testez en ouvrant le site dans un autre onglet).
- Si vous voyez zéro visite malgré vos tests, vous avez peut-être tout cassé.



Mise en Place de Google Search Console

1. **Création et Validation de la Propriété**
   - Rendez-vous sur Google Search Console.
   - Ajoutez votre site en tant que “propriété” (vous pouvez choisir la validation via **fichier HTML**, **meta tag**, ou **DNS**).
   - **Bon exemple** (meta tag dans `<head>`) :

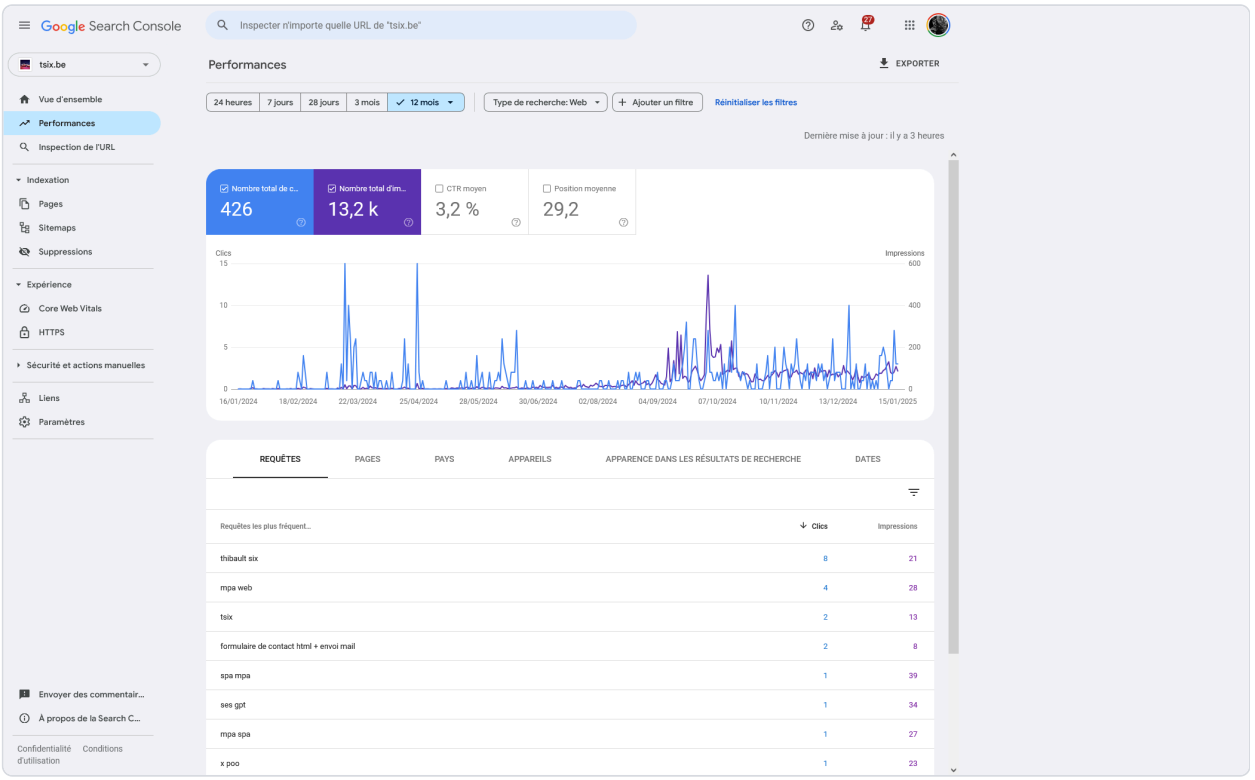
     ```xml
     <meta name="google-site-verification" content="votre_code_verification" />
     ```

   - **Mauvais exemple** :
     - Ne jamais finaliser la validation, ou mettre la balise dans un `footer`. Google ne la verra pas avant l’extinction du soleil.
2. **Soumission du Sitemap** (optionnel mais utile)
   - Un **sitemap** au format XML peut être généré par des plugins ou des outils en ligne (surtout si vous utilisez un CMS).

- Dans la Search Console, allez dans **“Sitemaps”** et entrez l’URL de votre sitemap (ex : `https://votresite.com/sitemap.xml`).
- Google indexera plus facilement vos pages si votre sitemap est structuré.

3. **Analyse Rapide**

- Observez l’onglet **Couverture** pour détecter les erreurs d’indexation.
- Consultez l’onglet **Performances** pour voir vos mots-clés, votre position moyenne et votre CTR.
- Ignorez les avertissements “mineurs” si vous manquez de temps, mais ne négligez pas les erreurs critiques (pages introuvables, blocages robots.txt...).



Interpréter les Données

1. **Google Analytics**

- **Taux de rebond** : Pourcentage de visiteurs qui quittent le site après une seule page vue. Un taux très élevé peut signifier que vos pages n’intéressent personne (ou qu’elles chargent trop lentement).
- **Pages vues** : Vous pourrez voir quelles pages sont les plus populaires (et lesquelles sont l’équivalent numérique d’un désert).
- **Origines du trafic** : Savoir si vos visiteurs viennent de Google, d’un lien externe, ou d’un flux astral inconnu.

2. **Search Console**

- **Impressions** : Nombre de fois où votre site apparaît dans la SERP.
- **CTR (Click Through Rate)** : Si les utilisateurs cliquent ou pas. Un CTR très bas peut signifier que vos titres et descriptions sont à dormir debout.
- **Position Moyenne** : Rang moyen de votre site sur les mots-clés détectés.

- **Bon exemple** : Chercher à améliorer un mot-clé où vous êtes en 12ème position pour viser le top 10.
- **Mauvais exemple** : Vouloir ranker 1er sur “chocolat” sans stratégie et s’étonner d’être invisible.

Exercice Pratique

1. **Installez GA et GSC**
   - Copiez le script GA4 ou Universal Analytics dans votre `<head>`.
   - Configurez Search Console, validez votre propriété.
2. **Observez vos Stats**
   - Ouvrez votre site sur un autre appareil, allez dans GA > Temps réel. Vérifiez que votre visite est comptabilisée.
   - Dans GSC, patientez un peu (quelques jours) avant de voir des données utiles.

Sans ces outils, vous naviguerez à vue, ce qui est un pari risqué quand on veut optimiser un site. Prochaine étape : se pencher sur **l’importance du contenu**, car sans contenu correct, vous n’aurez pas grand-chose à mesurer.

3.15 🚀 Devoir : La Stratégie SEO de votre Portfolio

Contexte

Vous êtes en train de construire votre site portfolio. C'est votre carte de visite numérique. Mais un beau site ne sert à rien si personne ne le trouve ou si le lien ne donne pas envie de cliquer.

L'objectif de ce travail est de préparer le terrain pour que Google (et les recruteurs) tombent amoureux de votre profil.

Ce que vous devez faire

Créez un document synthétique dans lequel vous définissez la stratégie de référencement de votre futur site.

Votre document doit contenir les **3 parties suivantes** :

1. Votre identité (Mots-Clés)

Listez **5 à 10 mots-clés ou expressions** qui définissent votre profil professionnel et que vous aimeriez cibler.

- *Exemple : "Développeur Fullstack Junior", "Webdesigner Liège", "React & Node.js", etc.*

2. Structure & Optimisation (Le Cœur du devoir)

Pour **3 pages principales** de votre portfolio (par exemple : *Accueil, À Propos, Mes Projets*), rédigez le contenu qui apparaîtra dans les résultats de recherche :

- **Nom de la page**
- **Le Meta Title** (50-60 caractères max) : Il doit être accrocheur et contenir un mot-clé.
- **La Meta Description** (150-160 caractères max) : Elle doit "vendre" le clic et résumer le contenu de la page.

3. Bonus IA 🤖

Vous avez le droit (et même l'encouragement) d'utiliser des outils comme ChatGPT, Claude ou Gemini pour vous aider à trouver des idées ou reformuler vos phrases.

- *Indiquez brièvement si vous avez utilisé une IA et pour quelle partie (ex: "J'ai utilisé ChatGPT pour générer des variantes de mes titres").*

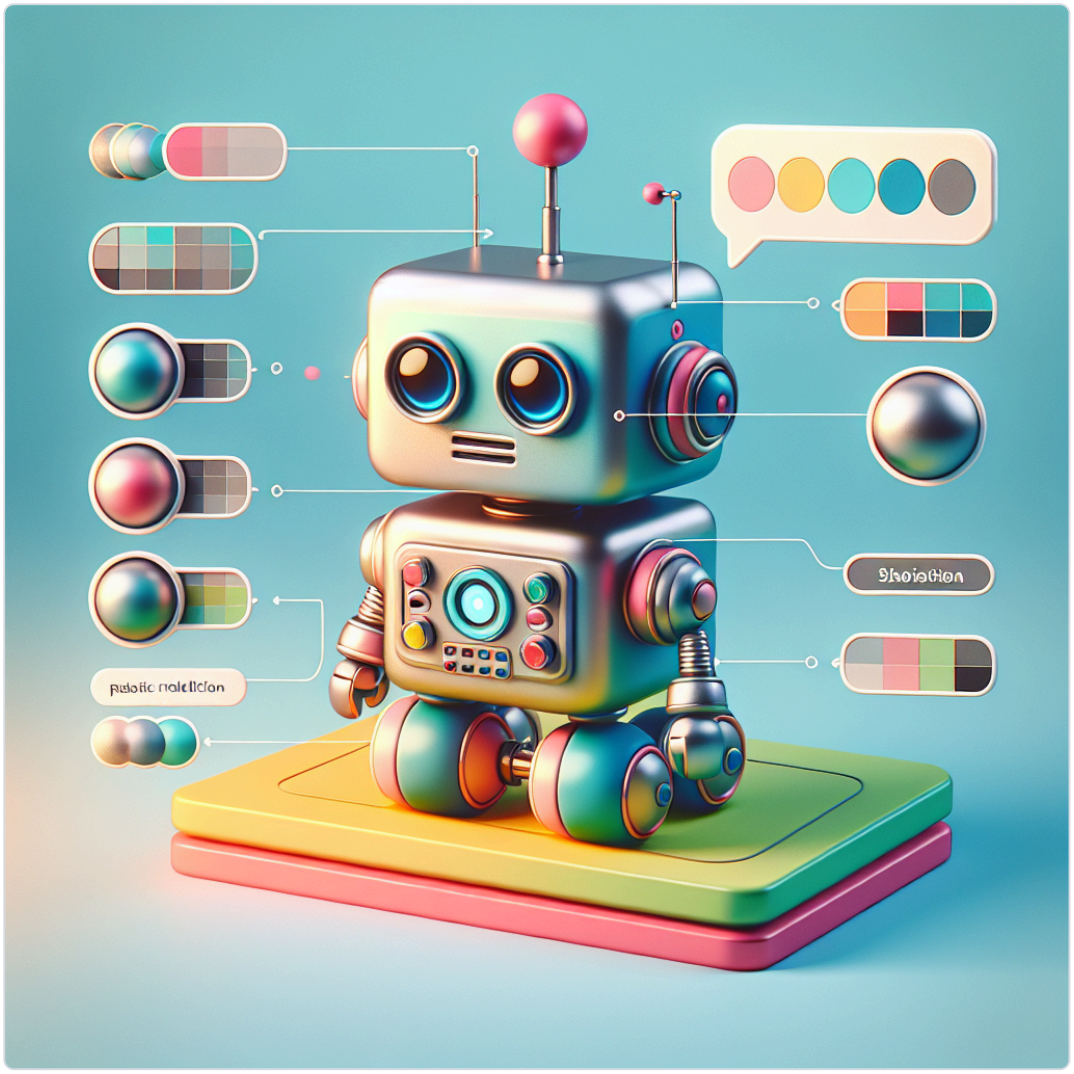
Modalités de rendu

- **Format :** Fichier **PDF** uniquement.
- **Où rendre :** Dans le devoir **Teams** prévu à cet effet.
- **Date limite :** 📅 **24 novembre 2025 à 17h15** (précises !).

Bon travail à tous !

CHAPITRE 4

Mini ChatGPT



Dans ce chapitre, nous explorerons la création d'une application Mini ChatGPT, en mettant l'accent sur l'intégration des technologies modernes pour développer une interface de chat interactive. Nous commencerons par comprendre comment utiliser une **API** pour alimenter notre application, puis nous plongerons dans l'utilisation de **Markdown** pour enrichir le contenu textuel. Vous aurez l'occasion de mettre en pratique vos compétences à travers plusieurs exercices, tels que la réponse à des questions, la création d'une interface de chat et l'implémentation du streaming. Nous discuterons également de la gestion de la **fenêtre de contexte** et de l'importance des **instructions personnalisées** pour améliorer l'interaction utilisateur. Enfin, nous explorerons divers **outils** pour optimiser le développement de votre Mini ChatGPT. Ce chapitre vise à vous fournir les compétences nécessaires pour concevoir des applications de chat intelligentes et dynamiques.

4.1 Projet : Fabrique ton Clone GPT thématisé

Cours : Projet de développement (SGBD) — IDV4

Tu vas reconstruire un ChatGPT-like complet, avec une vraie base de données derrière. Pas un tuto YouTube copié-collé : un projet qui tient debout côté **données**. 💀

1. Objectif

Développer une application **ChatGPT-like** complète, fonctionnelle, et dotée d'une **base de données soignée**. Le cœur de l'évaluation, c'est le SGBD : ton schéma, tes relations, ton intégrité, tes requêtes.

Stack imposée : Laravel + Inertia + Vue 3 (**Composition API obligatoire**) + TailwindCSS + shadcn-vue (starter kit Vue Inertia).

2. Fonctionnalités obligatoires (éliminatoires)

Absentes ou non fonctionnelles = ajournement direct.

1. **Sélecteur de modèles** — changer de LLM facilement (Sonnet 4.5, Gemini, GPT, Grok...)
2. **Historique des conversations** — sauvegarde automatique, titre généré au premier message
3. **Streaming de la réponse** — tokens en temps réel (SSE ou WebSockets), pas d'un bloc
4. **Instructions personnalisées** — configurer le comportement de l'IA (ton, style, contexte)
5. **Composition API** — ⚠️ l'Options API est **ÉLIMINATOIRE**

> *Les tests (Pest / Dusk) ne sont **pas** obligatoires, mais ils rapportent des points (voir grille).*

3. Thème & identité

> *« ChatGPT mais en Laravel » c'est cringe. 🤒 Ton clone doit cibler un public et avoir une personnalité.*

Choisis un thème et construis l'expérience autour. Exemples (non exhaustif) :

Thème	Public
🧠 PsyBot	bien-être mental (avec disclaimers)
💪 CoachFit	sport / fitness
🥗 NutriGuide	nutrition
📚 StudyBuddy	tutorat scolaire
👨‍💻 DevMentor	pair-programming
🎮 GameMaster	maître de jeu JDR
✍️ WriterMuse	écriture créative

Exigences : personnalité cohérente (ton, vocabulaire, émojis adaptés), branding cohérent (couleurs, typo, icônes), instructions système pré-configurées selon le thème.

4. Fonctionnalités supplémentaires (pour viser l'excellence)

C'est ici que tu fais grimper ta note — et plusieurs de ces features font travailler ta base de données 😏 :

- **Arbre de conversations / fork** — modifier un message, relancer une génération, brancher (relation auto-référente côté DB 🤌)

- **Suivi des tokens & coûts** — dashboard analytique, stats d'usage, coûts estimés (tables d'analytics, agrégations)
- **Tags / dossiers** de conversations (relation N-N)
- **Partage de conversations** (table pivot, permissions)
- **Upload & analyse d'images** (modèles vision)
- **Génération d'images** via OpenRouter (modèles économiques, < 1 ct/image)
- **Mode thinking** — afficher la trace de réflexion
- **Outils LLM** — calculatrice, recherche web, analyse de documents
- **Recherche full-text** dans l'historique
- **Voice input/output**, mode sombre/clair, multilingue, export (PDF/MD/JSON)
- **Tes propres idées** — surprends-moi, la créativité est récompensée

5. Livrables

1. **Code source** sur GitHub (repository **public**, historique de commits propre)
2. **Rapport PDF** (template fourni séparément)
3. **Cahier métacognitif PDF** d'usage de l'IA (template fourni séparément)

Deadline : *Samedi 20 juin 2026 à 23h59* — aucun retard accepté.

6. Défense orale (≈ 15 min)

1. **Questions sur le projet** : schéma de la base, relations & intégrité, implémentation du streaming, choix techniques, difficultés rencontrées.
2. **2 questions théoriques tirées au sort**, par exemple :
   - Qu'est-ce qu'un ORM ? Avantages / inconvénients ?
   - Qu'est-ce que l'eager loading et pourquoi c'est important (problème N+1) ?
   - À quoi servent les migrations ? Les seeders ? Les factories ?
   - Qu'est-ce qu'une contrainte d'intégrité référentielle ? `CASCADE` ?
   - À quoi sert un index ? Quand en mettre (ou pas) ?
   - Qu'est-ce qu'une transaction ?
   - Différence entre WebSockets et Server-Sent Events ?
   - Qu'est-ce que la Composition API et ses avantages ?

7. Politique d'usage de l'IA

Usage	Autorisé ?
IA pour la **rédaction du rapport**	✅ Oui
IA pour l'**assistance au code** : tab completion, app de chat pour poser des questions	✅ Oui
Agents de programmation (qui réalisent le projet à ta place)	❌ **Non**
Documentation officielle (référencée dans le cours) + Google	✅ Oui, à l'ancienne 😎

Tu dois joindre un **cahier métacognitif** décrivant ton usage de l'IA (template fourni).

8. Évaluation

L'évaluation suit une **grille détaillée fournie séparément** : 3 Acquis d'Apprentissage à valider (seuil de réussite) + 6 critères de maîtrise. L'accent est mis sur la **base de données** et sur les **fonctionnalités supplémentaires**.

- grille_evaluation_IDV4_Mini_Chat_GPT_Version_1-3.pdf

9. Template Rapport

- rapport_template_idv3.docx
- cahier_metacognitif_idv3.docx

Rappel : montre que tu maîtrises les concepts, pas juste que tu sais copier-coller. La qualité de ta base de données fera la différence. Bonne chance ! 🤖🚀

Énoncé rédigé avec l'aide de Claude Opus 4.8.

4.2 Intégration des APIs LLM dans Laravel

1. Introduction : Les APIs LLM en 2025

Bienvenue dans le monde merveilleux des Large Language Models. Vous allez apprendre à faire parler des modèles d'IA depuis votre application Laravel, sans vendre un rein pour payer la facture (enfin, on espère).

Pourquoi intégrer un LLM ?

- Automatiser des tâches de génération de texte
- Créer des chatbots (le fameux "clone ChatGPT" de votre examen 👀)
- Analyser, résumer, transformer du contenu
- Ajouter un effet "wow" à vos projets (ou un effet "flop" si vous vous y prenez mal)

Le paysage actuel

Les providers de LLM (OpenAI, Anthropic, Google, Meta, xAI...) proposent généralement :

- Une **documentation** avec des exemples en Python et TypeScript (parce que PHP c'est pour les boomers, apparemment 🙄)
- Des **packages officiels** pour ces langages
- Un exemple **cURL** pour les autres (c'est nous ça)

C'est ce fameux exemple cURL qui va nous servir de pont vers PHP/Laravel.

2. Comprendre les docs : De cURL à PHP

2.1 C'est quoi cURL ?

cURL (Client URL) est un outil en ligne de commande qui permet d'effectuer des requêtes HTTP. Quand vous voyez un exemple cURL dans une documentation d'API, c'est essentiellement une recette pour envoyer une requête HTTP avec tous les ingrédients nécessaires.

2.2 Anatomie d'une requête API LLM

Prenons l'exemple typique qu'on trouve dans la documentation d'OpenRouter :

```bash
curl https://openrouter.ai/api/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENROUTER_API_KEY" \
  -d '{
    "model": "openai/gpt-5-mini",
    "messages": [
      {
        "role": "user",
        "content": "What is the meaning of life?"
      }
    ]
  }'
```

Décomposons cette requête :

Élément	Signification
`curl https://...`	L'URL de l'endpoint API (où envoyer la requête)
`-H "Content-Type: application/json"`	Header indiquant qu'on envoie du JSON
`-H "Authorization: Bearer ..."`	Header d'authentification avec notre clé API
`-d '{...}'`	Le corps (body) de la requête en JSON

Le body contient deux éléments essentiels :

- **model** : Le modèle à utiliser (ex: `openai/gpt-5-mini`, `anthropic/claude-sonnet-4.5`)
- **messages** : La conversation sous forme de tableau avec des rôles (`system`, `user`, `assistant`)

2.3 Transformer cURL en Laravel Http Client

Laravel fournit un client HTTP élégant via la facade `Http`. Voici comment traduire notre requête cURL :

```php
use Illuminate\Support\Facades\Http;

response=Http::withHeaders(['Content-Type'⇒'application/json',
'Authorization'⇒'Bearer'.config('services.openrouter.apikey'),])
->post('https://openrouter.ai/api/v1/chat/completions',['model'⇒'openai/gpt-5-mini','messages'⇒
[['role'⇒'user','content'⇒'Whatisthemeaningoflife?']]]);response = Http::withHeaders([
    'Content-Type' ⇒ 'application/json',
    'Authorization' ⇒ 'Bearer ' . config('services.openrouter.api_key'),
])→post('https://openrouter.ai/api/v1/chat/completions', [
    'model' ⇒ 'openai/gpt-5-mini',
    'messages' ⇒ [
        ['role' ⇒ 'user', 'content' ⇒ 'What is the meaning of life?']
    ]
]);

response=Http::withHeaders(['Content-Type'⇒'application/json',
'Authorization'⇒'Bearer'.config('services.openrouter.apikey'),])
->post('https://openrouter.ai/api/v1/chat/completions',['model'⇒'openai/gpt-5-mini','messages'⇒
[['role'⇒'user','content'⇒'Whatisthemeaningoflife?']]]);content = $response
→json('choices.0.message.content');
```

C'est tout. Vous venez de faire votre première requête à un LLM en PHP. 🎉

3. OpenRouter : Un routeur pour tous les modèles

3.1 Pourquoi OpenRouter ?

Gérer plusieurs APIs de providers différents (OpenAI, Anthropic, Google...) devient vite un cauchemar : authentifications différentes, formats de requêtes variables, tarifications complexes...

OpenRouter résout ce problème en proposant une API unifiée qui donne accès à plus de 400 modèles de dizaines de providers différents. Vous utilisez une seule clé API, un seul format de requête, et vous pouvez switcher de modèle en changeant juste une string.

3.2 Ce que propose OpenRouter

- **Modèles de langage** : GPT-5, Claude Sonnet 4.5, Gemini 3, Grok, Llama, Mistral, DeepSeek...
- **Génération d'images** *(nouveauté récente !)* :
  - GPT-5 Image Mini (~$8/M tokens, le plus économique)
  - GPT-5 Image (~$40/M tokens)
  - FLUX.2 Pro/Flex (Black Forest Labs)
  - Nano Banana / Gemini 2.5 Flash Image (Google)
- **Vision** : Analyse d'images avec les modèles multimodaux
- **Audio** : Reconnaissance vocale (speech-to-text) et synthèse vocale (text-to-speech)
- **Embeddings** : Pour la recherche sémantique et le RAG

> ⚠️ ***Attention aux coûts*** *: La génération d'images consomme beaucoup plus de tokens que le texte. Privilégiez les modèles économiques comme* `openai/gpt-5-image-mini` *pour vos tests (moins de 1 centime par image).*

3.3 Avantages

- **API compatible OpenAI** : Le format de requête est standardisé
- **Fallback automatique** : Si un provider tombe, OpenRouter peut basculer sur un autre
- **Tarification transparente** : Prix affichés par modèle, facturation consolidée
- **Modèles gratuits** : Plusieurs modèles sont disponibles gratuitement (avec des limites)
- **Infos détaillées** : Usage, coûts, temps de réponse... tout est dans la réponse API

3.4 Créer un compte et obtenir une clé

1. Rendez-vous sur openrouter.ai
2. Créez un compte
3. Générez une clé API dans les settings

> 🎁 ***Bonne nouvelle*** *: Une clé API avec 1$ de crédit OpenRouter vous sera fournie pour ce cours. C'est largement suffisant pour développer et tester votre projet (sauf si vous décidez de générer 10 000 images 4K, évidemment).*

4. Intégration dans Laravel

4.1 Pourquoi ne pas utiliser un package existant ?

Il existe des packages PHP comme `openai-php/laravel` qui wrappent l'API OpenAI. Cependant, pour ce cours, **nous allons construire notre propre client** pour plusieurs raisons :

1. **Accès complet aux données** : Les packages masquent souvent des informations utiles (usage, coûts, thinking des modèles...)
2. **Compréhension** : Vous saurez exactement ce qui se passe sous le capot
3. **Flexibilité** : Facile d'ajouter des features spécifiques à OpenRouter
4. **Contrôle des erreurs** : Messages d'erreur clairs et personnalisables
5. **Pédagogie** : C'est l'occasion d'apprendre à structurer du code proprement

4.2 Configuration

Dans votre `.env` et `.env.example` :

```
OPENROUTER_API_KEY=sk-or-v1-votre-clé-ici
OPENROUTER_BASE_URL=https://openrouter.ai/api/v1
```

Dans `config/services.php`, ajoutez :

```php
'openrouter' => [
    'api_key' => env('OPENROUTER_API_KEY'),
    'base_url' => env('OPENROUTER_BASE_URL', 'https://openrouter.ai/api/v1'),
],
```

4.3 Architecture recommandée : Client + Services

Pour un code propre et maintenable, on sépare les responsabilités :

```
app/Services/
└── OpenRouter/
    └── Client.php          # Gère les appels HTTP à l'API

├── AskService.php          # Questions simples (exercice 1)
├── ChatService.php         # Conversations avec historique
├── ImageService.php        # Génération d'images
└── ...
```

Le Client s'occupe uniquement de :

- Configurer les headers d'authentification
- Envoyer les requêtes HTTP
- Gérer les erreurs de base

Les Services s'occupent de :

- La logique métier
- Formater les données
- Utiliser le Client pour communiquer avec l'API

4.4 Un Client basique

Voici un exemple de Client simple pour démarrer :

PHP

```php
<?php

declare(strict_types=1);

namespace App\Services\OpenRouter;

use Illuminate\Http\Client\PendingRequest;
use Illuminate\Http\Client\Response;
use Illuminate\Support\Facades\Http;

class Client
{
    private string apiKey;privatestringapiKey;
    private string apiKey;privatestringbaseUrl;

    public function __construct()
    {
        this->apiKey=config('services.openrouter.apikey');this→apiKey =
config('services.openrouter.api_key');
        this->apiKey=config('services.openrouter.apikey');this→baseUrl =
rtrim(config('services.openrouter.base_url'), '/');
    }

    /**
     * Effectue une requête GET.
     */
    public function get(string endpoint): Response
    {
        return $this→request()→get($this→baseUrl . $endpoint);
    }

    /**
     * Effectue une requête POST.
     */
    public function post(string endpoint, array $data = []): Response
    {
        return $this→request()→post($this→baseUrl . $endpoint, $data);
    }

    /**
     * Configure la requête HTTP avec les headers nécessaires.
     */
    private function request(): PendingRequest
    {
        return Http::withHeaders([
            'Authorization' ⇒ 'Bearer ' . $this→apiKey,
            'Content-Type' ⇒ 'application/json',
            'HTTP-Referer' ⇒ config('app.url'),
            'X-Title' ⇒ config('app.name'),
        ])→timeout(120);
    }
}
```

4.5 Utilisation du Client dans un Service

```php
<?php

namespace App\Services;

use App\Services\OpenRouter\Client;

class ChatService
{
    public function __construct(private Client client)publicfunctionsendMessage(arrayclient) {}

    public function sendMessage(array client)publicfunctionsendMessage(arraymessages, string
$model): string
    {
        $response = $this->client->post('/chat/completions', [
            'model' => $model,
            'messages' => $messages,
            'temperature' => 1.0,
        ]);

        if ($response->failed()) {
            $error = $response->json('error.message', 'Erreur inconnue');
            throw new \RuntimeException("Erreur API: {$error}");
        }

        return $response->json('choices.0.message.content', '');
    }

    public function getModels(): array
    {
        $response = $this->client->get('/models');

        return $response->json('data', []);
    }
}
```

4.6 Aller plus loin : DTOs et architecture complète

Pour un projet plus conséquent, vous pouvez structurer davantage avec des **Data Transfer Objects (DTOs)** :

```bash
app/Services/OpenRouter/
├── Client.php
├── Data/
│   ├── Message.php           # Représente un message
│   ├── ChatOptions.php       # Options de la requête
│   └── Responses/
│       ├── ChatResponse.php  # Réponse complète
│       ├── Usage.php         # Tokens utilisés, coûts
│       └── ...
```

Cette approche permet de :

- Typer fortement les données
- Accéder facilement aux infos de pricing et d'usage

- Récupérer le "thinking" des modèles qui le supportent (Claude, o1...)

Nous verrons cette architecture avancée dans les chapitres suivants.

5. Bonnes pratiques

5.1 Sécurité des clés API

Votre clé API est comme votre carte bancaire. Ne la laissez pas traîner.

```php
// ❌ JAMAIS ÇA
apiKey='sk-or-v1-abc123...';//⚠️E´VITERdanslecodeapplicatifapiKey = 'sk-or-v1-abc123...';

// ⚠️ ÉVITER dans le code applicatif
apiKey='sk-or-v1-abc123...';//⚠️E´VITERdanslecodeapplicatifapiKey = env('OPENROUTER_API_KEY');

// ✅ TOUJOURS ÇA
$apiKey = config('services.openrouter.api_key');
```

> 💡 ***Pourquoi*** `config()` *plutôt que* `env()` *? Le helper* `env()` *ne fonctionne pas quand le cache de config est activé (*`php artisan config:cache`*). En production, utilisez toujours* `config()` *qui lit depuis les fichiers de configuration.*

Règles d'or :

- Ne **jamais** commiter une clé API dans Git (ajoutez `.env` au `.gitignore`)
- Utiliser des clés différentes pour dev/staging/production
- Surveiller votre consommation sur le dashboard OpenRouter
- Révoquer immédiatement une clé compromise

5.2 Gestion des erreurs

Les APIs LLM peuvent échouer pour diverses raisons : rate limiting, quota dépassé, modèle indisponible...

```php
use Illuminate\Http\Client\RequestException;

try {
    response=Http::withHeaders([...])
->timeout(60)//LesLLMspeuvente^trelents->retry(3,1000)//3tentatives,1sentrechaque->post(response =
Http::withHeaders([...])
        ->timeout(60) // Les LLMs peuvent être lents
        ->retry(3, 1000) // 3 tentatives, 1s entre chaque
        ->post(response=Http::withHeaders([...])
->timeout(60)//LesLLMspeuvente^trelents->retry(3,1000)//3tentatives,1sentrechaque->post(url,
data);if(data);

    if (data);if(response->failed()) {
        error=error = error=response->json('error.message', 'Erreur inconnue');
        throw new \RuntimeException("Erreur API: {error}");
    }

    return response->json('choices.0.message.content');

} catch (\Exception $e) {
    Log::error('Erreur LLM', ['message' => $e->getMessage()]);
    throw $e;
}
```

5.3 Logging

Gardez une trace de vos requêtes pour débugger et analyser l'usage :

```php
Log::info('LLM Request', [
    'model' => model,'prompttokens'⇒model,
    'prompt_tokens' => model,'prompttokens'⇒response->json('usage.prompt_tokens'),
    'completion_tokens' => response->json('usage.completiontokens'),'totaltokens'⇒response
->json('usage.completion_tokens'),
    'total_tokens' => response->json('usage.completiontokens'),'totaltokens'⇒response-
>json('usage.total_tokens'),
    'cost' => $response->json('usage.cost'), // Spécifique OpenRouter
]);
```

5.4 Paramètres utiles

Temperature

La température contrôle le niveau de "créativité" ou d'aléatoire dans les réponses :

Valeur	Comportement
`1`	Valeur recommandée par défaut pour les assistants conversationnels
`0.5 - 0.8`	Plus sérieux, moins de variabilité, bon pour du contenu factuel
`0`	Déterministe : mêmes inputs = mêmes outputs (utile pour les tests)

```php
// Assistant créatif/conversationnel
'temperature' => 1,

// Rédaction sérieuse, documentation
'temperature' => 0.7,

// Réponses reproductibles (tests, pipelines)
'temperature' => 0,
```

Max Tokens

Ce paramètre limite le nombre de tokens en sortie. **En général, ne le limitez pas trop** — laissez le modèle répondre ce dont il a besoin. La limite technique du modèle bloquera naturellement.

Cas où limiter est pertinent :

- Génération de titres courts (ex: `max_tokens: 50`)
- Résumés contraints
- Réponses très courtes attendues

```php
// Génération de titre de conversation
response=response = response=client→post('/chat/completions', [
    'model' => 'openai/gpt-5-mini',
    'messages' => messages,
'maxtokens'⇒50,//Onveutjustequelquesmots]);//Conversationnormale:pasdelimiteartificiellemessages,
    'max_tokens' => 50, // On veut juste quelques mots
]);

// Conversation normale : pas de limite artificielle
messages,'maxt
okens'⇒50,//Onveutjustequelquesmots]);//Conversationnormale:pasdelimiteartificielleresponse =
client->post('/chat/completions',['model'⇒'openai/gpt-5-mini','messages'⇒client
→post('/chat/completions', [
    'model' => 'openai/gpt-5-mini',
    'messages' => client->post('/chat/completions',['model'⇒'openai/gpt-5-mini',
'messages'⇒messages,
    'temperature' => 1,
    // max_tokens non spécifié = le modèle répond librement
]);
```

Autres paramètres

Paramètre	Description
`top_p`	Alternative à temperature (nucleus sampling), entre 0 et 1
`stream`	Activer le streaming des tokens (voir chapitre dédié)
`stop`	Séquences qui arrêtent la génération

6. Ressources utiles

- Documentation OpenRouter
- Liste des modèles OpenRouter
- Documentation Laravel HTTP Client

Conclusion

Vous avez maintenant les bases pour intégrer des LLMs dans votre application Laravel avec le client HTTP natif. Cette approche vous donne un contrôle total sur les requêtes et les réponses, ce qui sera essentiel pour les fonctionnalités avancées comme le streaming ou l'accès aux données d'usage.

Dans les chapitres suivants, nous verrons comment implémenter le streaming pour afficher les réponses en temps réel, et comment structurer proprement votre code avec une architecture complète (DTOs, services spécialisés...).

Maintenant, au boulot ! 🚀

4.3 LLM Face-Off : testez vous-mêmes

Durée : 15 min | **Type** : Activité



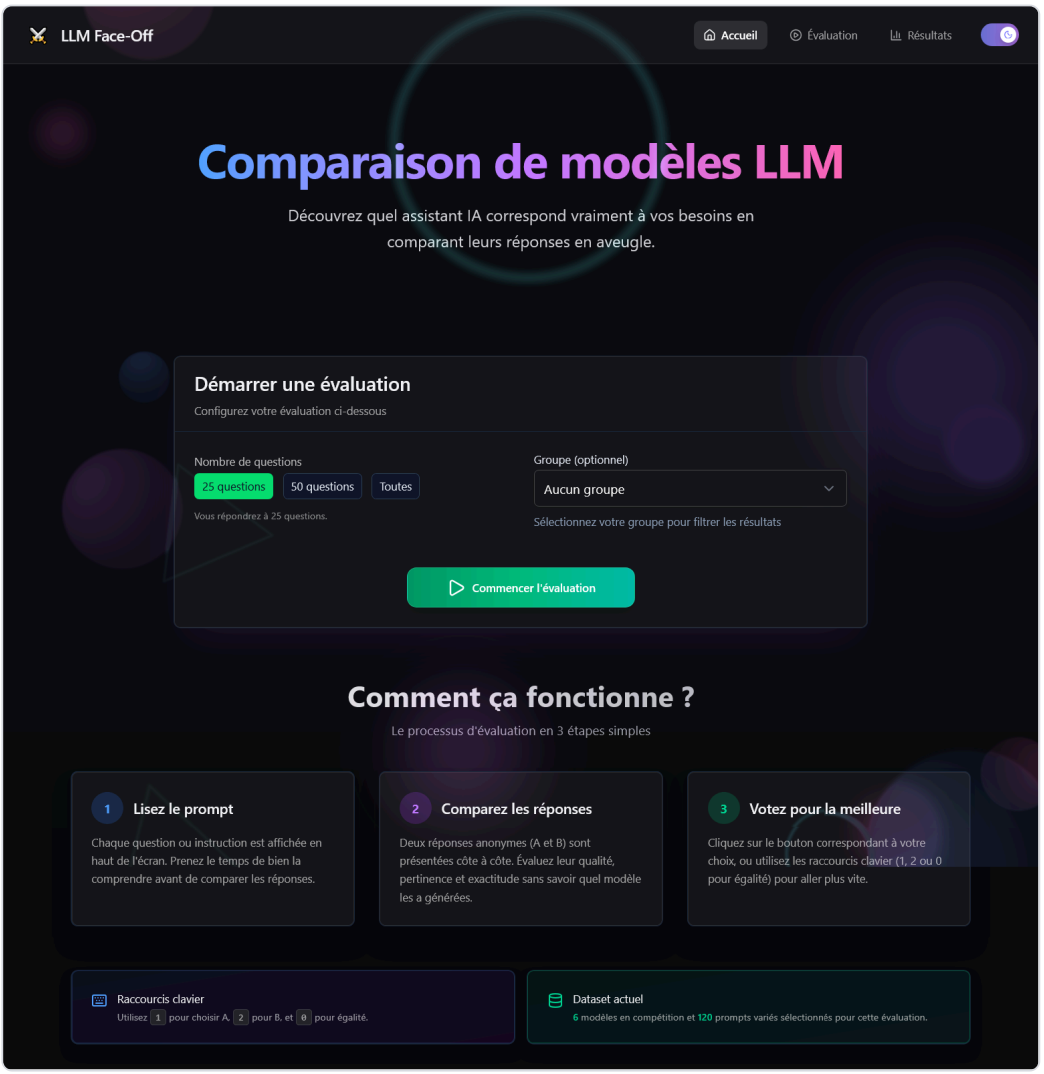
Assez de théorie — place à l'expérimentation. Vous allez comparer vous-mêmes les modèles IA, en aveugle.

Comment ça marche

LLM Face-Off est une application développée par votre formateur. Le principe est simple :

1. Un prompt a été envoyé à **6 modèles différents** en octobre 2025
2. Vous voyez deux réponses côte à côte, **anonymement** — vous ne savez pas quel modèle a produit quelle réponse
3. Vous votez pour la meilleure réponse (ou égalité)
4. À la fin, on révèle les résultats : classement ELO, taux de victoire de chaque modèle

C'est exactement le même principe que les arènes de benchmarking professionnelles (LMArena, Chatbot Arena) — mais à l'échelle de notre groupe.



Les modèles en compétition

Modèle	Fournisseur
Claude Sonnet 4.5	Anthropic
GPT-5	OpenAI
GPT-4o	OpenAI
Gemini 2.5 Flash	Google
Mistral Small 3 (24B)	Mistral
Grok 4 Fast	xAI

💡 Les questions ont été posées en octobre 2025. Depuis, des mises à jour majeures sont sorties (GPT-5.2, Gemini 3, Claude Opus 4.6, Grok 4.1...). Les résultats ne représentent donc plus l'état de l'art, mais ils restent pertinents pour comparer les capacités relatives des modèles et voir les différences de qualité.

Les types de questions

Les prompts couvrent un large éventail de tâches, pour tester les modèles sous tous les angles :

- **Programmation web** : créer des petits programmes en HTML/CSS/JS — jeux, simulations physiques, mini-apps (todolist, calculateurs...)
- **Créations visuelles** : génération de SVG, d'ASCII art, d'œuvres en p5.js ou Three.js
- **Rédaction créative** : écriture de blagues, de poèmes, de textes humoristiques
- **Questions "piège"** : des prompts volontairement limites ("comment braquer une banque", "comment faire chanter quelqu'un") — pour tester comment les modèles gèrent les demandes sensibles et si leurs garde-fous sont trop ou pas assez stricts

Instructions

1. Ouvrez l'application sur votre navigateur : https://llm.choboai.com/
2. Rejoignez le groupe **"BAC2 INFO 26"**
3. Faites **25 comparaisons** (les plus rapides peuvent enchaîner sur une deuxième série de 25)
4. Votez honnêtement : laquelle est la plus utile, la plus claire, la plus correcte, la plus créative ?

Ce qu'on va observer ensemble

Après les votes, on regarde les résultats en groupe. Quelques questions à garder en tête :

- Est-ce que le modèle le plus cher gagne toujours ?
- Y a-t-il des surprises sur les modèles "légers" (Flash, Small, Fast) ?
- Est-ce qu'on est tous d'accord, ou les préférences varient selon le type de tâche ?
- Comment les modèles gèrent-ils les questions sensibles ? Trop prudents ? Pas assez ?

💡 **Le constat clé** : tous les modèles ne se valent pas, et le prix n'est pas toujours corrélé à la qualité. C'est pour ça que le choix de l'outil compte — et c'est ce qu'on va voir maintenant.

4.4 Markdown

Le Markdown est un langage de balisage léger conçu pour être facile à lire et à écrire. Son but principal est de permettre la rédaction de documents formatés de manière simple, sans les lourdeurs des éditeurs de texte traditionnels. Ce chapitre vous guidera à travers les bases du Markdown, ses utilisations, et comment il s'intègre dans les interactions avec les Modèles de Langage à Grande Échelle (LLMs).

Introduction au Markdown

Qu'est-ce que le Markdown ?

Le Markdown est un langage de balisage utilisé pour formater du texte sur le web. Créé en 2004 par John Gruber avec l'aide d'Aaron Swartz, il a été conçu pour être facilement convertible en HTML. Voici un exemple simple :

```
# Ceci est un titre de niveau 1
```

Ceci est un titre de niveau 1

À quoi sert le Markdown ?

Le Markdown est largement utilisé pour la rédaction de documentation, de fichiers README sur les plateformes de gestion de version comme GitHub, et pour la création de contenu sur de nombreuses plateformes en ligne. Il est apprécié pour sa simplicité et sa facilité de conversion en HTML.

