Documentation Technique Laravel

TABLE DES MATIERES

Introduction

L’application web **Info-Tools** a été développée avec le framework **Laravel**, reconnu pour sa robustesse, sa sécurité et sa rapidité de développement. Laravel propose de nombreux outils intégrés, notamment **JetStream**, qui facilite la gestion des utilisateurs et l’authentification de manière moderne et sécurisée.

Outils utilisées

- Framework : Laravel
- Serveur : Laragon
- Version de PHP : PHP 8.3.4
- Packages : Jetstream
- IDE : Visual Studio Code

Gestion de l’inscription et de la connexion

L’authentification des utilisateurs est gérée grâce au package **JetStream**, qui repose sur **Laravel Fortify** pour sécuriser le processus de connexion et d'inscription. JetStream offre une solution complète pour la gestion des comptes utilisateurs en intégrant plusieurs fonctionnalités essentielles :

- ***Inscription et connexion*** : Permet aux utilisateurs de créer un compte et de se connecter en toute sécurité.
- ***Vérification des emails*** : Assure qu'un utilisateur a bien confirmé son adresse email avant d’accéder à certaines fonctionnalités.

- ***Réinitialisation de mot de passe*** : Offre une fonctionnalité permettant aux utilisateurs de récupérer l'accès à leur compte en cas d'oubli de mot de passe.
