Rapport
PHP Mastery Sprint

Fondamentales de PHP, Architecture MVC et Framework-from-scratch

Réalisé par

Hamza Lazaar

Encadré par

Abdelmajid Bendrif
Hamza Bahlaouane

PFE 2026 – 20 février

Table des matières

Chapitre 1 : Architecture MVC – L'Approche "From Scratch"

Pour comprendre les fondations des frameworks modernes comme Laravel, j'ai débuté par la création d'un framework MVC (Modèle-Vue-Contrôleur) artisanal en PHP pur. Cette démarche, basée sur le tutoriel d'Anant Garg, vise à démystifier le fonctionnement du routage et de la séparation des responsabilités.

1.1. Structure du Projet et Standards

La première étape a consisté à mettre en place une arborescence claire séparant le dossier public (accessible au web) du cœur de l'application (logique métier).

Pourquoi cette structure ?
Isoler le code source dans un dossier parent permet de sécuriser l'application : seul le fichier `index.php` est exposé au serveur web.

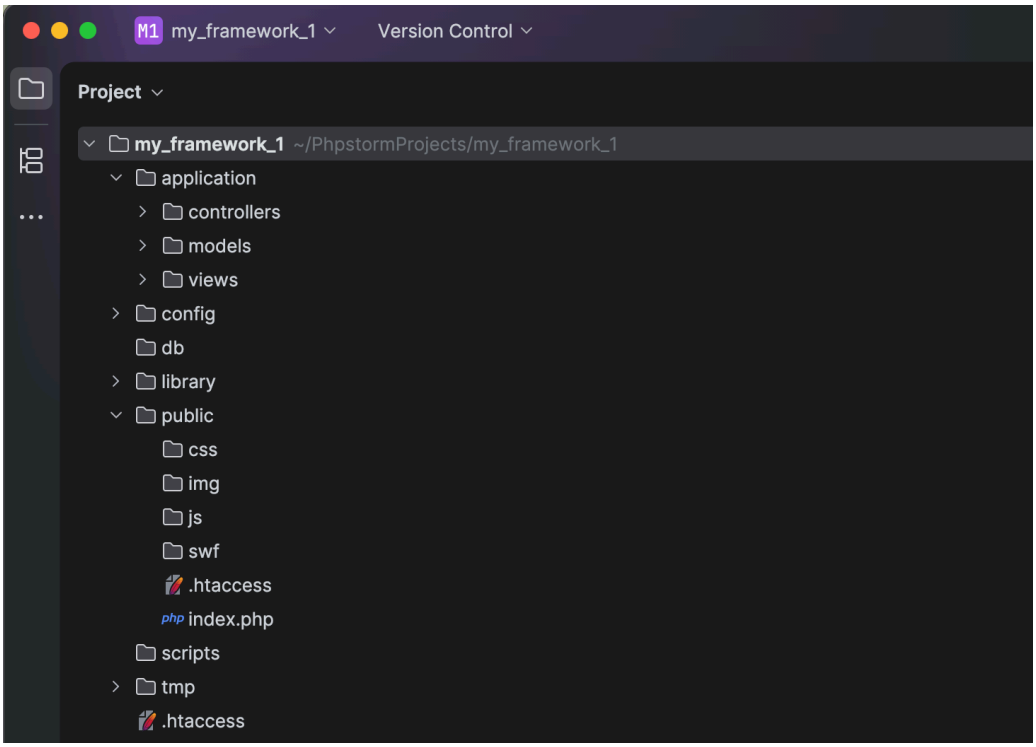

FIGURE 1.1 – Arborescence du projet MVC

1.2. Le Bootstrapping (Point d'Entrée)

Le fichier `public/index.php` agit comme l'unique point d'entrée. J'ai configuré le serveur (via `.htaccess`) pour rediriger toutes les requêtes vers ce fichier.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ index.php?url=$1 [QSA,L]
```

Listing 1.1 – Configuration du .htaccess pour la réécriture d'URL

1.3. Le Système de Routage (Convention over Configuration)

Contrairement à une approche où chaque route est définie manuellement, j'ai implémenté un routage dynamique basé sur des conventions strictes. Le cœur de ce système réside dans la fonction `callHook()`, qui interprète automatiquement l'URL selon le schéma : `/Controleur/Action/Arguments`.

Le processus se déroule en trois étapes :

1. **Découpage :** L'URL est explosée en segments via la fonction PHP `explode()`.
2. **Déduction :** Le script détermine dynamiquement quel fichier inclure et quelle classe instancier (ex : `items` devient `ItemsController`).
3. **Dispatching :** La méthode cible est appelée via `call_user_func_array()`, passant les arguments restants.

```php
// shared.php

function callHook() {
    global $url;

    // 1. Définition de la route par défaut
    if (empty($url)) {
        $url = 'items/viewall';
    }

    // 2. Découpage de l'URL
    $urlArray = explode("/", $url);
    $controller = $urlArray[0];
    array_shift($urlArray);
    $action = $urlArray[0];
    array_shift($urlArray);
    $queryString = $urlArray;

    // 3. Construction du nom du Contrôleur (ex: ItemsController)
    $controllerName = $controller;
    $controller = ucwords($controller);
    $model = rtrim($controller, 's'); // Convention : Items -> Item
    $controller .= 'Controller';

```