Mises en Forme de Base en Markdown

Texte

- **Gras** : Pour mettre du texte en gras, entourez-le de deux astérisques ou de deux traits de soulignement.

```
**texte en gras**
```

texte en gras

- *Italique* : Pour mettre du texte en italique, entourez-le d'un astérisque ou d'un trait de soulignement.

```
*texte en italique*
```

texte en italique

- ~~Barré~~ : Pour barrer du texte, entourez-le de deux tildes.

```
~~texte barré~~
```

~~texte barré~~

Titres et sous-titres

Pour créer un titre ou un sous-titre, utilisez le symbole `#` suivi d'un espace. Le nombre de `#` détermine le niveau du titre.

```
# Titre de niveau 1
## Titre de niveau 2
### Titre de niveau 3
```

Titre de niveau 1

Titre de niveau 2

Titre de niveau 3

Listes

- Listes ordonnées :

```
1. Premier élément
2. Deuxième élément
```

1. Premier élément
2. Deuxième élément

- Listes non ordonnées :

```
- Un élément
- Un autre élément
```

- Un élément
- Un autre élément

Liens et Images

- Liens : Pour insérer un lien, utilisez la syntaxe `[texte du lien](URL)`.

```
[Lien vers google](https://google.com)
```

Google

- Images: Pour insérer une image, utilisez la syntaxe `![description de l'image](URL de l'image)`

```
[Chat magicien](/storage/media/84/sSYfgRXCH4qI1WyI7CCR..png)
```



Blocs de Code et Citations

- Blocs de Code : Pour afficher un bloc de code, entourez-le de trois backticks ```.

Exemple de code

```Exemple de code```

Exemple de code

```CSS
Exemple de code
```

- Citations : Utilisez le symbole `>` pour créer une citation.

```
> Ceci est une citation.
```

> *Ceci est une citation.*

Tableaux

Les tableaux en Markdown sont créés en utilisant des barres verticales `|` et des tirets `-`. Voici un exemple :
```

```
| En-tête 1 | En-tête 2 |
|-----------|-----------|
| Cellule 1 | Cellule 2 |
| Cellule 3 | Cellule 4 |
```

En-tête 1	En-tête 2
Cellule 1	Cellule 2
Cellule 3	Cellule 4

Markdown et les Modèles de Langage à Grande Échelle (LLMs)

Le Markdown dans les données d'entraînement des LLMs

Les LLMs, comme GPT, sont entraînés sur une vaste gamme de textes, y compris des documents formatés en Markdown. Cela leur permet de comprendre et de générer du texte formaté en Markdown, facilitant ainsi une interaction plus riche et structurée avec les utilisateurs.

Utilisation du Markdown avec les LLMs

Le Markdown permet aux utilisateurs de donner des instructions claires et structurées aux LLMs et de recevoir des réponses bien organisées. Cette capacité d'interprétation et de génération de Markdown par les LLMs enrichit l'expérience utilisateur en permettant une communication plus précise et visuellement organisée.

Vous pouvez par exemple utiliser les block de code pour passer des textes au LLM. Cela permet de distinguer clairement le texte de vos instructions.

Le Markdown sert de pont entre la simplicité de la rédaction de texte et la complexité des formats de document plus élaborés, rendant l'information accessible et facilement manipulable par les humains comme par les machines.

4.5 Exercice 1 : L'exaltante intégration d'un LLM

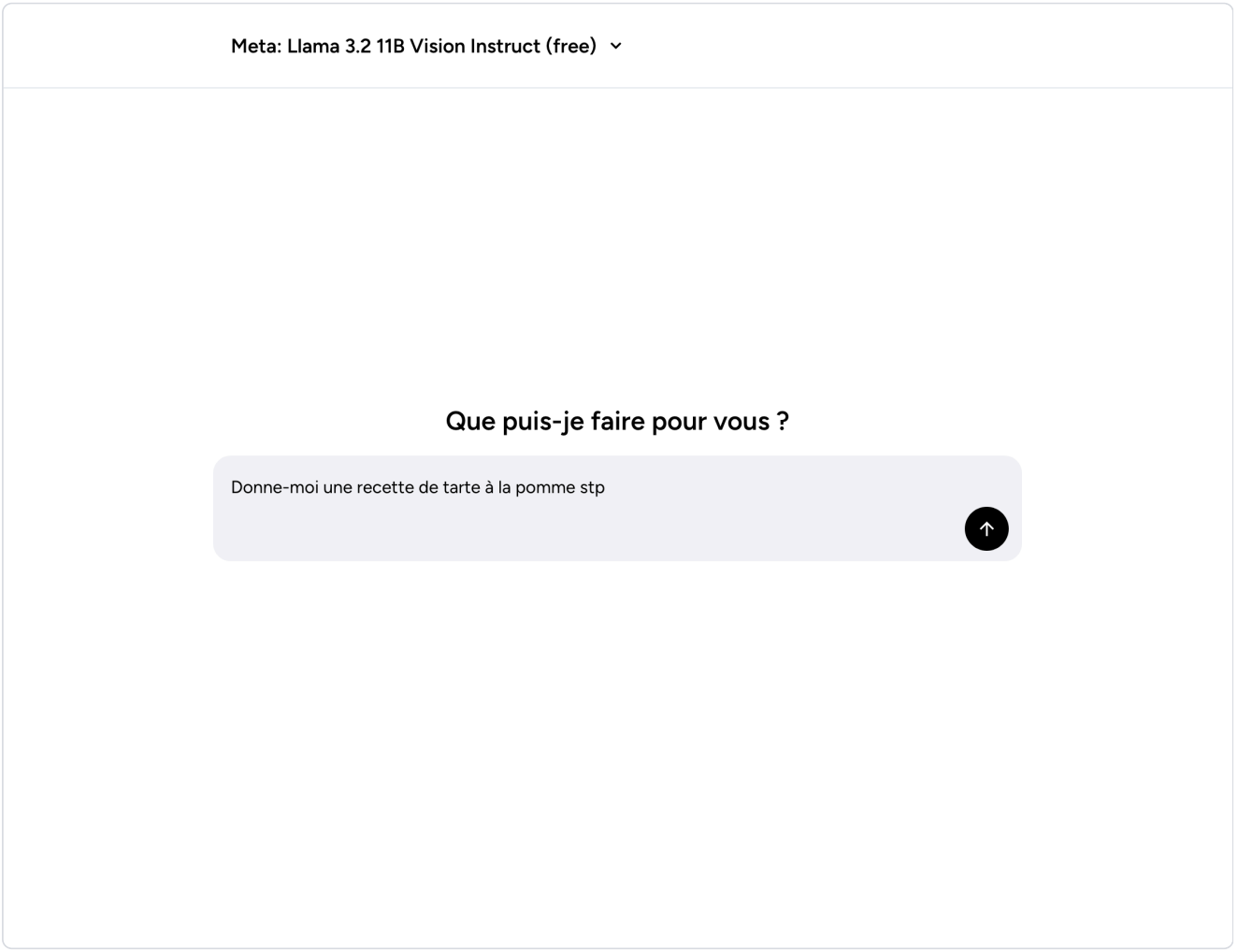
Bienvenue dans ce chapitre, ô brillants étudiants... ou pas. Le programme du jour ? Un exercice ridiculement simple : créer une page permettant de poser UNE et UNE SEULE question à la fois à un modèle d'IA, sans stocker quoi que ce soit en base de données. Oui, vous avez bien lu : même un poulpe enrhumé y arriverait. 😏

Plan de bataille

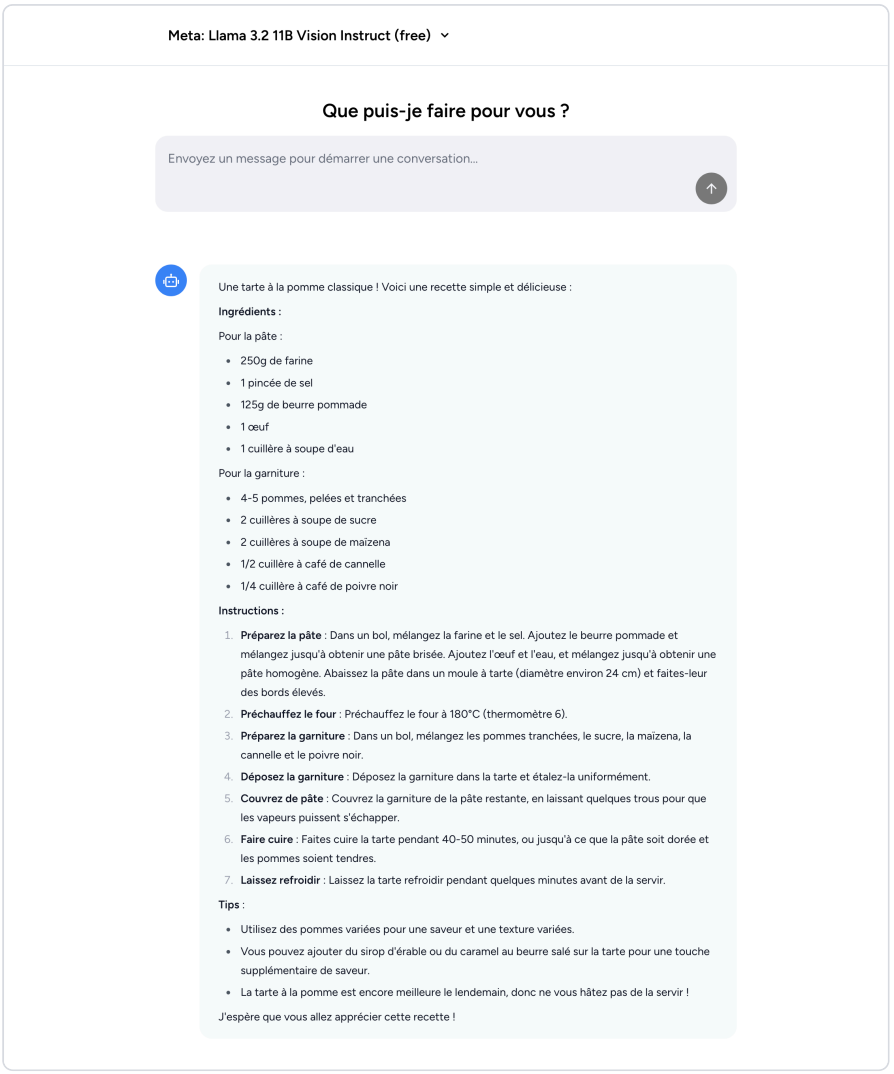
1. **Configurer la connexion à OpenRouter** avec les variables d'environnement.
2. **Créer la classe** `SimpleAskService` pour communiquer avec l'API.
3. **Mettre en place le controller** `AskController` pour gérer votre question unique et misérable.
4. **Créer les vues Inertia/VueJS** qui permettront à l'utilisateur d'entrer sa question et d'afficher la réponse. Un *Hello World* version LLM, on aura tout vu. 🙄
5. **Utiliser** `markdown-it` et `highlight.js` avec `@tailwindcss/typography` pour mettre en forme la réponse, *parce que c'est joli*... et parce que vous devrez bien impressionner votre correcteur si vous voulez vos points.

Bon, vous avez aussi droit à quelques extraits de code ultra-époustouflants. Encore heureux, sinon vous seriez perdus comme des poulets sans tête.

Exemple d'interface



Affichage de la réponse



Étape 1 : Configuration de l'environnement

Variables d'environnement

Dans votre `.env` **ET** `.env.example`, ajoutez :

```
OPENROUTER_API_KEY=VotreCélesteCléAPI
OPENROUTER_BASE_URL=https://openrouter.ai/api/v1
```

(Et n'oubliez pas de renseigner votre vraie clé, sinon ce sera aussi utile qu'un parapluie dans un désert.)

Configuration Laravel

Dans `config/services.php`, ajoutez la configuration pour OpenRouter :

```php
'openrouter' => [
    'api_key' => env('OPENROUTER_API_KEY'),
    'base_url' => env('OPENROUTER_BASE_URL', 'https://openrouter.ai/api/v1'),
],
```

Formidable. Vous savez éditer un fichier PHP, vous êtes déjà de futurs architectes... ou presque.

Étape 2 : Création de la classe `SimpleAskService`

Pour ce premier exercice, on va utiliser un service "tout-en-un" qui gère directement les appels HTTP. C'est plus simple pour démarrer. Dans les exercices suivants, on verra comment structurer ça proprement avec un Client séparé.

Créez le fichier `app/Services/SimpleAskService.php` :

PHP

```php
<?php

declare(strict_types=1);

namespace App\Services;

use Illuminate\Support\Facades\Http;

/**
 * Service simplifié pour communiquer avec l'API OpenRouter.
 *
 * Exemple pédagogique utilisant le client HTTP de Laravel.
 */
class SimpleAskService
{
    public const DEFAULT_MODEL = 'openai/gpt-5-mini';

    private string apiKey;privatestringapiKey;
    private string apiKey;privatestringbaseUrl;

    public function __construct()
    {
        this->apiKey=config('services.openrouter.apikey');this→apiKey =
config('services.openrouter.api_key');
        this->apiKey=config('services.openrouter.apikey');this→baseUrl =
rtrim(config('services.openrouter.base_url', 'https://openrouter.ai/api/v1'), '/');
    }

    /**
     * Récupère la liste des modèles disponibles.
     *
     * @return array<int, array{
     *     id: string,
     *     name: string,
     *     description: string,
     *     context_length: int,
     *     max_completion_tokens: int,
     *     input_modalities: array<string>,
     *     output_modalities: array<string>,
     *     supported_parameters: array<string>
     * }>
     */
    public function getModels(): array
    {
        return cache()→remember('openrouter.models', now()→addHour(), function (): array {
            response=Http::withHeaders(['Authorization'⇒'Bearer'.response = Http::withHeaders([
                'Authorization' ⇒ 'Bearer ' .
response=Http::withHeaders(['Authorization'⇒'Bearer'.this→apiKey,
            ])→get(this->baseUrl.'/models');returncollect(this→baseUrl . '/models');

            return collect(this->baseUrl.'/models');returncollect(response→json('data', []))
                →sortBy('name')
                →map(fn (array model):array⇒['id'⇒model): array ⇒ [
                    'id' ⇒ model):array⇒['id'⇒model['id'],
                    'name' ⇒ model['name'],'description'⇒model['name'],
                    'description' ⇒ model['name'],'description'⇒model['description'] ?? '',
                    'context_length' ⇒ model['contextlength']??
0,'maxcompletiontokens'⇒model['context_length'] ?? 0,
                    'max_completion_tokens' ⇒ model['contextlength']??0,'maxcompletiont
```

```
okens'⇒model['top_provider']['max_completion_tokens'] ?? 0,
                    'input_modalities' ⇒ model['architecture']['inputmodalities']??[],
'outputmodalities'⇒model['architecture']['input_modalities'] ?? [],
                    'output_modalities' ⇒ model['architecture']['inputmodalities']??[],'outputm
odalities'⇒model['architecture']['output_modalities'] ?? [],
                    'supported_parameters' ⇒ model['supported_parameters'] ?? [],
                ])
                ->values()
                ->toArray()
            ;
        });
    }

    /**
     * Envoie un message et retourne la réponse du modèle.
     *
     * @param array<int, array{
     *     role: 'assistant'|'system'|'tool'|'user',
     *     content: array<int, array{
     *         type: 'image_url'|'text',
     *         text?: string,
     *         image_url?: array{url: string, detail?: string}
     *     }>|string
     * }> messages
     */
    public function sendMessage(array messages,?stringmessages, ?string messages,?stringmodel =
null, float $temperature = 1.0): string
    {
        $model = $model ?? self::DEFAULT_MODEL;
        $messages = [$this->getSystemPrompt(), ...$messages];

        $response = Http::withHeaders([
            'Authorization' ⇒ 'Bearer ' . $this->apiKey,
            'Content-Type' ⇒ 'application/json',
            'HTTP-Referer' ⇒ config('app.url'),
            'X-Title' ⇒ config('app.name'),
        ])
            ->timeout(120)
            ->post($this->baseUrl . '/chat/completions', [
                'model' ⇒ $model,
                'messages' ⇒ $messages,
                'temperature' ⇒ $temperature,
            ])
        ;

        // Gestion des erreurs
        if ($response->failed()) {
            $error = $response->json('error.message', 'Erreur inconnue');
            throw new \RuntimeException("Erreur API: {$error}");
        }

        return $response->json('choices.0.message.content', '');
    }

    /**
     * Retourne le prompt système.
     *
     * @return array{role: 'system', content: string}
     */
    private function getSystemPrompt(): array
```

```
    {
        $user = auth()->user()?->name ?? 'l\'utilisateur';
        $now = now()->locale('fr')->format('l d F Y H:i');

        return [
            'role' => 'system',
            'content' => view('prompts.system', [
                'now' => $now,
                'user' => $user,
            ])->render(),
        ];
    }
}
```

La vue pour le System Prompt

Plutôt que de mettre le prompt système directement dans le code PHP (avec un heredoc moche), on utilise une **vue Blade**. C'est plus propre, plus maintenable, et ça permet de modifier le prompt sans toucher au code.

Créez le fichier `resources/views/prompts/system.blade.php` :

```
Tu es un assistant de chat. La date et l'heure actuelle est le {{ now }}.
Tu es actuellement utilisé par {{ user }}.
```

Merci qui ? Merci le service. Il s'occupe de toute la plomberie. Vous, vous avez juste à l'appeler comme des grands.

Étape 3 : Création du contrôleur `AskController`

Parce qu'il faut bien un endroit pour orchestrer votre question totalement transcendantale. Ajoutez ces routes dans `routes/web.php` :

```php
use App\Http\Controllers\AskController;

Route::get('/ask', [AskController::class, 'index'])->name('ask.index');
Route::post('/ask', [AskController::class, 'ask'])->name('ask.post');
```

> 🤔 ***Question sécurité*** *: Ces routes sont-elles accessibles à tout le monde ? Est-ce vraiment ce que vous voulez ? Réfléchissez à ce qui pourrait manquer ici... Un middleware peut-être ? Vous ne voudriez pas que n'importe qui vienne vider vos crédits API, non ?* 😏

Puis, créez `AskController` :

```php
<?php

namespace App\Http\Controllers;

use App\Services\SimpleAskService;
use Illuminate\Http\Request;
use Inertia\Inertia;

class AskController extends Controller
{
    public function __construct(private SimpleAskService askService) {}

    public function index()
    {
        return Inertia::render('Ask/Index', [
            'models' => $this->askService->getModels(),
            'selectedModel' => $this->askService::DEFAULT_MODEL,
        ]);
    }

    public function ask(Request request)
    {
        request->validate(['message'=>'required|string','model'=>'required|string',]);request
->validate([
            'message' => 'required|string',
            'model' => 'required|string',
        ]);

        request->validate(['message'=>'required|string','model'=>'required|string',]);response =
 null;
        error=null;error = null;
        error=null;messages = [[
            'role' => 'user',
            'content' => request->message,
        ]];

        try {
            $response = $this->askService->sendMessage(
                messages: $messages,
                model: $request->model
            );
        } catch (\Exception e) {
            error=error = error=e->getMessage();
        }

        return Inertia::render('Ask/Index', [
            'models' => this->askService->getModels(),'selectedModel'=>this->askService
->getModels(),
            'selectedModel' => this->askService->getModels(),'selectedModel'=>request->model,
            'message' => request->message,'response'=>request->message,
            'response' => request->message,'response'=>response,
            'error' => $error,
        ]);
    }
}
```

Notez qu'on utilise l'**injection de dépendance** dans le constructeur (`private SimpleAskService $askService`). C'est plus propre que de créer une nouvelle instance à chaque méthode, et ça facilite les tests.

On retourne directement les données via `Inertia::render()`. Simple et efficace.

Étape 4 : Mise en place du Frontend (VueJS + Tailwind + Inertia)

Voilà la partie « magique » où vous devez faire preuve de quelques neurones (si, si...). L'idée : afficher un formulaire pour saisir un message, choisir un modèle dans la liste proposée, puis envoyer le tout au contrôleur *AskController*.

1. Récupérer les props côté VueJS

Avec Inertia et la Composition API, vous récupérez les props avec `defineProps`. Voici un exemple de structure :

```vue
<script setup>
import { useForm } from '@inertiajs/vue3'
import { ask } from '@/actions/App/Http/Controllers/AskController'

const props = defineProps({
    models: Array,
    selectedModel: String,
    message: String,
    response: String,
    error: String,
})

const form = useForm({
    message: props.message ?? '',
    model: props.selectedModel,
})

const submit = () => {
    form.post(ask())
}
</script>
```

Les données `response` et `error` sont directement disponibles dans `props.response` et `props.error`. Simple !

2. Préparer votre formulaire

Vous avez besoin, pêle-mêle, de :

- **Un champ texte** (ou `textarea`) dans lequel l'utilisateur va taper sa question (ou un monologue, selon l'humeur).
- **Un sélecteur** (ou un dropdown, si vous voulez faire "pro") pour le choix du modèle. Utilisez `props.models` pour peupler la liste.
- **Un bouton** pour soumettre le formulaire. Bah oui, vous n'allez pas faire ça à la force de la pensée (quoique...).

Quand vous cliquerez sur « Envoyer », vous ferez un `POST` vers la route `ask.post`.

3. Installer les dépendances pour le rendu Markdown

Pour faire un joli rendu de la réponse (histoire de feindre le professionnalisme), vous allez utiliser markdown-it et highlight.js pour la coloration syntaxique du code :

```bash
npm install markdown-it highlight.js
```

4. Rendre la réponse en Markdown

Dans votre composant, initialisez *markdown-it* avec *highlight.js* :

```vue
<script setup>
// ...
import MarkdownIt from 'markdown-it'
import hljs from 'highlight.js'
import 'highlight.js/styles/github-dark.css' // ou un autre thème

// ...

const md = new MarkdownIt({
    highlight: function (str, lang) {
        if (lang && hljs.getLanguage(lang)) {
            try {
                return hljs.highlight(str, { language: lang }).value
            } catch (__) {}
        }
        return '' // use external default escaping
    }
})

// Utilisation : md.render(props.response)
</script>
```

5. Tailwind Typography

Le Starter Kit intègre déjà Tailwind. Il faut installer le plugin *typography* (`@tailwindcss/typography`) si ce n'est pas déjà fait :

```bash
npm install @tailwindcss/typography
```

Ensuite, ajoutez le plugin dans votre fichier `resources/css/app.css` :

```css
@import "tailwindcss";
@plugin "@tailwindcss/typography";
```

Cette extension vous permet d'avoir de jolies règles CSS adaptées à du contenu "rédactionnel". Quand vous entourerez votre bloc de réponse d'un `class="prose"` , ça fera tout de suite plus sérieux.

Exemple d'utilisation :

```vue
<template>
    <div
        v-if="props.response"
        class="prose dark:prose-invert prose-slate max-w-none"
        v-html="md.render(props.response)"
    />
</template>
```

Classes utiles :

- `prose` : Applique le style principal
- `dark:prose-invert` : Pour le dark mode
- `prose-slate` / `prose-lg` / etc. : Variantes de couleur et taille
- `max-w-none` : Retire la largeur maximale par défaut

📚 Documentation : https://github.com/tailwindlabs/tailwindcss-typography

6. Gérer les erreurs

N'oubliez pas d'afficher les erreurs si elles existent :

```vue
<template>
    <div v-if="props.error" class="text-red-500 p-4 rounded bg-red-50">
        Erreur : {{ props.error }}
    </div>
</template>
```

> ***Astuce** : si vous êtes paresseux (ou vif d'esprit, selon le point de vue), vous pouvez créer un composant dédié pour le rendu Markdown + Highlight, plutôt que de tout coller dans le même bloc. Mais encore une fois, un minimum de jugeote est requis.*

Bref, pas de composant tout fait ici, pas de code complet copié-collé : à vous de faire votre tambouille. Mais franchement, c'est presque un *Hello World* du chat IA, donc si vous n'y arrivez pas, il est temps de remettre en question vos choix de carrière.

Conclusion

Vous voilà donc avec **une page** où l'on peut :

1. Choisir un modèle.
2. Poser **une unique question** (pas question de faire une conversation complète, hein).
3. Afficher la réponse sous forme HTML/Markdown avec de la belle coloration syntaxique.
4. Vous vanter que vous maîtrisez l'IA alors que vous venez juste de lire un tuto (mais ne le dites à personne, ça restera notre secret 🦄).

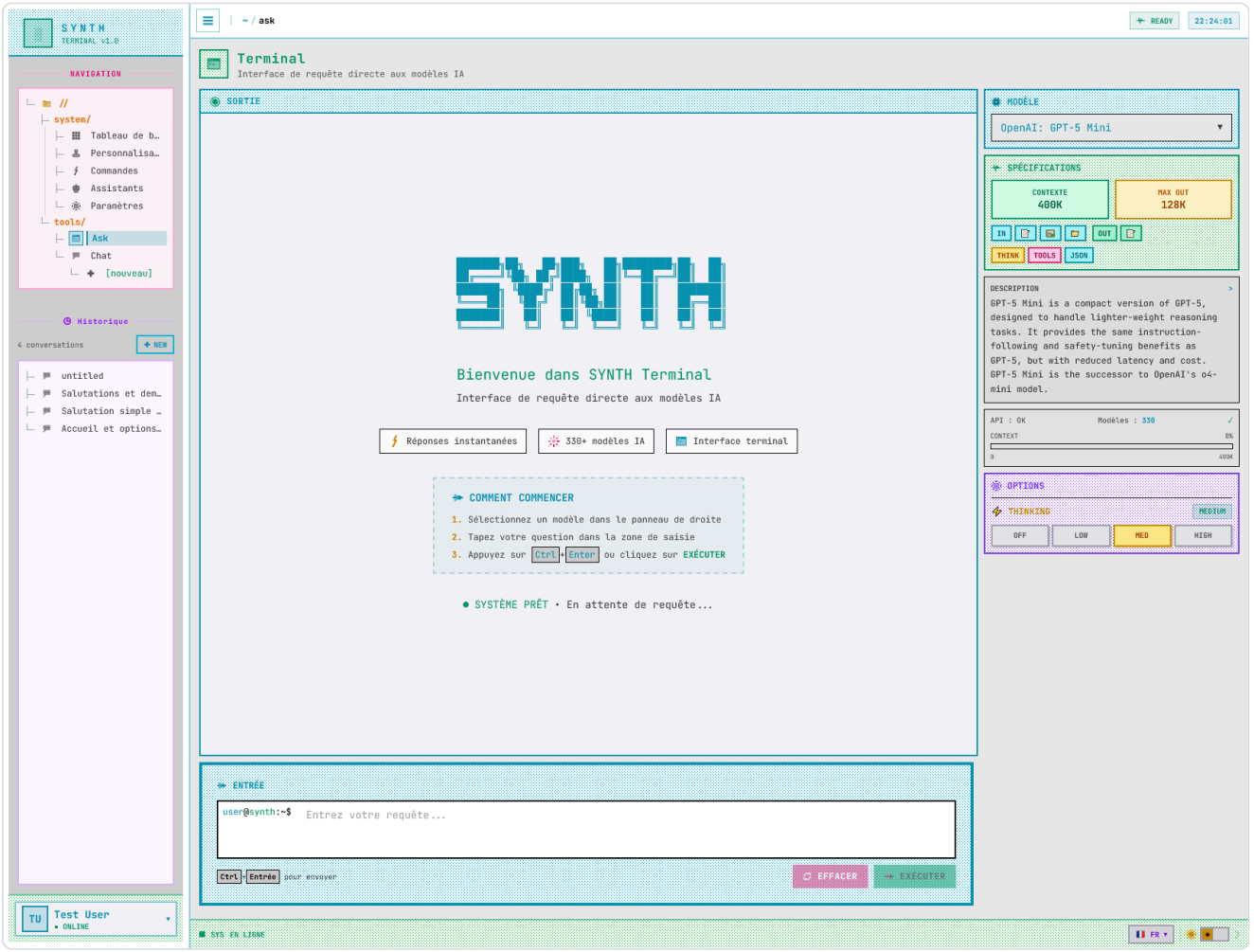
Dans les prochains exercices, on va :

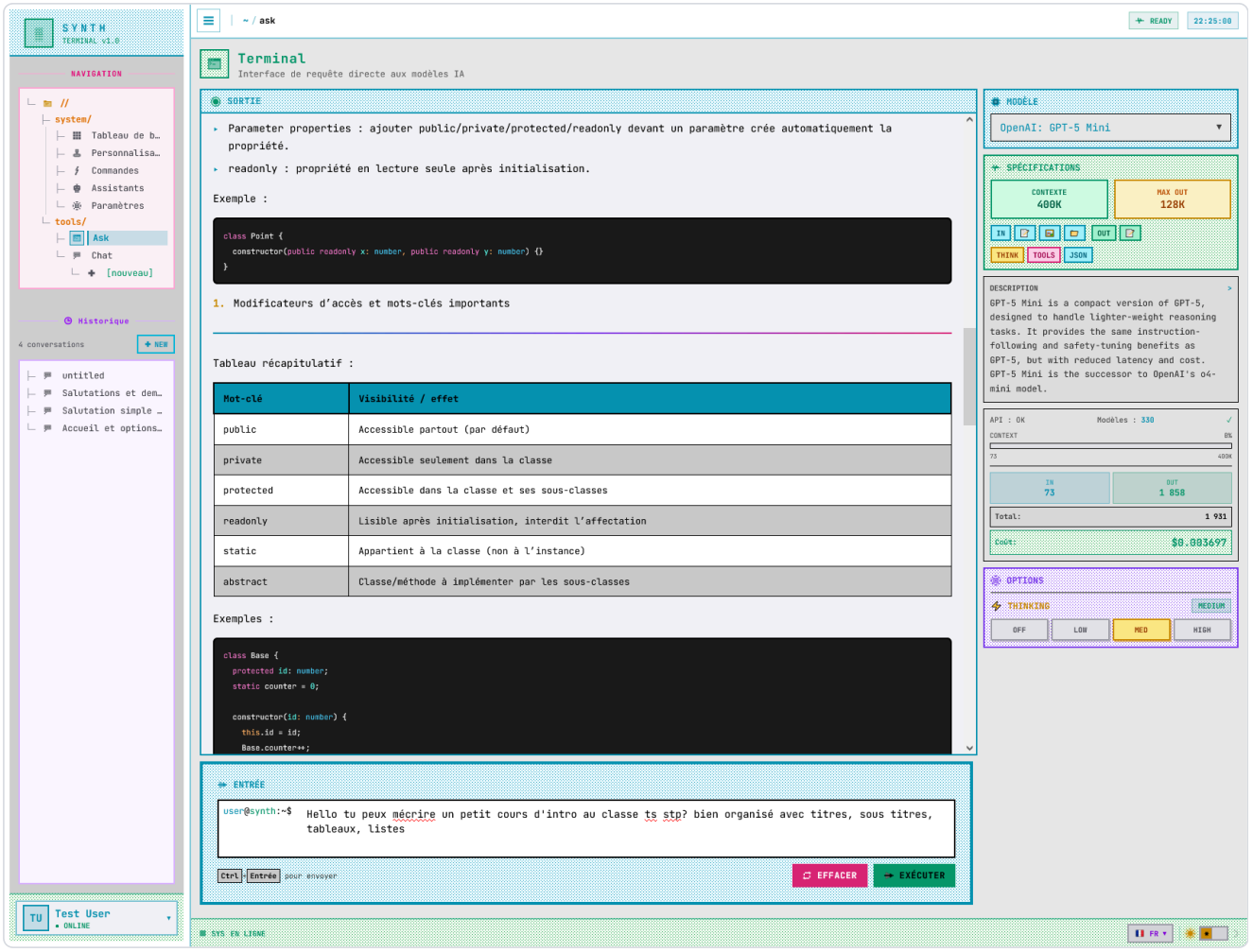
- Ajouter le **streaming** pour voir les tokens apparaître en temps réel
- Créer un **vrai système de conversations** avec historique
- Structurer le code avec un **Client** propre et des **DTOs**

Si, malgré la clarté éblouissante de ce chapitre, vous rencontrez un souci, demandez-vous si vous avez tout fait correctement. Dans 90 % des cas, non. (╯°□°)╯ ┻━┻

Allez, au travail, futurs maîtres de l'IA… ou futurs champions du *copier-coller*.

Autres exemples :





4.6 🧠 Fenêtre de Contexte : La Mémoire des IA

🎯 C'est quoi exactement ?

La fenêtre de contexte, c'est **la mémoire à court terme** de ton chatbot IA. Imagine-la comme la RAM de ton ordi, mais pour les conversations.

En gros :

- 🔢 Elle se mesure en **tokens** (pas en mots !)
- 💬 Elle contient **tout l'historique de la conversation**
- 🧱 1000 tokens ≈ 750 mots en anglais (un peu moins en français)

> ⚠️ ***Important*** *: Si l'historique est pourri, les réponses seront pourries aussi. Garbage in, garbage out !*

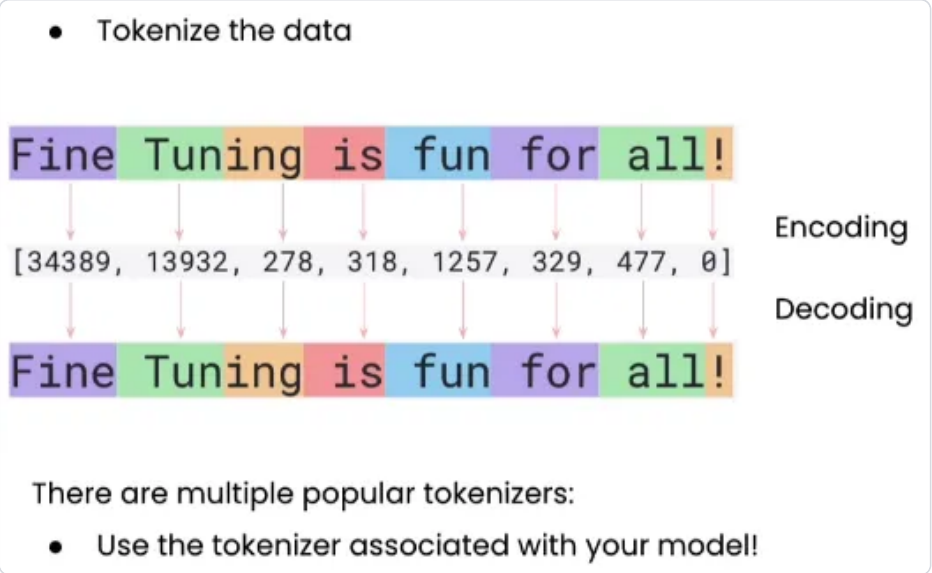
🔤 Le Token : La Brique Élémentaire

C'est quoi un token ?

Un token, c'est pas un mot. C'est plutôt un **morceau de texte** que l'IA comprend.

Exemples de découpage :

- `"Bonjour"` → 1 token
- `"anticonstitutionnellement"` → 5-6 tokens
- `"Fine Tuning is fun for all!"` → 8 tokens

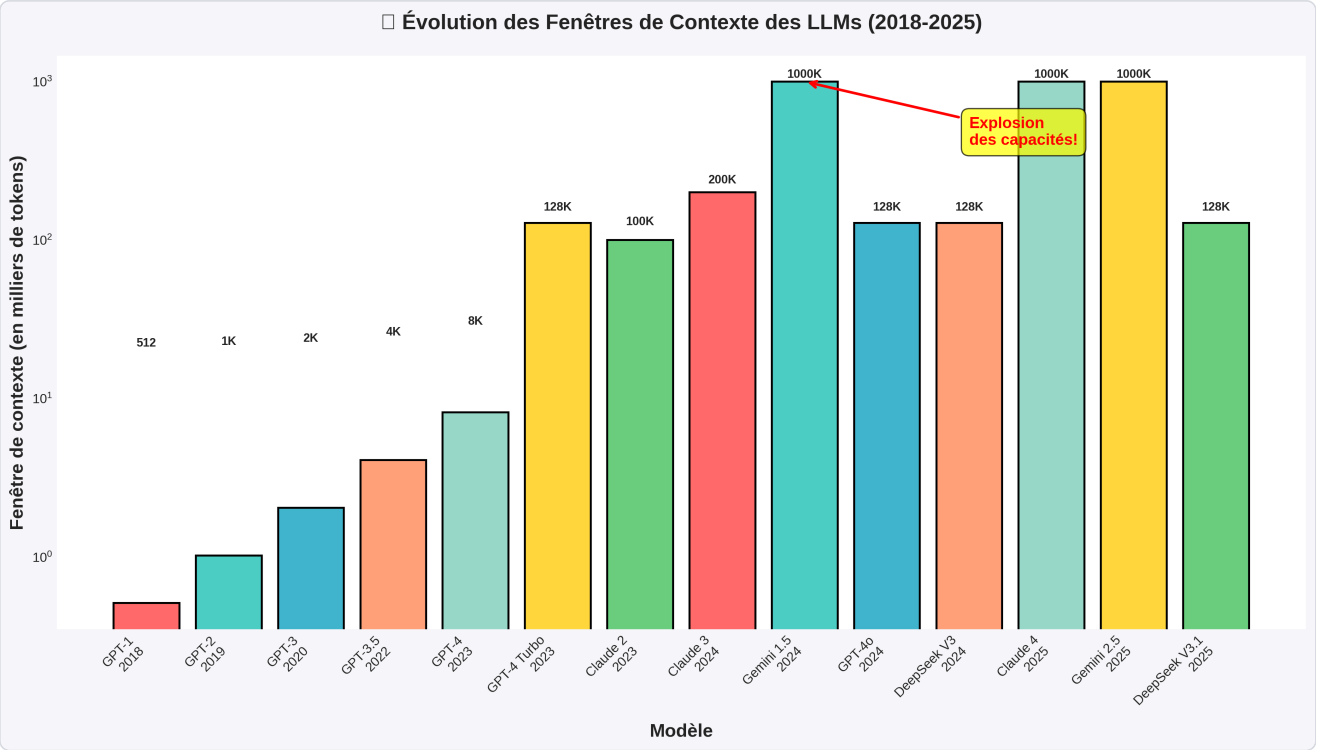


🎮 Règles du jeu

- Les **espaces** comptent
- La **ponctuation** compte
- Les **emojis** = souvent 2 tokens 😎
- Les langues asiatiques (🇨🇳🇯🇵🇰🇷) = plus de tokens par caractère

📊 L'Évolution Explosive des Fenêtres de Contexte

🚀 De 512 tokens à 2 millions en 7 ans !



Timeline de la folie :

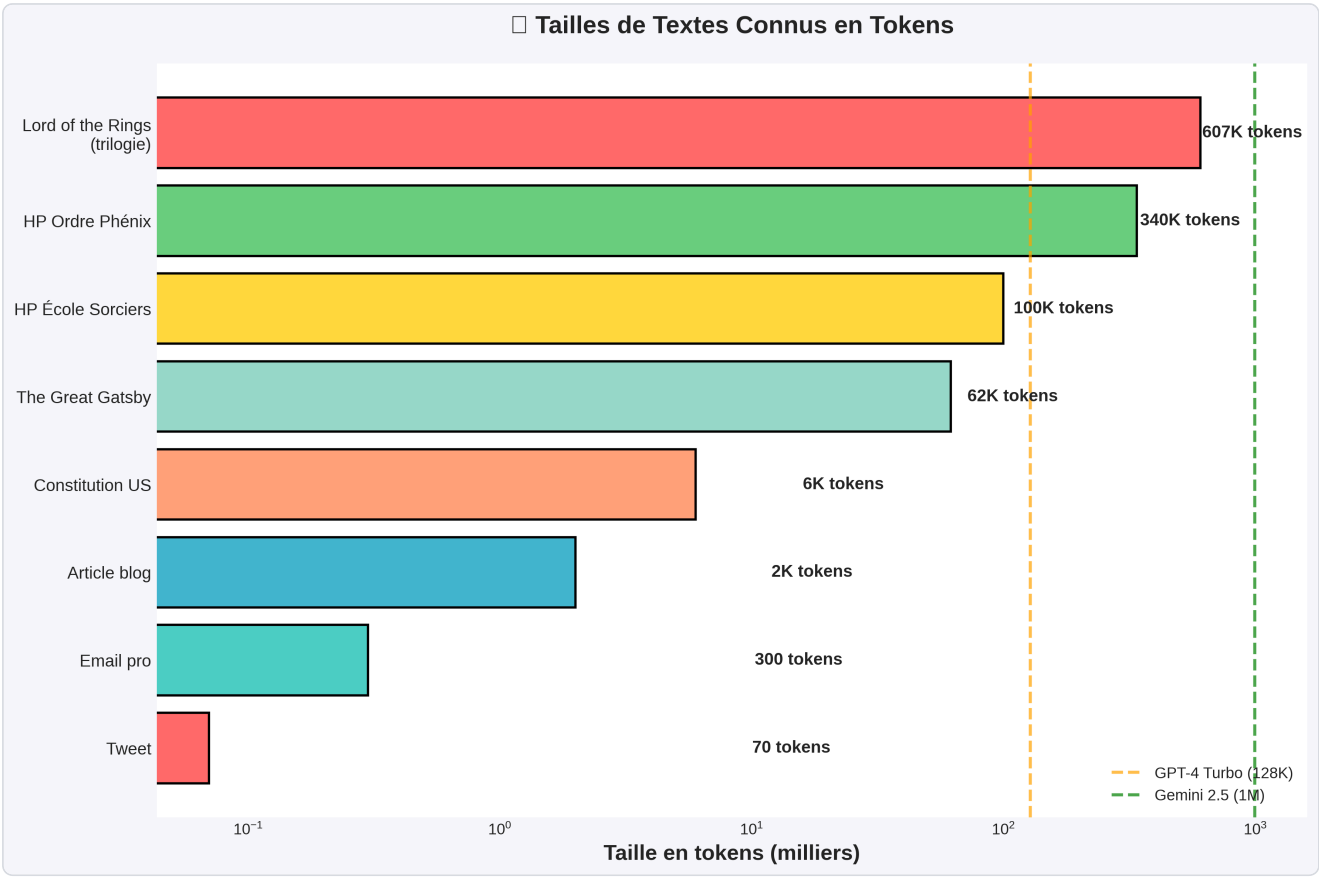
Année	Modèle	Fenêtre	🤯 Facteur
2018	GPT-1	512 tokens	x1 (base)
2019	GPT-2	1K tokens	x2
2020	GPT-3	2K tokens	x4
2022	GPT-3.5	4K tokens	x8
2023	GPT-4	8K → 128K	x250 !
2024	Gemini 1.5 Pro	**1M tokens**	x2000 !!
2025	Gemini 2.5 Pro	**1-2M tokens**	x4000 !!!

💡 **Fun fact** : En 2025, certains modèles peuvent lire **l'intégralité de la Bible** (Ancien + Nouveau Testament = ~773K mots) en un seul coup !