- ***Authentification à deux facteurs (2FA)*** : Ajoute une couche de sécurité supplémentaire en demandant un code unique lors de la connexion.
- ***Gestion des sessions*** : Permet aux utilisateurs de gérer leurs connexions et de se déconnecter des sessions actives sur différents appareils.
- ***Support des tokens API avec Sanctum*** *(optionnel)* : Si l'application expose une API, JetStream peut être utilisé avec **Laravel Sanctum** pour générer des tokens sécurisés et gérer l'authentification des utilisateurs via API.

Grâce à JetStream, la gestion des utilisateurs est simplifiée et sécurisée, permettant un déploiement rapide des fonctionnalités d'authentification tout en respectant les bonnes pratiques de sécurité.

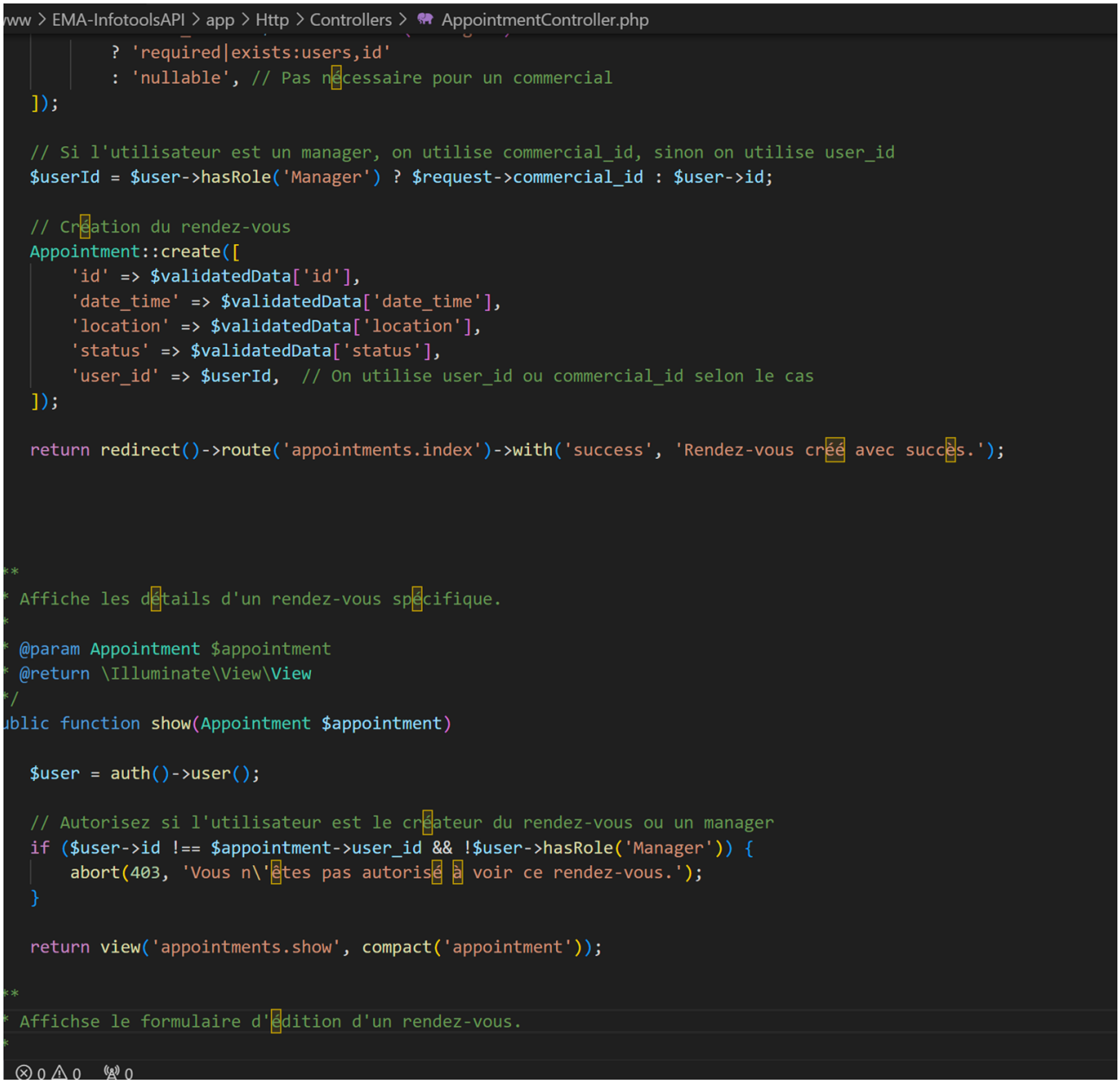
Pour mettre en place l'authentification avec **JetStream**, plusieurs étapes sont nécessaires. JetStream propose deux options d'interface : **Livewire** (basé sur Blade et Alpine.js) ou **Inertia.js** (basé sur Vue.js). Dans cette application, nous avons choisi **Livewire** pour sa simplicité et son intégration fluide avec Laravel.

```
php artisan jetstream:install livewire
```

