Hedvi Nadenane Année 2024-2025

Documentation AP Parking

Sommaire

Présentation du Projet :

1.1 En quoi consiste ce Projet ? :

Le projet consiste à développer une **application de gestion des places de parking** permettant aux membres du personnel de **réserver une place numérotée** de manière automatisée.

Lorsqu'un utilisateur fait une demande, une place libre lui est attribuée immédiatement, et en cas d'indisponibilité, il est placé en file d'attente.

Un **administrateur** supervise l'ensemble du système, valide les inscriptions, gère les réservations et peut attribuer des places manuellement. L'application devra être **sécurisée, accessible via un réseau local** et proposer une interface claire et responsive. Elle inclura également une **base de données** pour stocker les informations des utilisateurs, des réservations et de l'historique des attributions.

Explication des différentes Pages

Explication d'une route :

En Laravel, une **route** permet de définir quelle action doit être exécutée lorsqu'un utilisateur accède à une URL spécifique. Elle associe une URL à une fonction ou à un contrôleur.Les routes sont définies dans les fichiers de routes situés dans le dossier routes/, comme Web.php

Explication d'un Controller :

Un **Controller** en Laravel est une classe qui gère la logique du site. Il fait le lien entre les routes (les URL) et les vues (les pages affichées).

Explication simple :

Quand un utilisateur fait une action (comme cliquer sur un bouton), la requête est envoyée à un **Controller**. Ce dernier traite la demande, récupère les données nécessaires, et renvoie la réponse appropriée

Explication d'une vue :

Une **Vue** en Laravel est un fichier qui contient le code HTML affiché à l'utilisateur. Elle sert à séparer l'affichage de la logique du projet, ce qui permet d'avoir un code plus propre et organisé.

Explication d'une migration (BDD) :

Une **migration** en Laravel est un fichier qui permet de créer, modifier ou supprimer des tables dans la base de données. Elle sert à gérer la structure de la base de données de manière organisée et versionnée, un peu comme un "git" pour la base de données.

Explication d'un seeders (BDD) :

Un **seeder** en Laravel est un fichier qui permet d'insérer des données automatiquement dans la base de données. Il sert à remplir les tables avec des valeurs par défaut, utiles pour tester l'application sans devoir entrer les données manuellement.

2.1.1 Contenue de la page : routes/auth.php:

Ce fichier utilise **Laravel Breeze** pour gérer l'authentification. Il définit des groupes de routes en fonction de l'état de connexion de l'utilisateur :

- **middleware('guest') :** Ces routes sont accessibles uniquement aux utilisateurs non connectés (ex. connexion, inscription).
- **middleware('auth') :** Ces routes sont accessibles uniquement aux utilisateurs connectés (ex. vérification d'email, déconnexion).

1. Routes accessibles aux invités (`guest`)

```php
Route::middleware('guest')->group(function () {
    Route::get('register', [RegisteredUserController::class, 'create'])
                ->name('register');

    Route::post('register', [RegisteredUserController::class, 'store']);
```

GET /register → Affiche le formulaire d'inscription.

POST /register → Enregistre un nouvel utilisateur

[RegisteredUserController::class, 'create'] : Appelle la méthode **create()** dans le contrôleur **RegisteredUserController** pour afficher le formulaire d'inscription.

2. Routes accessibles aux client (`login`)

```php
Route::post('login', [AuthenticatedSessionController::class, 'store']);

Route::get('forgot-password', [PasswordResetLinkController::class, 'create'])
            ->name('password.request');
```

`GET /login` → Affiche le formulaire de connexion.

`POST /login` → Authentifie l'utilisateur et le connecte

3. Mot de passe oublié (`forgot-password`)

```php
Route::get('forgot-password', [PasswordResetLinkController::class, 'create'])
            ->name('password.request');

Route::post('forgot-password', [PasswordResetLinkController::class, 'store'])
            ->name('password.email');
```

`GET /forgot-password` → Affiche le formulaire pour entrer son email et recevoir un lien de réinitialisation.

`POST /forgot-password` → Envoie un email contenant le lien pour réinitialiser le mot de passe.