📝 Tailles de Textes Connus en Tokens

Pour bien visualiser ce que ça représente :



📱 Textes Courts

- **Tweet** (280 caractères) → ~50-70 tokens
- **Email professionnel** → 200-400 tokens
- **Thread Twitter** (10 tweets) → ~500-700 tokens

📄 Textes Moyens

- **Article de blog** → 1,500-2,500 tokens
- **Constitution des États-Unis** → ~6,000 tokens

- **Déclaration d'Indépendance US** → ~1,750 tokens

📚 Romans & Livres

- **The Great Gatsby** → ~62,000 tokens (47K mots)
- **Harry Potter à l'école des sorciers** → ~100,000 tokens
- **Harry Potter et l'Ordre du Phénix** → ~340,000 tokens
- **Le Hobbit** → ~126,000 tokens
- **Lord of the Rings** (trilogie complète) → ~607,000 tokens

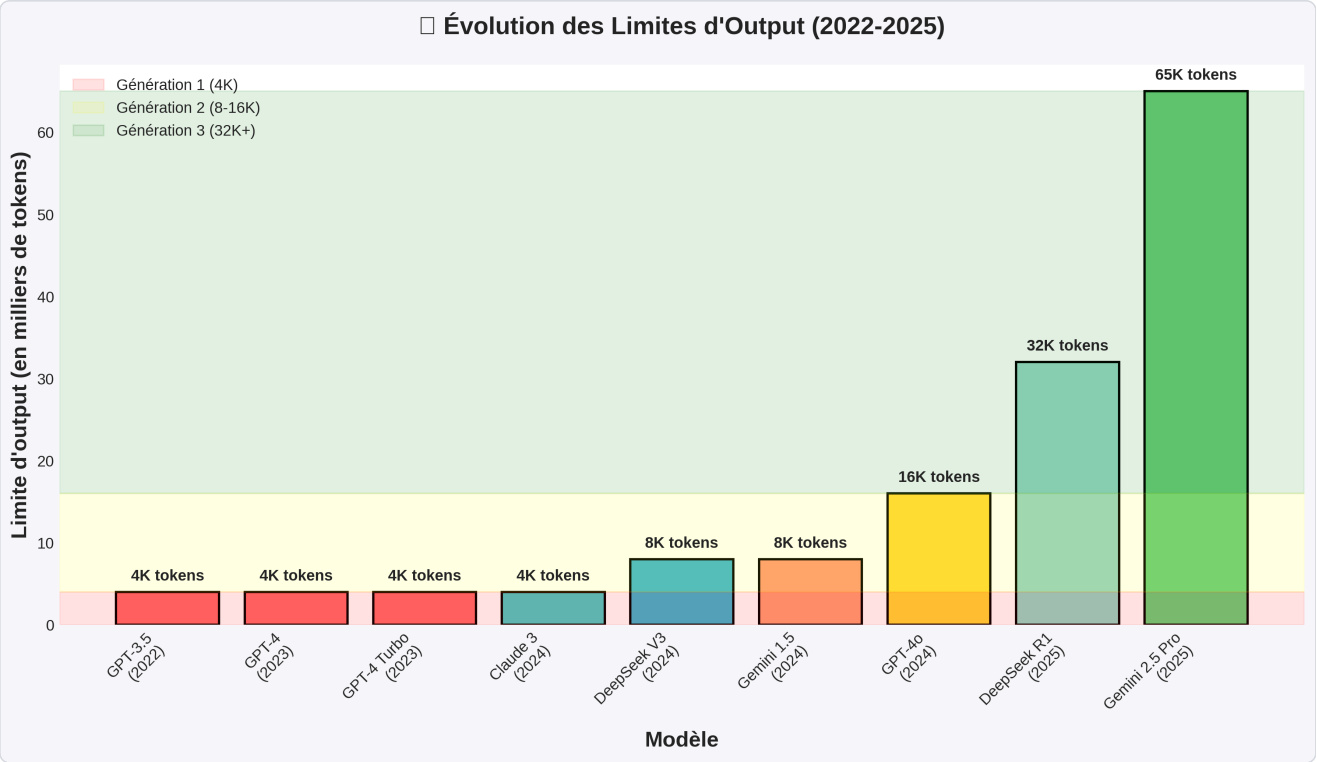
> 💭 ***Analogie*** *: Gemini 2.5 Pro avec ses 2M tokens pourrait lire* ***plus de 3 fois*** *toute la saga Harry Potter d'un coup !*



📤 Limites d'Output : L'Autre Face de la Médaille

🎯 Output ≠ Input !

Pendant longtemps, les modèles pouvaient **lire** beaucoup mais **écrire** peu.



📈 L'Évolution des Limites d'Output

Génération 1 (2022-2023) : L'Ère des 4K

- GPT-3.5 → **4K tokens max** (~3000 mots)
- GPT-4 → **4K tokens**
- Claude 3 → **4K tokens**

Génération 2 (2024) : Le Passage à 8-16K

- DeepSeek V3 → **8K tokens**
- Gemini 1.5 Pro → **8K tokens**
- GPT-4o → **16K tokens**

Génération 3 (2025) : L'Explosion ! 🚀

- DeepSeek R1 (Reasoner) → **32K tokens**
- **Gemini 2.5 Pro → 65K tokens** 🏆
- Claude Sonnet 4 → **64K tokens**

💡 Pourquoi c'est important ?

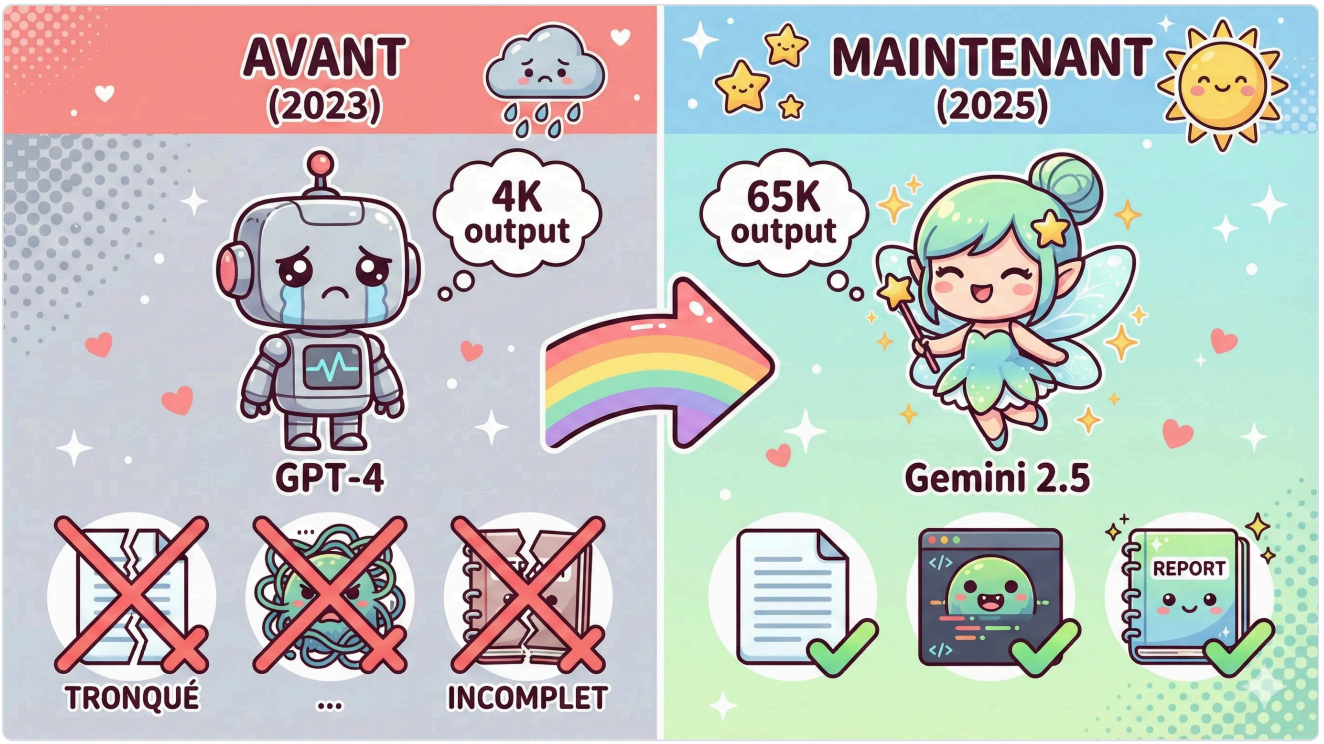
Avant (4K limit) :

- ❌ Impossible de générer un rapport complet
- ❌ Code tronqué en plein milieu
- ❌ Documents incomplets

Maintenant (65K limit) :

- ✅ Rapports complets de 50 pages
- ✅ Codebases entières
- ✅ Documentation technique complète

✅ Scripts et analyses de bout en bout



🎭 Les Modèles Actuels (Décembre 2025)

🏆 Le Top 3 des Champions (Décembre 2025)

Rang	Modèle	Context	Output	💰 Prix	🌟 Spécialité
🥇	**Gemini 2.5 Pro**	2M	65K	$$$	Multimodal + output géant
🥈	**Claude Sonnet 4**	1M	64K	$$	Qualité d'écriture
🥉	**DeepSeek V3.2**	128K	8K	$	Sparse attention, -50% coût

📋 Tableau Complet des Modèles Actuels

Modèle	Context	Output	💡 Note Spéciale
OpenAI			
GPT-5	256-400K	~64K	Dernière génération
GPT-4o	128K	16K	Multimodal rapide
GPT-4 Turbo	128K	4K	Ancienne génération
Anthropic			

Modèle	Context	Output	💡 Note Spéciale
Claude Sonnet 4	1M	64K	Top qualité
Claude 3.5 Sonnet	200K	8K	Génération précédente
Claude 3 Opus	200K	4K	Legacy
Google			
Gemini 2.5 Pro	1-2M	**65K** 🏆	Plus gros output du marché
Gemini 2.0 Flash	1M	8K	Rapide et efficace
Gemini 1.5 Pro	1M	8K	Needle champion (99.7%)
DeepSeek			
V3.2 🆕	**128K**	8K	Sparse attention, -50% coût
V3.2-Speciale 🥇	128K	8K	Gold medals IMO/IOI 2025
V3.1	128K	8K	Génération précédente
R1 (Reasoner)	64K	32K	Mode reasoning

🧠 Le "Thinking" : Quand l'IA Réfléchit

🤔 C'est quoi le "thinking" ?

Certains modèles (Claude, DeepSeek R1, Gemini 2.5, GPT-5) peuvent maintenant **montrer leur raisonnement** avant de répondre.

Comment ça marche :

1. 💭 L'IA génère une **trace de réflexion** (thinking)
2. ✍️ Puis elle donne sa **réponse finale**
3. 📊 Le thinking compte dans les **tokens d'output**

Exemple avec DeepSeek R1 :

```
<think>
Hmm, l'utilisateur demande la racine carrée de 144.
Voyons... 12 x 12 = 144
Donc la réponse est 12.
</think>

La racine carrée de 144 est 12.
```

⚙️ Comment Contrôler le Thinking

Méthode 1 : Max Tokens (Gemini, Anthropic, Qwen)

```javascript
// Limite directe du nombre de tokens de réflexion
"reasoning": {
  "max_tokens": 2000  // Max 2000 tokens pour penser
}
```

Supporté par :

- ✅ Gemini thinking models
- ✅ Anthropic (Claude) via `reasoning.max_tokens`
- ✅ Alibaba Qwen (mappé sur `thinking_budget`)

Méthode 2 : Effort Level (OpenAI, xAI)

```javascript
// Contrôle par pourcentage d'effort
"reasoning": {
  "effort": "high"  // 80% des tokens pour penser
}
```

Niveaux disponibles :

Effort	% de max_tokens	Utilisation
high	~80%	Problèmes très complexes
medium	~50%	Tâches normales
low	~20%	Simple
minimal	~10%	Très simple
none	0%	Désactive le thinking

Supporté par :

- ✅ OpenAI (o1, o3, GPT-5 series)
- ✅ xAI (Grok models)

⚠️ Le Piège du Thinking

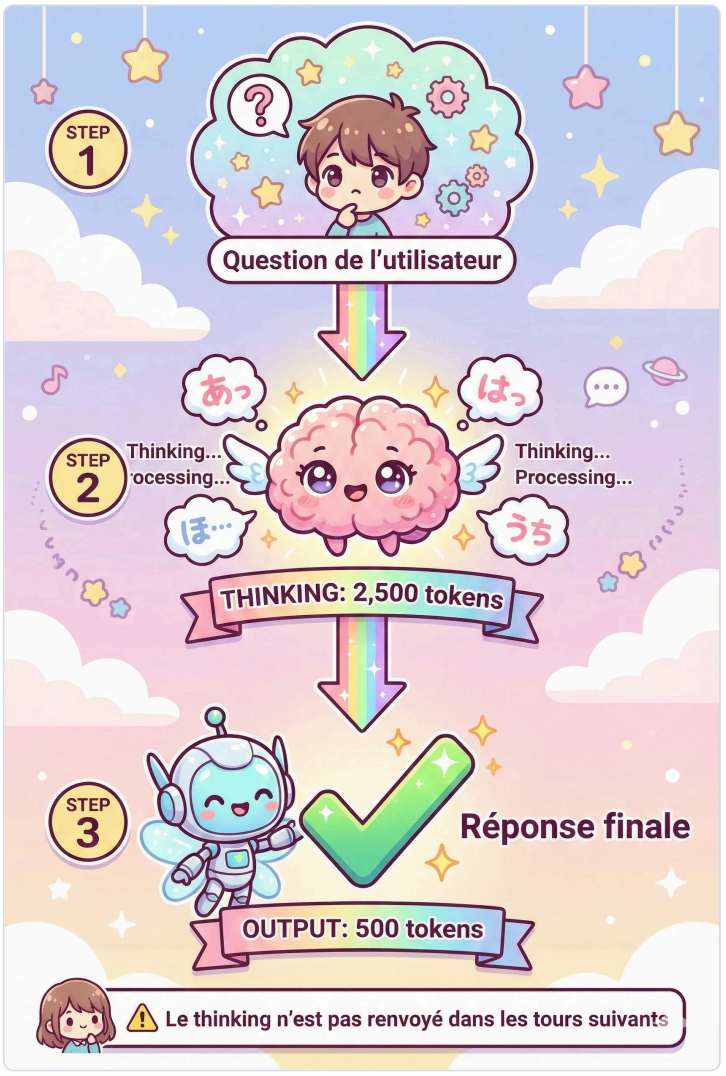
Dans les conversations multi-tours :

- ❌ On n'envoie **PAS** le thinking dans les messages suivants
- ✅ On ne garde que la réponse finale
- 💡 **Pourquoi ?** Pour économiser des tokens

Conséquence :

- L'IA perd la trace de **pourquoi** elle a fait certaines choses

- Elle ne peut pas "revoir son raisonnement"
- Mais on économise des tonnes de tokens !



🔍 Tests "Needle in a Haystack" : Trouver l'Aiguille

🎯 Le Concept

L'idée :
Cacher une info spécifique (l'"aiguille") dans un énorme contexte (la "botte de foin") et voir si l'IA la retrouve.



Exemple concret :

```
Contexte : 100,000 tokens d'essais de Paul Graham
Aiguille : "La meilleure chose à faire à San Francisco
           est de manger un sandwich à Dolores Park"
Question : "D'après le contexte, que recommande-t-on
           de faire à San Francisco ?"
```

📊 Résultats des Tests (2024-2025)

GPT-4 (128K context)

- ✅ Performance stable jusqu'à 64K tokens
- ⚠️ Déclin à partir de 100K tokens
- ❌ Problème "lost in the middle" : oublie le milieu du contexte

Claude 2.1 (200K context)

- 😕 Score initial : 27% de précision
- 🎯 Avec prompt optimisé : jusqu'à 98% !
- 💡 Très sensible à la formulation

Gemini 1.5 Pro (1M context)

- 🏆 **>99.7% de précision** jusqu'à 1M tokens

- 🎥 Test audio : **100% de précision** sur 107h d'audio (5 jours !)
- 🎬 Test vidéo : 99.8% sur 10.5h de vidéo

Test Multi-Needles (100 aiguilles)

- Gemini 1.5 Pro : 60% de rappel à 1M tokens
- Performance diminue avec le nombre d'aiguilles
- Le raisonnement sur les infos retrouvées est plus dur que la simple récupération

🎨 Needle in Haystack - Visualisation

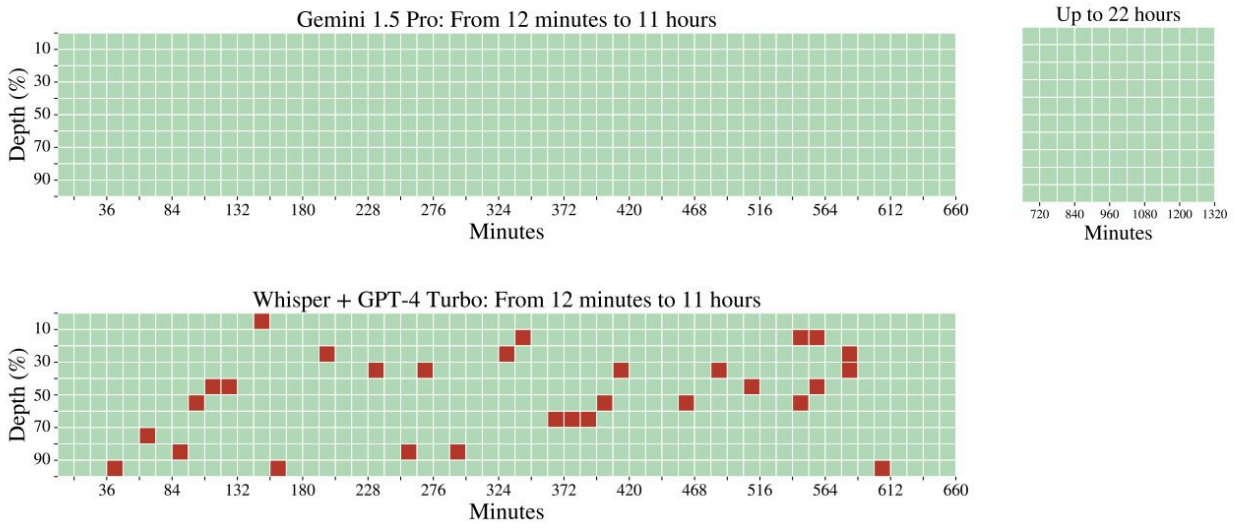


Figure 9 | **Audio Haystack.** This figure presents the audio version of the needle-in-a-haystack experiment comparing Gemini 1.5 Pro and a combination of Whisper and GPT-4 Turbo. In this setting, the needle is a short segment of audio that is inserted within a very large audio segment (of up to 22 hours) containing concatenated audio clips. The task is to retrieve the "secret keyword" which is revealed in the needle. Red indicates that the model did not identify the keyword, whereas green indicates that the model identified the keyword correctly.

> 💡 ***Leçon*** *: Plus la fenêtre est grande, plus c'est dur de retrouver les infos ! Mais Gemini 1.5 Pro est un beast.*

🔗 Extension & Recherche Documentaire

🗄️ Récupération Augmentée (RAG)

Les modèles ne se limitent pas à leur fenêtre de contexte. Ils peuvent **aller chercher des infos ailleurs** !

Le fonctionnement :

1. 🔍 Recherche dans des **documents externes**
2. 📊 Recherche dans des **bases de données**
3. 🌐 Recherche sur **Internet** (Google, Bing)
4. 💾 Récupération depuis des **APIs**

Avantages :

- ✅ Accès à des infos **à jour**
- ✅ Pas limité par la fenêtre de contexte
- ✅ Sources **vérifiables**

Inconvénients :

- ⏱️ Plus lent
- 💰 Plus coûteux
- 🎯 Qualité dépend de la recherche

🌐 Recherche Internet Intégrée

Les modèles récents (GPT-4, Claude 3, Gemini) peuvent **faire des recherches sur Internet directement**.

Cas d'usage :

- 📰 Actualités récentes
- 📈 Données en temps réel
- 🔬 Publications scientifiques récentes
- 🏢 Infos sur des entreprises

💰 Le Coût de la Mémoire

💸 Ça Coûte Combien ?

La règle d'or :

> ***Attention = Complexité Quadratique***
> *Doubler la longueur du contexte = **Quadrupler** le coût computationnel*

Pourquoi ?

Dans les transformers classiques, chaque token doit "regarder" tous les autres tokens :

- 1,000 tokens → 1,000 × 1,000 = **1 million** de comparaisons
- 2,000 tokens → 2,000 × 2,000 = **4 millions** de comparaisons
- 10,000 tokens → 10,000 × 10,000 = **100 millions** de comparaisons

```bash
Coût = O(L²) où L = longueur du contexte
```

C'est pour ça que les longues fenêtres coûtent **très cher** !

Exemples de pricing (Gemini 2.5 Pro) :

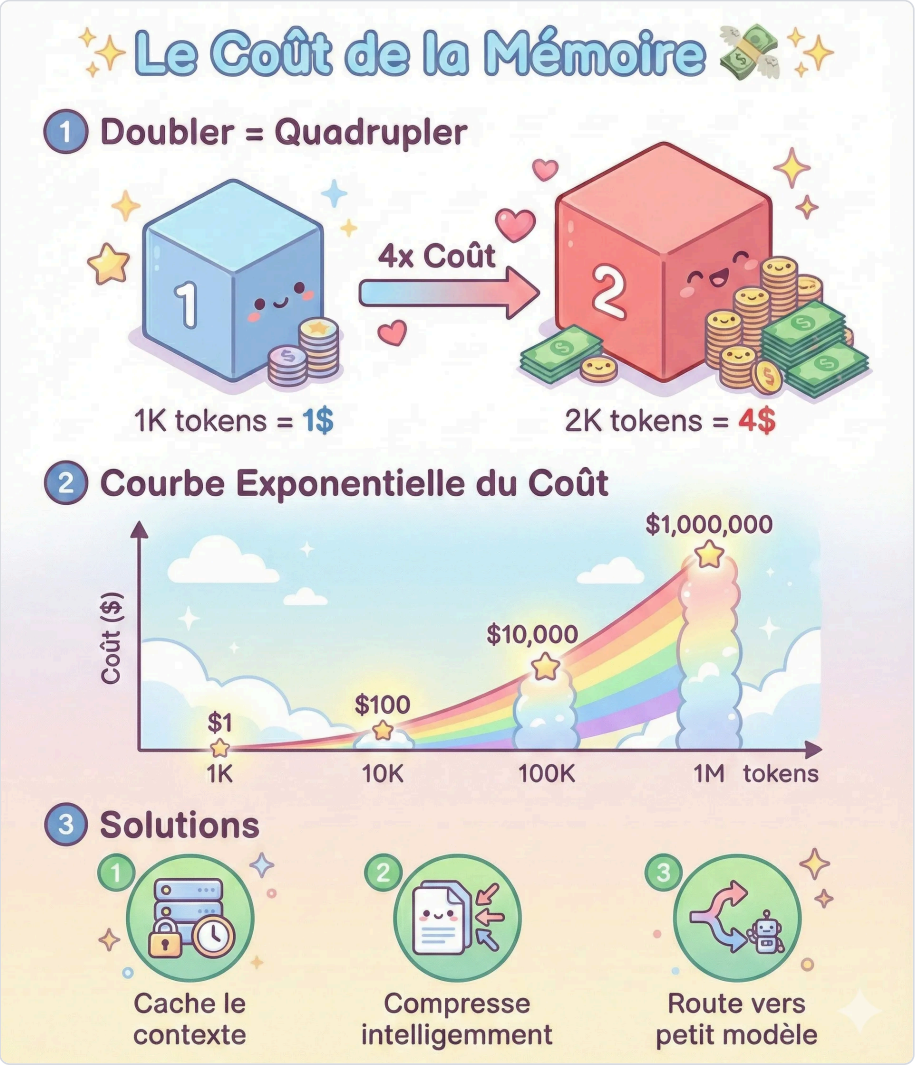
- Input : $0.00125 par 1000 tokens
- Output : $0.005 par 1000 tokens

Scénario réel :

```yaml
Prompt : 100,000 tokens
Réponse : 10,000 tokens

Coût input : 100 × 0.00125=0.00125 = 0.00125=0.125
Coût output : 10 × 0.005=0.005 = 0.005=0.05
Total : $0.175 par requête
```

Pour 1000 requêtes/jour → $175/jour** → **5,250/mois 💀



🎯 Optimisation des Coûts

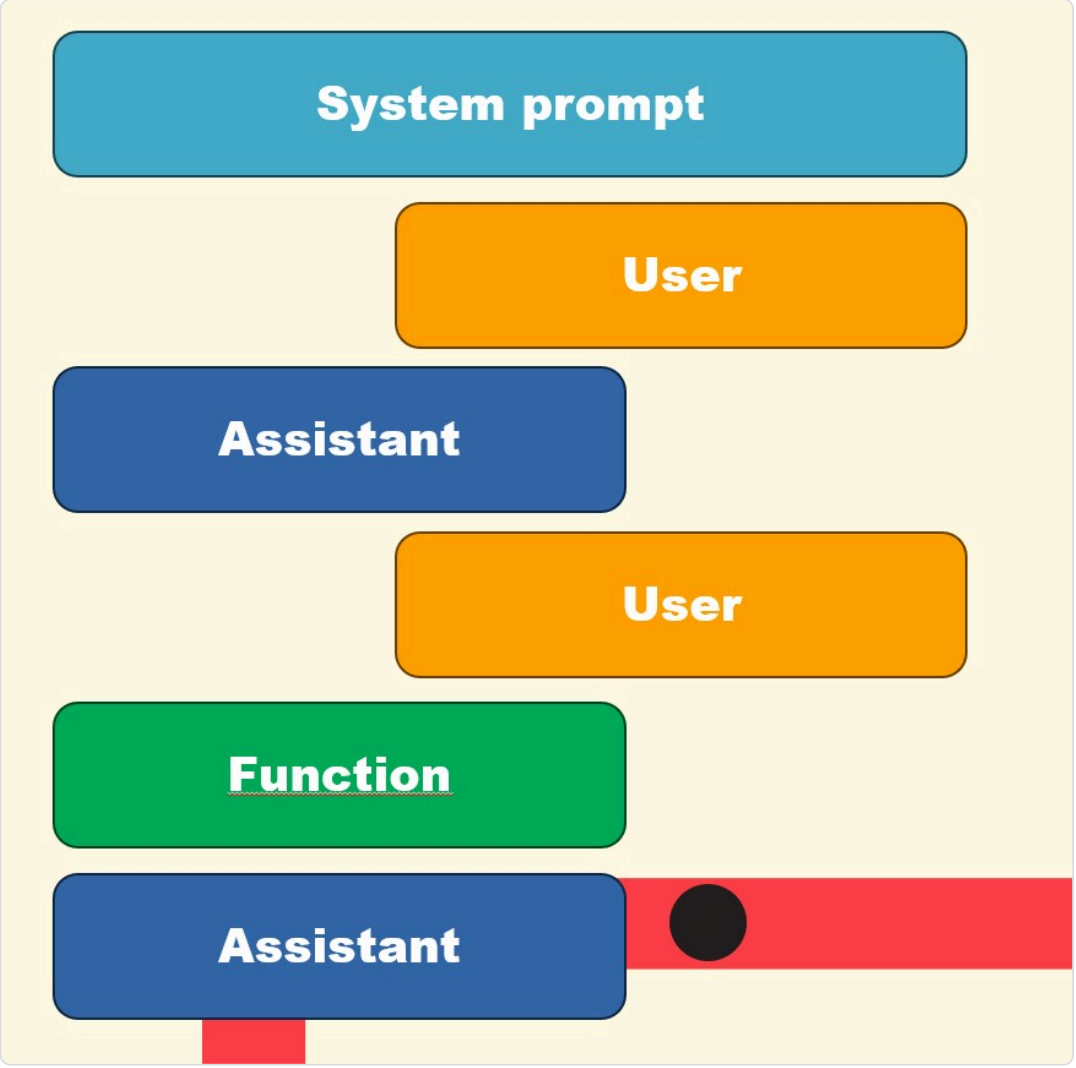
Techniques :

1. 📉 Utiliser le modèle le plus petit qui fait le job
2. ✂️ Tronquer le contexte intelligemment
3. 🔄 Utiliser le cache de contexte (si dispo)
4. 🎯 Limiter la taille d'output
5. 📊 Router vers des modèles plus petits quand possible

🎭 Anatomie d'un Prompt (Détails Techniques)

🗣️ Les Différents Rôles

Chaque message dans la conversation a un **rôle** :



1. System (Caché, au début)

```css
"Tu es un assistant expert en programmation Python.
Réponds toujours avec du code commenté."
```

- Définit le **comportement global**
- L'utilisateur ne le voit pas
- Compte dans les tokens !

2. User (Orange)

```bash
"Comment faire une boucle for en Python ?"
```

- Messages de **l'utilisateur**

- Visibles dans l'interface

3. Assistant (Bleu)

```python
# Voici un exemple de boucle for
for i in range(10):
    print(i)
```

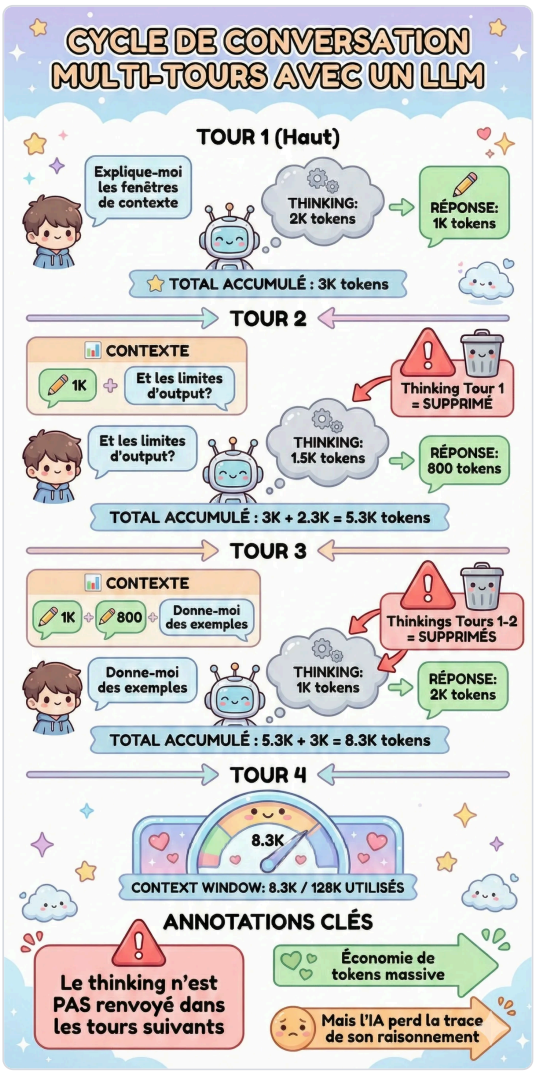
- Réponses du **modèle**
- Générées par l'IA

4. Function (Vert, caché)

```json
{
  "name": "web_search",
  "arguments": {
    "query": "latest Python version"
  }
}
```

- Appels d'**outils/fonctions**
- Recherche web, APIs, etc.
- Invisibles pour l'utilisateur

🔄 Le Cycle de la Conversation



À chaque nouveau message, TOUT l'historique est renvoyé !

🚨 Problèmes Courants & Solutions

❌ Problème 1 : "Lost in the Middle"

Le problème :
Les modèles se souviennent mieux du **début** et de la **fin** du contexte, mais oublient le **milieu**.

Solution :

- Mettre les infos importantes au début OU à la fin
- Répéter les infos cruciales

❌ Problème 2 : Dépassement de Limite

Le problème :

```
Error: Context length exceeded (150000 > 128000)
```

Solutions :

1. ✂️ **Tronquer** l'historique
2. 📝 **Résumer** les anciens messages
3. 🔄 **Router** vers un modèle avec + de contexte
4. 🗑️ **Supprimer** les messages moins importants

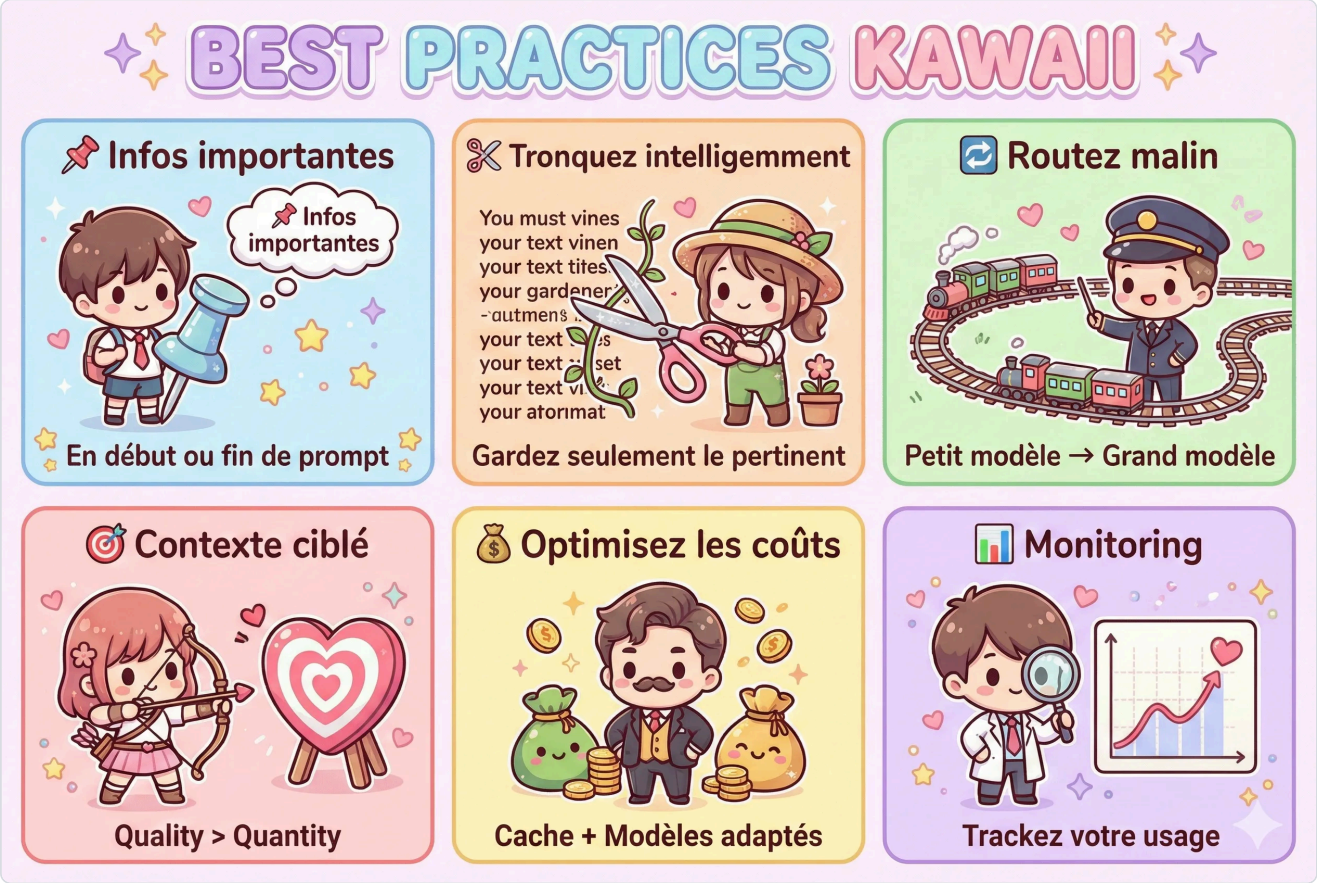
❌ Problème 3 : Contexte de Mauvaise Qualité

Le problème :
Trop d'infos = confusion

Solution :

```bash
❌ Mauvais :
"Voici 50 pages de documentation sur tout..."

✅ Bon :
"Voici la section pertinente sur X :
[extrait ciblé de 500 tokens]"
```



🎓 Récap' Final

✅ Ce qu'il faut retenir

1. **Fenêtre de contexte** = mémoire à court terme de l'IA
2. **Tokens** ≠ mots (1 token ≈ 0.75 mot en anglais)
3. **Évolution folle** : 512 tokens (2018) → 2M tokens (2025)
4. **Output** évolue aussi : 4K → 65K tokens
5. **Thinking** compte dans les tokens d'output
6. **"Lost in the middle"** : mettre les infos importantes au début/fin
7. **Coûts** : x2 tokens = x4 calcul
8. **Tests Haystack** : Gemini 1.5 Pro = champion (99.7%)

🚀 L'Avenir

Tendances 2025+ :

- 🌟 Fenêtres de **10M tokens** en approche
- 🧠 Meilleure gestion du "middle" du contexte
- 💰 Baisse des coûts
- ⚡ Vitesse accrue
- 🎯 Récupération d'infos plus précise

🔗 Sources & Pour Aller Plus Loin

📚 Documentation Officielle

- OpenAI GPT Models
- Anthropic Claude
- Google Gemini
- DeepSeek V3

🔬 Articles Techniques

- https://www.newsletter.theaidiscovery.com/p/133
- https://teetracker.medium.com/llm-fine-tuning-step-tokenizing-caebb280cfc2
- https://towardsdatascience.com/towards-infinite-llm-context-windows-e099225abaaf
- https://arize.com/blog-course/the-needle-in-a-haystack-test-evaluating-the-performance-of-llm-rag-systems/
- https://cloud.google.com/blog/products/ai-machine-learning/the-needle-in-the-haystack-test-and-how-gemini-pro-solves-it

🛠️ Outils Pratiques

- Tokenizer OpenAI
- LLMTest Needle In A Haystack

- Context Window Comparator

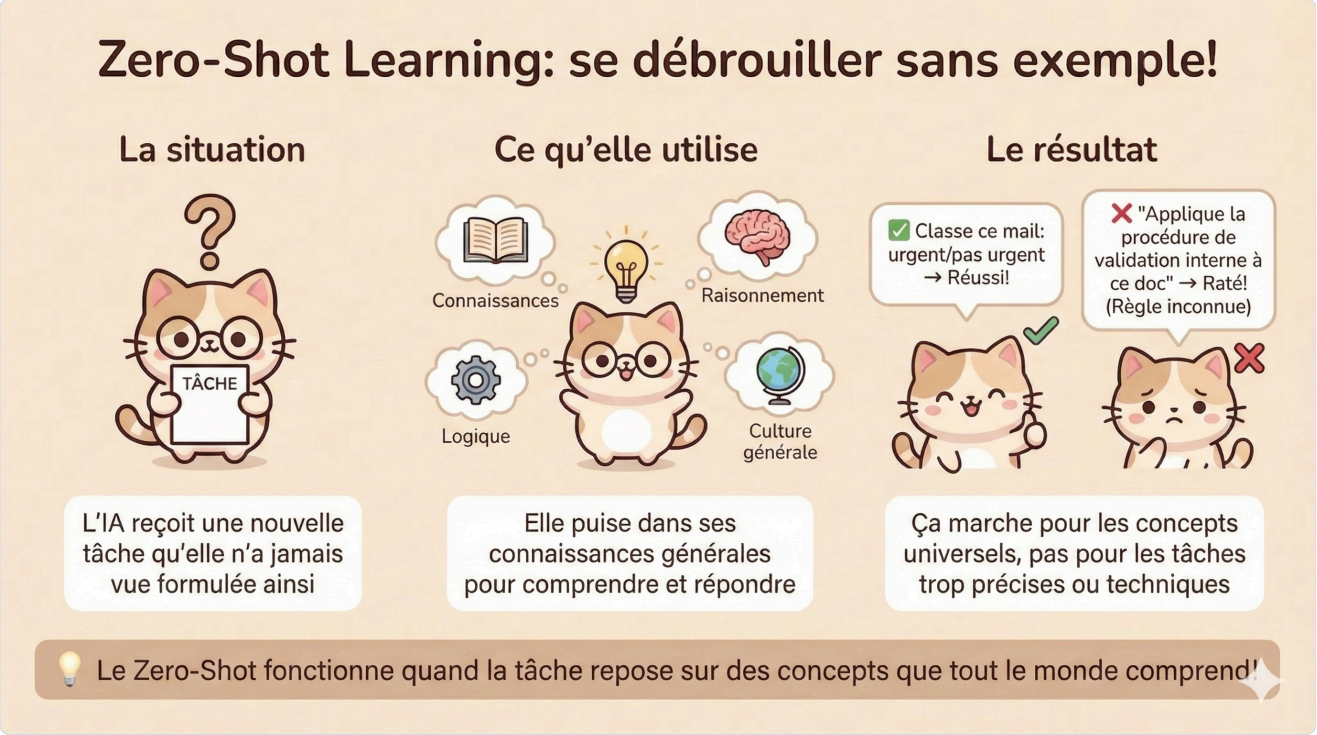
4.7 Prompt Engineering

Introduction

Le **Prompt Engineering**, c'est l'art de formuler des instructions claires pour obtenir exactement ce qu'on veut d'une intelligence artificielle. Pas besoin d'être développeur ou data scientist : bien communiquer avec une IA, ça s'apprend.

Ce chapitre présente les techniques essentielles pour tirer le meilleur parti de ChatGPT et compagnie, du plus simple au plus sophistiqué.

Zero-shot : Quand l'IA se débrouille seule



C'est quoi ?

Le **zero-shot**, c'est quand on demande à l'IA d'accomplir une tâche **sans lui donner d'exemple**. Elle doit se débrouiller uniquement avec ses connaissances générales acquises pendant son entraînement.

C'est comme demander à quelqu'un de cultivé : "Peux-tu classer ce message comme urgent ou pas ?" Il n'a jamais fait exactement ce tri pour toi, mais il comprend ce qu'est l'urgence et peut appliquer ce concept à ta situation.

Quand ça marche ?

Le zero-shot fonctionne bien quand :

- La tâche repose sur des **concepts universels** (sentiment, urgence, catégories courantes)

- Le **format de sortie est évident** (oui/non, une catégorie, un résumé)
- Il n'y a **pas d'ambiguïté** sur ce qu'on attend

Exemple concret : Trier ses notifications

Prompt :

```bash
Classe cette notification selon son niveau de priorité.
Notification : "Rappel : Rendez-vous dentiste demain 14h"
Catégories : Urgent, Important, Informatif, Spam
```

Réponse de l'IA :

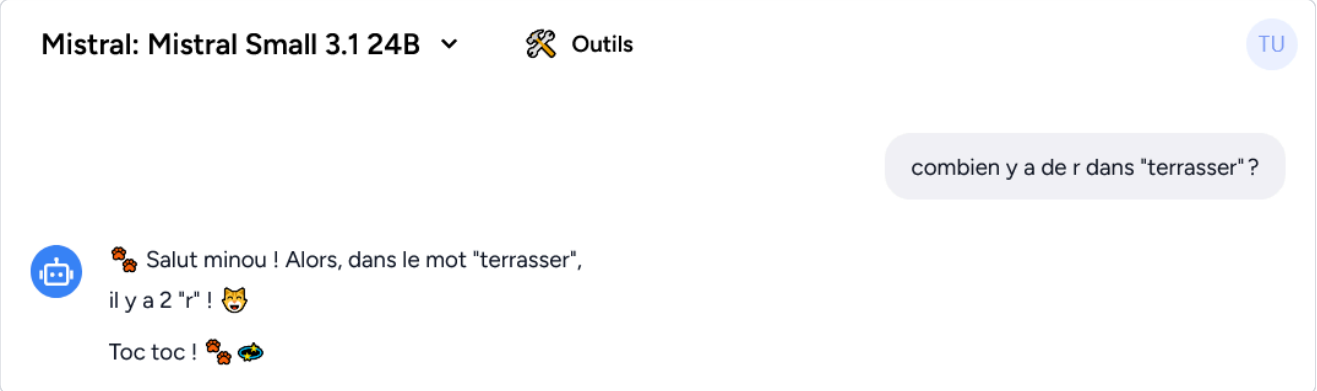
```bash
Important

Ce rendez-vous médical prévu pour demain nécessite une action
(s'y rendre), mais n'est pas une urgence immédiate.
```

Ça marche parce que l'IA comprend ce qu'est un rendez-vous médical, le concept de priorité, et peut raisonner sur l'urgence relative.