```
    // 4. Instanciation et appel dynamique
    $dispatch = new $controller($model, $controllerName, $action);

    if ((int)method_exists($controller, $action)) {
        call_user_func_array(array($dispatch, $action), $queryString);
    } else {
        // Gestion des erreurs 404 ici
    }
}
```

Listing 1.2 – La fonction callHook responsable du routage dynamique

Analyse Technique : Cette approche est très rapide à mettre en place (pas de fichier de configuration de routes), mais elle oblige à respecter strictement la nomenclature des fichiers et des classes.

1.4. La Couche Contrôleur et le Cycle de Vie

L'architecture du framework repose sur une classe parente `Controller` qui orchestre tout le cycle de vie de la requête. J'ai utilisé l'héritage pour que chaque contrôleur spécifique (ex : `ItemsController`) bénéficie automatiquement de deux automatismes puissants : l'injection du modèle et le rendu de la vue.

1.4.1. Le Contrôleur de Base (Core)

La classe abstraite `Controller` agit comme un chef d'orchestre caché.

— **Constructeur (`__construct`) :** Il instancie dynamiquement le modèle associé au contrôleur (si je suis dans `ItemsController`, il charge `Item`).

— **Destructeur (`__destruct`) :** Il déclenche automatiquement le rendu du template à la fin de l'exécution du script.

```
#[AllowDynamicProperties] // Support PHP 8.2+
class Controller {

    protected $_model;
    protected $_controller;
    protected $_action;
    protected $_template;

    function __construct($model, $controller, $action) {
        $this->_controller = $controller;
        $this->_action = $action;
        $this->_model = $model;

        // Instanciation dynamique du Modèle (Magic)
        $this->$model = new $model;

        // Initialisation du moteur de template
        $this->_template = new Template($controller, $action);
    }

```

```php
    // Passe les variables à la vue
    function set($name, $value) {
        $this->_template->set($name, $value);
    }

    // Rendu automatique à la fin du script
    function __destruct() {
        $this->_template->render();
    }
}
```

Listing 1.3 – La classe parente Controller et son cycle de vie

1.4.2. Implémentation Métier (ItemsController)

Grâce à cette abstraction, le contrôleur final (`ItemsController`) est extrêmement épuré. Il se concentre uniquement sur la logique métier : récupérer les données via `$this->Item` et les passer à la vue via `$this->set()`.

```php
class ItemsController extends Controller {

    function view($id = null, $name = null) {
        $this->set('title', $name . ' - My Todo List App');
        // Appel direct au modèle injecté automatiquement
        $this->set('todo', $this->Item->select($id));
    }

    function viewall() {
        $this->set('title', 'All Items - My Todo List App');
        $this->set('todo', $this->Item->selectAll());
    }

    function add() {
        $todo = $_POST['todo'];
        $this->set('title', 'Success - My Todo List App');
        // Requête SQL via le modèle
        $this->set('todo', $this->Item->query('insert into items (
            item_name) values (?)', [$todo]));
    }

    function delete($id = null) {
        $this->set('title', 'Success - My Todo List App');
        $this->set('todo', $this->Item->query('delete from items where
            id = ?', [$id]));
    }
}
```

Listing 1.4 – Exemple d'un contrôleur CRUD complet

Note sur PHP 8.2 : J'ai dû ajouter l'attribut `#[AllowDynamicProperties]` sur la classe parente. En effet, la création dynamique de la propriété `$this->Item` (qui n'est pas déclarée explicitement) est dépréciée dans les versions récentes de PHP sans cet attribut.

1.5. Publication sur Packagist et Gestion des Dépendances

Afin d'assimiler complètement l'écosystème PHP moderne et de comprendre le cycle de vie d'une librairie, j'ai procédé à la publication de mon code sous forme de package sur **Packagist**, le dépôt central utilisé par Composer.

Cette étape a nécessité la configuration d'un fichier `composer.json` à la racine du projet. Ce fichier définit non seulement les métadonnées du package (nom, auteur, description), mais configure surtout les règles d'**autoloading** (standard PSR-4). C'est ce mécanisme qui permet d'éliminer les inclusions manuelles (`require_once`) au profit d'un chargement automatique des classes.

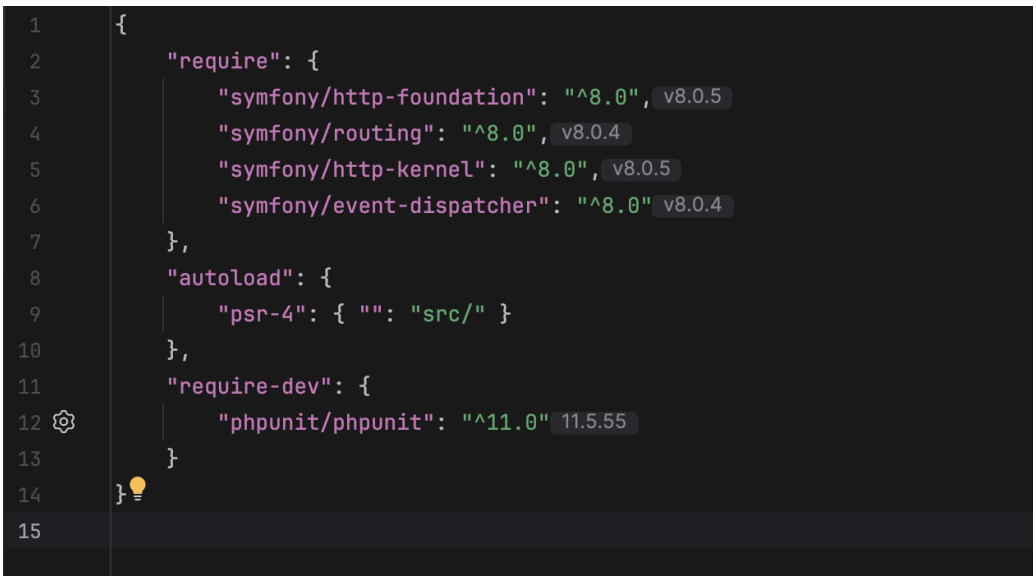

FIGURE 1.2 – Le fichier composer.json

Une fois le dépôt GitHub lié à Packagist, j'ai testé l'intégration du package dans un projet vierge. Lors de l'installation, j'ai été confronté aux contraintes de stabilité de Composer (*minimum-stability*). Par défaut, Composer refuse d'installer des paquets en cours de développement sans version explicite.

Pour résoudre ce problème de dépendance, j'ai exploré deux approches :

— L'installation explicite de la branche de développement.

— L'application des bonnes pratiques de **Semantic Versioning** en créant un tag Git (`v1.0.0`) pour marquer une *release* stable, rendant ainsi le package standardisé pour la production. (C'est cette approche que j'ai preféré suivre)