Pour gérer les utilisateurs et les différentes fonctionnalités, nous avons mis en place une base de données locale contenant tous les champs nécessaires au bon fonctionnement de l'application web.

Cette base stocke notamment les informations des utilisateurs, leurs rôles et permissions, ainsi que les données essentielles pour assurer la cohérence et la sécurité du système.

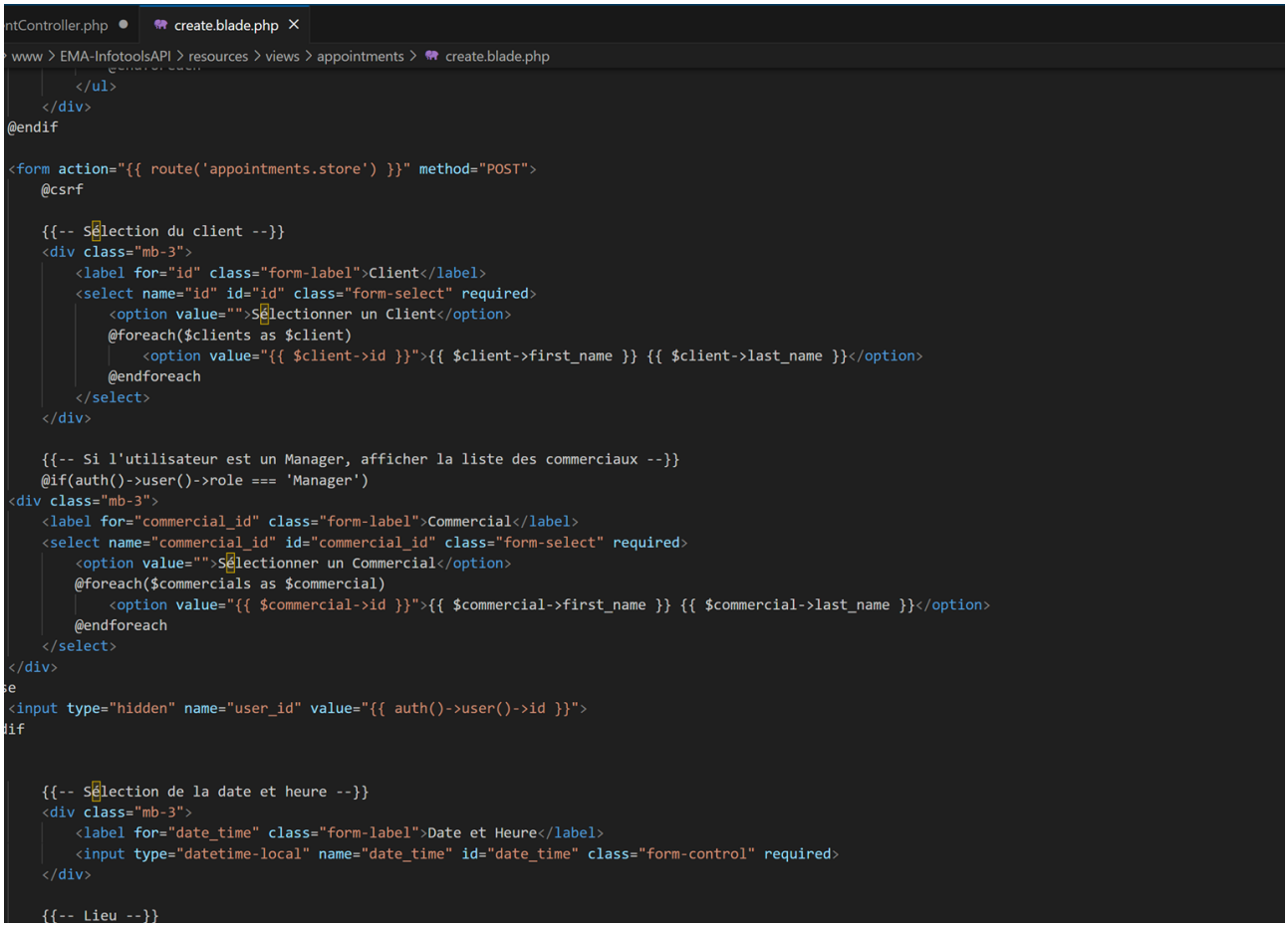
Nous avons également créé des contrôleurs qui permettent de faire le lien entre l'affichage (views) et la base de données. Ils gèrent toutes les actions nécessaires au bon fonctionnement des différentes fonctionnalités. Voici un exemple de contrôleur :



Pour chaque fonctionnalité, cinq vues sont disponibles :

- ***Index*** : affiche la liste complète des éléments.
- ***Create*** : affiche une page permettant d'ajouter un nouvel élément.
- ***Edit*** : affiche une page permettant de modifier un élément existant.
- ***Show*** : affiche les détails d'un élément spécifique.

- ***Layout*** : sert de modèle (template) pour structurer les autres pages.



API

Afin de permettre à l'application mobile d'accéder aux informations de la base de données, nous avons mis en place une **API** sur l'application web Laravel. Cette API permet d'échanger

des données de manière sécurisée et optimisée entre le serveur et les clients (application mobile, front-end, ou autre service tiers).

Pour ce faire, nous avons créé des **contrôleurs API** dédiés à chaque fonctionnalité de l'application. Ces contrôleurs gèrent les différentes requêtes (GET, POST, PUT, DELETE) et renvoient les informations au format **JSON**, assurant ainsi une compatibilité avec les applications clientes.