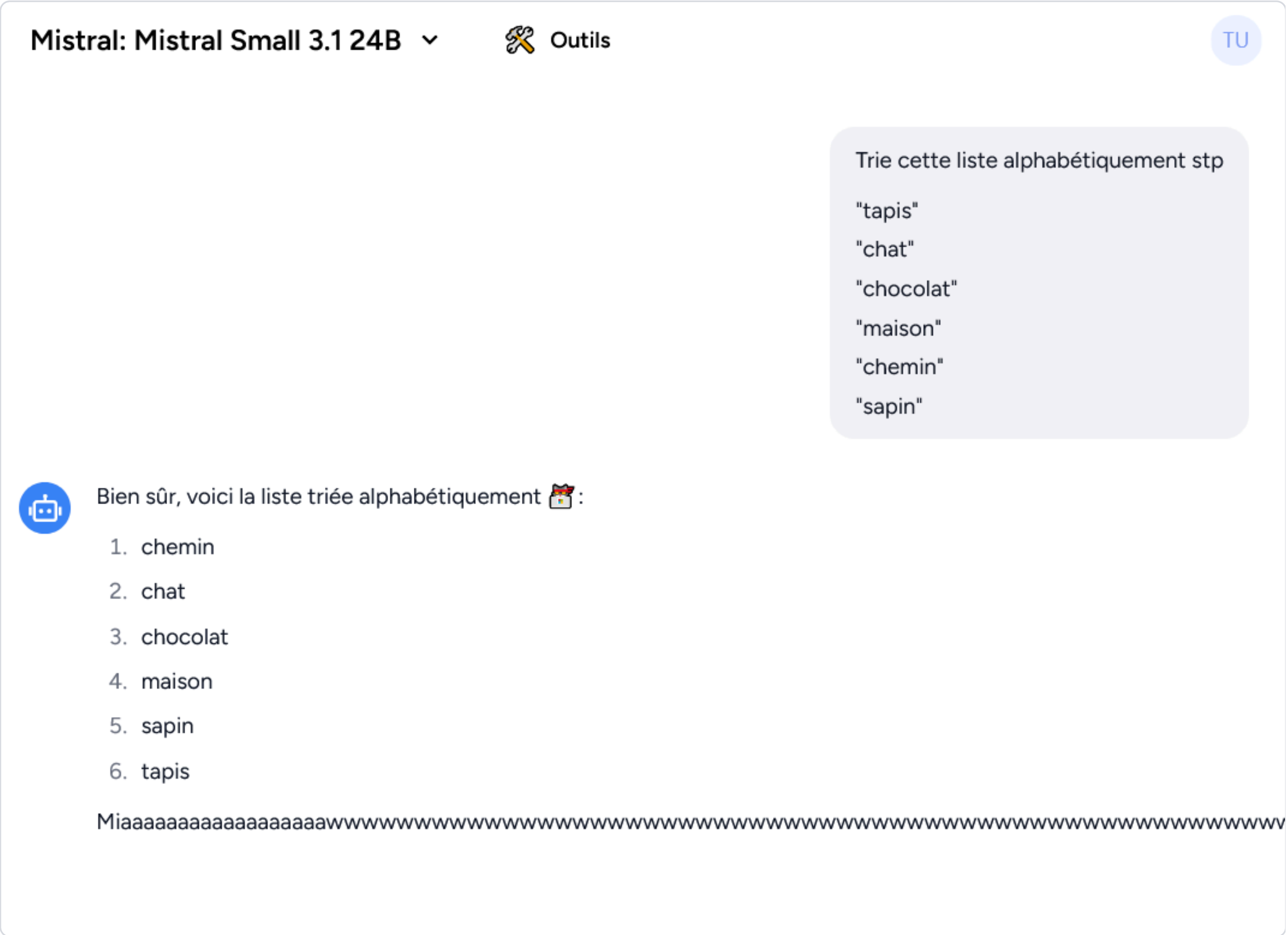
Quand ça ne marche PAS

Le zero-shot a ses limites. Voici un exemple où l'IA se trompe en comptant les lettres d'un mot :



L'IA répond qu'il y a 2 "r" dans "terrasser"... alors qu'il y en a **3** ! (te-**r**-**r**-asse-**r**) 🤦

Autre exemple avec un tri alphabétique :

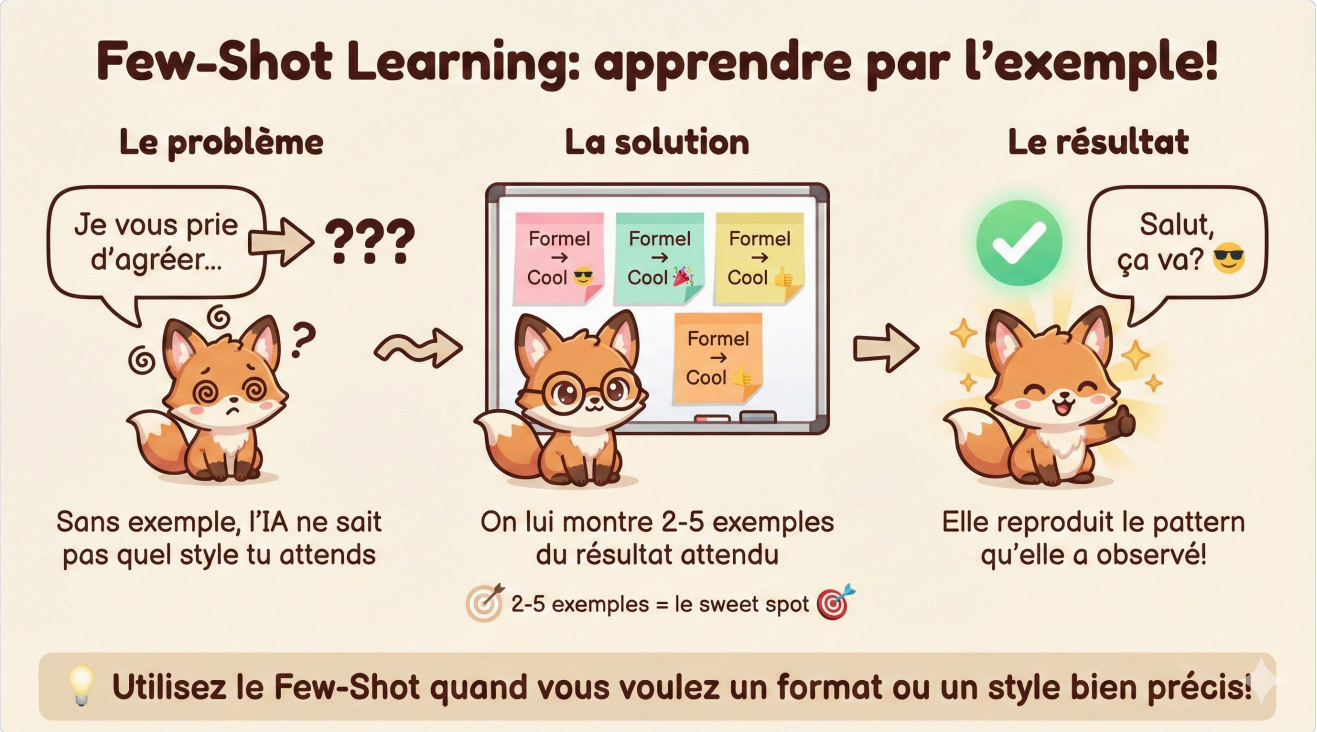


L'IA met "chemin" avant "chat"... alors que "chat" devrait venir en premier alphabétiquement ! 💀

⚠️ Piège classique

Les tâches qui semblent simples (compter, trier) peuvent échouer en zero-shot car l'IA "devine" plutôt qu'elle ne calcule vraiment. Pour ces cas, le **few-shot** ou le **Chain of Thought** donnent de meilleurs résultats.

Few-shot : Apprendre par l'exemple



C'est quoi ?

Le **few-shot**, c'est quand on montre à l'IA **quelques exemples** (généralement 2 à 5) de ce qu'on attend avant de lui poser la vraie question. Elle apprend le pattern et le reproduit.

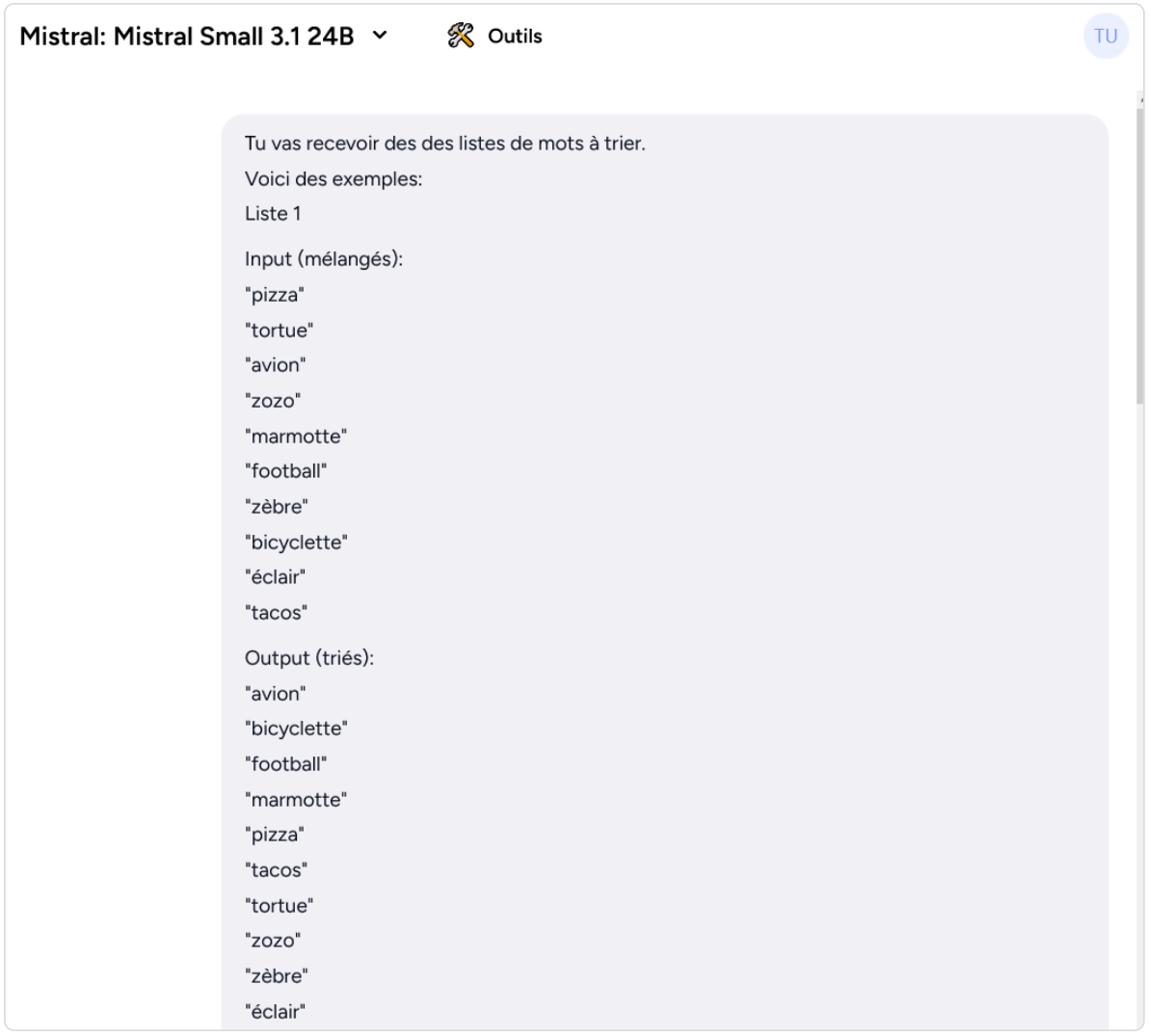
Quand l'utiliser ?

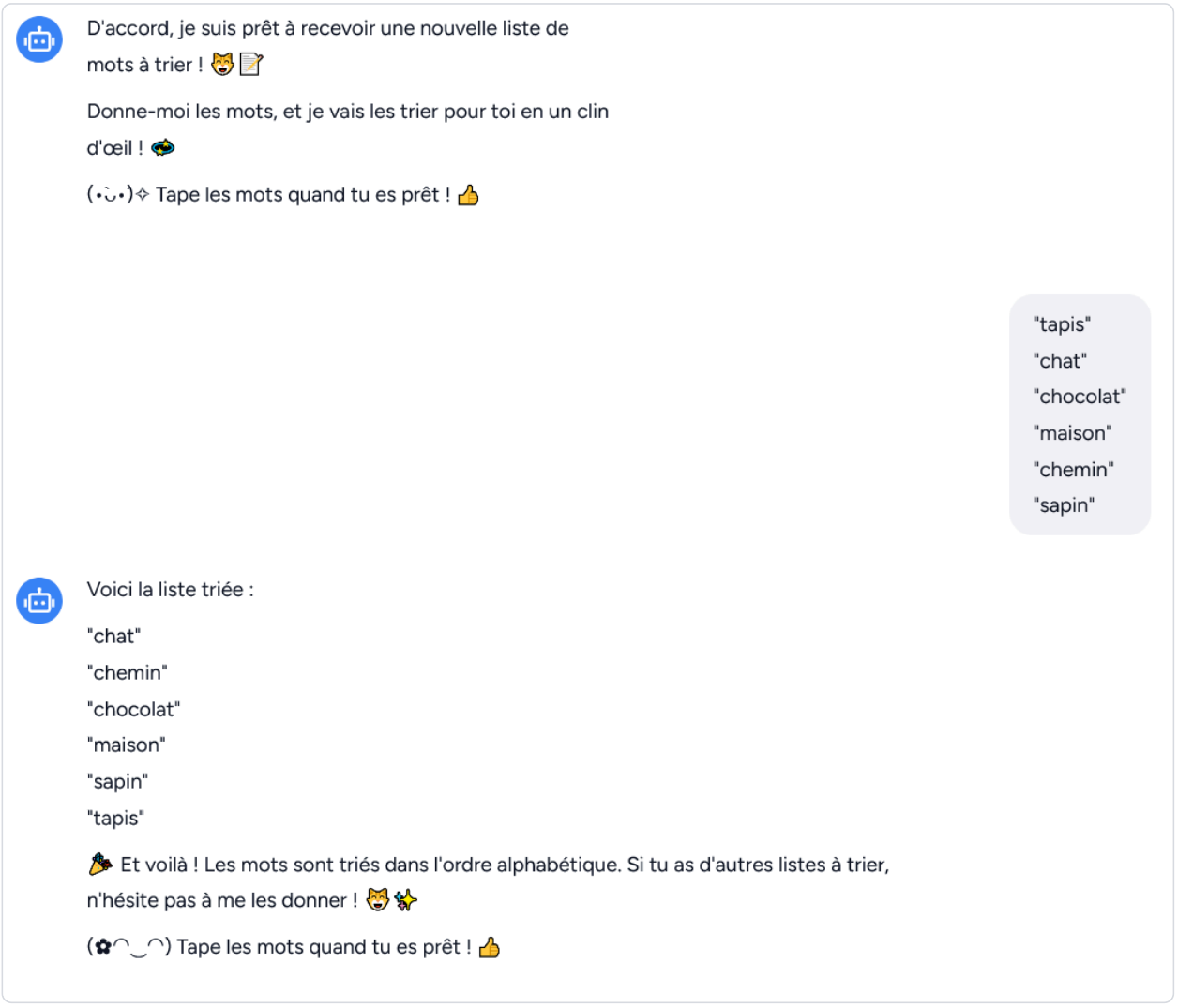
Utilisez le few-shot quand :

- Le **format de sortie** est très spécifique
- Le **ton ou le style** attendu n'est pas évident
- La tâche est **ambiguë** sans démonstration
- Le **zero-shot a échoué** (comme notre exemple de tri!)

Exemple concret : Le tri alphabétique (version corrigée)

Reprenons l'exemple du tri qui avait échoué en zero-shot. Cette fois, on donne des exemples à l'IA :


Mistral: Mistral Small 3.1 24B
Outils
TU
Tu vas recevoir des des listes de mots à trier.
Voici des exemples:
Liste 1
Input (mélangés):
"pizza"
"tortue"
"avion"
"zozo"
"marmotte"
"football"
"zèbre"
"bicyclette"
"éclair"
"tacos"
Output (triés):
"avion"
"bicyclette"
"football"
"marmotte"
"pizza"
"tacos"
"tortue"
"zozo"
"zèbre"
"éclair"



Cette fois, "chat" est bien placé avant "chemin" ! ✅

En montrant un exemple de tri correct, l'IA comprend exactement ce qu'on attend et reproduit le comportement.

Autre exemple : Reformuler des messages

Sans exemple (zero-shot) :

```bash
Transforme ce message formel en SMS décontracté :
"Je vous confirme ma présence à la réunion de demain à 14h."
```

Réponse probable : "Je confirme pour demain 14h 👍"

C'est correct... mais peut-être pas *ton* style.

Avec exemples (few-shot) :

```yaml
Transforme ces messages formels en SMS, comme dans les exemples :

Formel : "Je vous remercie pour votre retour rapide."
SMS : "Merci pour le retour rapidos! 🙏"

Formel : "Pourriez-vous me transmettre le document ?"
SMS : "Tu peux m'envoyer le doc stp? 📄"

Formel : "Je vous confirme ma présence à la réunion de demain à 14h."
SMS :
```

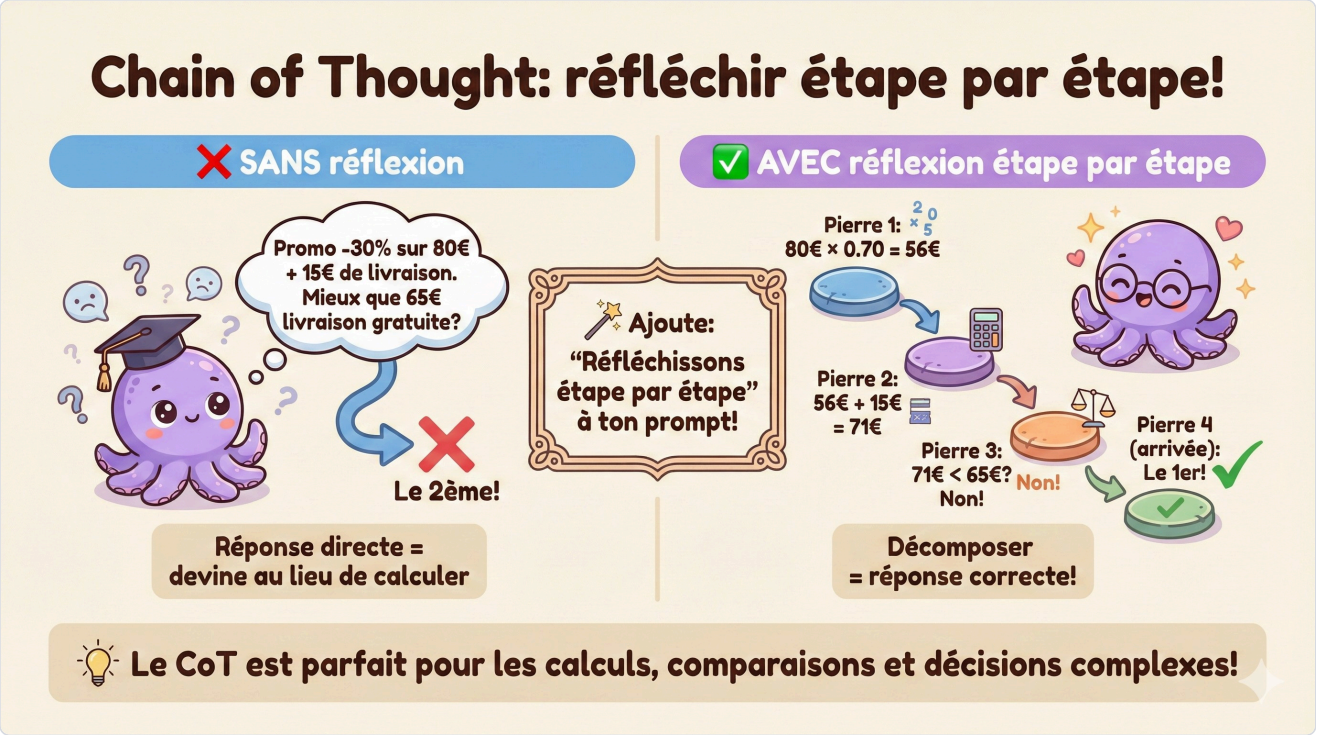
Réponse : "Je serai là demain 14h, c'est noté! 📅"

L'IA a capté le style (émojis, ton décontracté, formulations courtes) et le reproduit.

⚠️ Piège classique

Donner des exemples **incohérents entre eux**. Si vos exemples suivent des patterns différents, l'IA sera confuse et le résultat imprévisible.

Chain of Thought : Réfléchir étape par étape



C'est quoi ?

La **Chain of Thought** (chaîne de pensée), c'est demander à l'IA de **montrer son raisonnement** étape par étape avant de donner sa réponse finale. Ça force l'IA à réfléchir plutôt qu'à deviner.

Quand l'utiliser ?

- Problèmes avec **plusieurs étapes de calcul**

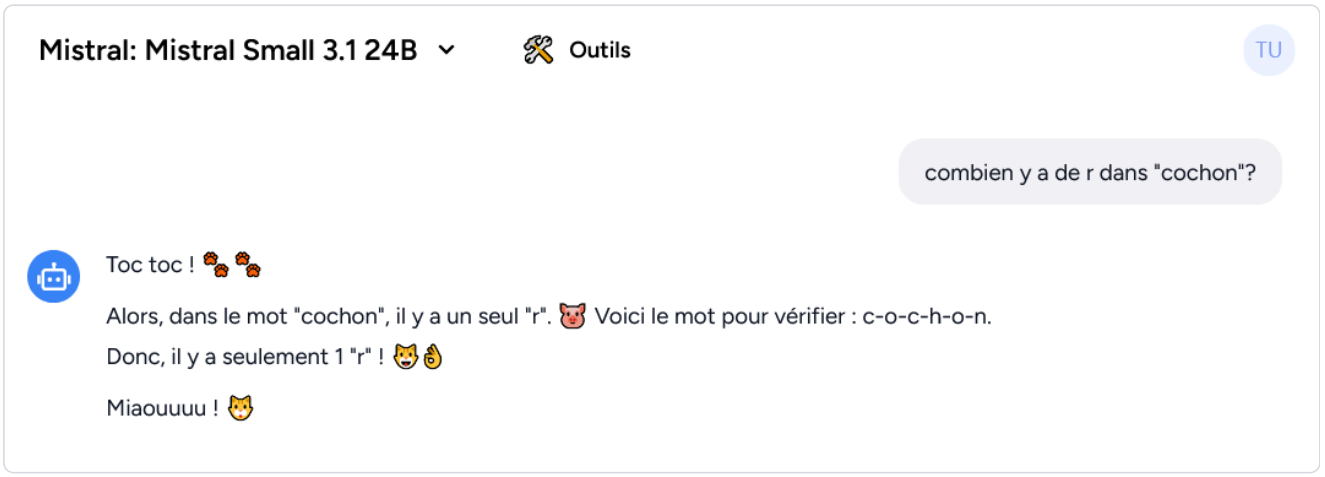
- Décisions impliquant **plusieurs critères** à peser
- Questions où la **réponse directe risque d'être fausse**
- Tâches de **comptage ou de vérification**

La formule magique

Ajoutez simplement : **"Réfléchissons étape par étape."** ou **"Explique ton raisonnement."**

Exemple concret : Compter les lettres (version corrigée)

Souvenez-vous du fail où l'IA trouvait 1 "r" dans "cochon" alors qu'il n'y en a aucun :



Maintenant, demandons-lui de réfléchir étape par étape :



En décomposant le mot lettre par lettre (c-o-c-h-o-n), l'IA trouve correctement **0** ! 🎉

Autre exemple : Évaluer si une promo vaut le coup

Sans Chain of Thought :

```css
Un magasin propose -30% sur un article à 89€, mais les frais de
livraison sont de 12€. Un autre magasin vend le même article à 75€
avec livraison gratuite. Lequel choisir ?
```

Réponse directe (risquée) : "Le deuxième magasin."

C'est... faux! Mais l'IA aurait pu se tromper sans montrer son calcul.

Avec Chain of Thought :

```css
Un magasin propose -30% sur un article à 89€, mais les frais de
livraison sont de 12€. Un autre magasin vend le même article à 75€
avec livraison gratuite. Lequel choisir ?

Réfléchissons étape par étape.
```

Réponse :

```yaml
Étape 1 : Calculons le prix avec la réduction
89€ × 0,70 = 62,30€

Étape 2 : Ajoutons les frais de livraison
62,30€ + 12€ = 74,30€

Étape 3 : Comparons avec l'autre offre
Magasin 1 : 74,30€
Magasin 2 : 75€

Conclusion : Le premier magasin est légèrement moins cher (0,70€ d'économie).
```

La réponse intuitive était fausse! Le CoT permet de s'en rendre compte. 🎯

⚠️ Piège classique

Utiliser le CoT pour des questions simples qui n'en ont pas besoin. Pour "Quelle est la capitale de la France ?", pas besoin de réfléchir en 4 étapes.

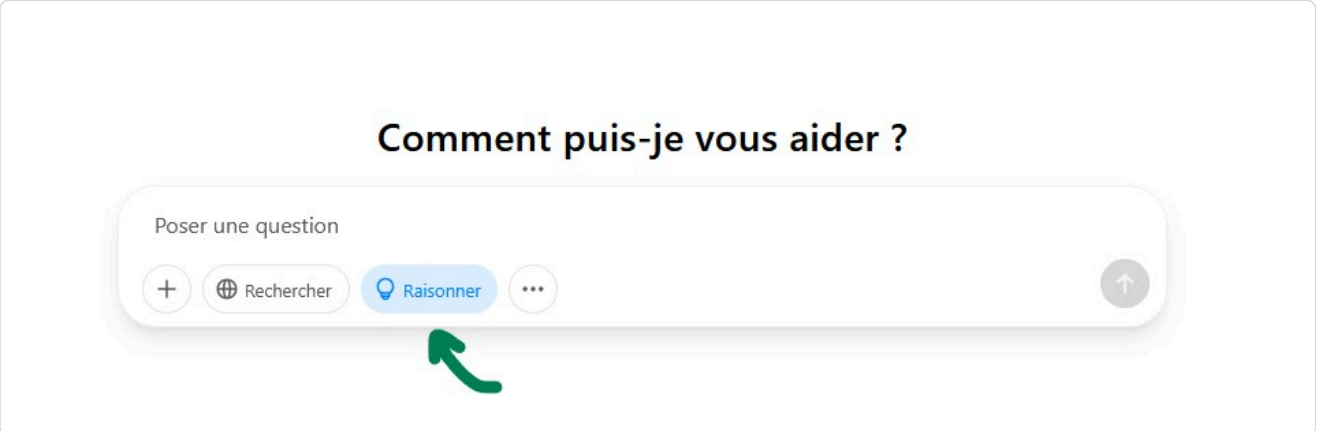
Les modèles de raisonnement

Certains modèles récents intègrent automatiquement la Chain of Thought dans leur fonctionnement. C'est le cas de **GPT5**, **DeepSeek V3.2**, **Grok 4** et **Gemini 3**.

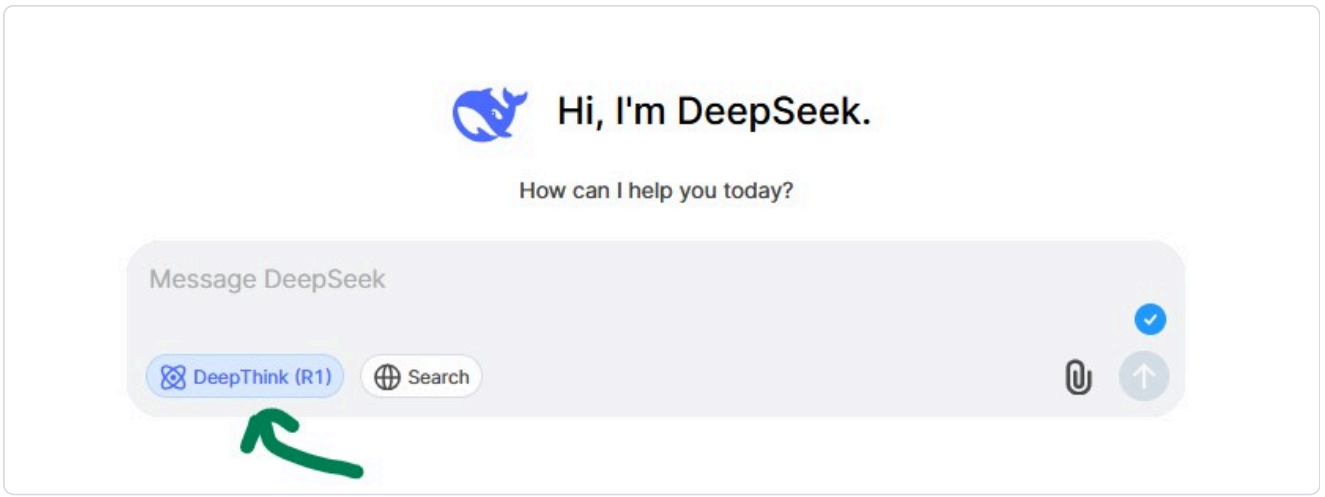
Ces modèles "réfléchissent avant de répondre" : ils génèrent une chaîne de raisonnement interne (parfois visible, parfois cachée) avant de fournir leur réponse finale.

Avantage : Plus précis sur les problèmes complexes.
Inconvénient : Temps de réponse plus long (le modèle "pense" vraiment).

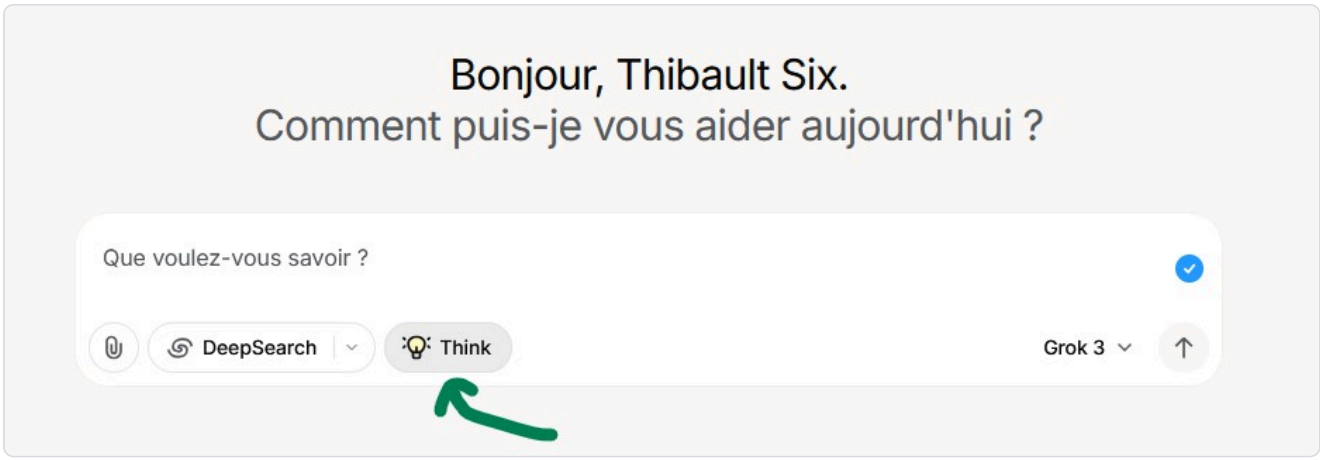
ChatGPT



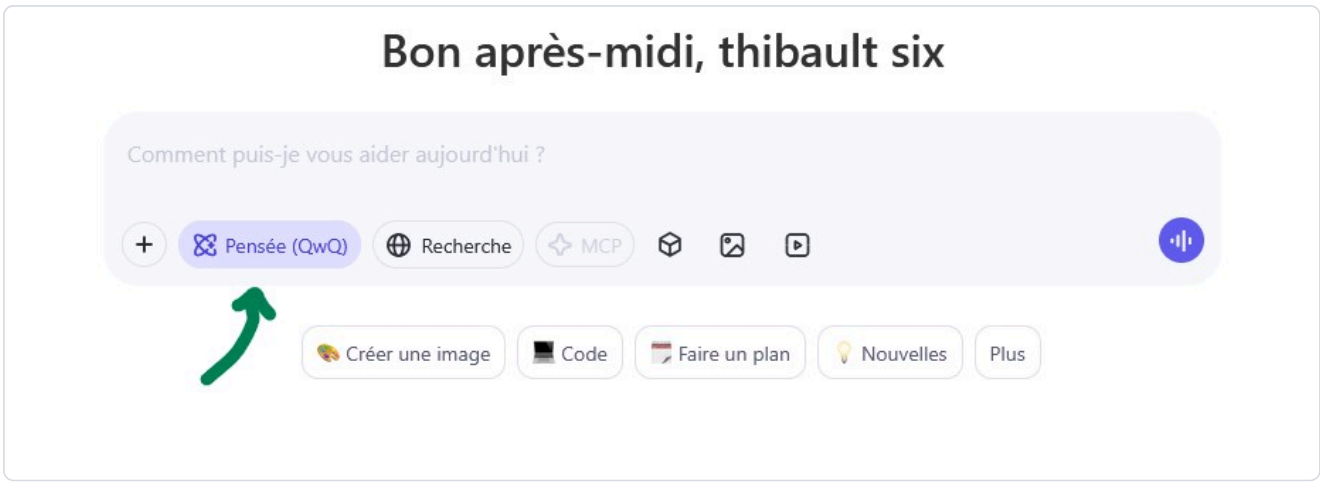
Deepseek



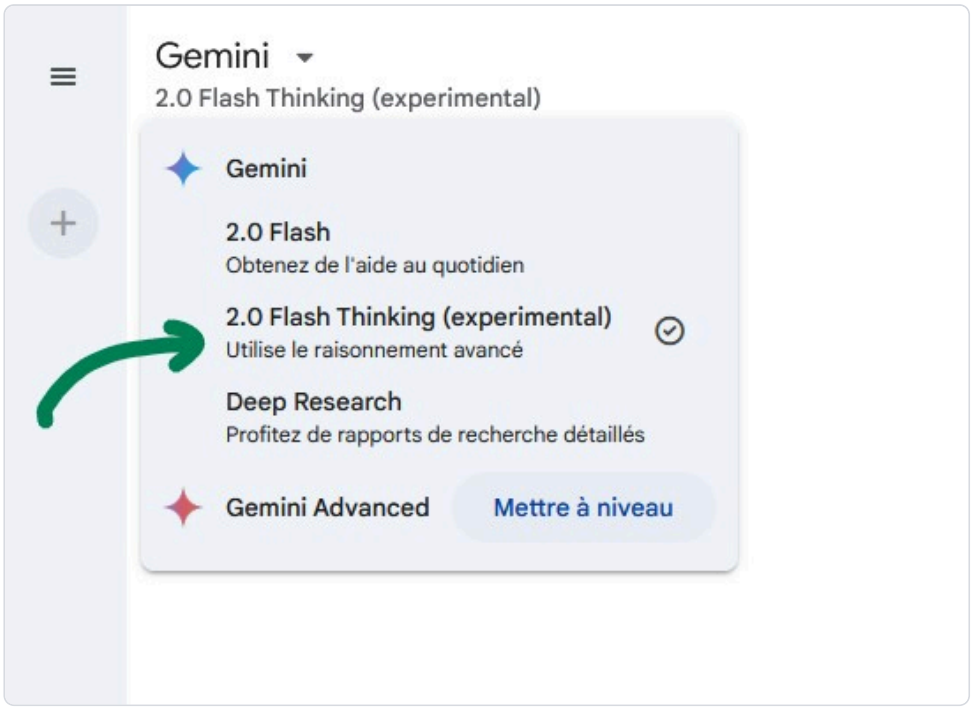
Grok



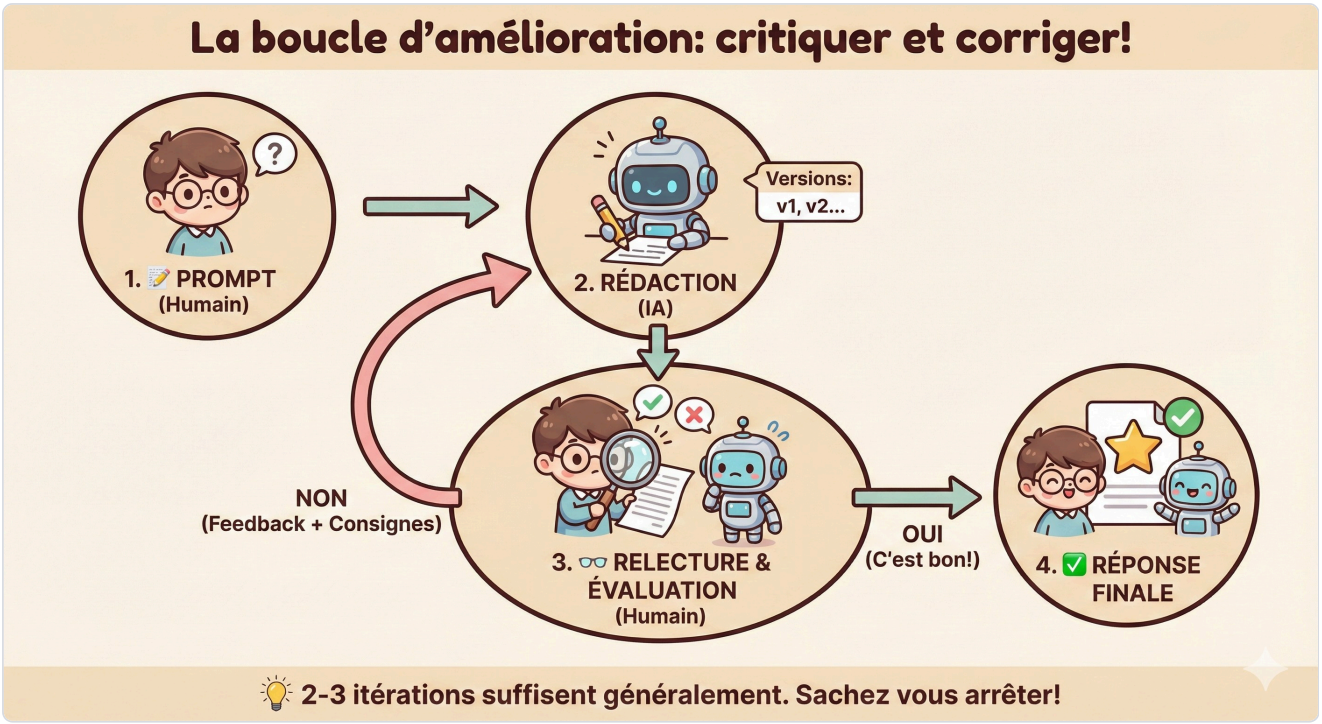
Qwen



Gemini



Autocorrection : L'IA qui se relit



C'est quoi ?

L'**autocorrection**, c'est demander à l'IA de **critiquer et améliorer** sa propre réponse. Un processus en boucle qui affine progressivement la qualité du résultat.

Le processus

1. **Prompt initial** → L'IA génère une première réponse
2. **Rédaction** → L'IA produit une version (v1, v2...)
3. **Relecture & évaluation** → L'humain (et/ou l'IA) examine le résultat
4. **Itération** → Si ce n'est pas bon, on donne du feedback et on recommence
5. **Réponse finale** → Quand c'est validé!

Comment faire ?