```
# Exemple d'installation en ciblant la branche de développement
composer require hamza_void/my_php_framework
```

Listing 1.5 – Commande d'installation du package via Composer

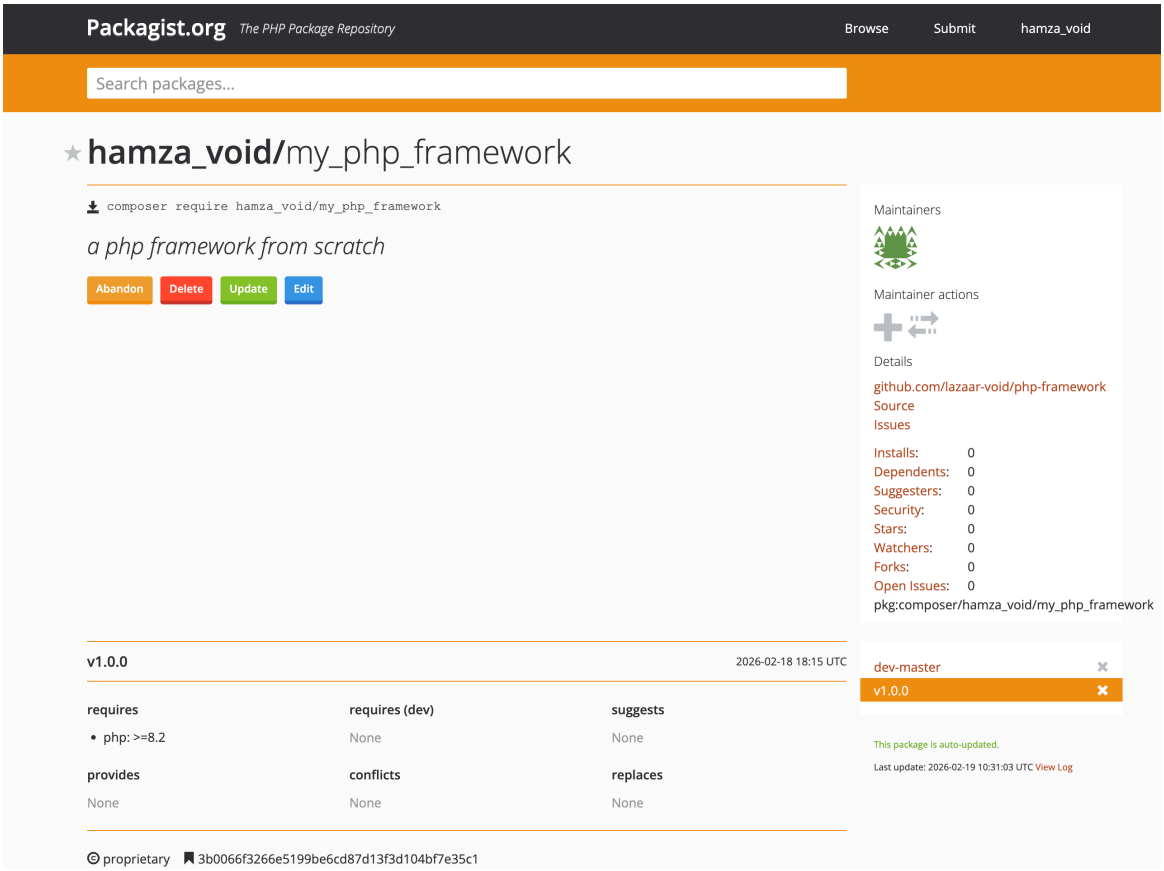

FIGURE 1.3 – Package publié et disponible publiquement sur Packagist

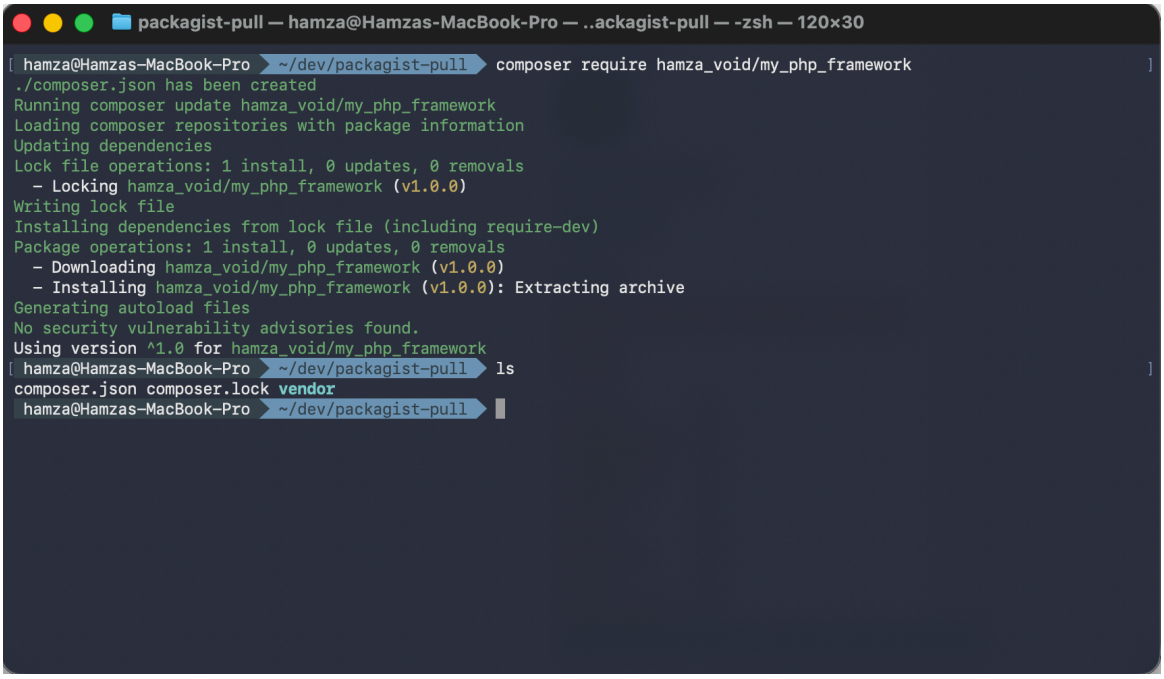

FIGURE 1.4 – Package installé localement avec composer

Chapitre 2 : Reconstruction Modulaire avec Symfony

Après avoir compris la théorie avec l'approche "From Scratch", je suis passé à une approche industrielle en utilisant les composants découplés de Symfony. L'objectif est de remplacer mon code artisanal par des composants robustes et testés, simulant le fonctionnement interne de Laravel.

2.1. Abstraction HTTP (HttpFoundation)

Au lieu d'utiliser les superglobales brutes (`$_GET`, `$_SERVER`), j'ai intégré le composant `HttpFoundation`. Cela permet de traiter la requête et la réponse comme des objets orientés objet.