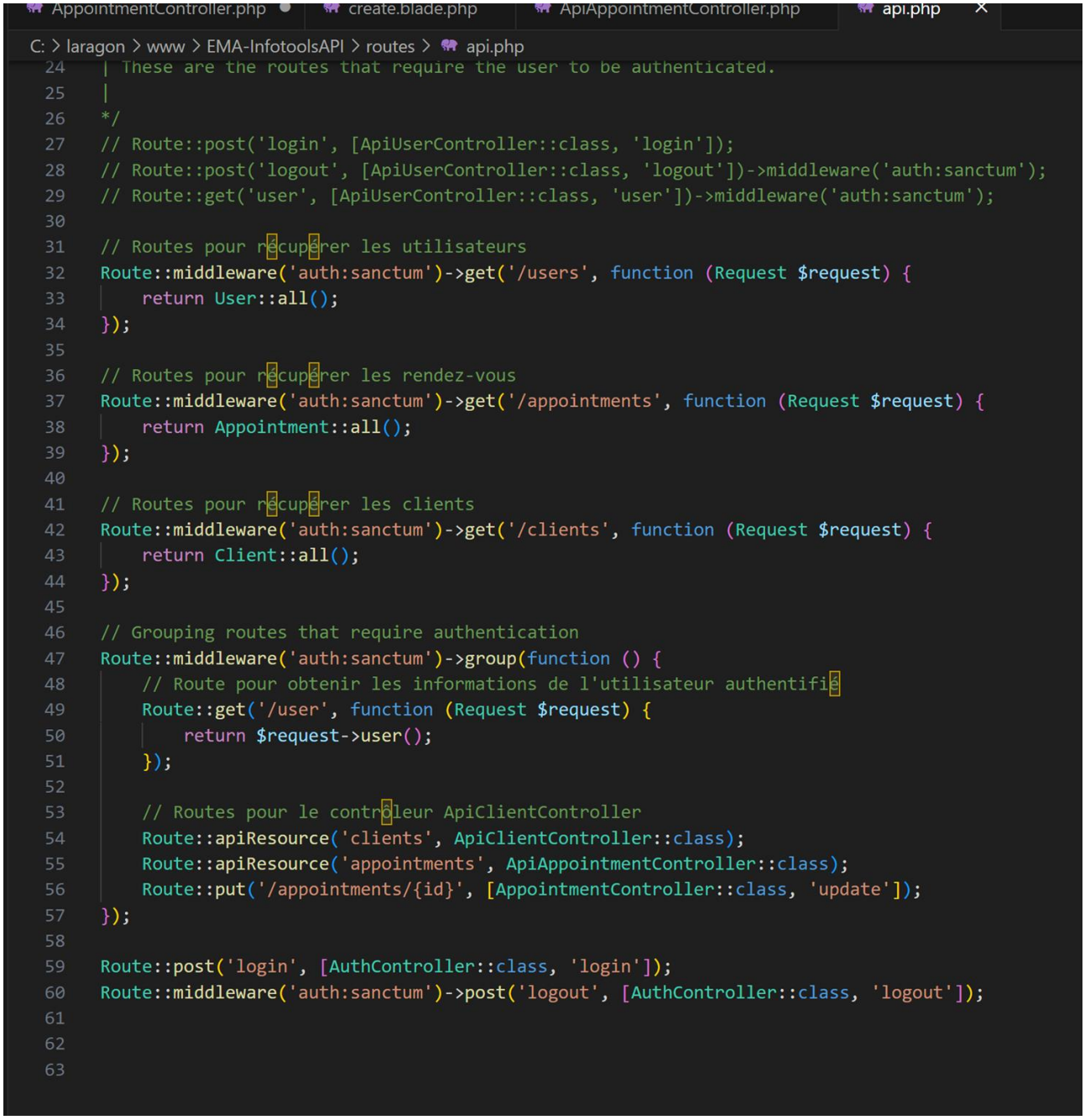
Voici un exemple de contrôleur API permettant de récupérer la liste des rendez-vous :

```php
hp

espace App\Http\Controllers\Api;

App\Http\Controllers\Controller;
App\Models\Appointment;
App\Models\Client;
App\Models\User;
Illuminate\Http\Request;

ss ApiAppointmentController extends Controller

/**
 * Affiche une liste de tous les rendez-vous.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    // Récupère tous les rendez-vous avec leurs clients associés
    $appointments = Appointment::with('client')->get();

    // Retourne la réponse JSON avec la liste des rendez-vous
    return response()->json([
        "success" => true,
        "message" => "Liste des rendez-vous",
        "data" => $appointments
    ]);
}

/**
 * Store a newly created resource in storage.
 */
public function store(Request $request)
{
    try {
        $user = auth()->user(); // Utilisateur connecté

        // Validation des données
        $validatedData = $request->validate([
            'id' => 'required|exists:clients,id',
```

Les contrôleurs API permettent également de **restreindre l'accès** aux ressources afin de garantir la sécurité des données. Seuls les utilisateurs authentifiés et disposant des **permissions nécessaires** peuvent effectuer des requêtes sur l'API.