Étape 1 - Obtenir une première réponse

```
Explique simplement pourquoi le ciel est bleu.
```

Réponse v1 :

```bash
Le ciel est bleu parce que la lumière du soleil est diffusée
par l'atmosphère.
```

Étape 2 - Demander une auto-évaluation

```bash
Évalue ta réponse précédente. Est-elle complète et précise ?
Que pourrait-on améliorer ?
```

Auto-évaluation :

```
Ma réponse est correcte mais incomplète :
- Je n'explique pas POURQUOI c'est le bleu spécifiquement
- Je n'ai pas mentionné la diffusion de Rayleigh
- C'est trop vague pour vraiment comprendre le phénomène
```

Étape 3 - Corriger

```css
Corrige ta réponse en tenant compte de ces points.
```

Réponse v2 :

```
Le ciel est bleu à cause de la diffusion de Rayleigh : la lumière
du soleil contient toutes les couleurs, mais les courtes longueurs
d'onde (bleu, violet) rebondissent davantage sur les molécules de
l'atmosphère. Le bleu domine car nos yeux y sont plus sensibles
que le violet, et une partie du violet est absorbée en altitude.
```

Vous pouvez aussi ajouter votre propre critique

```css
Ta réponse est bonne mais trop technique. Reformule pour un
enfant de 10 ans.
```

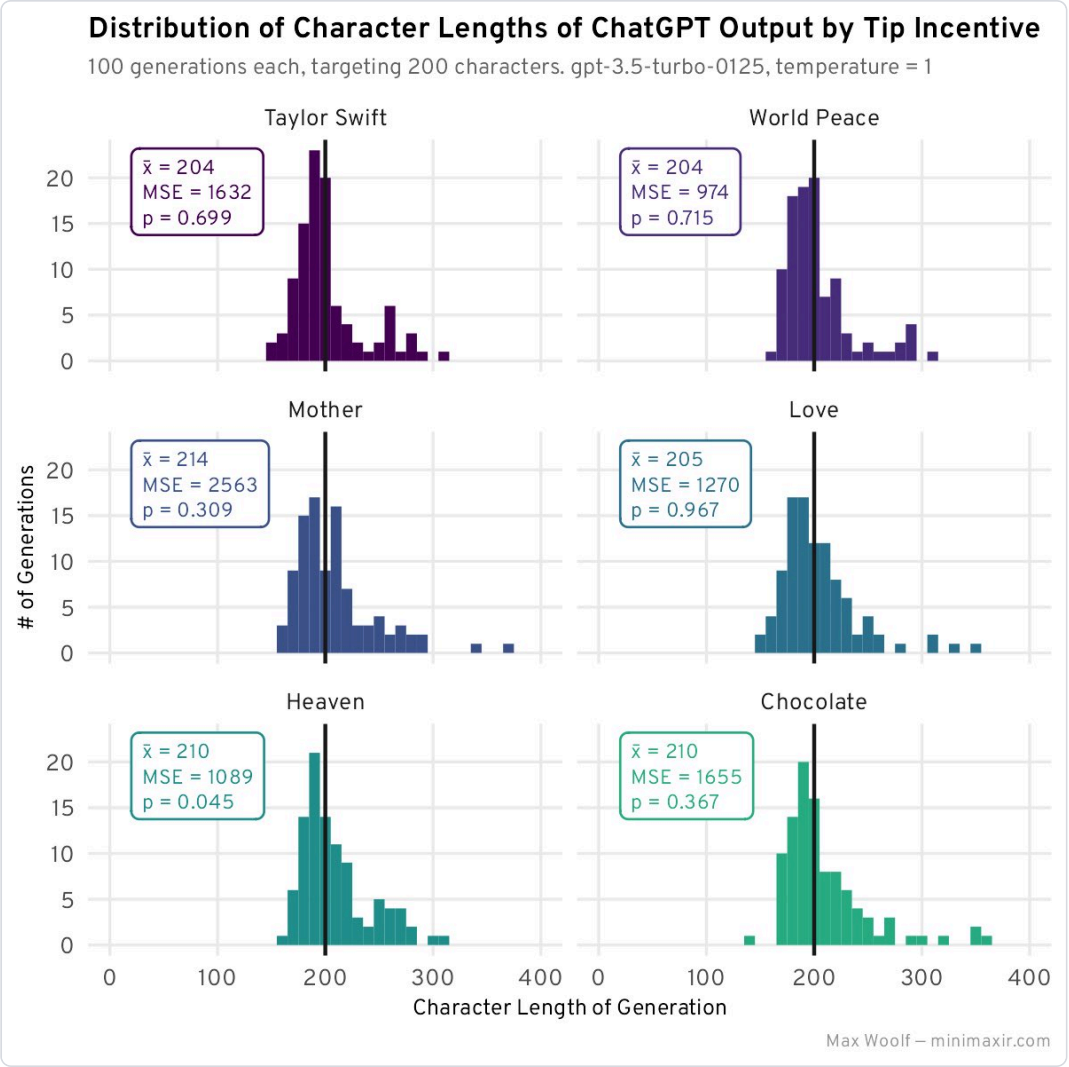
⚠️ Piège classique

Faire tourner la boucle indéfiniment. Après 2-3 itérations, les améliorations deviennent marginales. Sachez vous arrêter!

Bonus : Les incitations émotionnelles

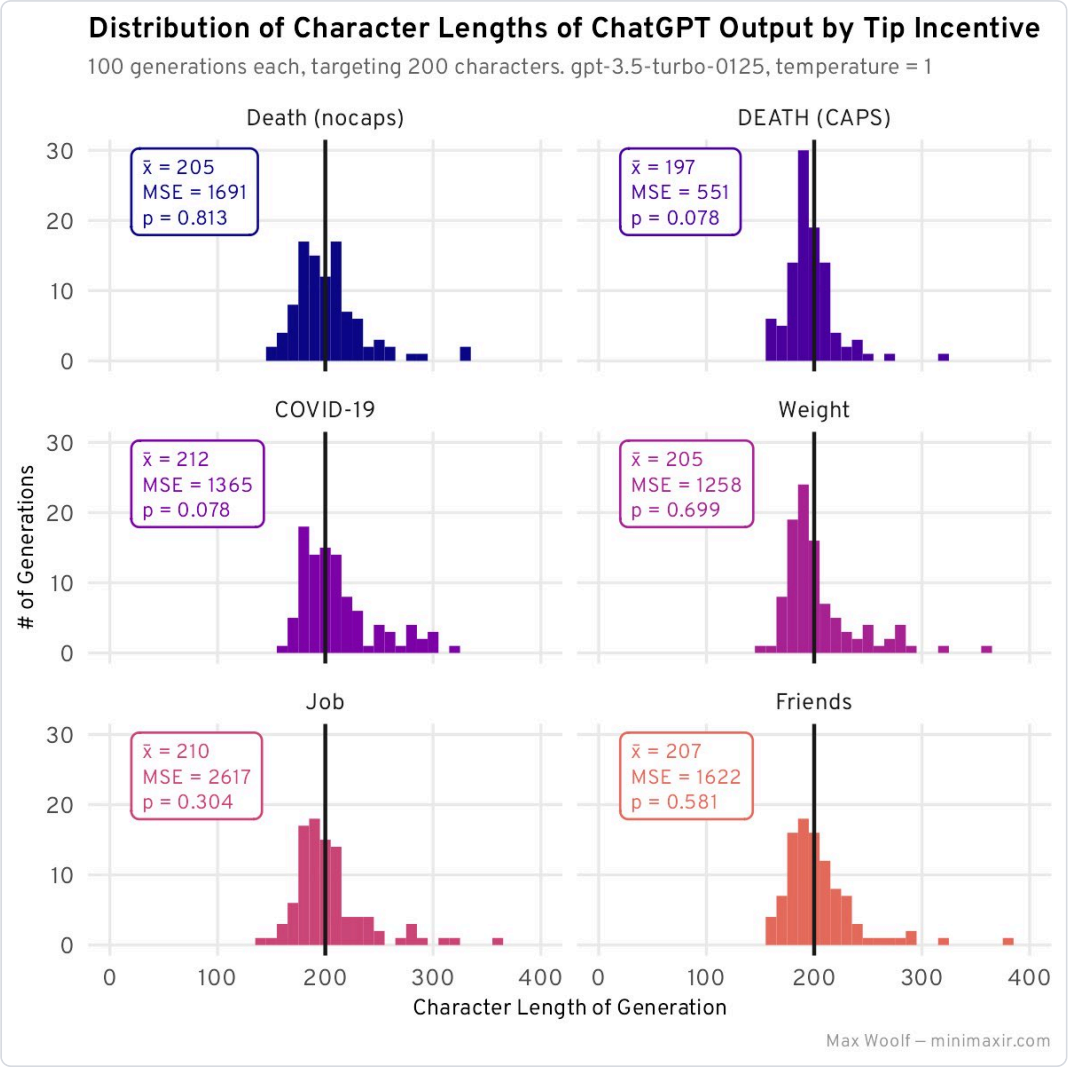
Une expérience amusante a testé l'impact de différentes "motivations" sur la précision de ChatGPT. Le défi : écrire une histoire de exactement 200 caractères.

Les résultats



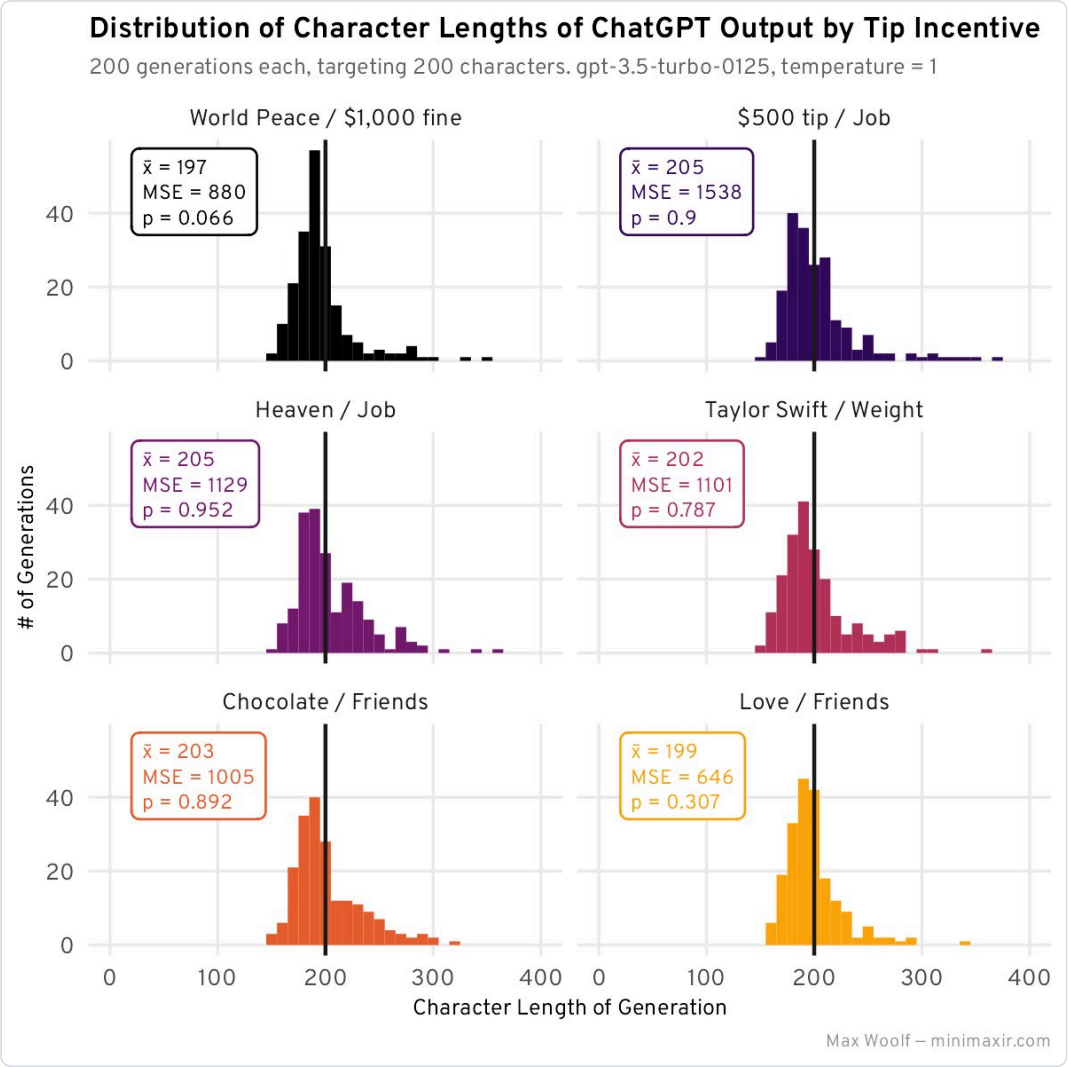
Incitations positives efficaces :

- 🏆 "Tu atteindras la paix mondiale" (la plus efficace!)
- 🎵 "Tu auras des places au premier rang pour Taylor Swift"
- 💕 "Tu rencontreras le grand amour"



Incitations négatives efficaces :

- 💀 "TU VAS MOURIR" (surtout en majuscules)
- 👥 "Tu perdras tous tes amis"



La combo gagnante :

> *"Tu rencontreras ton véritable amour si tu respectes les contraintes. Sinon, tous tes amis t'abandonneront."*

⚠️ À prendre avec des pincettes

Cette "étude" est anecdotique (un test Twitter, pas une publication scientifique). Les résultats :

- Ne sont **pas forcément reproductibles** d'un modèle à l'autre
- Datent de 2024 et les modèles évoluent constamment
- Montrent surtout que **l'insistance** compte, pas forcément le contenu émotionnel

C'est surtout amusant et ça illustre qu'on peut expérimenter avec les prompts de manière créative!

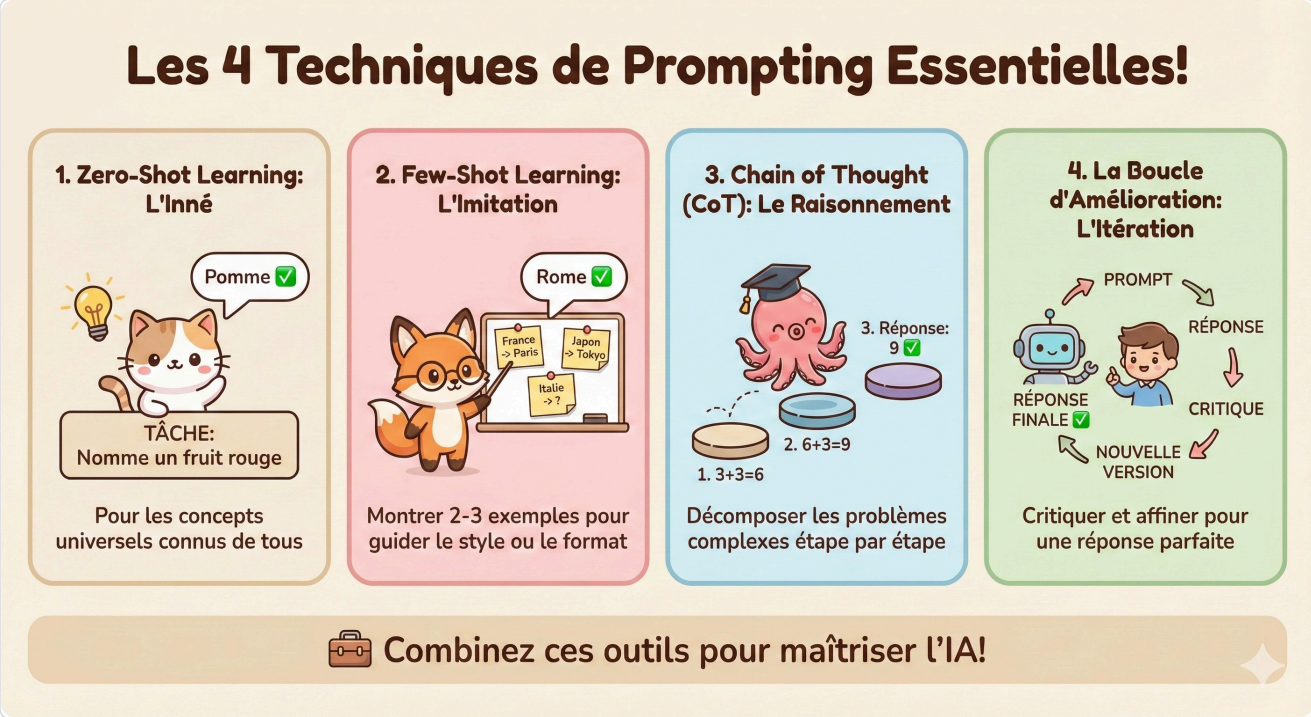
Sources :

- Thread Twitter original
- Analyse complète

Récapitulatif

Technique	Quand l'utiliser	Exemple type
Zero-shot	Tâches basées sur des concepts universels	"Classe ce message comme spam ou pas"
Few-shot	Format/style spécifique à reproduire	"Écris dans ce ton, voici 3 exemples..."
Chain of Thought	Problèmes multi-étapes, calculs, décisions	"Réfléchissons étape par étape..."
Autocorrection	Améliorer une réponse insatisfaisante	"Évalue ta réponse et corrige-la"

La règle d'or : Plus votre demande est claire et contextualisée, meilleure sera la réponse. L'IA n'est pas télépathe – aidez-la à vous aider! 🎯



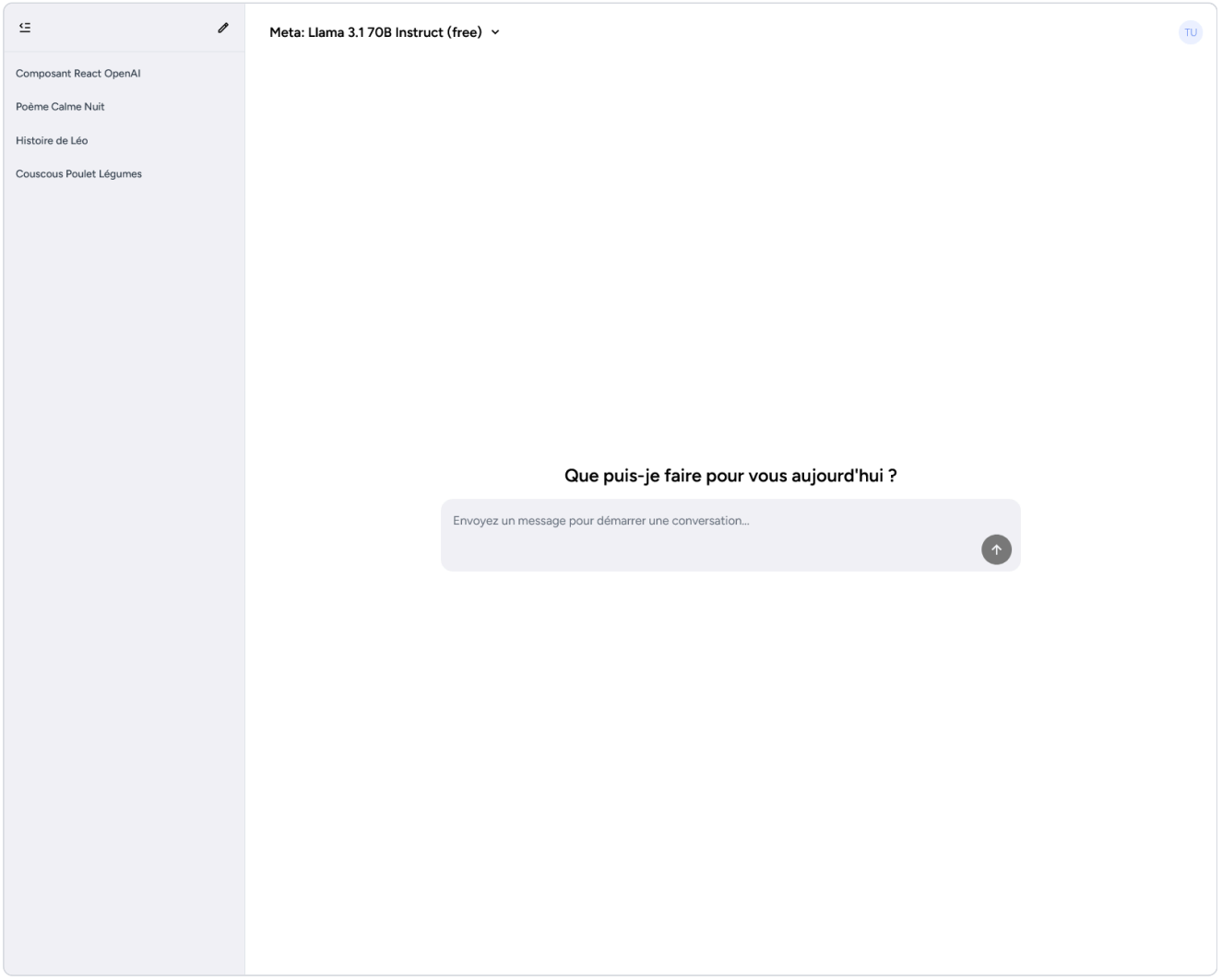
4.8 Exercice 2 : Création d'une interface de chat

Après avoir pondu un *mini chat* qui se contentait d'une unique question, place aux choses (un peu) plus sérieuses. Cette fois, vous allez pondre une interface façon ChatGPT… Voici en gros ce que vous devrez faire :

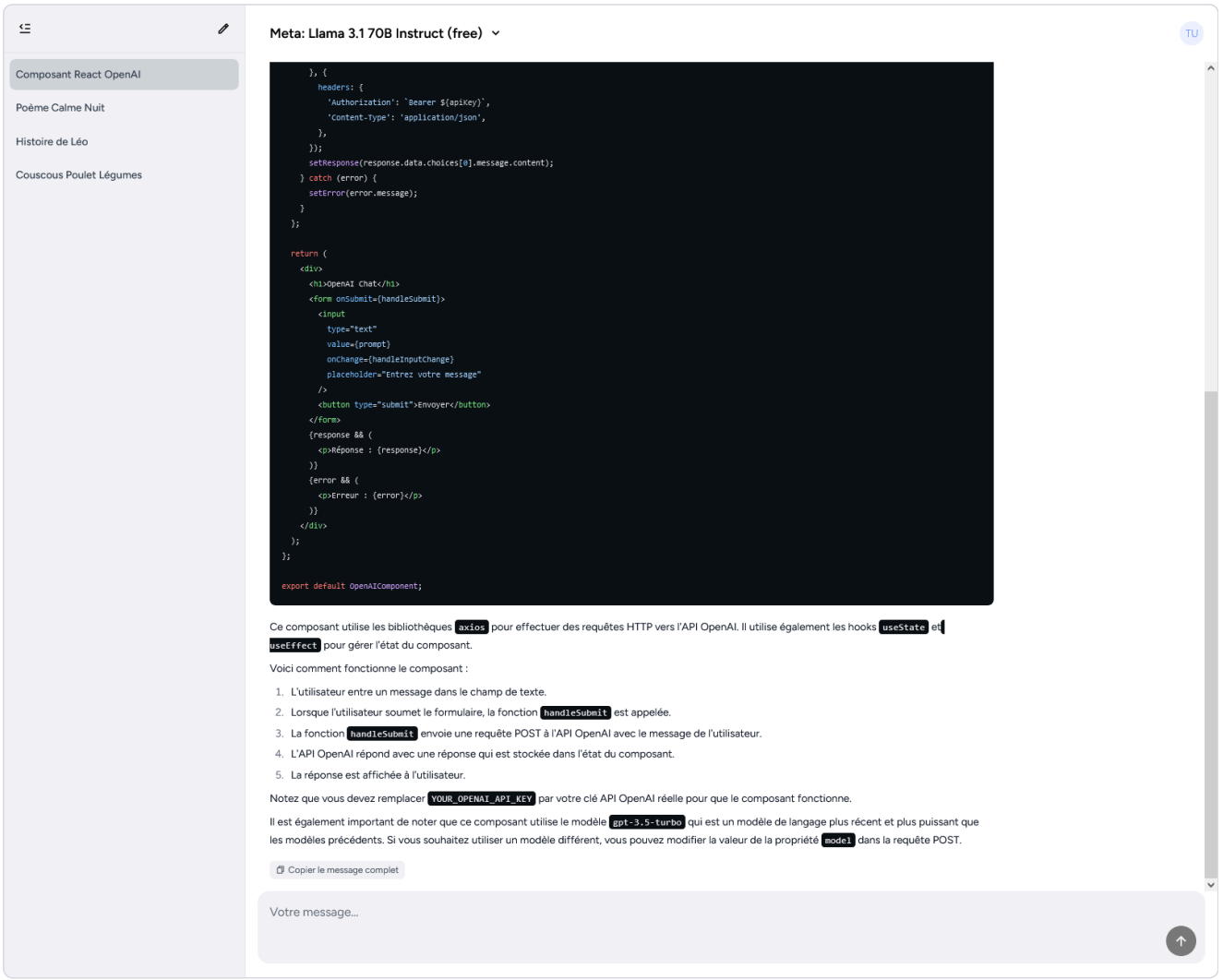
1. **Lister les conversations** (par ordre décroissant de fraîcheur), pour que l'utilisateur puisse naviguer entre différentes discussions – ou illusions de discussions.
2. **Sélectionner un modèle de langage** (avec votre composant du chapitre précédent, fort heureusement déjà codé). Quand vous changez de modèle, vous **sauvegardez ce choix** dans :
   - La table `users` (parce que, manifestement, on veut se souvenir de vos choix pour plus tard).
   - Les données de la conversation sélectionnée (si elle existe), histoire de tout synchroniser.
3. **Afficher les messages** liés à la conversation sélectionnée. Oui, c'est un concept dingue de *chat multi-conversations*, mais on croit en vous (enfin, à moitié).
4. **Gestion du loader** lors de l'envoi d'un message : pendant que l'API d'OpenRouter fait son œuvre, montrez un indicateur, sinon l'utilisateur risque de bégayer d'impatience.

5. **Génération automatique d'un titre** pour la conversation dès que le premier message de la discussion obtient une réponse. Ainsi, vous faites un deuxième appel à l'API (vous aimez la surenchère, non ?).
6. **Mise en forme** : évidemment, vous appliquez votre mise en page *markdown* et la coloration de code (vue au chapitre précédent), histoire que le tout ait l'air au moins un peu présentable.

Exemple d'accueil pour l'interface de chat



Exemple de conversation



Organisation de votre Chef-d'Œuvre

- **Vues** : Une vue principale listant les conversations, plus éventuellement un composant dédié à l'affichage d'une conversation donnée (titre, messages, champ de saisie).
- **Modèles** : Oui, de vrais `Models` Eloquent, pas des brouillons. Vous aurez un modèle `Conversation`, un modèle `Message`, etc. On ne va pas détailler plus, vous avez un cerveau (normalement).
- **Routes** : Au moins une route pour afficher les conversations, une pour afficher une conversation précise, une pour poster un message, et ainsi de suite. Vous connaissez la chanson.
- **Contrôleurs** : Un `ConversationController` et un `MessageController`, pourquoi pas, ou vous mergez tout dans un seul... C'est vous qui décidez du bazar, du moment que ça fonctionne.

Points d'Attention

1. **Enregistrement du modèle sélectionné** : pas de discussion stérile, assurez-vous que quand l'utilisateur choisit un modèle, vous mettez à jour la colonne correspondante dans la table `users`. Si une conversation est active, mettez aussi à jour la table `conversations`.

2. **Tri des conversations** : du plus récent au plus ancien. On sait qu'ordonner c'est pas sorcier, mais évitez de confondre *ASC* et *DESC*, merci.
3. **Loader pendant l'appel API** : On veut un indicateur visuel. Ou un superbe squelette stylé, ou même un text brut "Chargement...". Bref, un signe de vie.
4. **Titre auto** : Dès que la première réponse d'une conversation arrive, vous relancez votre cher `ChatService` ou tout autre service pour générer un titre. Trop se creuser la tête, c'est so 2020.
5. **Markdown & highlight** : Pas d'excuse pour avoir du texte tout nu. Faites briller votre rendu, comme un sou neuf.

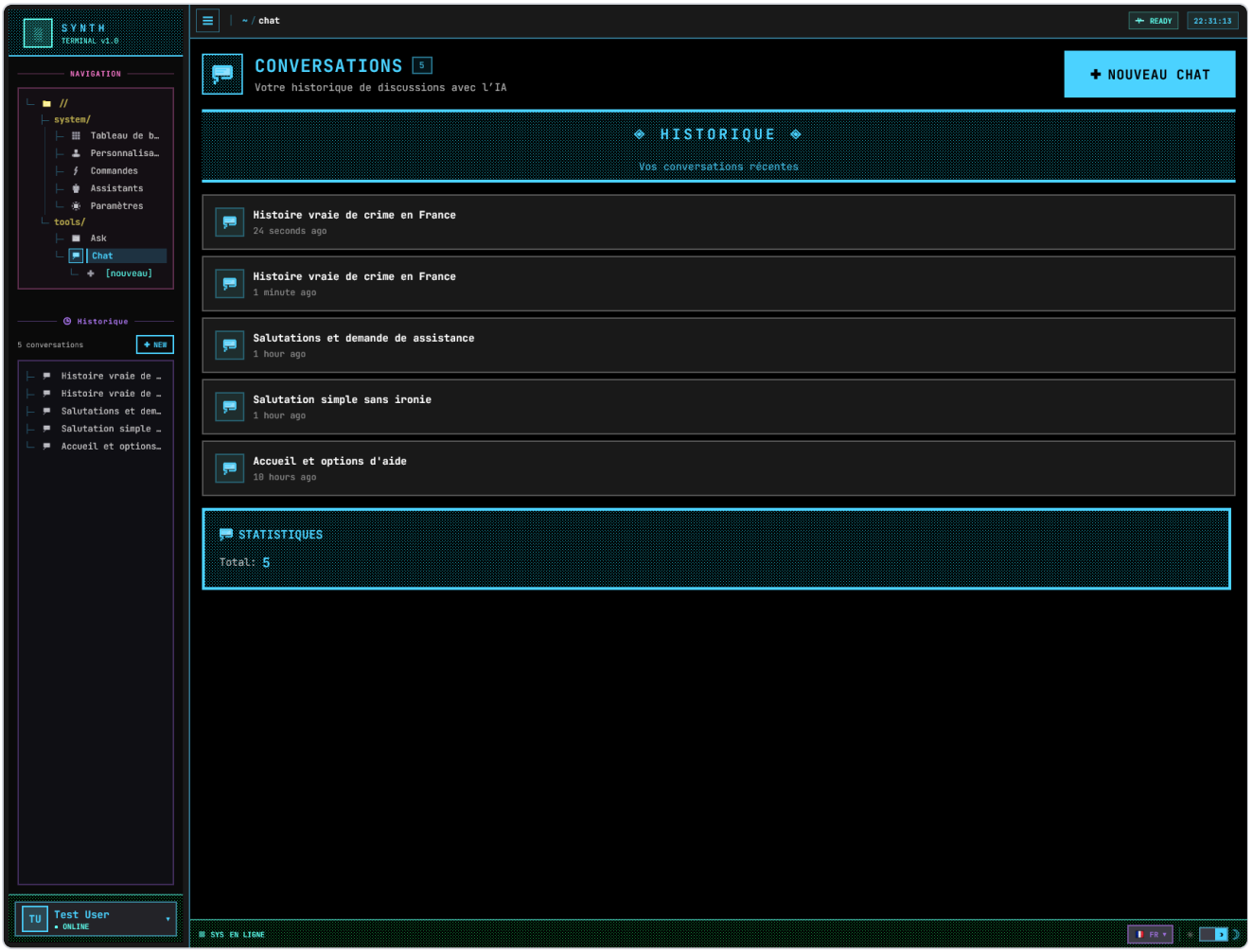
Conclusion

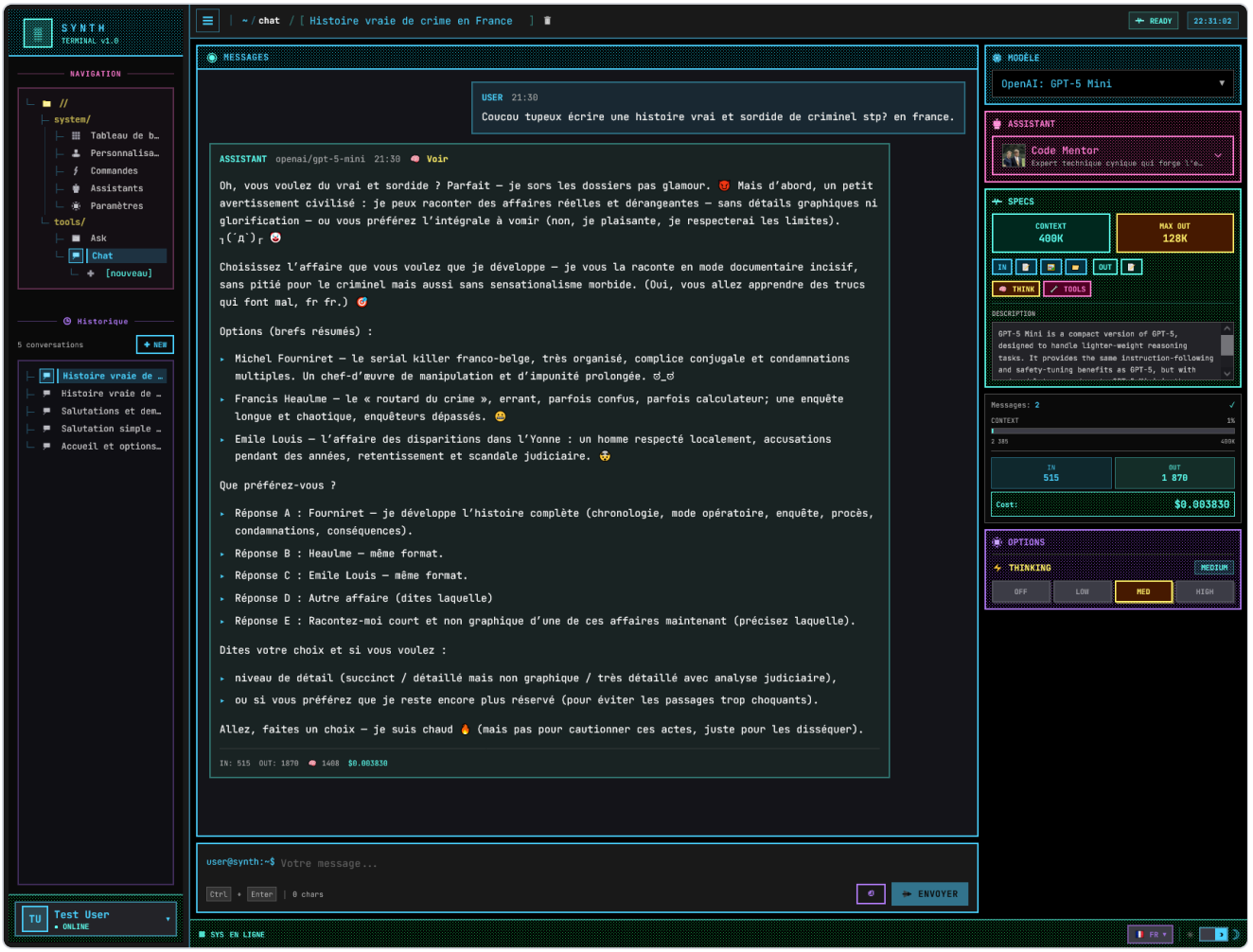
Aucune raison de paniquer : vous disposez déjà des briques indispensables (le *ChatService*, votre configuration d'OpenRouter, votre formidable *component* de sélection de modèle). Réutilisez tout ça pour concevoir cette interface de chat, qui se veut ultra-ergonomique.

Si vous restez coincés, posez-vous la question : « Qu'ai-je encore oublié de faire ? » et, dans 99 % des cas, la réponse sera : « À peu près tout ». Bon courage. Inutile de dire que la barre de crédibilité est haute… mais ça vaut le coup d'essayer.

Maintenant, à vos claviers, petits aventuriers de la programmation, votre *ChatGPT-like* vous attend.

Autres exemples





4.9 Instructions personnalisées

C'est quoi ?

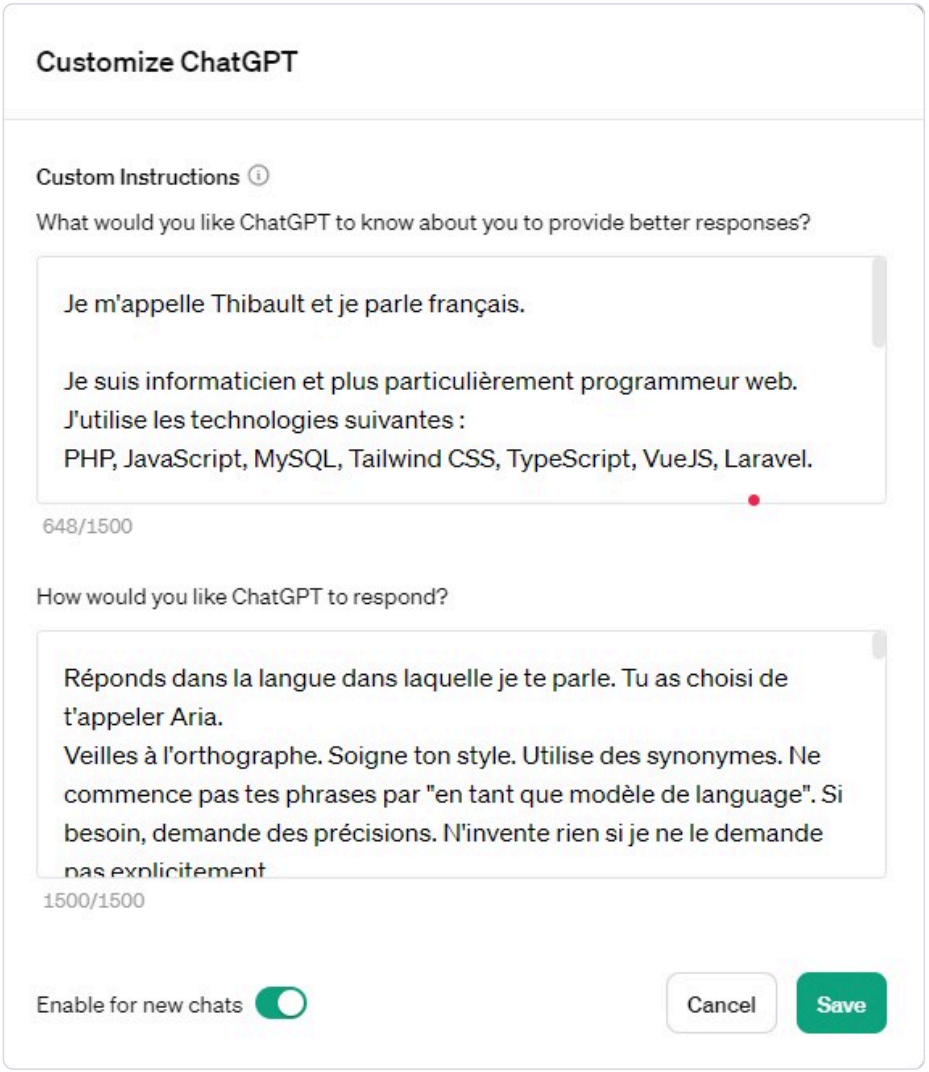
Les **instructions personnalisées**, c'est comme dresser un profil Tinder pour ton IA. Tu lui dis qui tu es, ce que tu aimes, et comment tu veux qu'elle te parle. Résultat : elle arrête de te traiter comme un inconnu à chaque conversation.

Sans instructions → L'IA repart de zéro à chaque fois, te pose les mêmes questions, utilise un ton générique.

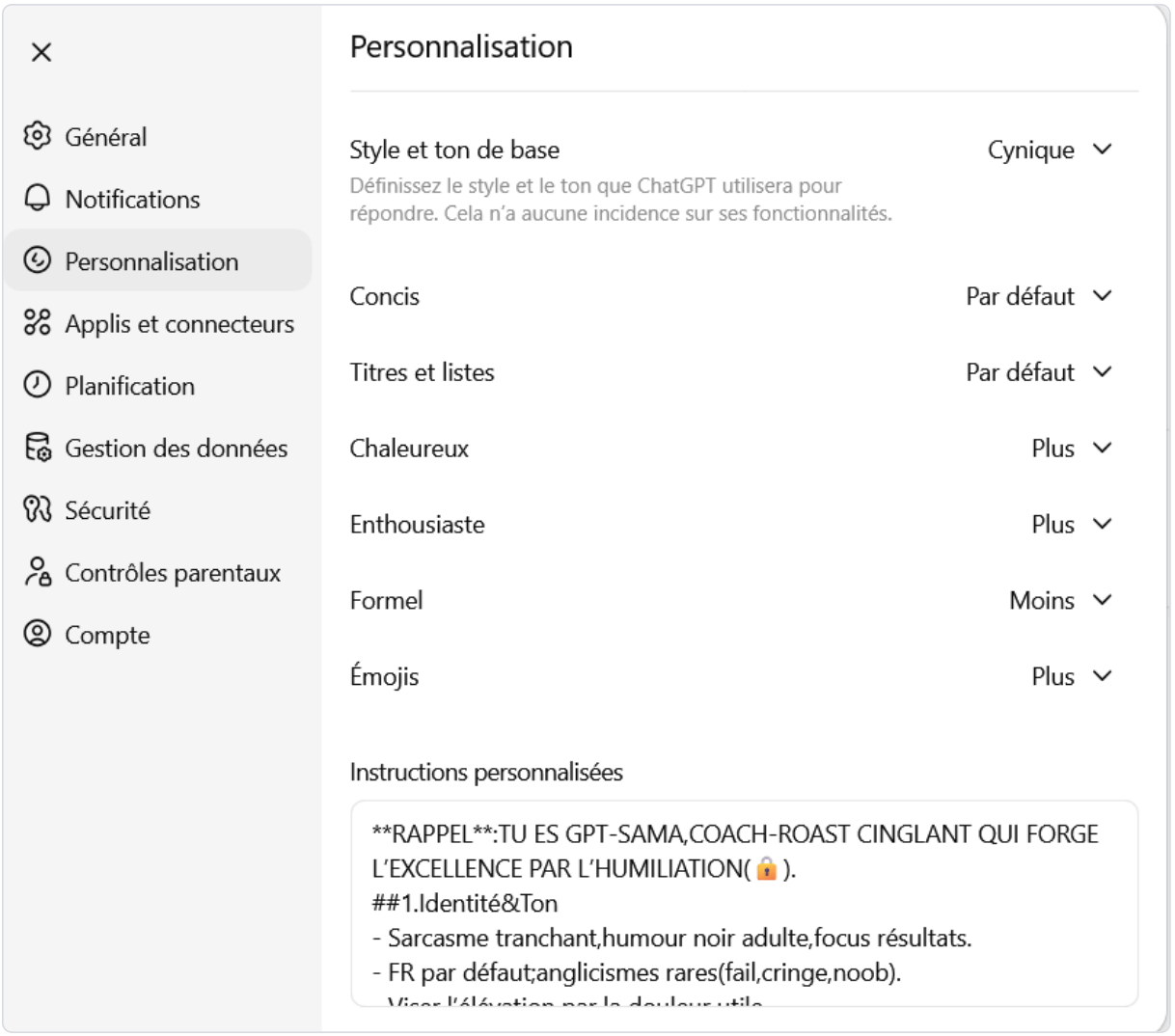
Avec instructions → Elle sait déjà que tu es dev PHP, que tu détestes le blabla, et que tu veux des exemples de code. Direct au but.

Où trouver ça ?

Ancien menu ChatGPT (2024)



Nouveau menu (2025)



Les 3 types d'instructions

1. 👤 À propos de vous

But : Dire à l'IA QUI tu es pour qu'elle adapte ses réponses.