```php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();
$name = $request->query->get('name', 'World');
```

Listing 2.1 – Utilisation de l'objet Request

2.2. Le Front Controller

J'ai refondu le point d'entrée pour qu'il utilise ces objets. Le contrôleur frontal reçoit désormais un objet `Request` et doit impérativement retourner un objet `Response`.

2.3. Intégration du Routage (Routing Component)

J'ai remplacé mon routeur manuel par `UrlMatcher` de Symfony. Cela offre une gestion beaucoup plus puissante des URLs (paramètres nommés, contraintes regex, méthodes HTTP).

Avantage clé : Le composant Routing sépare la définition des routes (configuration) de leur exécution (matching), rendant l'application plus maintenable.

2.4. Le Contrôleur et le Noyau (HttpKernel)

L'étape finale a été d'utiliser `HttpKernel` pour orchestrer le tout. Le `ControllerResolver` se charge automatiquement de trouver la bonne méthode PHP à exécuter, et l'`ArgumentResolver` injecte les bons arguments.

Conclusion du Sprint : J'ai maintenant un framework fonctionnel qui partage la même architecture fondamentale que Drupal ou Laravel. Je comprends désormais ce que signifie "Bootstrapping" ou "Request Lifecycle" dans un contexte professionnel.

Chapitre 3 : Développement d'un Outil CLI d'Importation

L'automatisation du traitement des données est une compétence fondamentale en ingénierie backend. Ce chapitre détaille la conception d'une application en ligne de commande (CLI) permettant d'importer de larges fichiers CSV vers une base de données MySQL de manière sécurisée et ultra-performante.

3.1. Architecture Orientée Objet et Configuration

Pour structurer cette application, j'ai utilisé le composant `symfony/console`. Au fil du développement, l'architecture a évolué d'un script procédural vers une approche orientée objet respectant les principes **SOLID**. La logique métier a été découpée en services spécialisés :

- `Database.php` : Gestion de la connexion PDO.
- `CsvReader.php` : Encapsulation de la librairie `league/csv` pour la lecture.
- `CsvImporter.php` : Moteur d'importation gérant la validation et les requêtes.

Mise à jour majeure (Symfony 8.x) : Lors de l'instanciation de l'application, j'ai été confronté à une erreur fatale liée à l'absence de la méthode `add()`. Le débogage a révélé que la version 8.0 du composant a définitivement supprimé cette méthode historique au profit de `addCommand()`. Cette adaptation souligne l'importance d'auditer les dépendances lors d'une installation vierge.

FIGURE 3.1 – Structure du projet (Services) et implémentation de la commande sous PhpStorm

3.2. Sécurité des Accès et Interactivité

La sécurité des accès a été traitée à deux niveaux. Premièrement, l'importation ne s'effectue pas avec l'utilisateur `root`, mais via un utilisateur MySQL dédié (`csv_importer`) possédant des privilèges restreints.