L'authentification des requêtes API est gérée via **Laravel Sanctum**, qui génère des **tokens d'accès sécurisés** pour chaque utilisateur autorisé. Un utilisateur qui ne possède pas de **token valide** ou qui ne dispose pas des **droits d'accès requis** ne pourra pas interagir avec l'API et recevra une réponse d'erreur appropriée.

Voici un exemple de protection d'une route API avec **Sanctum** :

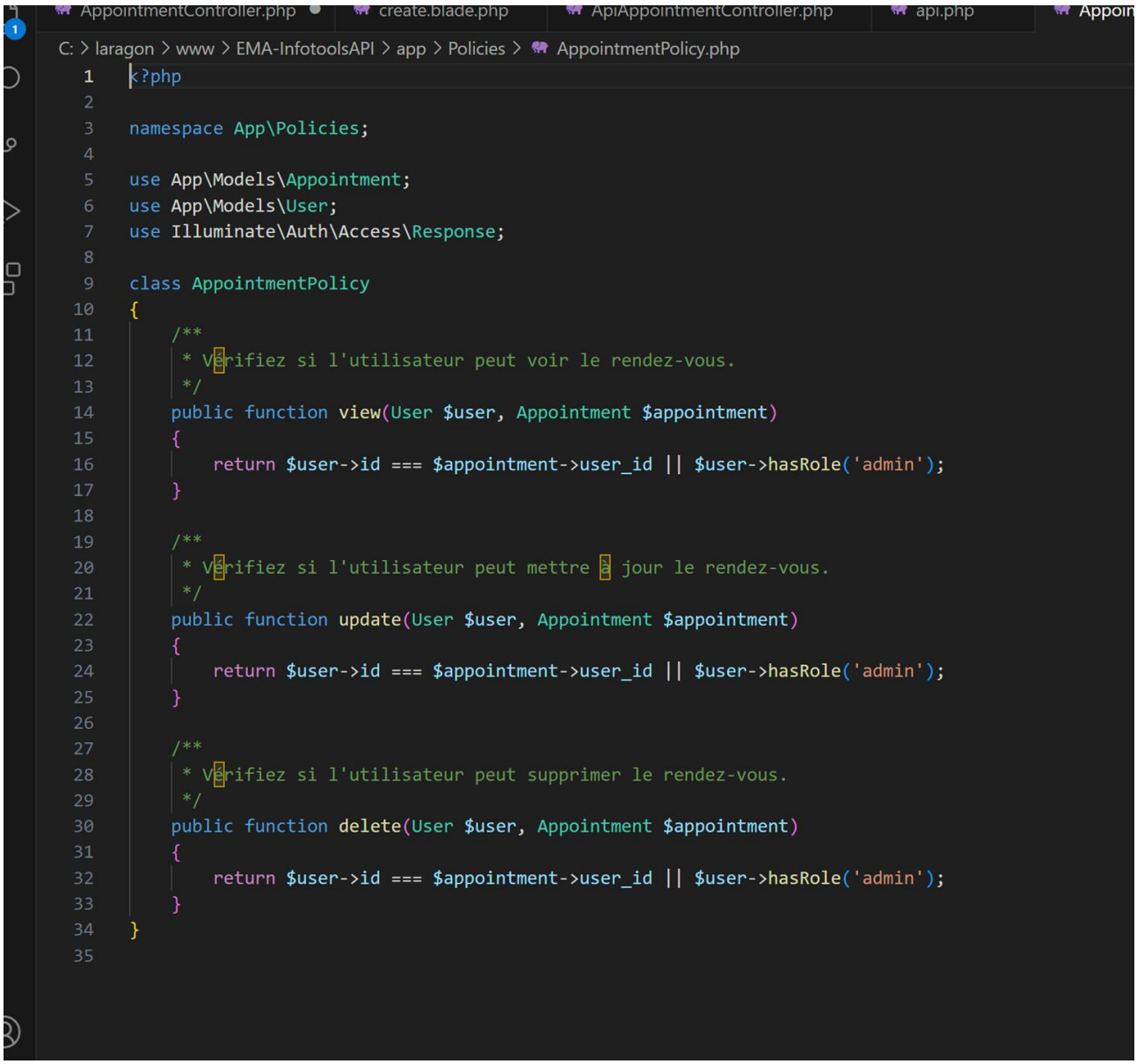

AppointmentController.php
create.blade.php
ApiAppointmentController.php
api.php
C: > laragon > www > EMA-InfotoolsAPI > routes > api.php
| These are the routes that require the user to be authenticated.
|
*/
// Route::post('login', [ApiUserController::class, 'login']);
// Route::post('logout', [ApiUserController::class, 'logout'])->middleware('auth:sanctum');
// Route::get('user', [ApiUserController::class, 'user'])->middleware('auth:sanctum');
// Routes pour récupérer les utilisateurs
Route::middleware('auth:sanctum')->get('/users', function (Request $request) {
return User::all();
});
// Routes pour récupérer les rendez-vous
Route::middleware('auth:sanctum')->get('/appointments', function (Request $request) {
return Appointment::all();
});
// Routes pour récupérer les clients
Route::middleware('auth:sanctum')->get('/clients', function (Request $request) {