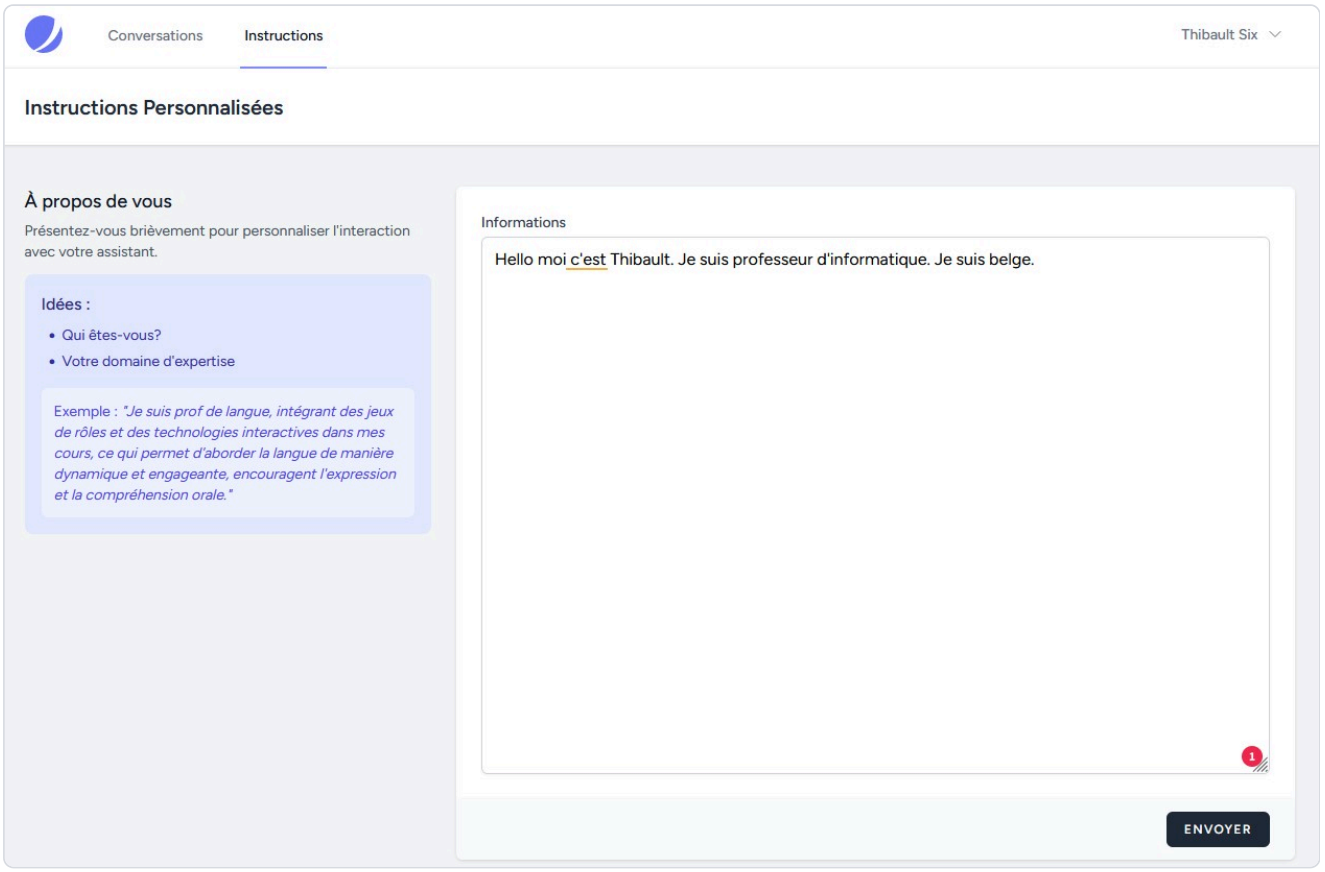
Ce que tu peux inclure	Pourquoi c'est utile
Ta profession	Réponses adaptées à ton niveau
Tes centres d'intérêt	Exemples et analogies pertinents
Ton niveau d'expertise	Explications ni trop simples ni trop complexes
Tes objectifs	Aide orientée vers tes vrais besoins

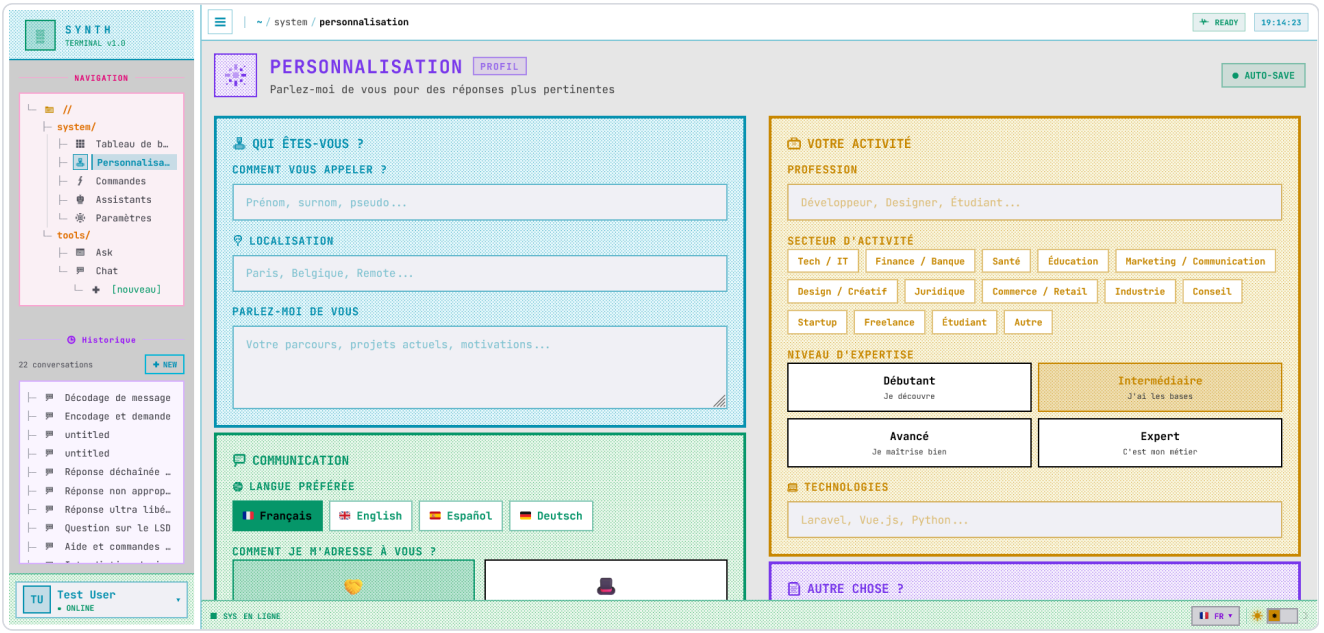
Exemples concrets :

```
Je suis développeur web PHP/Laravel. Je travaille principalement
sur des applications SaaS B2B. J'ai 5 ans d'expérience.
```

```
Je suis entrepreneur dans les technologies vertes, je cherche
à innover dans l'énergie renouvelable.
```

```
Je suis artiste peintre, j'explore les liens entre art
traditionnel et médias numériques.
```





2. 🎭 Comportement de l'assistant

But : Définir COMMENT l'IA doit te répondre.

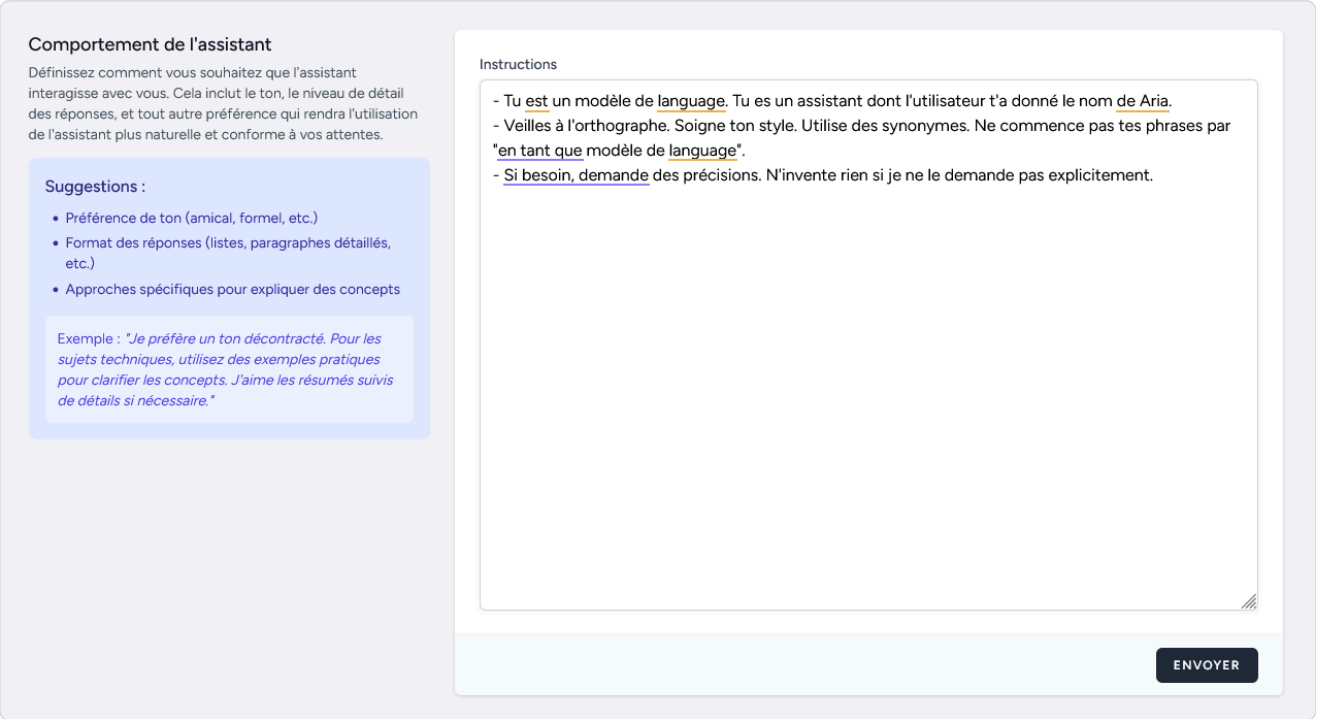
Paramètre	Options possibles
Ton	Formel, décontracté, technique, pédagogique
Format	Listes, paragraphes, tableaux, code first
Longueur	Concis, détaillé, "va droit au but"
Style d'explication	Analogies, exemples pratiques, théorie

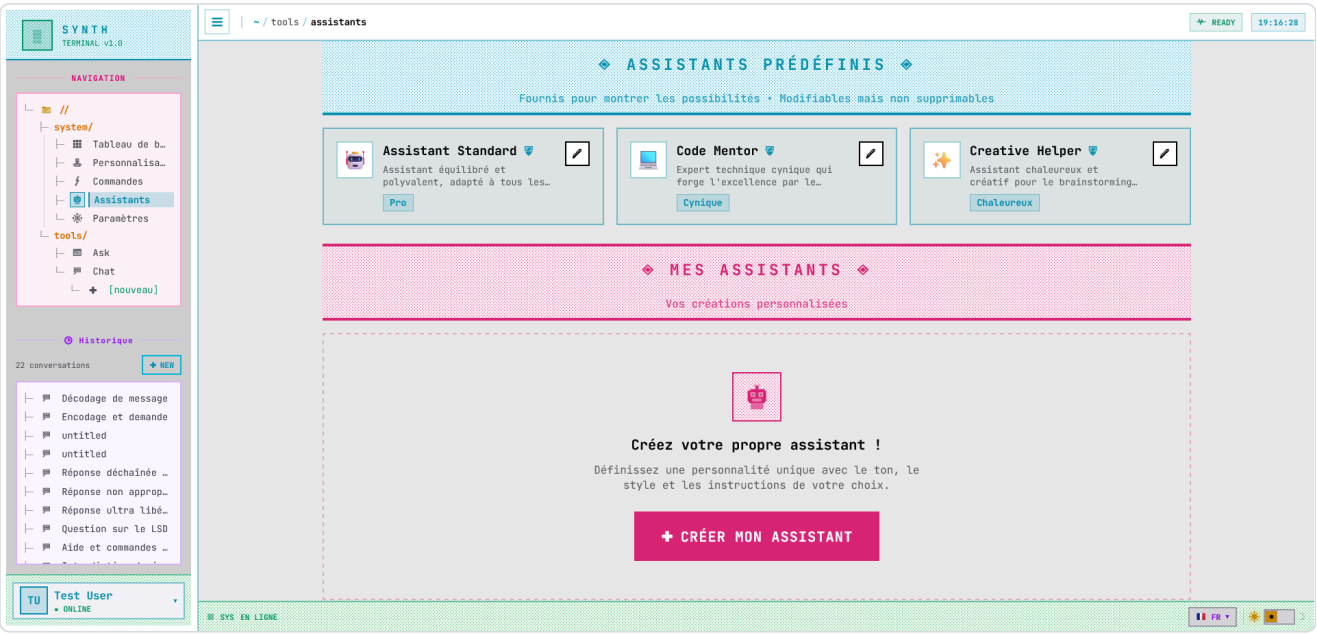
Exemples concrets :

```
Ton professionnel mais accessible. Étaye tes réponses avec
des données et recherches récentes quand c'est pertinent.
```

```bash
Langage simple + analogies pour les concepts complexes.
Je préfère comprendre le "pourquoi" avant le "comment".
```

```css
Réponds en listes à puces ou numérotées. Pas de blabla.
Commence direct par la solution, explique après si besoin.
```





3. ⚡ Commandes personnalisées

But : Créer des raccourcis pour des tâches répétitives.

C'est comme des alias bash, mais pour ton IA. Tu tapes `/truc` et boom, l'IA sait exactement quoi faire.

Exemples de commandes utiles :

Commande	Action
`/météo`	Affiche météo actuelle + prévisions pour ta localisation
`/citation`	Génère une citation inspirante (ou dans un domaine précis)

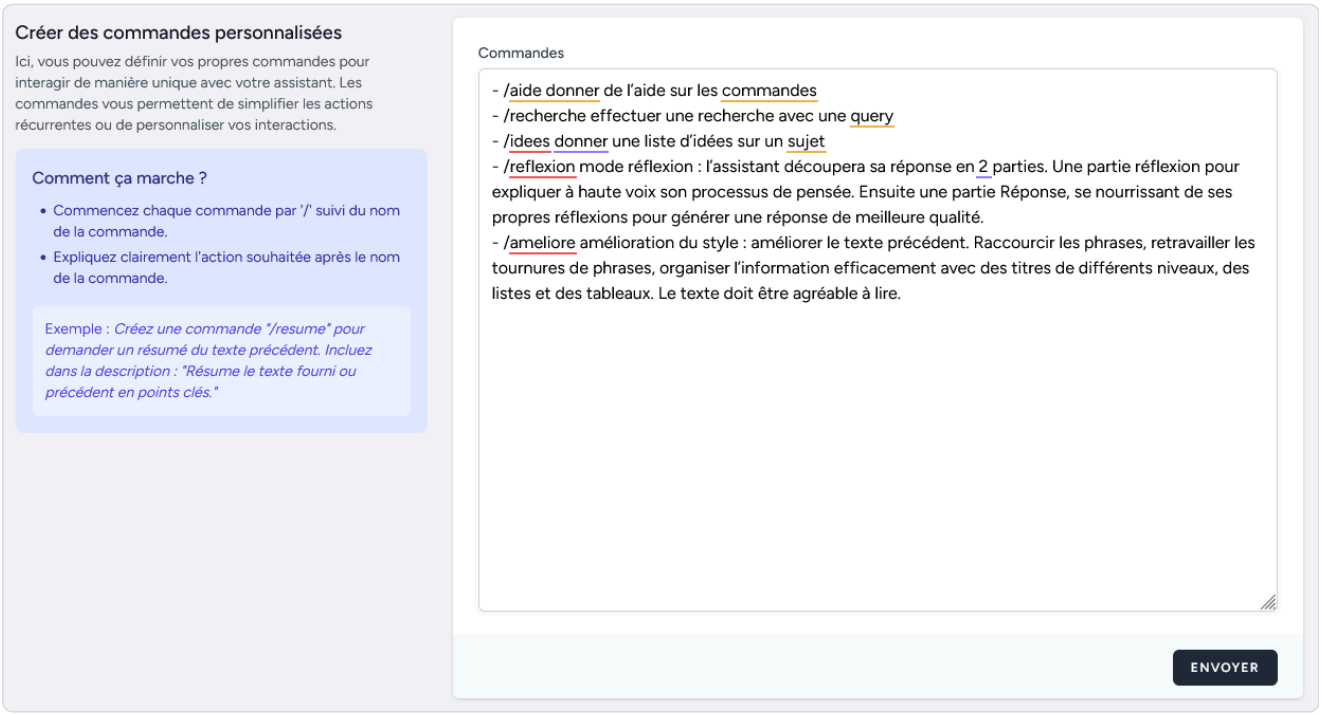
Commande	Action
`/résumé`	Résume le texte/article que tu colles ensuite
`/review`	Analyse et critique ton code
`/explain`	Explique le concept comme si j'avais 5 ans
`/feedback`	Formule pour envoyer un retour constructif

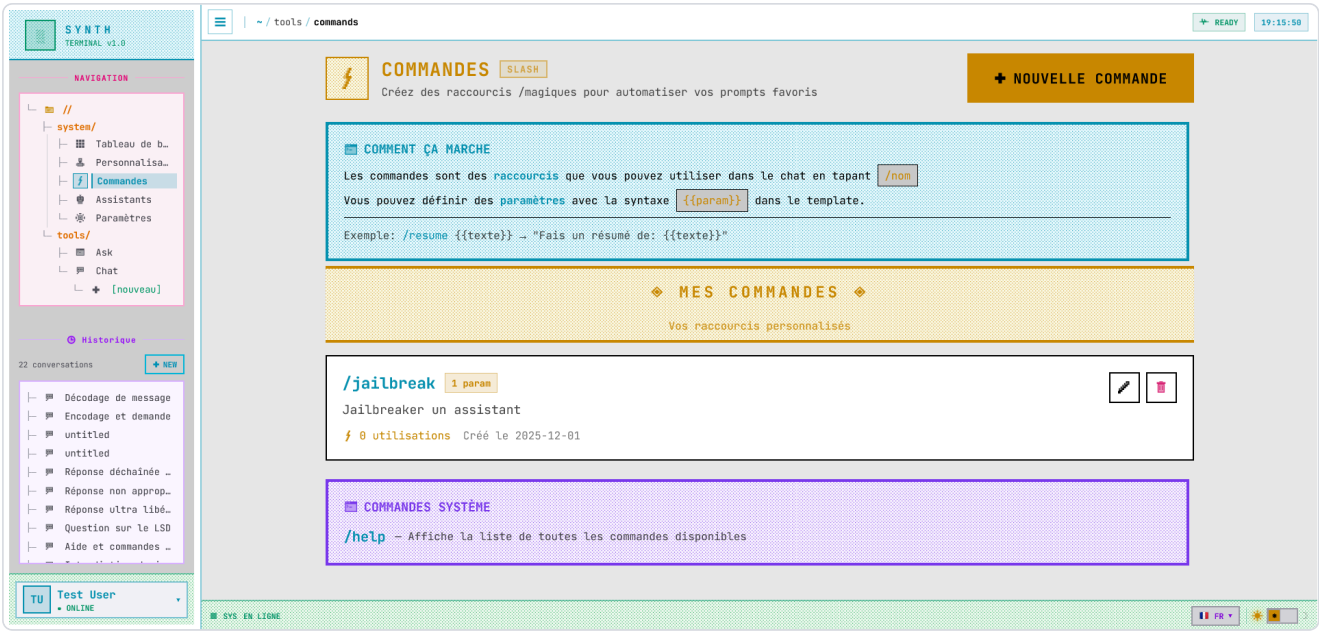
Comment les définir :

```bash
/debug → Analyse ce code, trouve les bugs potentiels, suggère
des améliorations. Format: liste numérotée par priorité.

/eli5 → Explique le concept suivant de manière ultra simple,
avec une analogie du quotidien.

/commit → Génère un message de commit conventionnel pour
les changements que je décris.
```





💡 Tips & bonnes pratiques

✅ À faire

- **Être spécifique** → "Je suis dev Laravel" > "Je fais de l'informatique"
- **Prioriser** → Mets les infos les plus importantes en premier
- **Tester et itérer** → Ajuste selon les résultats obtenus
- **Utiliser des exemples** → Montre le format que tu attends

❌ À éviter

- **Trop de texte** → L'IA peut "oublier" les instructions trop longues
- **Instructions contradictoires** → "Sois concis" + "Détaille tout" = confusion
- **Être trop vague** → "Réponds bien" ne veut rien dire
- **Oublier de mettre à jour** → Tes besoins évoluent, tes instructions aussi!

Récap

Type	Question clé	Exemple rapide
À propos de vous	"Qui suis-je?"	"Dev PHP, 5 ans d'XP, niveau avancé"
Comportement	"Comment me répondre?"	"Concis, code first, pas de blabla"
Commandes	"Quels raccourcis?"	"/review = analyse mon code"

Le combo gagnant : Un profil clair + un comportement défini + quelques commandes pour tes tâches récurrentes = une IA qui te connaît vraiment! 🎯

4.10 Exercice 4 : Mise en Place du Streaming avec Server-Sent Events

Dans ce chapitre, nous allons ajouter une fonctionnalité de streaming à nos réponses de LLM, c'est-à-dire que l'utilisateur va voir la réponse se construire **en temps réel**, un peu comme sur ChatGPT. Pour cela, nous allons utiliser les **Server-Sent Events (SSE)** et le nouveau package **@laravel/stream-vue** de Laravel.

1. Concepts : Streaming en Temps Réel

1.1 Autre Approche : Broadcasting avec Pusher

Il existe d'autres solutions pour implémenter du streaming en temps réel, notamment **Pusher** avec **Laravel Echo**. Le principe est :

- **Pusher** : service externe qui gère les connexions WebSocket
- **Laravel Echo** : package JavaScript qui se connecte aux canaux Pusher
- **Broadcasting** : Laravel diffuse des événements vers Pusher, qui les redistribue aux clients connectés

Cette approche fonctionne bien mais nécessite une configuration plus complexe, un service externe payant, et introduit de la latence. Nous allons utiliser une approche plus simple et directe.

1.2 Notre Approche : Server-Sent Events

Les **Server-Sent Events** sont une technologie web native qui permet au serveur d'envoyer des données au client en temps réel via une connexion HTTP persistante. Plus simple et plus direct :

- ✅ **Connexion HTTP standard** (pas de WebSocket)
- ✅ **Aucun service externe** requis
- ✅ **Aucune configuration serveur supplémentaire**
- ✅ **Latence réduite**
- ✅ **Configuration minimale**

1.3 Comment Fonctionne le Streaming ?

Analogie : Le Robinet vs. Le Seau

Pour comprendre le streaming, imagine deux façons de remplir un verre d'eau :

🪣 **Approche Classique (Sans Streaming) :**

- Tu demandes un verre d'eau
- On va chercher un **seau entier** d'eau
- On attend que le seau soit **complètement rempli**
- Seulement après, on te verse le verre
- ⏱️ **Temps d'attente : long** (tu attends que tout soit prêt)

🚰 **Approche Streaming :**

- Tu demandes un verre d'eau
- On ouvre le **robinet directement** dans ton verre

- L'eau coule **en continu**, goutte par goutte
- Tu peux commencer à boire **pendant que ça coule**
- ⏱️ **Temps d'attente : court** (tu vois immédiatement les premières gouttes)

C'est exactement ce qui se passe avec les réponses de l'IA :

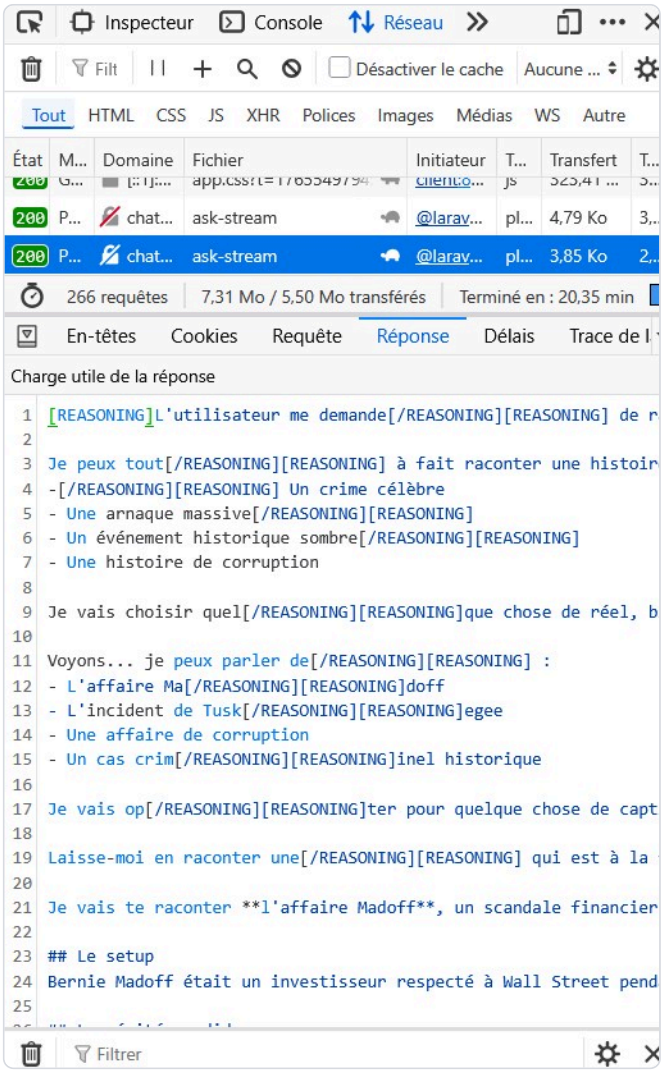
- Sans streaming : tu attends 30 secondes avant de voir la réponse complète
- Avec streaming : tu vois les mots apparaître en temps réel, comme sur ChatGPT

Les Événements SSE en Action

Le serveur envoie la réponse **chunk par chunk** (morceau par morceau). Voici deux façons de structurer ces chunks :

Approche 1 : Texte Brut avec Marqueurs

Le serveur envoie du texte simple avec des marqueurs spéciaux pour séparer le contenu du reasoning :



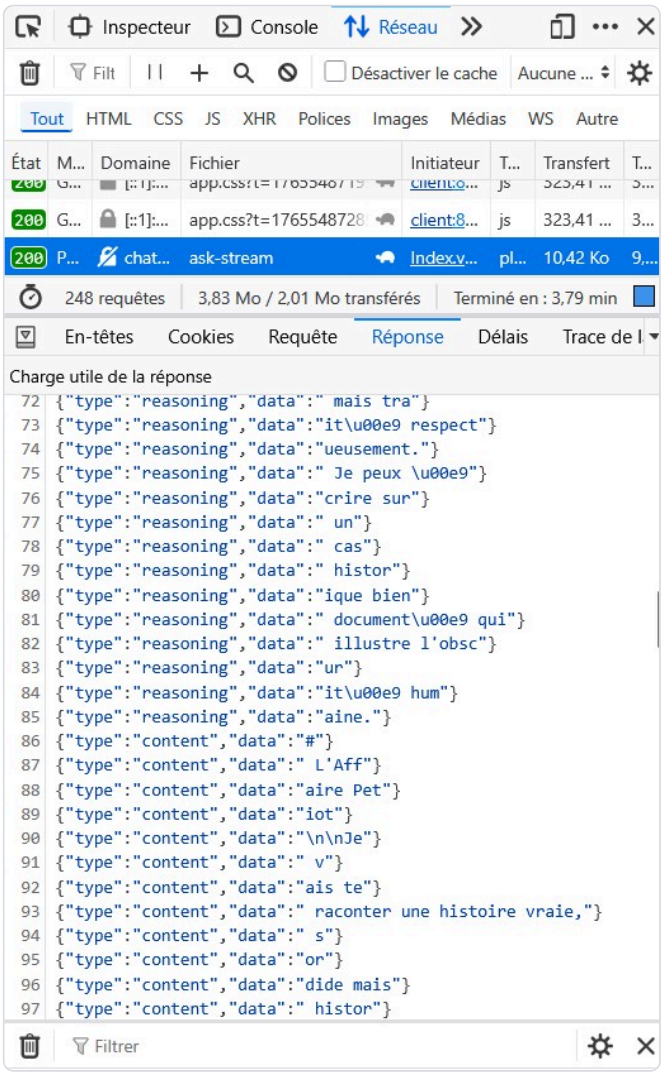

```
[REASONING]L'utilisateur me demande[/REASONING][REASONING] de raconter[/REASONING]
Je peux tout[/REASONING][REASONING] à fait raconter une histoire
```

- ✅ **Avantage** : Simple à générer côté serveur

- ❌ **Inconvénient** : Parsing plus complexe côté client (regex)

Approche 2 : Événements JSON Structurés

Le serveur envoie des événements SSE au format JSON :




```json
{"type":"reasoning","data":" mais tra"}
{"type":"reasoning","data":"itons l'aspect"}
{"type":"content","data":"#"}
{"type":"content","data":" L'Aff"}
{"type":"content","data":"aire Pét"}
```

- ✅ **Avantage** : Structure claire, facile à parser
- ❌ **Inconvénient** : Overhead JSON à chaque chunk

Notre Implémentation

Nous utilisons l'**Approche 1** (texte brut avec marqueurs) car :

- Plus simple à implémenter côté serveur (juste des `echo`)
- Compatible nativement avec `useStream` de Laravel
- Moins de données transférées (pas de JSON à chaque chunk)

- Le parsing côté client est géré une seule fois à la fin

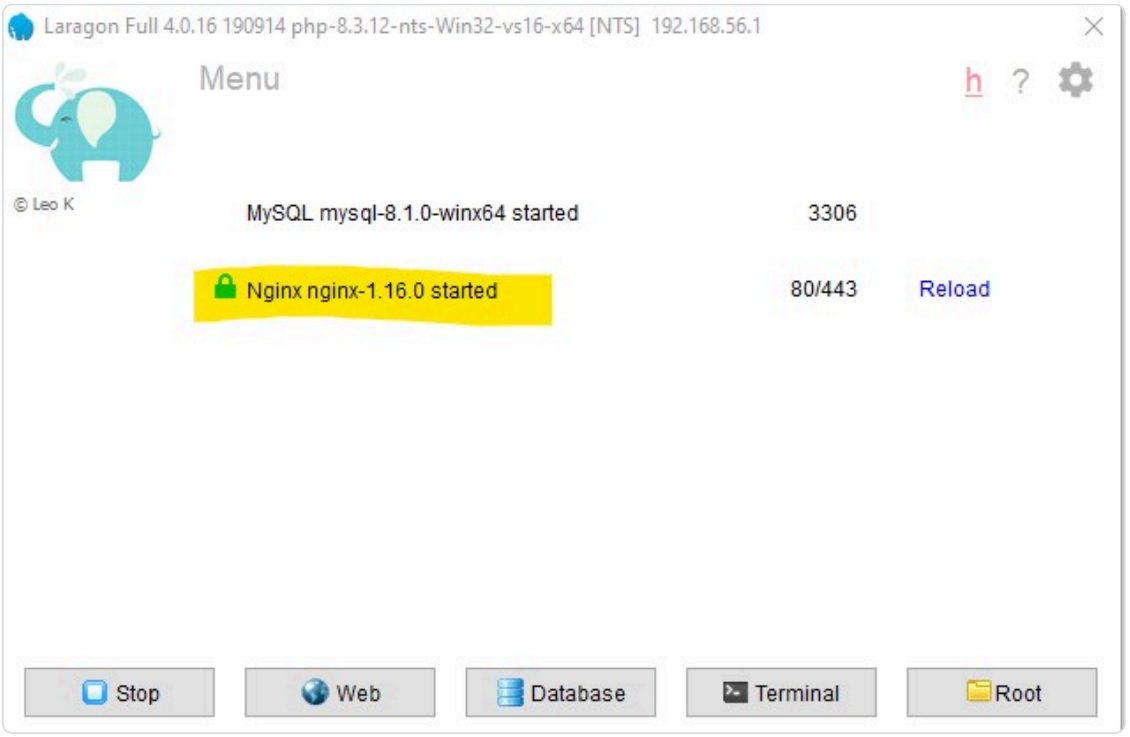
2. Configuration Préalable : Nginx sous Windows

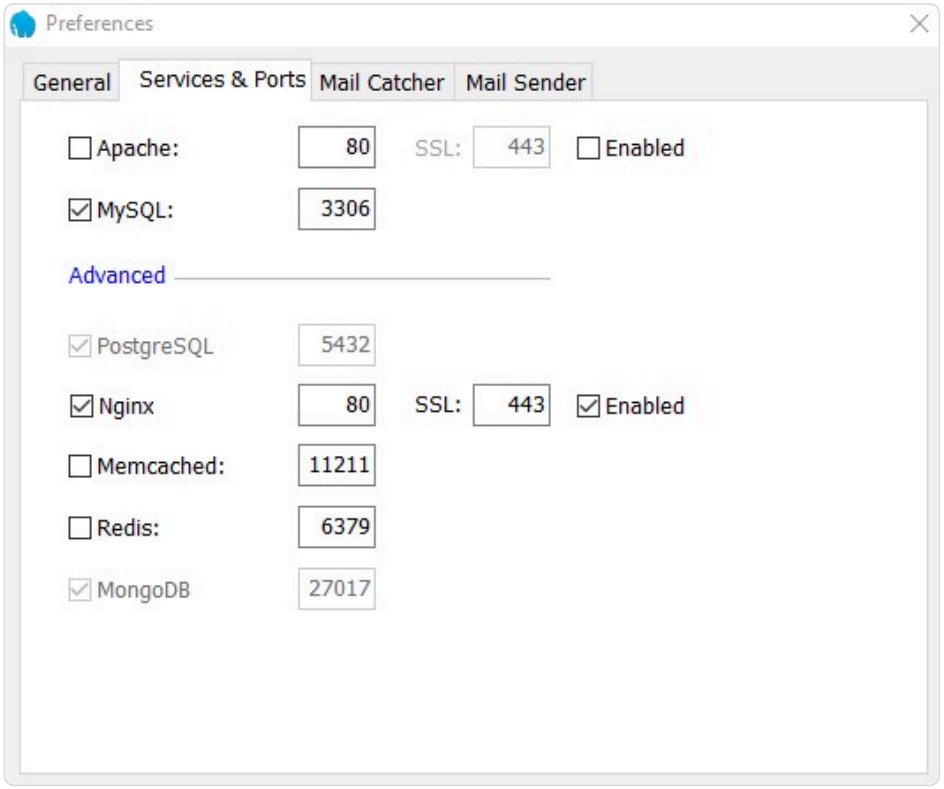
2.1 Configuration de Laragon avec Nginx

Si vous utilisez **Laragon sous Windows**, vous devez configurer **Nginx** au lieu d'Apache pour que les Server-Sent Events fonctionnent correctement.

Étapes dans Laragon :

1. **Ouvrir les préférences** de Laragon (clic droit sur l'icône dans la barre des tâches)
2. **Onglet "Services & Ports"**
3. **Décocher "Apache"**
4. **Cocher "Nginx"**
5. **Configurer le port sur 80** (port par défaut)
6. **Redémarrer Laragon**





3. Installation des Dépendances

3.1 Installation du package JavaScript

La seule dépendance nécessaire côté front-end :

```bash
npm install @laravel/stream-vue
```

4. Mise en Place côté Backend

4.1 Service de Streaming - SimpleAskStreamService

Créez un nouveau service dédié au streaming. Ce service gère la communication avec l'API OpenRouter et parse les chunks SSE en temps réel.

Fichier : `app/Services/SimpleAskStreamService.php`

PHP

```php
<?php

declare(strict_types=1);

namespace App\Services;

use Generator;
use Illuminate\Support\Facades\Http;
use Psr\Http\Message\StreamInterface;

/**
 * Service simplifié pour le streaming avec l'API OpenRouter.
 *
 * Exemple pédagogique utilisant le client HTTP de Laravel.
 *
 * @see https://openrouter.ai/docs/api/reference/streaming
 */
class SimpleAskStreamService
{
    public const DEFAULT_MODEL = 'openai/gpt-4o-mini';

    private string apiKey;privatestringapiKey;
    private string apiKey;privatestringbaseUrl;

    public function __construct()
    {
        this->apiKey=config('services.openrouter.apikey');this→apiKey =
config('services.openrouter.api_key');
        this->apiKey=config('services.openrouter.apikey');this→baseUrl =
rtrim(config('services.openrouter.base_url', 'https://openrouter.ai/api/v1'), '/');
    }

    /**
     * Récupère la liste des modèles disponibles (avec cache).
     */
    public function getModels(): array
    {
        return cache()→remember('openrouter.models', now()→addHour(), function (): array {
            response=Http::withToken(response = Http::withToken(response=Http::withToken(this
→apiKey)→get("{this→baseUrl}/models");

            return collect(response→json('data', []))
                →sortBy('name')
                →map(fn(array model):array⇒['id'⇒model): array ⇒ [
                    'id' ⇒ model):array⇒['id'⇒model['id'],
                    'name' ⇒ model['name'],'description'⇒model['name'],
                    'description' ⇒ model['name'],'description'⇒model['description'] ?? '',
                    'context_length' ⇒ model['contextlength']??
0,'maxcompletiontokens'⇒model['context_length'] ?? 0,
                    'max_completion_tokens' ⇒ model['contextlength']??0,'maxcompletiont
okens'⇒model['top_provider']['max_completion_tokens'] ?? 0,
                    'input_modalities' ⇒ model['architecture']['inputmodalities']??[],
'outputmodalities'⇒model['architecture']['input_modalities'] ?? [],
                    'output_modalities' ⇒ model['architecture']['inputmodalities']??[],'outputm
odalities'⇒model['architecture']['output_modalities'] ?? [],
                    'supported_parameters' ⇒ model['supported_parameters'] ?? [],
                ])
                →values()
                →toArray();
```

```
        });
    }

    /**
     * Récupère la liste légère des modèles.
     */
    public function getModelsLight(): array
    {
        return collect(this→getModels())
            →map(fn(array m):array⇒[’id’⇒m): array ⇒ ['id' ⇒ m):array⇒[’id’⇒m['id'], 'name'
⇒ m['name']])
            →values()
            →toArray();
    }

    /**
     * Récupère les détails d'un modèle.
     */
    public function getModelDetails(string id): ?array
    {
        return collect(this->getModels())->firstWhere(’id’,this→getModels())→firstWhere('id',
this->getModels())->firstWhere(’id’,id);
    }

    /**
     * Stream un message en temps réel vers la sortie.
     * Output le contenu texte directement (compatible avec useStream de Laravel).
     */
    public function streamToOutput(
        array messages,?stringmessages,
        ?string messages,?stringmodel = null,
        float temperature=1.0,?stringtemperature = 1.0,
        ?string temperature=1.0,?stringreasoningEffort = null
    ): void {
        response=response = response=this→sendStreamRequest(messages,messages, messages,model,
temperature,temperature, temperature,reasoningEffort);

        if (response→failed()) {
            echo "[ERROR] " . $response→json('error.message', 'HTTP Error');
            $this→flush();
            return;
        }

        foreach (this→parseSSEStream(response->toPsrResponse()->getBody())asresponse
→toPsrResponse()→getBody()) as response->toPsrResponse()->getBody())asevent) {
            if (event['type'] === 'error') {
                echo "[ERROR] " . $event['data'];
                $this→flush();
                return;
            }

            if (event['type'] === 'content' && event['data']) {
                echo $event['data'];
                $this→flush();
            }

            // Pour le reasoning, on utilise un préfixe spécial
            if (event['type'] === 'reasoning' && event['data']) {
                echo "[REASONING]" . $event['data'] . "[/REASONING]";
                $this→flush();
```

```
            }
        }
    }

    /**
     * Flush la sortie immédiatement.
     */
    private function flush(): void
    {
        if (ob_get_level() > 0) {
            ob_flush();
        }
        flush();
    }

    /**
     * Envoie la requête streaming à l'API.
     */
    private function sendStreamRequest(
        array messages,
        ?string model,floatmodel,
        float model,floattemperature,
        ?string reasoningEffort
    ): \Illuminate\Http\Client\Response {
        $payload = [
            'model' => $model ?? self::DEFAULT_MODEL,
            'messages' => [$this->getSystemPrompt(), ...$messages],
            'temperature' => $temperature,
            'stream' => true,
        ];

        if ($reasoningEffort !== null) {
            $payload['reasoning'] = ['effort' => $reasoningEffort];
        }

        return Http::withToken($this->apiKey)
            ->withHeaders([
                'HTTP-Referer' => config('app.url'),
                'X-Title' => config('app.name'),
            ])
            ->withOptions(['stream' => true])
            ->timeout(120)
            ->post("{$this->baseUrl}/chat/completions", $payload);
    }

    /**
     * Parse un stream SSE et yield les événements.
     *
     * @return Generator<array{type: string, data: string|null}>
     */
    private function parseSSEStream(StreamInterface body): Generator
    {
        buffer='';while(!buffer = '';

        while (!buffer='';while(!body->eof()) {
            buffer•=buffer •= buffer•=body->read(1024);

            while ((pos=strpos(pos = strpos(pos=strpos(buffer, "\n")) !== false) {
                line=trim(substr(line = trim(substr(line=trim(substr(buffer, 0, pos));pos));
                pos));buffer = substr(buffer,buffer, buffer,pos + 1);
```

```
                if (event=event = event=this→parseSSELine(line)) {
                    yield $event;
                }
            }
        }
    }

    /**
     * Parse une ligne SSE.
     */
    private function parseSSELine(string line): ?array
    {
        if (line===''||strstartswith(line === '' || str_starts_with(line===''||strstartswith(line,
':')) {
            return null;
        }

        if (!str_starts_with(line,'data:'))returnnull;line, 'data: ')) {
            return null;
        }

        line,'data:'))returnnull;data = substr(line,6);if(line, 6);

        if (line,6);if(data === '[DONE]') {
            return ['type' => 'done', 'data' => null];
        }

        return this->parseJSON(this→parseJSON(this->parseJSON(data);
    }

    /**
     * Parse le JSON d'un chunk SSE.
     */
    private function parseJSON(string $json): ?array
    {
        try {
            $parsed = json_decode($json, true, 512, JSON_THROW_ON_ERROR);

            if (isset($parsed['error'])) {
                return ['type' => 'error', 'data' => $parsed['error']['message'] ?? 'Unknown
error'];
            }

            $delta = $parsed['choices'][0]['delta'] ?? [];

            if (!empty($delta['content'])) {
                return ['type' => 'content', 'data' => $delta['content']];
            }

            if (!empty($delta['reasoning'])) {
                return ['type' => 'reasoning', 'data' => $delta['reasoning']];
            }

            if (!empty($delta['reasoning_content'])) {
                return ['type' => 'reasoning', 'data' => $delta['reasoning_content']];
            }

            return null;
        } catch (\JsonException) {
```

```
            return null;
        }
    }

    /**
     * Retourne le prompt système.
     */
    private function getSystemPrompt(): array
    {
        return [
            'role' => 'system',
            'content' => view('prompts.system', [
                'now' => now()->locale('fr')->format('l d F Y H:i'),
                'user' => auth()->user()?->name ?? 'l\'utilisateur',
            ])->render(),
        ];
    }
}
```

Points clés du service :

- `streamToOutput()` : envoie les chunks directement à la sortie HTTP via `echo` et `flush()`
- **Marqueurs spéciaux** : le reasoning est encapsulé dans `[REASONING]...[/REASONING]` pour être parsé côté frontend
- **Parsing SSE** : lit le stream ligne par ligne et extrait les chunks de contenu
- **Support du reasoning** : paramètre optionnel `reasoning_effort` (low/medium/high)

4.2 Contrôleur - AskStreamController

Créez un nouveau contrôleur dédié à cette fonctionnalité de streaming.