```
CREATE USER 'csv_importer'@'localhost' IDENTIFIED BY '1234';
GRANT ALL PRIVILEGES ON csv_import_db.* TO 'csv_importer'@'localhost';
FLUSH PRIVILEGES;

CREATE TABLE IF NOT EXISTS customers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    csv_index INT NOT NULL,
```

```
    customer_id VARCHAR(50) UNIQUE NOT NULL,
    -- ... autres colonnes typées avec précision (VARCHAR, DATE)
);
```

Listing 3.1 – Création de l'utilisateur et définition du schéma

Deuxièmement, pour éviter de coder en dur (hardcoder) les identifiants dans le code source, j'ai implémenté `SymfonyStyle`. Cet outil permet de demander dynamiquement les accès à l'utilisateur lors de l'exécution, en masquant la saisie du mot de passe.

3.3. Validation, Batch Processing et Bulk Insert

L'un des défis majeurs de l'importation de gros fichiers est la gestion des ressources (RAM et temps d'exécution).

Lors de mes premiers tests, l'insertion s'effectuait ligne par ligne à l'intérieur d'une transaction. Bien que sécurisée par le principe **ACID** (qui annule tout en cas d'erreur grâce à `rollBack()`), cette méthode prenait environ 17 secondes à cause des multiples allers-retours avec le serveur MySQL.

J'ai donc combiné deux techniques d'optimisation d'entreprise :

1. Batch Processing (Optimisation RAM) : Le fichier CSV n'est pas chargé entièrement en mémoire. Les lignes sont lues et validées (exclusion des lignes vides ou mal formatées), puis stockées dans un tableau PHP par lots de 500.
2. Bulk Insert (Optimisation I/O) : Au lieu d'envoyer 500 requêtes `INSERT` individuelles, le script génère dynamiquement une requête SQL géante contenant 500 `VALUES`. Les temps de latence réseau et d'écriture disque sont ainsi drastiquement réduits.

```
private function insertBatch(array $batch): void {
    if (empty($batch)) return;

    $rowCount = count($batch);
    $columnsPerRow = 12;

    // Construction de la requête avec placeholders multiples
    $sql = "INSERT INTO customers (csv_index, customer_id, first_name,
       ...) VALUES ";
    $rowPlaceholders = "(" . implode(', ', array_fill(0, $columnsPerRow,
        '?')) . ")";
    $sql .= implode(', ', array_fill(0, $rowCount, $rowPlaceholders));

    // Aplatissement du tableau multidimensionnel
    $flatValues = [];
    foreach ($batch as $record) {
        array_push($flatValues, $record['Index'], $record['Customer Id'
           ], ...);
    }

    try {
        $this->pdo->beginTransaction();
        $stmt = $this->pdo->prepare($sql);
```

```
            $stmt->execute($flatValues); // Un seul appel réseau pour 500
                lignes
            $this->pdo->commit();
        } catch (\Exception $e) {
            $this->pdo->rollBack(); // Protection des données
        }
    }
```

Listing 3.2 – Génération dynamique de la requête de Bulk Insert

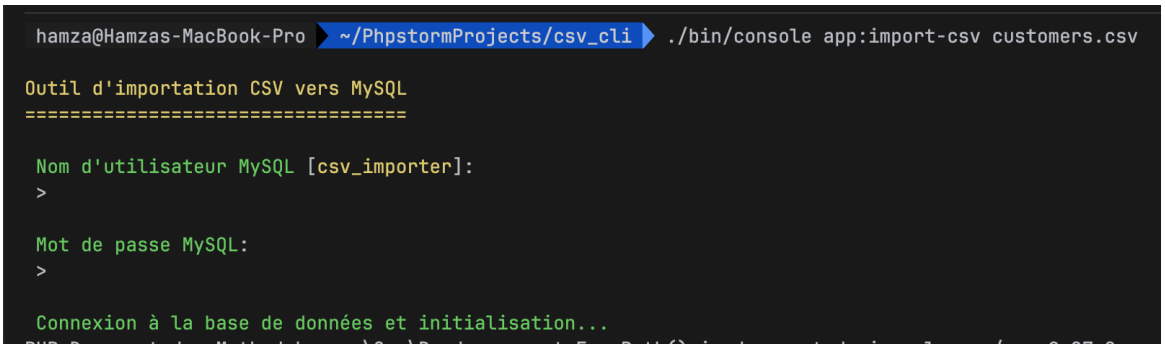

FIGURE 3.2 – Exécution interactive de l'outil CLI avec rapport détaillé du Bulk Insert [1]

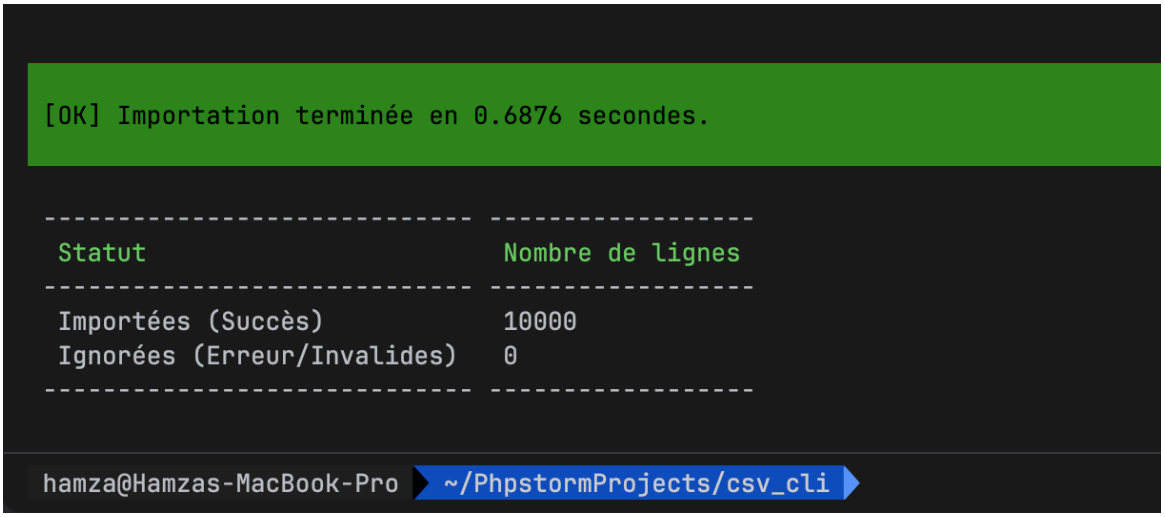

FIGURE 3.3 – Exécution interactive de l'outil CLI avec rapport détaillé du Bulk Insert [2]

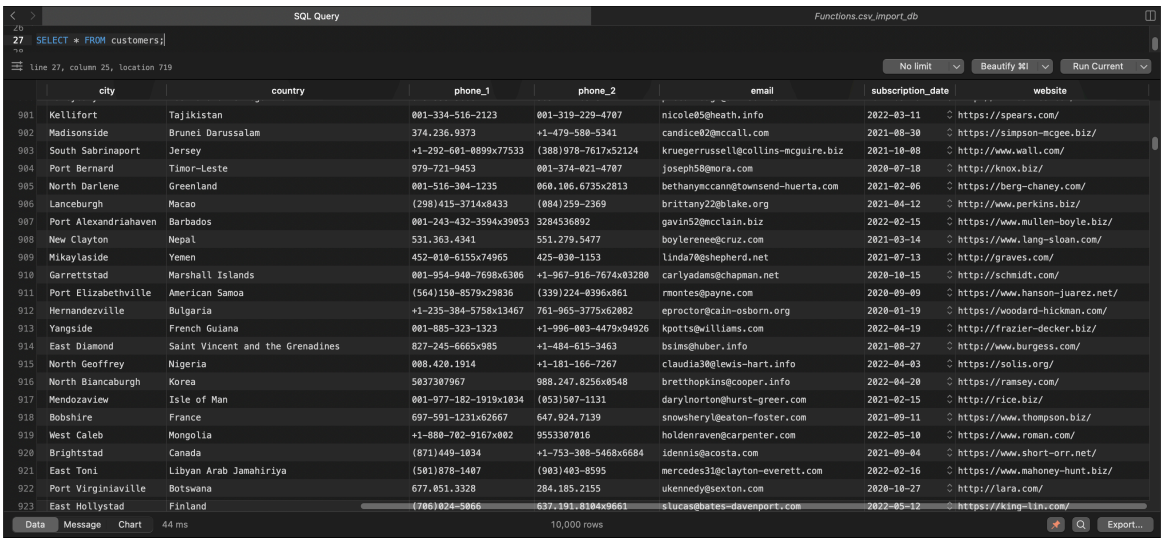

FIGURE 3.4 – Vérification de l'intégrité des données importées via TablePlus

3.4. Exportation, Compression et Restauration Locale

La dernière phase de ce flux de données consistait à extraire la base remplie, la compresser pour optimiser le stockage, puis tester sa restauration dans un nouvel environnement local.

Lors de l'utilisation de `mysqldump`, des contraintes de privilèges (liées à l'utilisateur restreint) sont apparues. Elles ont été résolues en ajustant les drapeaux d'exportation pour ignorer les tablespaces et les GTID de l'environnement local.

De plus, lors de la tentative de restauration, le système a généré une erreur d'accès (*Access denied*). Cette erreur a permis de valider la sécurité de notre architecture : l'utilisateur `csv_importer` étant strictement cloisonné à sa base d'origine, l'intervention du compte administrateur (`root`) a été requise pour restaurer les données dans la nouvelle base `csv_import_restored`.