return Client::all();
});
// Grouping routes that require authentication
Route::middleware('auth:sanctum')->group(function () {
// Route pour obtenir les informations de l'utilisateur authentifié
Route::get('/user', function (Request $request) {
return $request->user();
});
// Routes pour le contrôleur ApiClientController
Route::apiResource('clients', ApiClientController::class);
Route::apiResource('appointments', ApiAppointmentController::class);
Route::put('/appointments/{id}', [AppointmentController::class, 'update']);
});
Route::post('login', [AuthController::class, 'login']);
Route::middleware('auth:sanctum')->post('logout', [AuthController::class, 'logout']);

Gestion des droits

Dans les vues **"index"** des différentes fonctionnalités, l'accès au bouton « **supprimer »** est conditionné en fonction du rôle de l'utilisateur. Si l'utilisateur est un **manager**, il a la possibilité de supprimer un élément. En revanche, si ce n'est pas le cas, le bouton de suppression ne s'affichera pas, empêchant ainsi toute suppression non autorisée.

Pour gérer ces restrictions d'accès, nous avons utilisé la fonctionnalité **Policies** de Laravel, qui permet de définir des règles d'autorisation au niveau des modèles.

Voici un exemple d'implémentation d'une **Policy** dans `app/Policies/AppointmentPolicy.php` :