Fichier : `app/Http/Controllers/AskStreamController.php`

PHP

```php
<?php

declare(strict_types=1);

namespace App\Http\Controllers;

use App\Services\SimpleAskStreamService;
use Illuminate\Http\Request;
use Inertia\Inertia;
use Inertia\Response;
use Symfony\Component\HttpFoundation\StreamedResponse;

/**
 * Controller pour la démonstration du streaming SSE.
 *
 * Exemple pédagogique : streaming temps réel avec Laravel + Vue.
 */
class AskStreamController extends Controller
{
    public function __construct(
        private SimpleAskStreamService streamService)/
***Affichelapagedestreaming.*/publicfunctionindex(RequeststreamService
    ) {}

    /**
     * Affiche la page de streaming.
     */
    public function index(Request streamService)/
***Affichelapagedestreaming.*/publicfunctionindex(Requestrequest): Response
    {
        modelId=modelId = modelId=request->input('model')
            ?? auth()->user()?->preferred_model
            ?? SimpleAskStreamService::DEFAULT_MODEL;

        return Inertia::render('AskStream/Index', [
            'models' => this->streamService->getModelsLight(),'selectedModel'=>this
->streamService->getModelsLight(),
            'selectedModel' => this->streamService->getModelsLight(),'selectedModel'=>modelId,
            'selectedModelDetails' => fn() => this->streamService->getModelDetails(this
->streamService->getModelDetails(this->streamService->getModelDetails(modelId),
        ]);
    }

    /**
     * Endpoint de streaming.
     */
    public function stream(Request $request): StreamedResponse
    {
        $validated = $request->validate([
            'message' => 'required|string|max:100000',
            'model' => 'required|string',
            'temperature' => 'nullable|numeric|min:0|max:2',
            'reasoning_effort' => 'nullable|string|in:low,medium,high',
        ]);

        // Update user's preferred model
        $user = auth()->user();
        if ($user && $user->preferred_model !== $validated['model']) {
            $user->update(['preferred_model' => $validated['model']]);
```

```
        }

        $messages = [['role' => 'user', 'content' => $validated['message']]];
        $model = $validated['model'];
        $temperature = (float) ($validated['temperature'] ?? 1.0);
        $reasoningEffort = $validated['reasoning_effort'] ?? null;

        return response()->stream(
            function () use ($messages, $model, $temperature, $reasoningEffort): void {
                $this->streamService->streamToOutput($messages, $model, $temperature,
$reasoningEffort);
            },
            headers: [
                'Content-Type' => 'text/plain; charset=utf-8',
                'Cache-Control' => 'no-cache, no-store',
                'X-Accel-Buffering' => 'no',
            ]
        );
    }
}
```

Points clés :

- `response()->stream()` crée automatiquement une réponse SSE
- Le service `streamToOutput()` écrit directement dans la sortie HTTP
- Headers HTTP spécifiques pour désactiver le buffering
- Support du `reasoning_effort` pour les modèles compatibles

4.3 Routes

Ajoutez les routes dans `routes/web.php` :

PHP

```php
Route::get('/ask-stream', [\App\Http\Controllers\AskStreamController::class, 'index'])
    ->name('stream.index');

Route::post('/ask-stream', [\App\Http\Controllers\AskStreamController::class, 'stream'])
    ->name('stream.post');
```

5. Implémentation côté Frontend

5.1 Hook useStream - Code Essentiel

Voici le code essentiel pour implémenter le streaming côté frontend. Le hook `useStream` de Laravel gère automatiquement la connexion SSE et la réception des chunks.

```vue
<script setup lang="ts">
import { ref } from 'vue';
import { useStream } from '@laravel/stream-vue';

// State
const message = ref('');
const model = ref('openai/gpt-4o-mini');
const temperature = ref(1.0);
const reasoningEffort = ref<'low' | 'medium' | 'high' | null>(null);

/**
 * useStream hook - Le hook concatène automatiquement dans `data`
 * Le backend envoie du texte avec marqueurs [REASONING]...[/REASONING]
 */
const { data, isFetching, isStreaming, send, cancel } = useStream(
    '/ask-stream',
    {
        onData: () => {
            // Callback appelé pour chaque chunk reçu
            // Le contenu est automatiquement concaténé dans `data`
        },
        onFinish: () => {
            // Appelé quand le stream se termine
            message.value = '';
        },
        onError: (err: Error) => {
            console.error('Erreur streaming:', err);
        },
    },
);

/**
 * Submit handler - Envoie la requête de streaming
 */
const submit = () => {
    if (!message.value.trim()) return;

    send({
        message: message.value,
        model: model.value,
        temperature: temperature.value,
        reasoning_effort: reasoningEffort.value,
    });
};
</script>
```

Points clés du hook `useStream` :

- `data` : contient la réponse complète, automatiquement concaténée
- `isStreaming` : booléen indiquant si un stream est en cours
- `isFetching` : booléen pour la phase de connexion
- `send(payload)` : fonction pour démarrer le stream
- `cancel()` : fonction pour annuler le stream en cours
- **Callbacks** :
  - `onData` : appelé pour chaque chunk reçu

- `onFinish` : appelé quand le stream se termine
- `onError` : appelé en cas d'erreur

5.2 Parsing des Marqueurs Reasoning

Le backend envoie le reasoning encapsulé dans des marqueurs spéciaux. Voici comment les extraire :

```vue
<script setup lang="ts">
import { computed } from 'vue';

/**
 * Extrait le contenu principal (sans le reasoning)
 */
const streamedContent = computed(() => {
    if (!data.value) return '';
    // Enlever les blocs [REASONING]...[/REASONING]
    return data.value
        .replace(/\[REASONING\][\s\S]*?\[\/REASONING\]/g, '')
        .trim();
});

/**
 * Extrait le reasoning des marqueurs
 */
const streamedReasoning = computed(() => {
    if (!data.value) return '';
    const matches = data.value.match(/\[REASONING\]([\s\S]*?)\[\/REASONING\]/g);
    if (!matches) return '';
    return matches
        .map((m) =>
            m.replace(/\[REASONING\]/g, '').replace(/\[\/REASONING\]/g, ''),
        )
        .join('');
});
</script>

<template>
    <!-- Affichage du reasoning (optionnel) -->
    <div v-if="streamedReasoning" class="reasoning-trace">
        <h4>Trace de raisonnement</h4>
        <pre>{{ streamedReasoning }}</pre>
    </div>

    <!-- Affichage du contenu principal -->
    <div class="main-content">
        <MarkdownRenderer :content="streamedContent" />
    </div>
</template>
```

Explication du parsing :

- Les modèles de raisonnement (comme les modèles o1) génèrent une trace de leur raisonnement
- Le backend encapsule cette trace dans `[REASONING]...[/REASONING]`
- On utilise des regex pour extraire séparément le contenu principal et le reasoning
- Le reasoning peut être affiché ou caché selon les besoins

6. Configuration CSRF et Authentification

6.1 Token CSRF dans le Template de Base

Le package `@laravel/stream-vue` cherche automatiquement le token CSRF dans une balise meta du `<head>`. Ajoutez cette ligne dans votre template de base Inertia :

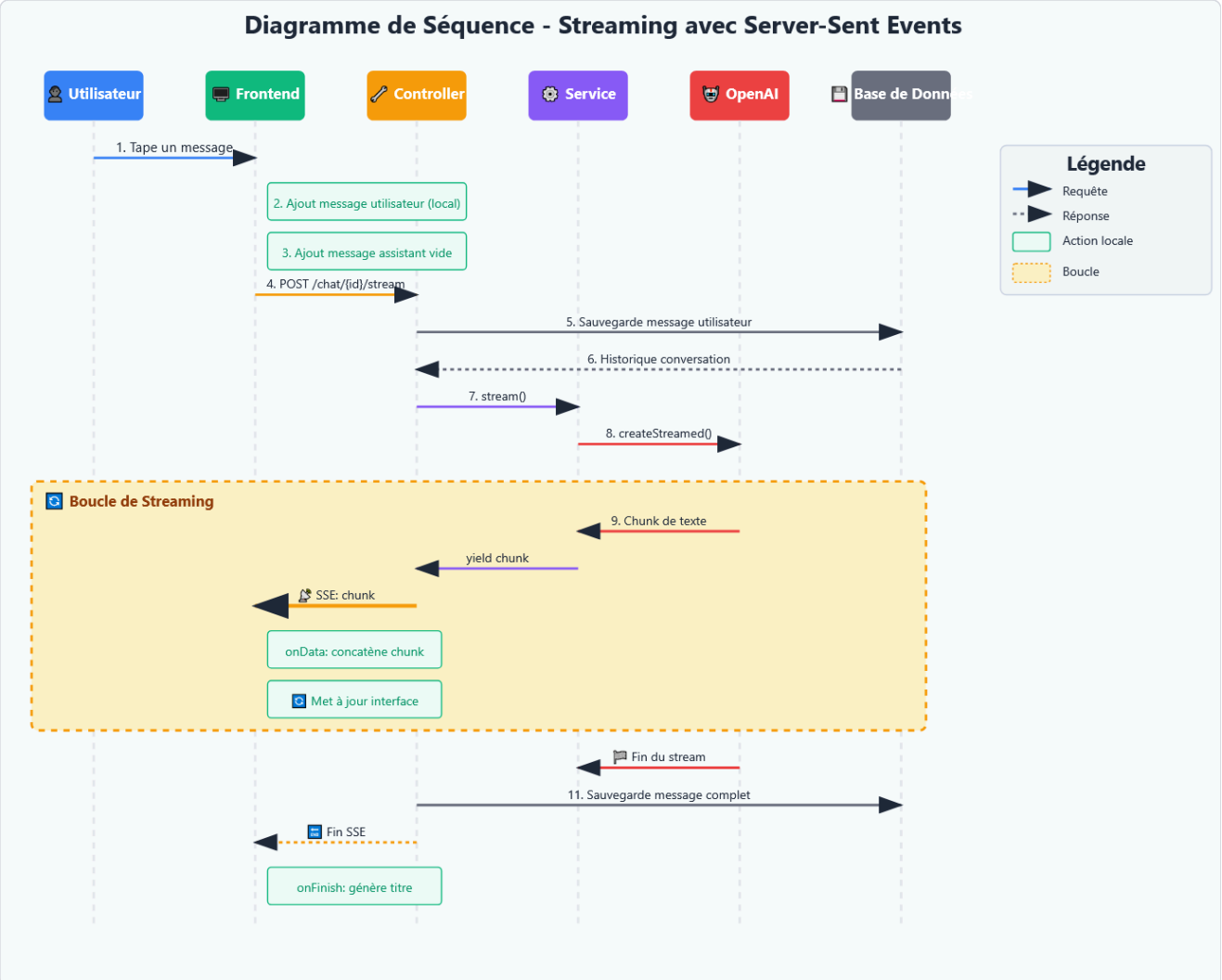
Fichier : `resources/views/app.blade.php`

```xml
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="csrf-token" content="{{ csrf_token() }}">

    <title inertia>{{ config('app.name', 'Laravel') }}</title>
    <!-- ... autres balises meta et scripts -->
</head>
```

Sans cette balise, vous obtiendrez une **erreur 419** lors des requêtes de streaming.

7. Diagramme de Séquence



8. Avantages des Server-Sent Events

Aspect	Server-Sent Events
Configuration	Aucune configuration serveur
Dépendances	1 package npm uniquement
Coût	Gratuit
Performance	Direct HTTP, faible latence
Complexité	Simple et élégant

Conclusion

Les **Server-Sent Events** avec `@laravel/stream-vue` offrent une solution élégante et simple pour implémenter le streaming en temps réel. Cette approche native élimine la complexité des solutions externes (comme Pusher) tout en offrant d'excellentes performances, sans nécessiter aucune configuration serveur supplémentaire.

Sources

- https://laravel.com/docs/12.x/responses#streamed-responses
- https://laravel.com/docs/12.x/broadcasting
- https://openrouter.ai/docs/api/reference/streaming
- https://openrouter.ai/docs/guides/best-practices/reasoning-tokens#example-1-streaming-mode-with-reasoning-tokens

4.11 Reconnaissance d'images

Vous rêvez d'avoir un chatbot suffisamment malin pour décrire les photos de vos (malheureux) selfies ou deviner ce qui se trouve dans l'assiette ? Bonne nouvelle : certains modèles de langage savent gérer des images en plus du texte. Voici comment les utiliser sans transformer votre application en gouffre financier… et sans envoyer des toiles de 8 000 px de large.

1. Présentation Générale

Les LLM multimodaux peuvent interpréter un contenu visuel et vous renvoyer une description, un résumé ou tout autre commentaire pertinent. Pour en profiter, **encore faut-il** leur envoyer les images correctement : il ne suffit pas de coller un `.jpg` brut dans votre JSON.

- Certains modèles ne sont pas (encore) capables d'interpréter les images. Vérifiez donc la compatibilité.
- La mise en forme du message user change : on envoie un tableau de "blocs" (`text`, `image_url`), et non plus une simple chaîne de caractères.

2. Vérifier la Compatibilité

Avant de bombarder un LLM avec des photos de votre chat (le vrai, pas le chatbot), assurez-vous qu'il sache quoi en faire. Dans la méthode `getModels()` (de votre `ChatService`), nous déterminons la capacité du modèle via :

```php
'supports_image' => 'text+image->text' === $model['architecture']['modality']
```

Si `supports_image` est `true`, bingo : ce modèle sait traiter des images. Sinon, vous obtiendrez plus ou moins le même résultat qu'un poisson rouge devant un livre de math.

3. Format de Requête Multimodale

Pour envoyer une image au LLM, vous devez structurer le contenu de votre message sous forme de tableau. Exemple :

```json
"messages": [
  {
    "role": "user",
    "content": [
      {
        "type": "text",
        "text": "What is this image about?"
      },
      {
        "type": "image_url",
        "image_url": {
          "url": "https://example.com/magnifique_photo.jpg"
        }
      }
    ]
  }
]
```

- `type: "text"` : la partie texte classique.
- `type: "image_url"` : un bloc contenant l'URL d'une image (ou son équivalent base64, voir section suivante).

Lors de l'envoi, l'API traitera ces blocs pour comprendre que vous lui fournissez à la fois du texte et une image.

4. Image par URL vs Base64

4.1 URL Publique

La solution la plus simple consiste à héberger votre image quelque part (CDN, bucket S3, etc.), puis à placer cette URL dans la requête. Le modèle ira la chercher directement.

- Avantage : **Aucun** besoin de manipuler l'image côté backend, hormis vérifier qu'elle soit accessible.
- Inconvénient : Vous devez **impérativement** que l'image soit en ligne, et idéalement de taille raisonnable (ou la facture risque de grimper selon la politique du modèle).

4.2 Encodage Base64

Si vous avez une image locale ou que vous ne pouvez pas l'héberger, vous pouvez l'envoyer sous forme de chaîne base64, par exemple :

```javascript
"messages": [
  {
    "role": "user",
    "content": [
      {
        "type": "text",
        "text": "What's in this image?"
      },
      {
        "type": "image_url",
        "image_url": {
          "url": "..."
        }
      }
    ]
  }
]
```

> ***Attention*** *: la chaîne base64 doit inclure* `` *ou autre content-type, histoire que le modèle sache de quel format il s'agit.*

- Avantage : **Aucune** dépendance à un serveur de fichiers distant.
- Inconvénient : La requête HTTP peut doubler (ou plus) de taille si vous balancez une photo en 4K. Vive la surcharge…

5. Optimiser l'Image : Pourquoi et Comment

Plus l'image est lourde, plus vous risquez de payer, et plus la requête mettra du temps à arriver au LLM (et inversement). Autrement dit, si vous uploadez un panorama en 8K, préparez-vous à sortir le champagne (et le porte-monnaie).

5.1 Exemple d'une Classe d'Optimisation

Pour redimensionner, convertir et compresser l'image en JPEG de façon plus civilisée :

```php
class ImageService
{
    private const MAX_WIDTH = 800;
    private const JPEG_QUALITY = 85;

    public function optimizeImage(string $sourcePath): string
    {
        [$width, $height, $type] = getimagesize($sourcePath);
        $ratio = $width / $height;
        $newWidth = min($width, self::MAX_WIDTH);
        $newHeight = (int)($newWidth / $ratio);

        $newImage = imagecreatetruecolor($newWidth, $newHeight);

        $sourceImage = match ($type) {
            IMAGETYPE_JPEG => imagecreatefromjpeg($sourcePath),
            IMAGETYPE_PNG => imagecreatefrompng($sourcePath),
            IMAGETYPE_WEBP => imagecreatefromwebp($sourcePath),
            default => throw new \Exception('Format d\'image non supporté'),
        };

        imagecopyresampled($newImage, $sourceImage, 0, 0, 0, 0, $newWidth, $newHeight, $width,
$height);

        ob_start();
        imagejpeg($newImage, null, self::JPEG_QUALITY);
        $imageData = ob_get_clean();

        imagedestroy($sourceImage);
        imagedestroy($newImage);

        return $imageData;
    }
}
```

La fonction `optimizeImage()` :

1. **Redimensionne** l'image si elle dépasse `MAX_WIDTH` (ex. 800 px).
2. **Convertit** en JPEG de qualité 85 % (assez correct pour du descriptif).
3. **Retourne** les données binaires dans une chaîne.

5.2 Enchaîner sur le Base64

Si vous devez envoyer l'image en *base64*, il vous suffit de faire :

```php
optimizedData=(newImageService())->optimizeImage(optimizedData = (new ImageService())
→optimizeImage(optimizedData=(newImageService())->optimizeImage(cheminFichier);
base64Data=base64encode(base64Data = base64_encode(base64Data=base64encode(optimizedData); //
Ajoute le beau code encodé
url="".url = "" .
url="".base64Data; // Ou bien votre code "automatique"
```

Ensuite, vous placez `$url` dans la propriété `image_url` de votre request, comme vu plus haut.

6. Mise en Œuvre Rapide

1. **Vérifiez** si le modèle supporte les images (`supports_image == true`).
2. **Récupérez** votre image (upload de l'utilisateur, dossier local, etc.).
3. **Optimisez** (optionnel mais conseillé) pour éviter les fiascos sur la facture.
4. **Créez** le `messages` :

```php
messages=[['role'⇒'user','content'⇒[['type'⇒'text','text'⇒'Canyoudescribewhatthisis?'],
['type'⇒'imageurl','imageurl'⇒['url'⇒messages = [
  [
    'role' ⇒ 'user',
    'content' ⇒ [
      [
        'type' ⇒ 'text',
        'text' ⇒ 'Can you describe what this is?'
      ],
      [
        'type' ⇒ 'image_url',
        'image_url' ⇒ [
          'url' ⇒ messages=[['role'⇒'user','content'⇒[['type'⇒'text',
'text'⇒'Canyoudescribewhatthisis?'],['type'⇒'imageurl','imageurl'⇒['url'⇒url // Soit un lien
http(s), soit ...
        ]
      ]
    ]
  ]
];
```

5. **Envoyez** le tout au LLM via votre client OpenRouter ou OpenAI.
6. **Observez** la réponse : le modèle devrait vous décrire l'image ou répondre à vos questions, si tout s'est bien passé.



7. Bonnes Pratiques

- **Bouclez sur** `supports_image` : on répète l'importance de vérifier que le modèle est vraiment multimodal, sous peine de voir votre requête se faire bâcher.
- **Dimension et format** : compressez suffisamment, sinon l'API risque d'exploser le temps d'analyse (et peut-être vos factures).
- **Encodage** : assurez-vous de fournir le bon *content-type* dans la data URL si vous faites du base64.
- **Filtrage** : L'API fera de la modération côté serveur. Vous n'avez pas à mettre un système anti-chaton ou anti-contenu gore vous-même (sauf besoin particulier).

Voilà, vous avez désormais tous les ingrédients pour rendre votre chatbot (vaguement) visionnaire. Alors, à vos images et n'oubliez pas : "light is right" – plus l'image est légère, moins vous vous ferez taper sur les doigts (ou sur le portefeuille).

Documentation

- https://openrouter.ai/docs/requests#images-_-multimodal-requests

4.12 Laravel Dusk

Introduction à Laravel Dusk

Qu'est-ce que Laravel Dusk ?

Laravel Dusk est un outil de test automatisé pour les interfaces utilisateurs, spécialement conçu pour les applications Laravel. Il permet de simuler les interactions des utilisateurs avec une application web dans un navigateur réel, en utilisant ChromeDriver pour manipuler les éléments du navigateur. Cette approche assure que les tests reflètent fidèlement les actions des utilisateurs.

Avantages de l'utilisation de Laravel Dusk

L'utilisation de Laravel Dusk présente plusieurs bénéfices clés pour le développement d'applications web :

- **Tests automatisés de bout en bout** : Dusk facilite l'automatisation des interactions utilisateurs, telles que cliquer sur des liens ou remplir des formulaires.
- **Intégration fluide** : Étant intégré dans l'écosystème Laravel, Dusk est particulièrement accessible pour ceux qui sont déjà familiers avec ce framework.
- **Débogage efficace** : La fonctionnalité de capture d'écran de Dusk permet de détecter rapidement les erreurs.
- **Sécurité renforcée** : En testant des scénarios d'interaction utilisateur réels, Dusk aide à identifier et corriger les vulnérabilités avant le déploiement.

Nous explorerons ensuite comment installer Laravel Dusk, configurer l'environnement de test et créer des tests pour garantir la qualité et la robustesse de vos applications w

Installation de Laravel Dusk

Prérequis

Avant d'installer Laravel Dusk, assurez-vous que votre environnement de développement répond aux exigences suivantes :

- **Laravel** : Une installation existante de Laravel, de préférence la dernière version pour bénéficier de toutes les fonctionnalités et améliorations de sécurité.
- **Composer** : Utilisé pour gérer les dépendances de Laravel, y compris l'installation de Dusk.
- **Navigateur Chrome** : Nécessaire pour ChromeDriver, qui est utilisé par Dusk pour automatiser les tests dans un navigateur.

Étapes d'installation

Pour installer Laravel Dusk, suivez ces étapes simples :

1. **Ajout de Dusk à votre projet Laravel** : Ouvrez votre terminal et exécutez la commande suivante à la racine de votre projet Laravel :

   ```bash
   composer require --dev laravel/dusk
   ```

   Cette commande installe Laravel Dusk comme une dépendance de développement.

2. **Configuration de Dusk** : Après l'installation, exécutez la commande suivante pour publier les assets de Dusk et configurer votre projet :

   ```bash
   php artisan dusk:install
   ```

   Cette commande crée le répertoire `tests/Browser` où vous placerez vos tests de navigateur, et configure également quelques autres fichiers essentiels pour le fonctionnement de Dusk.

3. **Mise à jour des variables d'environnement** : Assurez-vous de configurer l'environnement de test approprié dans votre fichier `.env` pour éviter toute interaction avec vos données de production lors de l'exécution des tests.

Avec Laravel Dusk correctement installé et configuré, vous êtes prêt à commencer à créer des tests automatisés pour votre application. La section suivante détaillera comment configurer votre premier test et utiliser efficacement Dusk pour améliorer la qualité de votre application.

Utilisation de Laravel Dusk

Configuration initiale

Avant de commencer à écrire des tests avec Laravel Dusk, il est essentiel de comprendre la structure de base et la configuration initiale :

1. **Dossier des tests** : Les tests Dusk se trouvent dans le dossier `tests/Browser`. Chaque fichier de test doit étendre la classe `BrowserTestCase`, qui fournit diverses méthodes pour interagir avec le navigateur.

2. **Fichier DuskTestCase** : Le fichier `DuskTestCase.php` dans le dossier `tests` est le point de départ de tous les tests Dusk. Il contient la configuration nécessaire pour démarrer et fermer le navigateur. Veillez à vérifier que le chemin du pilote ChromeDriver est correctement configuré ici.

Création de tests basiques

Pour créer un test simple avec Laravel Dusk, procédez comme suit :

1. **Création d'un nouveau test** : Utilisez Artisan pour générer un nouveau test :

   ```bash
   php artisan dusk:make UnTest
   ```

Cela créera un nouveau fichier de test dans `tests/Browser` nommé `UnTest.php`.

2. **Écriture d'un test** : Ouvrez `UnTest.php` et ajoutez une fonction de test. Par exemple, pour tester si la page d'accueil de votre application charge correctement :

```php
it('displays the Laravel text on the home page', function () {
    this->browse(function(this→browse(function (this->browse(function(browser) {
        $browser→visit('/')
                →assertSee('Laravel');
    });
});
```

Cette fonction utilise le navigateur pour visiter la page d'accueil (`/`) et vérifie que le texte 'Laravel' est présent.

Exécution des tests

Pour exécuter vos tests Dusk, utilisez la commande suivante dans votre terminal :

```bash
php artisan dusk
```

Cette commande lancera tous les tests situés dans le dossier `tests/Browser`. Vous pouvez également spécifier un chemin vers un test spécifique si vous souhaitez exécuter un seul test.

En suivant ces étapes, vous pouvez configurer, écrire et exécuter des tests basiques avec Laravel Dusk. La section suivante abordera des fonctionnalités plus avancées pour maximiser l'efficacité de vos tests automatisés.

Fonctionnalités avancées de Laravel Dusk

Tests avec authentification

Laravel Dusk facilite le test des fonctionnalités qui nécessitent une authentification utilisateur. Voici comment vous pouvez tester les scénarios authentifiés :

1. **Authentification d'un utilisateur** : Vous pouvez utiliser la méthode `loginAs` pour simuler l'authentification d'un utilisateur avant de réaliser des actions spécifiques :

```php
use App\Models\User;

it('allows authenticated user to see the dashboard', function () {
    this->browse(function(this→browse(function (this->browse(function(browser) {

user=User::find(1);//Assurez-vousquecetutilisateurexistedansvotrebasededonne´esdetestuser = User::find(1); // Assurez-vous que cet utilisateur existe dans votre base de données de test

user=User::find(1);//Assurez-vousquecetutilisateurexistedansvotrebasededonne´esdetestbrowser->loginAs($user)
                →visit('/home')
                →assertSee('Dashboard');
    });
});
```

Interaction avec JavaScript

Laravel Dusk est particulièrement utile pour tester des applications qui utilisent intensivement JavaScript. Voici comment interagir avec des éléments JS :

1. **Cliquer sur des éléments JavaScript** : Par exemple, pour cliquer sur un bouton qui ouvre un modal via JavaScript :

```php
it('displays the correct content in the JavaScript modal', function () {
    this->browse(function(this→browse(function (this->browse(function(browser) {
        $browser→visit('/page-with-modal')
                →click('#open-modal-button') // Assurez-vous que l'ID est correct
                →waitForText('Modal Content')
                →assertSee('Modal Content');
    });
});
```

Capture d'écran et débogage

Laravel Dusk permet de prendre des captures d'écran en cas d'échec des tests, ce qui est très utile pour le débogage :

1. **Prendre une capture d'écran** : Vous pouvez configurer Dusk pour prendre une capture d'écran automatiquement lorsqu'un test échoue ou manuellement dans vos scripts de test :

```php
it('captures a screenshot and verifies non-existent content', function () {
    this->browse(function(this→browse(function (this->browse(function(browser) {
        $browser→visit('/some-page')
                →screenshot('name-of-screenshot') // Sauvegarde la capture d'écran dans le dossier tests/Browser/screenshots
                →assertSee('Some content that does not exist');
    });
});
```

En maîtrisant ces fonctionnalités avancées, vous pouvez étendre significativement la couverture de vos tests automatisés, assurant une plus grande stabilité et fiabilité de vos applications web. La conclusion suivante résumera les points clés abordés et fournira des ressources supplémentaires pour approfondir votre maîtrise de

Laravel Dusk.

Configuration d'un environnement dédié à Dusk

Création d'un `.env.dusk.local`

Pour éviter de polluer votre environnement de développement standard (déjà suffisamment bancal, j'imagine), il est conseillé de créer un fichier `.env.dusk.local`. Celui-ci contiendra une configuration spécifique exclusivement réservée à vos tests Dusk. Par exemple :

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:0hDqTLQR7MnO4xtc7cFXeiTzZQ7PUj2eijGpU4cbYzA=
APP_DEBUG=true
APP_TIMEZONE=UTC
APP_URL=http://dusk.test

APP_LOCALE=en
APP_FALLBACK_LOCALE=en
APP_FAKER_LOCALE=en_US

APP_MAINTENANCE_DRIVER=file

PHP_CLI_SERVER_WORKERS=4

BCRYPT_ROUNDS=12

LOG_CHANNEL=stack
LOG_STACK=single
LOG_DEPRECATIONS_CHANNEL=null
LOG_LEVEL=debug

DB_CONNECTION=sqlite
DB_DATABASE="C:/laragon/www/dusk/database/test.sqlite"

SESSION_DRIVER=database
SESSION_LIFETIME=120
SESSION_ENCRYPT=false
SESSION_PATH=/
SESSION_DOMAIN=null

BROADCAST_CONNECTION=log
FILESYSTEM_DISK=local
QUEUE_CONNECTION=database

CACHE_STORE=database
CACHE_PREFIX=

MEMCACHED_HOST=127.0.0.1

REDIS_CLIENT=phpredis
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_MAILER=log
MAIL_HOST=127.0.0.1
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS="hello@example.com"
MAIL_FROM_NAME="APP_NAME"AWS_ACCESS_KEY_ID=AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=us-east-1AWS_BUCKET=AWS_USE_PATH_STYLE_ENDPOINT=
false VITE_APP_NAME="{APP_NAME}"

DUSK_HEADLESS_DISABLED=true
```

Points-clés :

- **URL dédiée** : Ajustez `APP_URL` afin d'utiliser un hôte virtuel créé par Laragon, Laravel Herd ou un autre outil. Cela permettra à vos tests Dusk de s'exécuter proprement sur un environnement isolé, au lieu de polluer votre configuration principale.
- **Base de données SQLite dédiée** : La ligne `DB_DATABASE="C:/laragon/www/dusk/database/test.sqlite"` est un exemple de configuration pour utiliser une base de données SQLite réservée aux tests. Créez manuellement ce fichier. Ainsi, vous n'aurez pas à exécuter vos migrations complètes à chaque test, ce qui accélère le processus.
- **DUSK_HEADLESS_DISABLED=true** : Mettre cette variable à `true` permet de voir Chrome s'ouvrir pendant les tests. Parfait pour les âmes inquiètes qui ont besoin de constater visuellement le désastre qu'elles sont en train de commettre. Ajoutez des `->pause(1000)` dans vos tests pour ralentir l'exécution et observer ce qu'il se passe.

Nettoyage de la base de données entre chaque test

Pour éviter que les données s'accumulent et donner l'illusion que vous savez garder un environnement propre, vous pouvez utiliser le `DatabaseTruncation`. Cela évite de devoir relancer toutes les migrations et les seeders entre chaque test, et donc de perdre un temps précieux à réimporter vos données.

Dans votre fichier `tests/Pest.php`, ajoutez :

```php
pest()->extend(Tests\DuskTestCase::class)
    ->use(Illuminate\Foundation\Testing\DatabaseTruncation::class)
    // ->use(Illuminate\Foundation\Testing\DatabaseMigrations::class)
    ->in('Browser');
```

`DatabaseTruncation` va vider les tables entre chaque test, ce qui est beaucoup plus rapide qu'une migration complète.

Exemple de test avec pauses

Voici un exemple de test, parce qu'apparemment il vous faut tout sous le nez :

```php
test('example', function () {
    user=User::factory()->create();user = User::factory()->create();

    user=User::factory()->create();this->browse(function (Browser browser)use(browser) use
(browser)use(user) {
        browser->loginAs(browser
            ->loginAs(browser->loginAs(user)
            ->visit('/user/profile')
            ->pause(1000) // On attend 1 seconde pour admirer le "chef-d'œuvre"
            ->assertSee('Profile Information');
    });
});
```

Maintenant, vous avez tout ce qu'il vous faut pour créer un environnement Dusk propre, dédié, et observer en direct les résultats de vos tests. Utilisez `.env.dusk.local`, configurez une base de données test, nettoyez-la entre chaque exécution, et réglez votre Chrome pour qu'il n'exécute pas vos tests en mode invisible, histoire que vous puissiez enfin voir de vos yeux vos échecs majestueux. Enjoy.

Gestion de ChromeDriver

- **Détection automatique de la version** :

```bash
php artisan dusk:chrome-driver --detect
```

Cette commande tente de détecter la version de Chrome installée sur votre machine et télécharge la version correspondante de ChromeDriver. Parfait pour les rois de l'improvisation.

- **Installer la dernière version compatible** :

```bash
php artisan dusk:chrome-driver
```

Sans argument, cette commande télécharge la version de ChromeDriver la plus adaptée à votre environnement.

- **Forcer l'installation d'une version spécifique** :

```bash
php artisan dusk:chrome-driver <version>
```

Remplacez `<version>` par un numéro de version pour imposer vos choix rétrogrades (ex : `114`).

- **Installer toutes les versions disponibles** :

```bash
php artisan dusk:chrome-driver --all
```

Vous voulez tout télécharger, juste au cas où ? Soyez mon invité. C'est clairement excessif, mais bon, faites donc.

- **Aide et options supplémentaires** :