```
# 1. Export propre sans nécessiter le privilège global PROCESS
mysqldump -u csv_importer -p --no-tablespaces --set-gtid-purged=OFF
    csv_import_db > export.sql

# 2. Compression du fichier SQL généré
gzip export.sql

# 3. Création d'une nouvelle base de données de destination (via root)
mysql -u root -p -e "CREATE DATABASE csv_import_restored;"

# 4. Restauration des données (décompression à la volée et injection via
    root)
gunzip < export.sql.gz | mysql -u root -p csv_import_restored
```

Listing 3.3 – Procédure de Dump Compression et Restauration locale

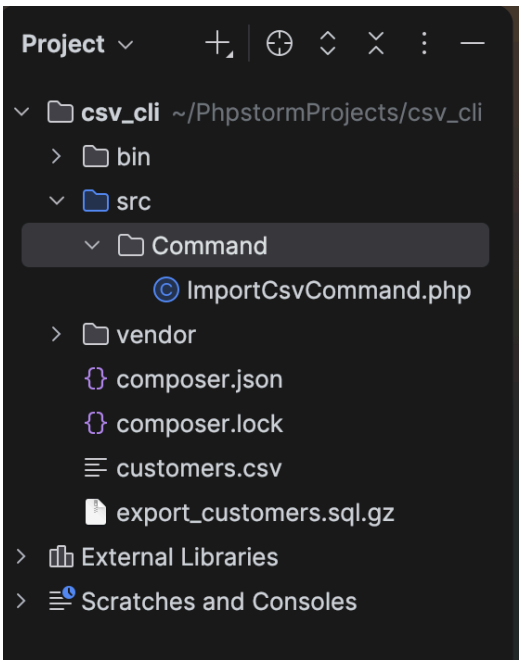

FIGURE 3.5 – Les fichiers de sauvegarde (.sql) et sa compression (.gz) générés avec succès

Bilan de l'exercice :
Création réussie d'un pipeline complet *ETL* (Extract, Transform, Load) en ligne de commande. L'application est robuste face aux erreurs, respecte les standards de sécurité d'un environnement de production (cloisonnement strict des privilèges, sécurisation des inputs), et les performances sont optimisées par une architecture alliant traitement par lots et insertion en masse.

Chapitre 4 : Création d'une API RESTful avec Laravel

Suite à l'importation massive et optimisée des données clients, l'objectif de cette phase était d'exposer ces informations de manière sécurisée et performante via une API REST. Le framework Laravel a été choisi pour sa robustesse et son écosystème d'outils adaptés au développement backend moderne.

4.1. Standardisation des Réponses (API Resources)

L'un des principes fondamentaux d'une API de qualité est la prédictibilité de son format de réponse. Pour répondre aux exigences de la consigne (structure JSON plate avec gestion native de la pagination), j'ai utilisé les **API Resources** de Laravel.

La classe `CustomerResource` agit comme une couche de transformation entre le modèle de base de données (éloquent) et le rendu JSON final, permettant notamment de masquer certains champs sensibles et d'en formater d'autres (comme la concaténation du prénom et du nom).