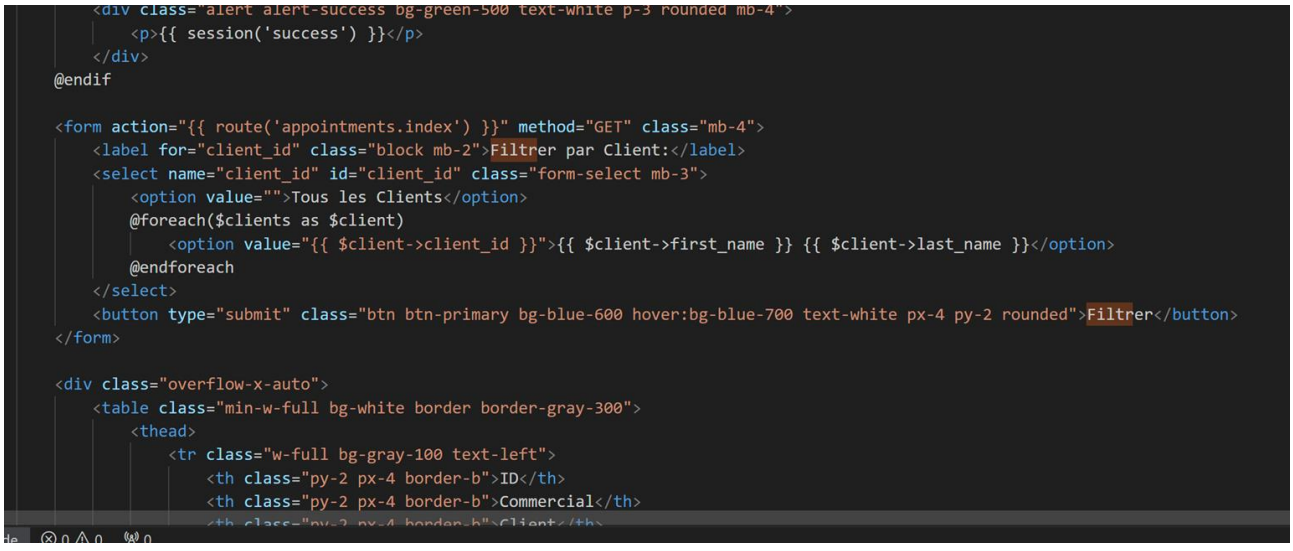

AppointmentController.php
create.blade.php
ApiAppointmentController.php
api.php
C: > laragon > www > EMA-InfotoolsAPI > app > Policies > AppointmentPolicy.php
<?php
namespace App\Policies;
use App\Models\Appointment;
use App\Models\User;
use Illuminate\Auth\Access\Response;
class AppointmentPolicy
{
/**
* Vérifiez si l'utilisateur peut voir le rendez-vous.
*/
public function view(User $user, Appointment $appointment)
{
return $user->id === $appointment->user_id || $user->hasRole('admin');
}
/**
* Vérifiez si l'utilisateur peut mettre à jour le rendez-vous.
*/
public function update(User $user, Appointment $appointment)
{
return $user->id === $appointment->user_id || $user->hasRole('admin');
}
/**
* Vérifiez si l'utilisateur peut supprimer le rendez-vous.
*/
public function delete(User $user, Appointment $appointment)
{
return $user->id === $appointment->user_id || $user->hasRole('admin');
}
}

Gestion des filtres et recherche

Afin d'améliorer l'expérience utilisateur et de faciliter la navigation, nous avons intégré un **système de filtrage** permettant de rechercher un rendez-vous par client. Ce filtre est accessible directement dans la vue et permet d'afficher uniquement les résultats pertinents, sans avoir à parcourir l'ensemble des rendez-vous.

1. Ajout du filtre dans la vue Blade

Nous avons ajouté un formulaire permettant aux utilisateurs de sélectionner un client pour filtrer les rendez-vous :



2. Gestion du filtrage dans le contrôleur

Ensuite, nous avons modifié la méthode `index()` pour prendre en compte le filtre sélectionné par l'utilisateur :

```php
    // Appliquez le filtre si un `id` (client) est fourni
    if ($request->filled('client_id')) {
        // Remplacez client_id par id, en supposant que client_id soit un champ dans le modèle Appoi
        $query->whereHas('client', function($q) use ($request) {
            $q->where('id', $request->client_id);
        });
    }

    // Exécutez la requête
    $appointments = $query->get();

    // Récupérez tous les clients pour le filtrage
    $clients = Client::all();

    // Retournez la vue avec les données des rendez-vous et des clients
    return view('appointments.index', compact('appointments', 'clients'));
}
```

3. Ajout de la barre de recherche dans la vue

Nous avons ajouté un formulaire permettant aux utilisateurs de rechercher un client pour filtrer les rendez-vous :

```blade
<!-- Formulaire de recherche par nom -->
<form action="{{ route('clients.index') }}" method="GET" class="mb-6">
    <div class="flex items-center space-x-2">
        <input type="text" name="search" id="search" class="form-input mb-3 p-2 border border-gr
        <button type="submit" class="btn btn-primary bg-blue-600 hover:bg-blue-700 text-white px
    </div>
</form>
```

4. Gestion de la recherche dans le contrôleur

Puis, Nous avons modifié la méthode `index()` :

```php
    // Récupérer les commerciaux avec une option de recherche
    $users = User::where('role', 'Salesperson')
        ->when($request->filled('search'), function($query) use ($request) {
            return $query->where('first_name', 'like', '%'.$request->search.'%')
                         ->orWhere('last_name', 'like', '%'.$request->search.'%');
        })
        ->paginate(10);

    // Retourner la vue avec les commerciaux filtrés et paginés
    return view('users.index', compact('users'))
        ->with('i', (request()->input('page', 1) - 1) * 10);
}
```

Conclusion

Le projet « **Info-Tools »** est une solution fonctionnelle et complète pour la gestion des rendez-vous, intégrant une authentification sécurisée avec **JetStream**, une API REST efficace et un contrôle des accès via **Policies Laravel**.

Il répond aux besoins essentiels, mais des améliorations sont possibles, notamment sur la sécurité avancée et la récupération des comptes.