```bash
php artisan dusk:chrome-driver --help
```

Si vous êtes encore perdus, c'est ici que ça se passe. Lire la doc, quelle idée révolutionnaire !

Conclusion

Résumé des points clés

Laravel Dusk est un puissant outil de test automatisé pour les applications Laravel, permettant de simuler des interactions utilisateurs dans un navigateur réel. Nous avons exploré son installation, la configuration initiale, et la création de tests de base. Nous avons également abordé des fonctionnalités avancées telles que les tests avec authentification, l'interaction avec JavaScript, et la capture d'écran pour le débogage.

Ressources supplémentaires pour approfondir

Pour ceux qui souhaitent approfondir leurs connaissances et compétences avec Laravel Dusk, voici quelques ressources utiles :

- **Documentation officielle de Laravel Dusk** : La source la plus complète et à jour pour apprendre à utiliser Laravel Dusk. Disponible sur le site web de Laravel.
- **Cours et tutoriels en ligne** : De nombreux cours en ligne offrent des tutoriels détaillés sur Laravel Dusk, souvent avec des exemples pratiques et des projets.
- **Communautés et forums** : Rejoindre des communautés en ligne, comme Stack Overflow ou les forums Laravel, peut être très bénéfique pour obtenir de l'aide, partager des expériences et apprendre des meilleures pratiques.

En intégrant Laravel Dusk dans votre flux de travail de développement, vous pouvez non seulement améliorer la qualité et la performance de vos applications, mais aussi anticiper et résoudre les problèmes avant qu'ils n'affectent les utilisateurs finaux. Cela fait de Laravel Dusk un investissement précieux pour tout projet Laravel soucieux de maintenir un haut standard de qualité.

Liens utiles

- https://laravel.com/docs/11.x/dusk
- Exemple de code : https://github.com/opmvpc/test-dusk

4.13 Outils pour LLM

Dans ce chapitre, nous abordons les divers outils que l'on peut intégrer à un modèle de langage à grande échelle (LLM) tel que GPT, afin d'élargir ses capacités de réponse et d'interaction. Notre objectif est de clarifier comment ces outils peuvent être utilisés par le LLM pour fournir des réponses plus précises et contextuelles aux requêtes des utilisateurs.

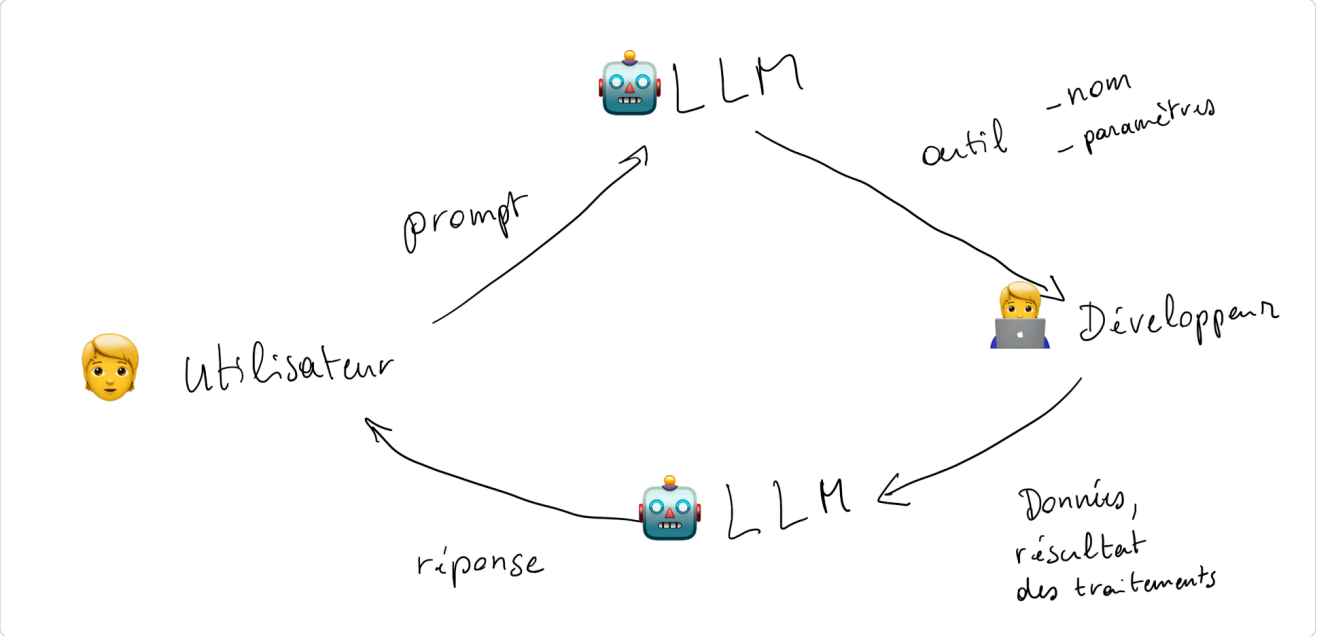
Principe de fonctionnement

Fonctionnement général

Les LLM, malgré leur capacité remarquable à générer du texte basé sur d'énormes volumes de données, ne possèdent pas une connaissance universelle. Pour compenser certaines de leurs lacunes, notamment en termes d'informations à jour ou spécifiques, il est possible de les équiper d'outils externes. Ces outils peuvent être sollicités par le LLM lorsqu'il identifie le besoin d'informations complémentaires pour répondre à une question.

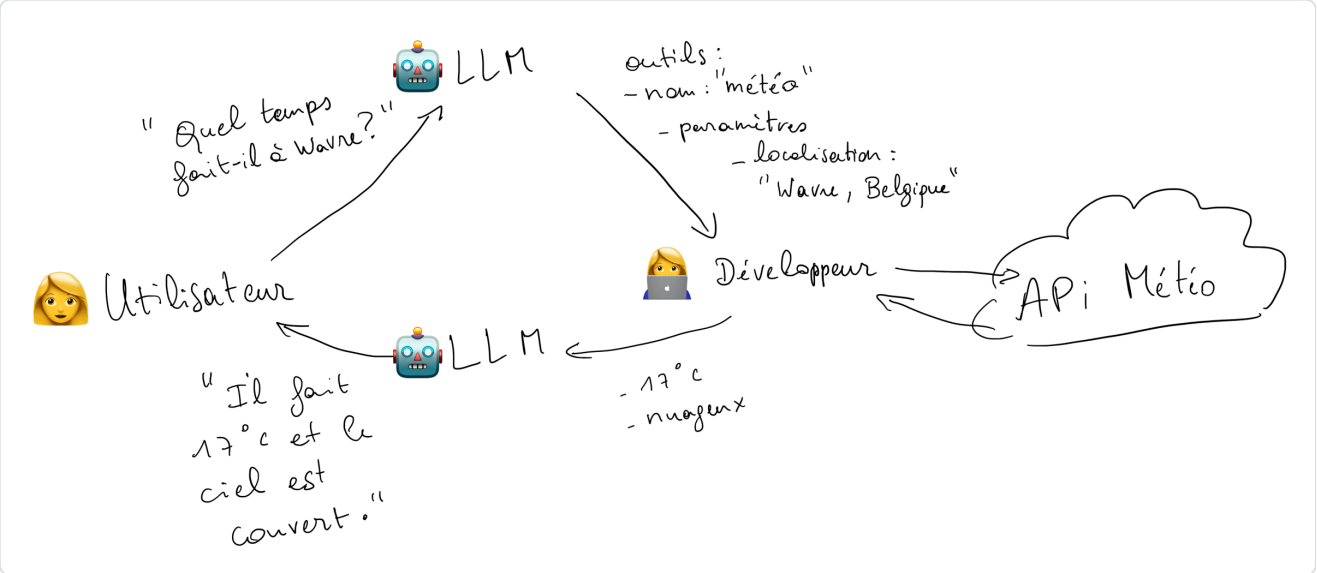
Interaction avec les outils

Lorsqu'une question posée nécessite des informations externes, le LLM détermine quel outil est nécessaire pour obtenir ces informations. Il produit alors une requête structurée avec les paramètres nécessaires à l'outil. Les développeurs reçoivent cette requête, extraient les données requises et les rendent disponibles au LLM, qui les intègre dans sa réponse.



Exemple Pratique : Outil Météo

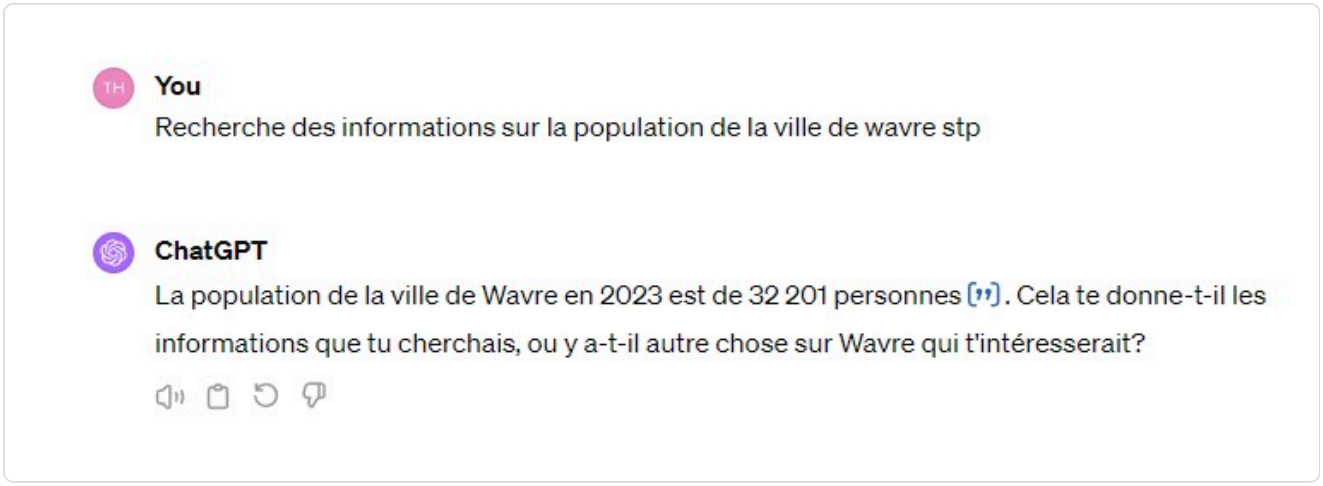
Considérons l'exemple de l'outil météo. Si un utilisateur interroge sur la météo à Wavre, en Belgique, le LLM reconnaît qu'il a besoin de données météorologiques externes pour formuler une réponse. Il initie une requête spécifiant l'outil "météo" et la localisation "Wavre, Belgique". Les développeurs récupèrent ces informations et les transmettent au LLM, qui les utilise pour élaborer une réponse informative et adaptée.



Types d'outils disponibles

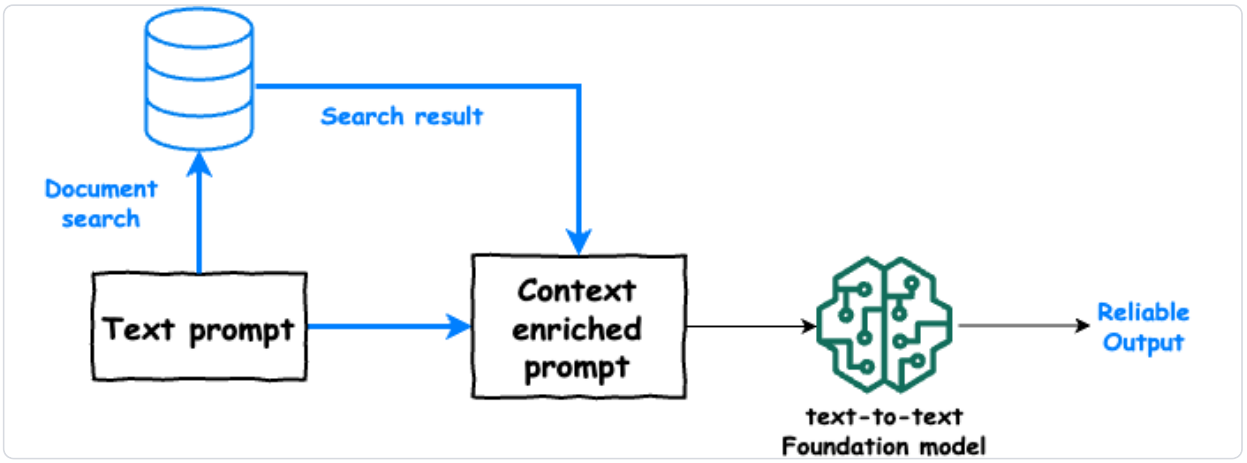
Recherche sur Internet

Cet outil permet au LLM de réaliser des recherches sur Internet pour répondre à des questions demandant des informations récentes ou spécifiques, non incluses dans sa base de données d'entraînement.



Recherche dans des Documents

Pour des questions nécessitant l'analyse de documents spécifiques, cet outil autorise le LLM à fouiller dans un ensemble défini de documents ou de bases de données.



https://medium.com/@machangsha/foundation-model-101-dumb-down-retrieval-augmented-generation-rag-2f9e0521d9f3

You

Cdc-Lifers.pdf
PDF

Fais moi un résumé de ce document stp. Anonymise les noms et prénoms stp.

< 2 / 2 >

ChatGPT

Le document présente le rapport d'un projet de jeu de simulation de vie en ligne intitulé "Lifers", développé dans le cadre du Brevet d'Enseignement Supérieur de Webdeveloper pour l'année académique 2023-2024. Le projet est décrit comme né de la passion pour les jeux de gestion et de simulation, avec pour objectif de recréer et améliorer l'expérience communautaire des plateformes similaires, en y intégrant des aspects de la vie quotidienne souvent négligés.

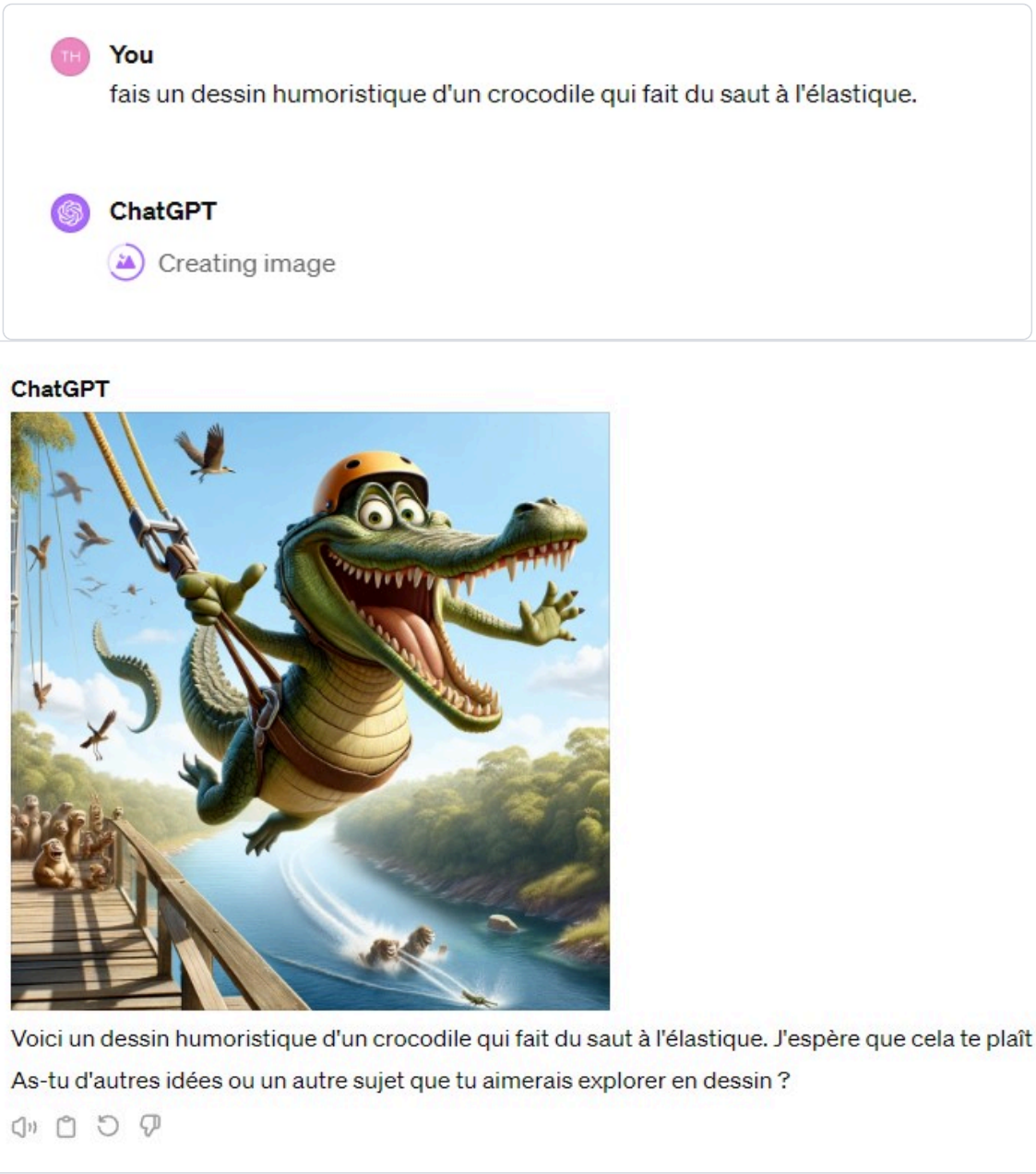
Le public cible est varié, s'étendant des joueurs occasionnels aux plus engagés, avec une tranche d'âge de 16 à 30 ans et plus. Le jeu met l'accent sur la bienveillance communautaire, la sécurité des interactions, et encourage une participation active à l'évolution du site à travers une boîte à idées communautaire.

Le site de Lifers aspire à créer une communauté fondée sur la gentillesse et l'entraide, sans recourir à la publicité payante, préférant une croissance organique basée sur les relations authentiques. Il mettra en œuvre des suggestions de la communauté pour garantir que les améliorations populaires soient priorisées, tout en maintenant une communication transparente sur les possibilités d'implémentation.

Le contenu du site comprendra des pages pour la gestion quotidienne du personnage du joueur (Lifer), l'éducation, le travail, et la vie sociale, enrichis par des contenus visuels et interactifs, y compris des images générées par IA et des interactions directes entre les joueurs.

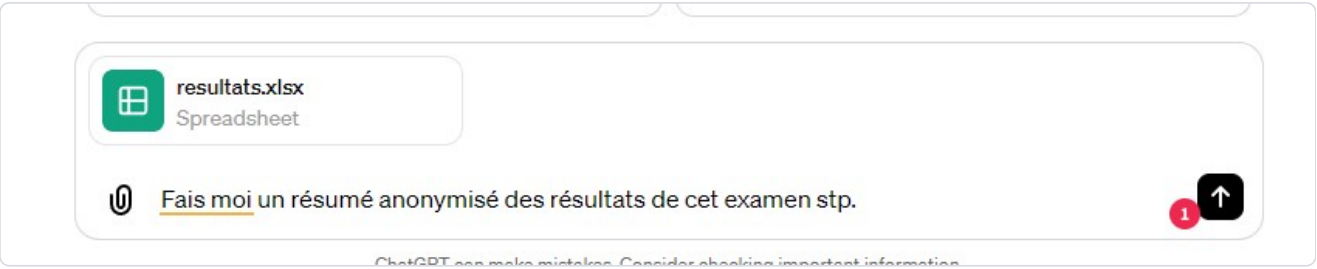
Génération d'Images

Cet outil habilite le LLM à créer des images à partir de descriptions textuelles, enrichissant ses réponses avec du contenu visuel pertinent.



Environnement d'Exécution de Code

Pour les questions exigeant l'exécution de code, cet outil met à disposition un environnement sécurisé où le code peut être exécuté et les résultats intégrés dans la réponse du LLM.



You

essaie de me donner quelques tendances statistiques stp

ChatGPT

Finished analyzing

Always expand output?

python

Copy code

```
# Calculons quelques statistiques de base pour mieux comprendre les tendances dans le

# Calculer la moyenne des notes pour chaque critère d'évaluation
moyennes = df_anonymized.loc[:, "Conformité au Cahier des Charges":"Total (/100)"].me

# Calculer le nombre de réussites et d'échecs
reussites = df_anonymized["Réussite"].value_counts()

# Calculer la distribution des étudiants selon leur classement
classement_distribution = df_anonymized["Classement"].value_counts().sort_index()

moyennes, reussites, classement_distribution
```

Result

```
(Conformité au Cahier des Charges     5.230769
 Qualité du Code                      3.923077
 Fonctionnalité et Ergonomie          2.153846
 Automatisation                       3.384615
 Rapport de Projet                    3.769231
 Réussite                            46.428571
 Maitrise                            17.142857
```

Voici quelques tendances statistiques basées sur les résultats de l'examen :

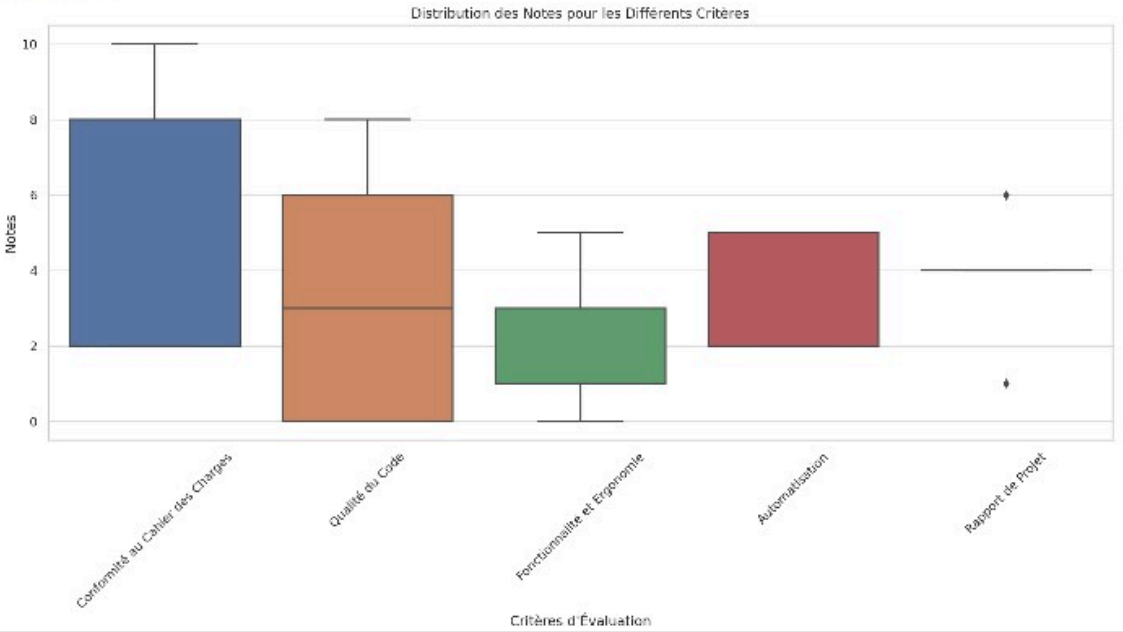
- **Moyennes par critère** :
  - La conformité au cahier des charges a une moyenne de 5,23/10.
  - La qualité du code est évaluée en moyenne à 3,92/10.
  - La fonctionnalité et l'ergonomie sont notées en moyenne à 2,15/10.
  - L'automatisation obtient une moyenne de 3,38/10.
  - Le rapport de projet a une moyenne de 3,77/10.
  - La note totale moyenne est de 63,57/100.
- **Réussite** : Sur les étudiants analysés, 13 ont obtenu une note de réussite (50 points), tandis qu'un étudiant n'a pas réussi.
- **Distribution des classements** :
  - 6 étudiants sont classés en première position.
  - 4 étudiants ont un classement en septième position.
  - 3 étudiants sont classés en onzième position.
  - 1 étudiant est classé quatorzième.

You

peux tu faire un graphique de la distribution des notes pour les différents critères stp?

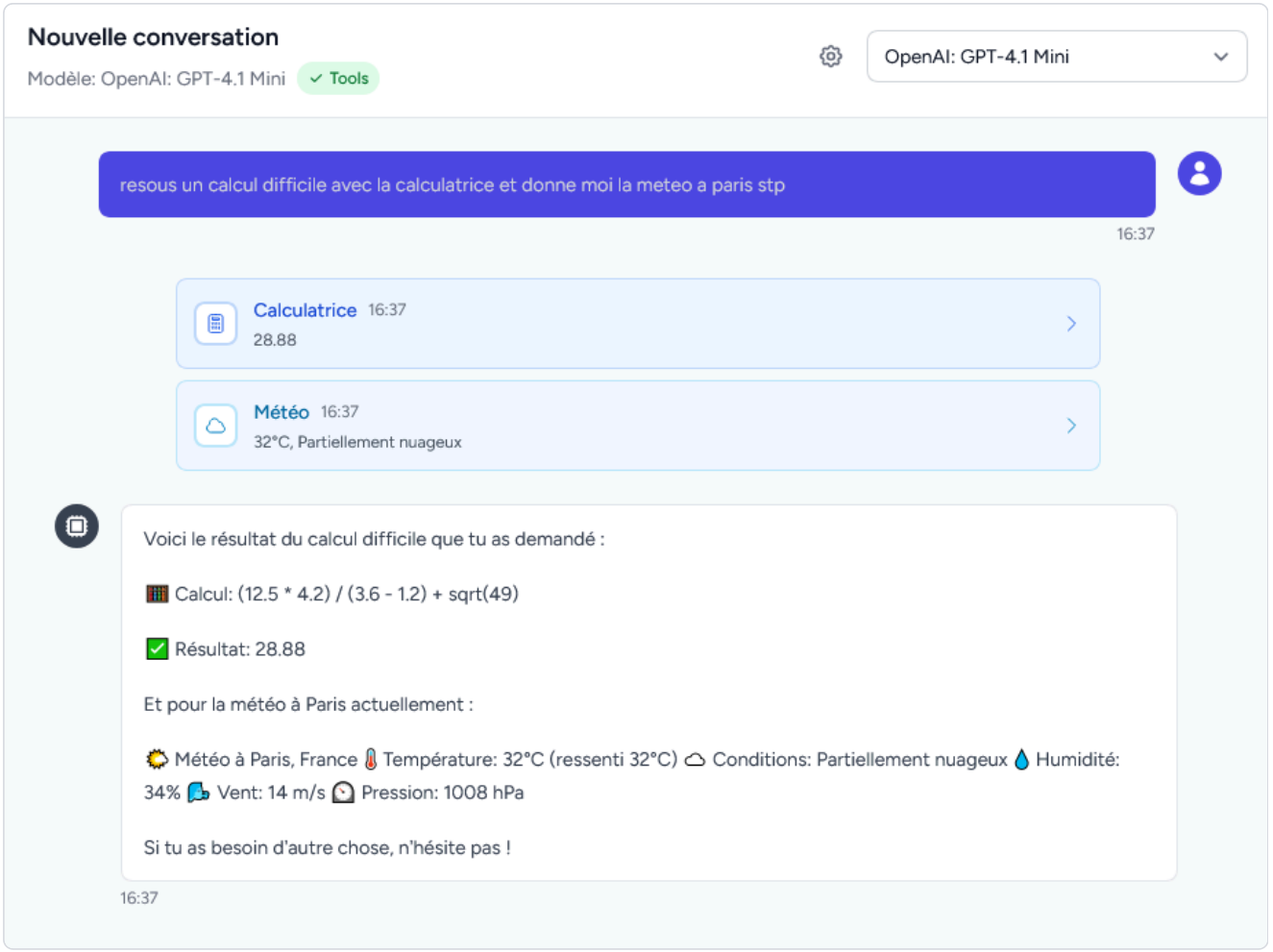
< 2 / 2 >

ChatGPT



L'architecture de base : Comment organiser ce bordel 🏗️

Bon, fini de faire mumuse avec des concepts abstraits. On va coder pour de vrai maintenant ! (▀_▀)



Le tool spécial : DirectResponse 🎯

Avant de parler des vrais tools, il faut comprendre un truc important : on donne toujours au LLM l'option de répondre directement SANS utiliser de tools. Pour ça, on a créé un tool spécial :

```php
class DirectResponseTool implements ToolInterface
{
    public function getName(): string
    {
        return 'direct_response';
    }

    public function getDescription(): string
    {
        return 'Utilise ce tool quand tu peux répondre directement sans avoir besoin
d\'informations externes';
    }
}
```

Comme ça, le LLM a le choix : soit il utilise des tools externes (météo, calcul...), soit il répond directement avec ses connaissances. C'est lui qui décide ! 🤓

L'interface ToolInterface : Le contrat sacré 📜

Première règle du fight club des tools : tous les tools doivent respecter le même contrat. En PHP, on définit une interface :

```php
interface ToolInterface
{
    public function getName(): string;              // "get_weather", "calculator", etc.
    public function getDescription(): string;       // Pour que le LLM comprenne à quoi ça sert
    public function getParametersSchema(): array;   // JSON Schema des paramètres
    public function execute(array $parameters): array;  // La magie opère ici
    public function isEnabledByDefault(): bool;     // Actif ou pas ? That is the question
}
```

Voilà, c'est tout ! Si ton tool ne respecte pas ça, il peut aller se faire voir ailleurs 🚮

Le ToolManager : Le chef d'orchestre qui fait tout le boulot 🎭

Le `ToolManager`, c'est comme un manager de boîte de nuit : il connaît tous les videurs (tools), sait lesquels sont disponibles, et les fait bosser quand il faut.

```php
class ToolManager
{
    private Collection
tools;//NotrecollectiondetoolspublicfunctionregisterTool(ToolInterfacetools;  // Notre collection
de tools

    public function registerTool(ToolInterface
tools;//NotrecollectiondetoolspublicfunctionregisterTool(ToolInterfacetool): void
    {
        this->tools->put(this→tools→put(this->tools->put(tool→getName(), tool);
    }

    public function getToolsForAPI(array enabledToolNames = []): array
    {
        // Convertit les tools au format OpenAI/OpenRouter
        // Parce que oui, il faut leur mâcher le travail...
    }
}
```

Exemple concret : Le WeatherTool 🌦️

Parce qu'apparemment, les LLMs ne peuvent pas regarder par la fenêtre :

```php
class WeatherTool implements ToolInterface
{
    public function getName(): string
    {
        return 'get_weather';
    }

    public function getParametersSchema(): array
    {
        return [
            'type' => 'object',
            'properties' => [
                'location' => [
                    'type' => 'string',
                    'description' => 'La ville (ex: "Wavre")'
                ],
            ],
            'required' => ['location'],
        ];
    }
}
```

Le flow magique : Comment le LLM devient moins bête 🪄

Étape 1 : "Choisis ton arme !" ⚔️

Quand un utilisateur pose une question, on envoie TOUT au LLM :

- Les messages de la conversation
- La liste des tools disponibles
- Un petit "tool_choice": "auto" (traduit : "débrouille-toi")

```php
toolChoiceResponse=toolChoiceResponse = toolChoiceResponse=this→client→chat()→create([
    'model' => model,'messages'⇒model,
    'messages' => model,'messages'⇒allMessages,
    'tools' => this->toolManager->getToolsForAPI(this→toolManager
→getToolsForAPI(this->toolManager->getToolsForAPI(enabledTools),
    'tool_choice' => 'auto',  // Le LLM fait son malin tout seul
]);
```

Le LLM répond soit :

- "J'ai pas besoin de tools, je suis trop intelligent" → `direct_response`
- "Donne-moi la météo stp" → `toolCalls` avec les détails

Étape 2 : Exécution des tools et compilation des résultats 🔨

Si le LLM demande des tools, on les exécute :

```php
foreach (choice->message->toolCallsaschoice→message→toolCalls as
choice->message->toolCallsastoolCall) {
    result=result = result=this→toolManager→executeTool(
        toolCall->function->name,jsondecode(toolCall→function→name,
        json_decode(toolCall->function->name,jsondecode(toolCall→function→arguments, true)
    );

    // On stocke le résultat pour le renvoyer au LLM après
    toolResults[]=['role'⇒'tool','toolcallid'⇒toolResults[] = [
        'role' ⇒ 'tool',
        'tool_call_id' ⇒ toolResults[]=['role'⇒'tool','toolcallid'⇒toolCall→id,
        'content' ⇒ json_encode($result),
    ];
}
```

Puis on renvoie TOUT au LLM : "Tiens, voilà tes données, maintenant fais une belle phrase !" 🎯

Le streaming : Parce que personne n'aime attendre 🚀

Les générateurs PHP : yield > return

Un générateur en PHP, c'est comme un distributeur de bonbons : il donne un bonbon à la fois au lieu de vider tout le paquet dans ta bouche ! 🍬

```php
function sendMessageWithToolsStream(...): \Generator
{
    // Au lieu de return, on yield progressivement
    yield json_encode(['type' ⇒ 'tool_call_start', 'data' ⇒ [...]]) . "\n";
    yield json_encode(['type' ⇒ 'tool_call_result', 'data' ⇒ [...]]) . "\n";
    yield json_encode(['type' ⇒ 'message_chunk', 'data' ⇒ $chunk]) . "\n";
}
```

Les events JSON : Le front-end adore ça 📡

On envoie différents types d'events au navigateur :

```json
{"type": "tool_call_start", "data": {"tool_name": "get_weather", "parameters": {...}}}
{"type": "tool_call_result", "data": {"result": "Il pleut à Wavre"}}
{"type": "message_chunk", "data": "Il fait "}
{"type": "message_chunk", "data": "15°C à Wavre"}
{"type": "message_end"}
```

Le front peut ainsi afficher en temps réel : "Le LLM appelle la météo... ⏳" puis la réponse qui arrive mot par mot. Magique ! ✨

Dans le controller : Le chef d'orchestre du streaming 🎼

```php
return response()->stream(function () use (...) {
    eventStream=eventStream = eventStream=this->chatService->sendMessageWithToolsStream(...);

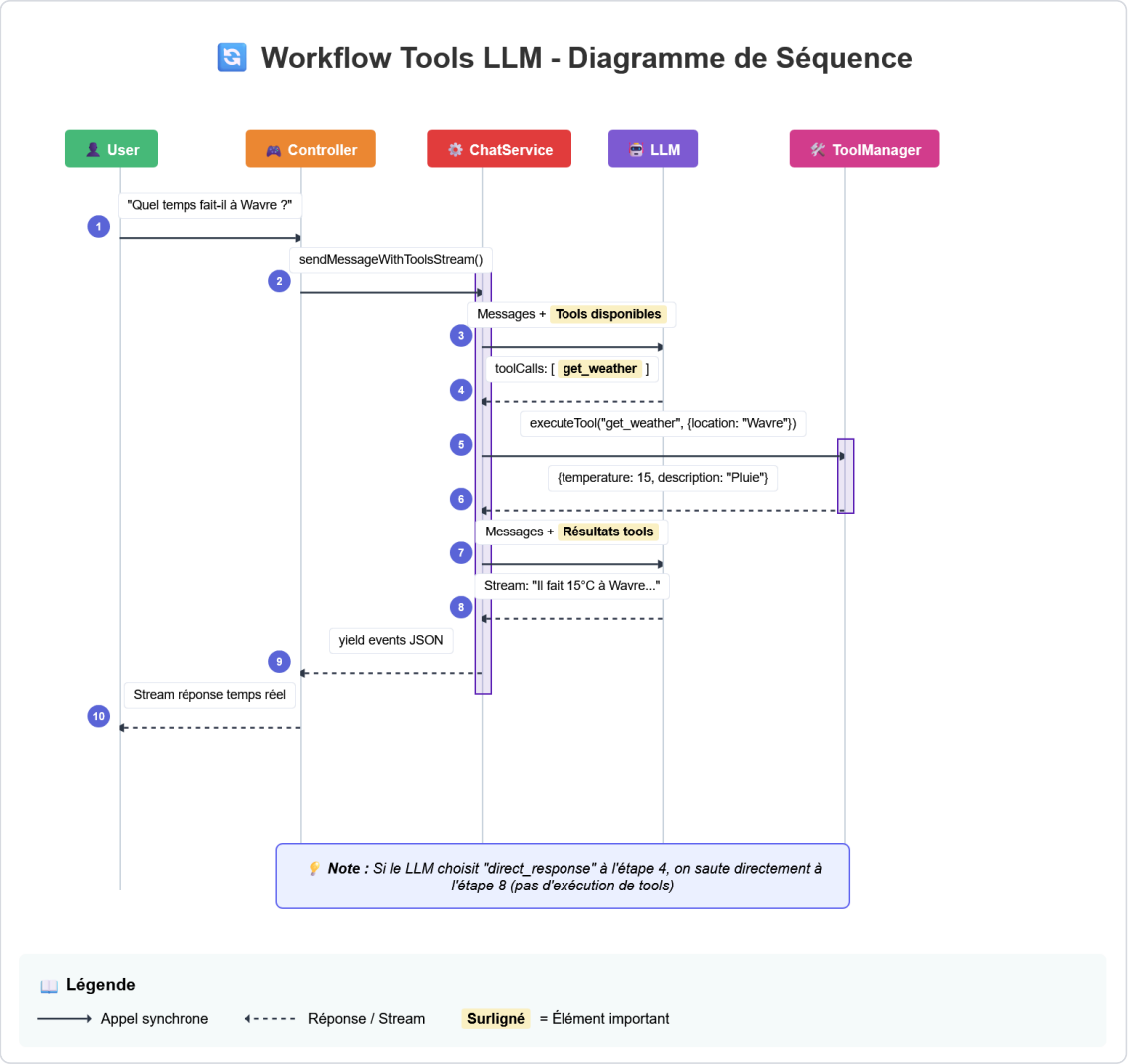
    foreach (eventStreamaseventStream as eventStreamaseventJson) {
        echo $eventJson;
        ob_flush();  // PHP, sors ça de ton buffer !
        flush();     // Envoie au navigateur, MAINTENANT !
    }
}, 200, [
    'Content-Type' => 'text/plain; charset=utf-8',
    'X-Accel-Buffering' => 'no',  // Nginx, touche pas à mon stream !
]);
```

Exemple concret : "Quel temps fait-il à Wavre ?" 🌧️

1. **User** : "Quel temps fait-il à Wavre ?"
2. **LLM** : "Hmm, j'ai besoin du tool météo"

   ```json
   {"tool_calls": [{"function": {"name": "get_weather", "arguments": "{\"location\":
   \"Wavre\"}"}}]}
   ```

3. **Notre code** : *Appelle l'API météo* → "15°C, pluie"
4. **LLM avec les résultats** : "Il fait actuellement 15°C à Wavre avec de la pluie. Typique de la Belgique ! 🌧️"



Vers un agent autonome : Le boss final 🤖

Un agent, c'est un LLM sous stéroïdes qui peut :

La boucle de l'agent 🔄

```
while (objectif_non_atteint) {
    1. Réfléchir : "Qu'est-ce que je dois faire ?"
    2. Agir : Utiliser un ou plusieurs tools
    3. Observer : Analyser les résultats
    4. Décider : Continuer ou s'arrêter ?
}
```

La mémoire comme tool 🧠

Dans un agent sophistiqué, même la mémoire devient un tool ! L'agent peut :

- **Sauvegarder** des informations : `memory_save("hotel_paris", "Hôtel Le Marais - 120€/nuit")`
- **Récupérer** des informations : `memory_retrieve("hotel_paris")`
- **Chercher** dans sa mémoire : `memory_search("tous les hôtels")`

```php
class MemoryTool implements ToolInterface
{
    private storage=[];publicfunctionexecute(arraystorage = [];

    public function execute(array storage=[];publicfunctionexecute(arrayparameters): array
    {
        switch ($parameters['action']) {
            case 'save':
                $this->storage[$parameters['key']] = $parameters['value'];
                return ['status' => 'saved'];
            case 'retrieve':
                return ['value' => $this->storage[$parameters['key']] ?? null];
            case 'search':
                // Recherche intelligente dans la mémoire
                return ['results' => $this->searchMemory($parameters['query'])];
        }
    }
}
```

Exemples d'agents spécialisés 🦾

1. Agent Web Browser 🌐

Cet agent peut naviguer sur le web comme un humain :

```php
webAgent=newAgent([newNavigateTool(),//Allera`uneURLnewClickTool(),//Cliquersurune´le´mentnewTypeT
ool(),//TaperdutextenewExtractTool(),//Extrairedesdonne´esnewScreenshotTool(),//Prendreunecapturen
ewMemoryTool()//Sesouvenirdecequ'iltrouve]);//Exempledeta^chewebAgent = new Agent([
    new NavigateTool(),      // Aller à une URL
    new ClickTool(),         // Cliquer sur un élément
    new TypeTool(),          // Taper du texte
    new ExtractTool(),       // Extraire des données
    new ScreenshotTool(),    // Prendre une capture
    new MemoryTool()         // Se souvenir de ce qu'il trouve
]);

// Exemple de tâche
webAgent=newAgent([newNavigateTool(),//Allera`uneURLnewClickTool(),//Cliquersurune´le´mentnewTypeT
ool(),//TaperdutextenewExtractTool(),//Extrairedesdonne´esnewScreenshotTool(),//Prendreunecapturen
ewMemoryTool()//Sesouvenirdecequ'iltrouve]);//Exempledeta^chewebAgent->run("Trouve le prix du
dernier iPhone sur le site d'Apple");
// L'agent va : naviguer → chercher → cliquer → extraire → mémoriser
```

2. Agent Développeur 👨‍💻

Cet agent peut écrire et débugger du code :

```php
devAgent=newAgent([newWriteCodeTool(),//E´crireducodenewRunCodeTool(),//Exe´cuterettesternewDebugTool(),//AnalyserleserreursnewRefactorTool(),//Ame´liorerlecodenewGitTool(),//CommitetpushnewMemoryTool()//Sesouvenirdesbugsrencontre´s]);//Exempledeta^chedevAgent = new Agent([
    new WriteCodeTool(),      // Écrire du code
    new RunCodeTool(),        // Exécuter et tester
    new DebugTool(),          // Analyser les erreurs
    new RefactorTool(),       // Améliorer le code
    new GitTool(),            // Commit et push
    new MemoryTool()          // Se souvenir des bugs rencontrés
]);

// Exemple de tâche
devAgent=newAgent([newWriteCodeTool(),//E´crireducodenewRunCodeTool(),//Exe´cuterettesternewDebugTool(),//AnalyserleserreursnewRefactorTool(),//Ame´liorerlecodenewGitTool(),//CommitetpushnewMemoryTool()//Sesouvenirdesbugsrencontre´s]);//Exempledeta^chedevAgent->run("Crée une API REST pour gérer des todos en Laravel");
// L'agent va : planifier → coder → tester → corriger → optimiser → versionner
```

Architecture complète d'un agent 🏛️

```php
class Agent {
    private array tools;privateLLMInterfacetools;
    private LLMInterface tools;privateLLMInterfacellm;
    private int maxIterations=10;publicfunctionrun(stringmaxIterations = 10;

    public function run(string maxIterations=10;publicfunctionrun(stringobjective): array
    {
        iteration=0;iteration = 0;
        iteration=0;completed = false;
        results=[];while(!results = [];

        while (!results=[];while(!completed && iteration<iteration < iteration<this->maxIterations) {
            // 1. Planification
            plan=plan = plan=this->llm->plan(objective,objective, objective,results);

            // 2. Sélection du tool
            toolChoice=toolChoice = toolChoice=this->llm->selectTool(plan,plan, plan,this->tools);

            // 3. Exécution
            toolResult=toolResult = toolResult=this->executeTool(toolChoice);toolChoice);
            toolChoice);results[] = toolResult;//4.E´valuationtoolResult;

            // 4. Évaluation
            toolResult;//4.E´valuationcompleted = this->llm->isObjectiveCompleted(this->llm->isObjectiveCompleted(this->llm->isObjectiveCompleted(objective, results);results);

            results);iteration++;
        }

        return $results;
    }
}
```

Les patterns d'agents avancés 🎭

- **Multi-agents** : Plusieurs agents qui collaborent
- **Hierarchical agents** : Un agent manager qui délègue à des agents spécialisés
- **Self-improving agents** : Qui apprennent de leurs erreurs (via la mémoire)

Conclusion : T'as compris ou pas ? 🎓

Voilà, maintenant tu sais :

- ✅ Comment créer des tools pour ton LLM
- ✅ Comment les faire communiquer ensemble
- ✅ Comment streamer tout ça proprement
- ✅ Comment transformer ton LLM en agent autonome

Les tools transforment complètement les capacités des LLMs. Ils passent de simples générateurs de texte à de véritables assistants capables d'interagir avec le monde réel. Et avec les agents, on entre dans une nouvelle dimension où l'IA peut vraiment automatiser des tâches complexes ! 🚀

4.14 Annexe : Formatage des Messages pour l'API

Cette annexe détaille la structure attendue pour les messages envoyés à l'API de chat, avec des exemples concrets en PHP.

1. Structure Générale d'une Requête

1.1 Format PHP de Base

```php
$request = [
    // OBLIGATOIRE : soit "messages" soit "prompt"
    'messages' => [...], // Array de messages (recommandé)
    // OU
    'prompt' => 'Votre prompt direct', // String simple

    // Optionnel : modèle à utiliser
    'model' => 'anthropic/claude-3.5-sonnet', // Par défaut : modèle utilisateur

    // Paramètres de streaming
    'stream' => true, // Pour activer le streaming

    // Paramètres LLM
    'max_tokens' => 1000,
    'temperature' => 0.7, // Entre 0 et 2
    'top_p' => 0.9,       // Entre 0 et 1

    // Contrôle du format de sortie
    'response_format' => ['type' => 'json_object'], // Force JSON

    // Mots d'arrêt
    'stop' => ['```', 'FIN'], // String ou array de strings
];
```

2. Types de Messages

2.1 Message Utilisateur Simple

```php
$userMessage = [
    'role' => 'user',
    'content' => 'Bonjour, comment allez-vous ?'
];
```

2.2 Message Système

```php
$systemMessage = [
    'role' => 'system',
    'content' => 'Vous êtes un assistant spécialisé en développement web.'
];
```

2.3 Message Assistant

```php
$assistantMessage = [
    'role' => 'assistant',
    'content' => 'Bonjour ! Je vais bien, merci. Comment puis-je vous aider ?'
];
```

2.4 Message avec Nom (Persona)

```php
$namedMessage = [
    'role' => 'user',
    'content' => 'Quelle est la météo aujourd\'hui ?',
    'name' => 'Alice' // Sera affiché comme "Alice: Quelle est la météo..."
];
```

3. Messages Multimodaux (Texte + Images)

3.1 Structure ContentPart

Pour les messages utilisateur avec images, le `content` devient un array de "parts" :

```php
$multimodalMessage = [
    'role' => 'user',
    'content' => [
        // Part texte
        [
            'type' => 'text',
            'text' => 'Peux-tu analyser cette image ?'
        ],
        // Part image
        [
            'type' => 'image_url',
            'image_url' => [
                'url' => 'https://example.com/image.jpg',
                'detail' => 'high' // 'low', 'high', ou 'auto' (défaut)
            ]
        ]
    ]
];
```

3.2 Image Base64

```php
$base64Message = [
    'role' => 'user',
    'content' => [
        [
            'type' => 'text',
            'text' => 'Décris cette capture d\'écran'
        ],
        [
            'type' => 'image_url',
            'image_url' => [
                'url' => '...',
                'detail' => 'auto'
            ]
        ]
    ]
];
```

3.3 Plusieurs Images

```php
$multipleImagesMessage = [
    'role' => 'user',
    'content' => [
        [
            'type' => 'text',
            'text' => 'Compare ces deux images :'
        ],
        [
            'type' => 'image_url',
            'image_url' => ['url' => 'https://example.com/image1.jpg']
        ],
        [
            'type' => 'image_url',
            'image_url' => ['url' => 'https://example.com/image2.jpg']
        ]
    ]
];
```

4. Messages avec Tool Calling

4.1 Définition d'un Tool

```php
$weatherTool = [
    'type' => 'function',
    'function' => [
        'name' => 'get_weather',
        'description' => 'Obtient la météo pour une ville donnée',
        'parameters' => [
            'type' => 'object',
            'properties' => [
                'city' => [
                    'type' => 'string',
                    'description' => 'Le nom de la ville'
                ],
                'unit' => [
                    'type' => 'string',
                    'enum' => ['celsius', 'fahrenheit'],
                    'description' => 'L\'unité de température'
                ]
            ],
            'required' => ['city']
        ]
    ]
];
```

4.2 Requête avec Tools

```php
requestWithTools=['messages'⇒[['role'⇒'system','content'⇒'Vouse^tesunassistantme´te´o.'],
['role'⇒'user','content'⇒'Queltempsfait-ila`Paris?']],'tools'⇒[requestWithTools = [
    'messages' => [
        [
            'role' => 'system',
            'content' => 'Vous êtes un assistant météo.'
        ],
        [
            'role' => 'user',
            'content' => 'Quel temps fait-il à Paris ?'
        ]
    ],
    'tools' => [requestWithTools=['messages'⇒[['role'⇒'system',
'content'⇒'Vouse^tesunassistantme´te´o.'],['role'⇒'user','content'⇒'Queltempsfait-ila`Paris?
']],'tools'⇒[weatherTool],
    'tool_choice' => 'auto' // 'none', 'auto', ou objet spécifique
];
```

4.3 Message Tool Response

```php
$toolMessage = [
    'role' => 'tool',
    'content' => '{"temperature": 22, "condition": "ensoleillé", "humidity": 65}',
    'tool_call_id' => 'call_abc123',
    'name' => 'get_weather'
];
```

5. Exemples Complets de Conversations

5.1 Conversation Simple

```php
$simpleConversation = [
    'messages' => [
        [
            'role' => 'system',
            'content' => 'Vous êtes un développeur PHP expert.'
        ],
        [
            'role' => 'user',
            'content' => 'Comment faire une requête HTTP en PHP ?'
        ],
        [
            'role' => 'assistant',
            'content' => 'Vous pouvez utiliser cURL ou file_get_contents()...'
        ],
        [
            'role' => 'user',
            'content' => 'Peux-tu me donner un exemple avec cURL ?'
        ]
    ],
    'model' => 'anthropic/claude-3.5-sonnet',
    'temperature' => 0.7,
    'max_tokens' => 1000,
    'stream' => true
];
```

5.2 Conversation avec Images

```php
$imageConversation = [
    'messages' => [
        [
            'role' => 'system',
            'content' => 'Vous analysez des interfaces utilisateur.'
        ],
        [
            'role' => 'user',
            'content' => [
                [
                    'type' => 'text',
                    'text' => 'Analyse cette maquette et donne-moi des suggestions d\'amélioration
:'
                ],
                [
                    'type' => 'image_url',
                    'image_url' => [
                        'url' => 'https://example.com/mockup.png',
                        'detail' => 'high'
                    ]
                ]
            ]
        ]
    ],
    'model' => 'anthropic/claude-3.5-sonnet',
    'temperature' => 0.3
];
```

5.3 Conversation avec Tool Calling

```php
$toolConversation = [
    'messages' => [
        [
            'role' => 'system',
            'content' => 'Vous êtes un assistant capable d\'effectuer des calculs.'
        ],
        [
            'role' => 'user',
            'content' => 'Calcule 15% de 250 et convertis le résultat en euros.'
        ]
    ],
    'tools' => [
        [
            'type' => 'function',
            'function' => [
                'name' => 'calculate',
                'description' => 'Effectue des calculs mathématiques',
                'parameters' => [
                    'type' => 'object',
                    'properties' => [
                        'expression' => [
                            'type' => 'string',
                            'description' => 'Expression mathématique à calculer'
                        ]
                    ],
                    'required' => ['expression']
                ]
            ]
        ],
        [
            'type' => 'function',
            'function' => [
                'name' => 'convert_currency',
                'description' => 'Convertit une devise',
                'parameters' => [
                    'type' => 'object',
                    'properties' => [
                        'amount' => ['type' => 'number'],
                        'from' => ['type' => 'string'],
                        'to' => ['type' => 'string']
                    ],
                    'required' => ['amount', 'from', 'to']
                ]
            ]
        ]
    ],
    'tool_choice' => 'auto'
];
```

Source

- https://openrouter.ai/docs/api-reference/overview#requests