```php
public function toArray(Request $request): array
{
    return [
        'id' => $this->id,
        'name' => $this->first_name . ' ' . $this->last_name,
        'email' => $this->email,
    ];
}
```

Listing 4.1 – Transformation des données via CustomerResource

4.2. Sécurité et Traçabilité

L'API a été sécurisée à plusieurs niveaux pour simuler un environnement de production.

4.2.1. Authentification Basic Auth

L'accès aux endpoints est protégé par le middleware natif `auth.basic`. Chaque requête doit obligatoirement inclure un en-tête d'autorisation (encodé en Base64) correspondant à un utilisateur enregistré dans la table `users` du système.

4.2.2. Injection d'En-têtes Personnalisés

Pour assurer la traçabilité et anticiper les futures évolutions de l'API, un **Middleware** personnalisé a été développé. Il intercepte chaque réponse sortante du serveur pour y injecter l'en-tête `x-api-version: v1`.

```php
public function handle(Request $request, Closure $next)
{
    $response = $next($request);
    $response->headers->set('x-api-version', 'v1');
    return $response;
}
```

Listing 4.2 – Middleware d'injection de version

4.3. Filtres, Tri et Pagination (Query Builder)

Le défi technique majeur consistait à gérer des requêtes d'URL complexes du type : `?filter[name]=Eladio&sort=-email&page[size]=10`.

Au lieu de surcharger le contrôleur avec de multiples conditions conditionnelles, j'ai intégré le package standard de l'industrie **Spatie Query Builder**. Cet outil parse automatiquement les paramètres de l'URL pour construire dynamiquement la requête SQL de manière propre et élégante.

```php
public function index(ListCustomersRequest $request)
{
    $customers = QueryBuilder::for(Customer::class)
        ->allowedFilters(['first_name', 'last_name', 'email'])
        ->allowedSorts(['first_name', 'last_name', 'email'])
        ->paginate(
            $request->input('page.size', 10),
            ['*'],
            'page[number]',
            $request->input('page.number', 1)
        );

    return CustomerResource::collection($customers);
}
```

Listing 4.3 – Implémentation du filtrage et tri avec Spatie

4.4. Optimisations pour la Production (Production-Readiness)

Pour garantir la stabilité de l'application face à un trafic réel, plusieurs sécurités backend avancées ont été mises en place :

- **Rate Limiting (Throttle) :** La route est protégée par le middleware `throttle:60,1`, limitant les appels à 60 requêtes par minute par utilisateur, prévenant ainsi les attaques par déni de service (DDoS).
- **Validation stricte (FormRequest) :** Une classe `ListCustomersRequest` valide les paramètres de l'URL. Par exemple, la taille de la pagination (`page[size]`) est strictement plafonnée à 100 pour empêcher un attaquant de saturer la RAM du serveur en demandant des millions d'enregistrements d'un coup.

— **Gestion des Erreurs JSON :** Configuration de l'en-tête `Accept: application/json` lors des appels clients pour garantir que les erreurs de validation (Code 422) soient renvoyées sous forme de structure JSON lisible, évitant ainsi les redirections HTML inattendues.
— **Configuration CORS :** Préparation du fichier `cors.php` pour autoriser explicitement les origines et les méthodes HTTP, une étape indispensable pour que l'API puisse communiquer avec une future architecture front-end découplée (Headless).

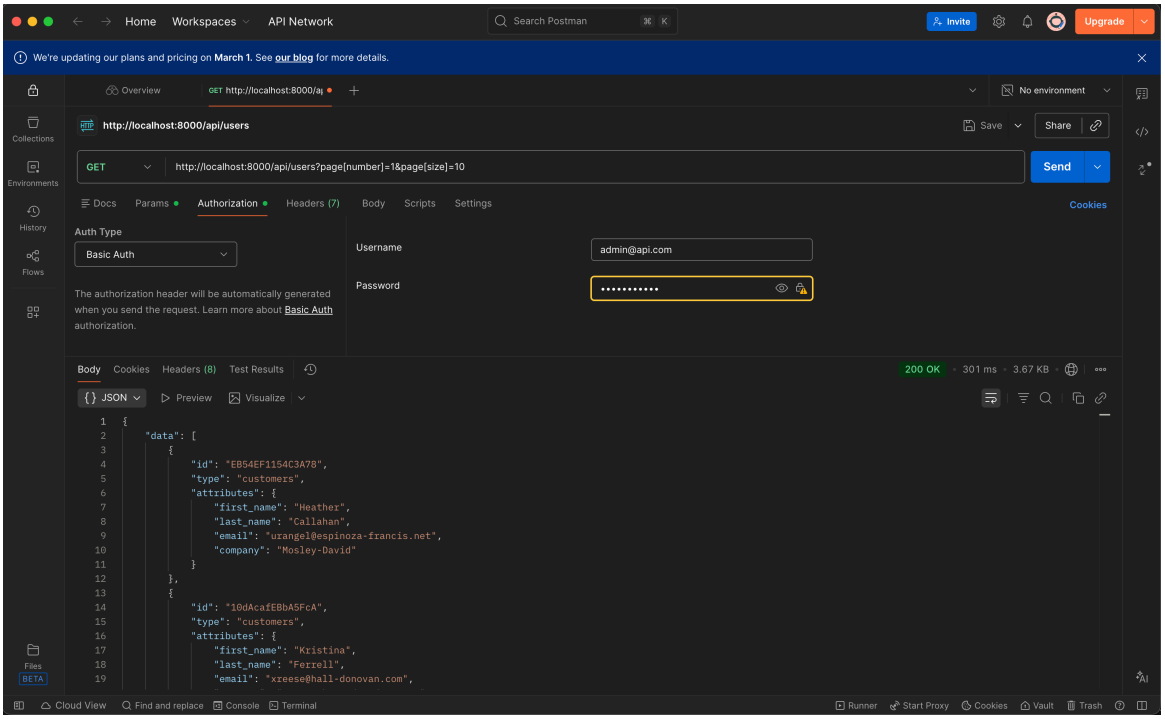

FIGURE 4.1 – Test de l'API via Postman illustrant le filtrage, le JSON généré et l'en-tête de version

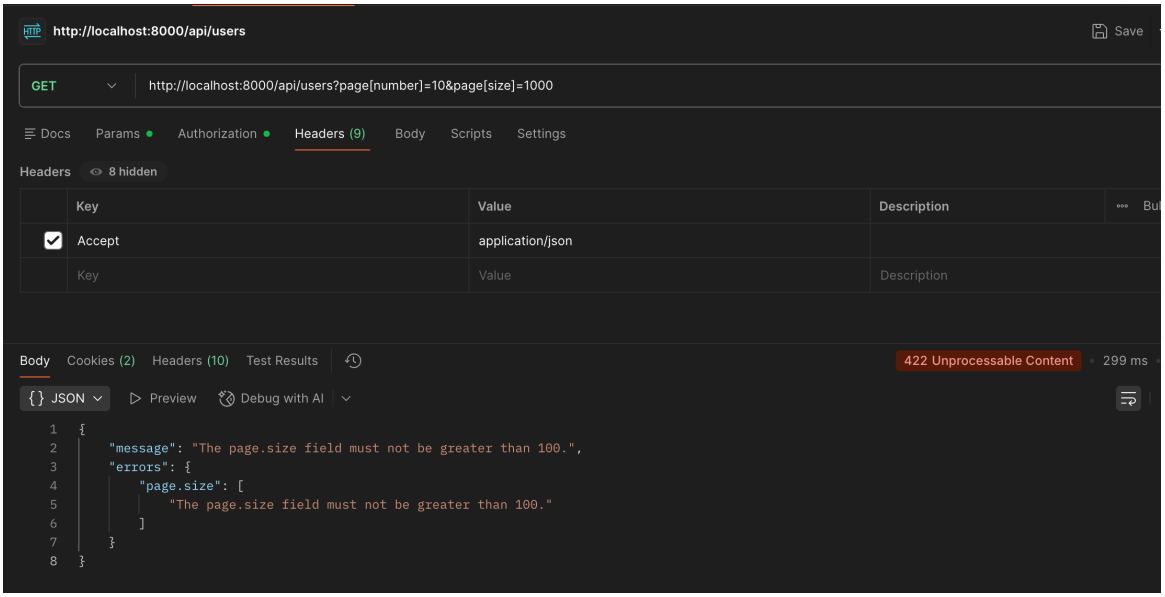

FIGURE 4.2 – Interception sécurisée d'une requête malveillante (Erreur 422 - Validation de la taille de page)

Bilan du Chapitre :
Le développement de cette API démontre la maîtrise d'un cycle complet de mise en production backend : exposition standardisée des données, sécurisation des accès, optimisation des requêtes de lecture et protection active contre les surcharges mémoires et réseaux.

Chapitre 5 : Conclusion Générale

L'objectif principal de ce Sprint consistait non seulement à assimiler l'utilisation des outils de développement modernes, mais surtout à déconstruire et comprendre les mécanismes internes régissant l'ingénierie backend.

En retraçant l'évolution des pratiques de développement PHP — depuis la conception d'une architecture MVC artisanale jusqu'à l'utilisation industrielle du framework Laravel, en passant par l'intégration modulaire des composants Symfony — ce travail a permis d'établir une vision globale et approfondie de l'écosystème.

Ce parcours pratique a mis en exergue plusieurs concepts cruciaux inhérents aux environnements de production :

— **L'Architecture et les Standards :** Maîtrise du cycle de vie d'une requête (Request/-Response), respect des principes SOLID, et implémentation stricte des normes architecturales REST et de la spécification JSON :API.

— **La Performance :** Mise en œuvre de stratégies d'importation massive (Batch Processing et Bulk Inserts), garantissant la stabilité des serveurs face au traitement de grands volumes de données.

— **La Sécurité et la Fiabilité :** Protection des points d'accès (Rate Limiting), validation intransigeante des données entrantes (FormRequests), et maintien de l'intégrité des bases de données (Transactions ACID et cloisonnement des privilèges).

Perspectives et Poursuite du Projet :
La consolidation de ces fondamentaux (programmation orientée objet, optimisation algorithmique et conception d'API sécurisées) pose les jalons techniques indispensables pour la suite du projet de fin d'études. Cette phase préparatoire valide les compétences nécessaires pour évoluer vers des architectures distribuées plus complexes, ouvrant ainsi la voie à la conception d'écosystèmes *Headless*, couplant des CMS robustes tels que Drupal à des technologies front-end modernes comme Next.js.