4. Réinitialisation du mot de passe (`reset-password`)

```php
Route::get('reset-password/{token}', [NewPasswordController::class, 'create']
            ->name('password.reset');

Route::post('reset-password', [NewPasswordController::class, 'store'])
            ->name('password.store');
```

`GET /reset-password/{token}` → Affiche le formulaire de réinitialisation avec le token.

`POST /reset-password` → Met à jour le mot de passe dans la base de données.

5. Routes accessibles aux utilisateurs connectés (`auth`)

Ces routes permettent de gérer la vérification d'email, le changement de mot de passe et la déconnexion.

```php
Route::middleware('auth')->group(function () {
    Route::get('verify-email', EmailVerificationPromptController::class)
            ->name('verification.notice');

    Route::get('verify-email/{id}/{hash}', VerifyEmailController::class)
            ->middleware(['signed', 'throttle:6,1'])
            ->name('verification.verify');

    Route::post('email/verification-notification', [EmailVerificationNotification
            ->middleware('throttle:6,1')
            ->name('verification.send');
```

`GET /verify-email` → Invite l'utilisateur à vérifier son email.

6. Confirmation du mot de passe (`confirm-password`)

```php
Route::get('confirm-password', [ConfirmablePasswordController::class, 'show'])
        ->name('password.confirm');

Route::post('confirm-password', [ConfirmablePasswordController::class, 'store']);
```

GET /confirm-password → Demande de saisir à nouveau son mot de passe pour des actions sensibles (ex. modifier email).

POST /confirm-password → Vérifie si le mot de passe est correct.

7. Mise à jour du mot de passe (`password`)

```
Route::put('password', [PasswordController::class, 'update'])->name('password.up
```

PUT /password → Permet de modifier son mot de passe depuis son profil.

.

2.1.2 Contenue de la page : routes/web.php:

Ce fichier définit toutes les **routes web** de votre application Laravel. Les routes sont utilisées pour lier les URL aux actions ou **méthodes** des contrôleurs. Voici une explication détaillée de chaque partie de ce fichier. 8. Routes pour la page d'accueil (`'/'`)

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

- Cette route est associée à l'URL `/`, la page d'accueil de votre application.
- Lorsque l'utilisateur accède à cette URL, Laravel affiche la vue **`welcome`**. Il s'agit généralement de la première page de l'application ou d'une page d'accueil par défaut.

9. Dashboard sécurisé (utilisateur connecté obligatoire)

```php
Route::middleware(['auth'])->group(function () {
    Route::get('/dashboard', [ProfileController::class, 'index'])->name('dashboar
    Route::post('/profile/password', [ProfileController::class, 'updatePassword']
});
```

`Route::middleware(['auth'])` : Ce groupe de routes est accessible uniquement aux utilisateurs authentifiés (connectés). Si un utilisateur n'est pas connecté, il sera redirigé vers la page de connexion.

Route `/dashboard` : Affiche le tableau de bord de l'utilisateur connecté. La méthode `index` du `ProfileController` gère l'affichage de cette page.

Route `/profile/password` : Permet à l'utilisateur de modifier son mot de passe via la méthode `updatePassword` du même contrôleur.

10. Routes nécessitant une authentification

Les routes suivantes sont également protégées par le middleware **`auth`,** donc elles ne sont accessibles qu'aux **utilisateurs connectés.**

```php
Route::middleware(['auth'])->group(function () {
    // Routes pour les utilisateurs
    Route::resource('users', UserController::class);

    // Routes pour le parking
    Route::resource('parking', ParkingController::class);

    // Routes pour les réservations
    Route::resource('reservations', ReservationController::class);

    // Routes pour l'historique
    Route::resource('historiques', HistoriqueController::class);

    // Routes pour l'attente
    Route::resource('attente', AttenteController::class);
```

Toutes ces **routes** permettent d'afficher une action selon le commentaire qui lui est inscrit.

11. Gestion du profil utilisateur

```php
Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit')
Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.up
Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.
```

- Ces routes permettent à l'utilisateur de modifier ou supprimer son profil (nom, email, etc.).

12. Routes spécifiques aux administrateurs

```php
Route::resource('admin', AdminController::class)
    ->middleware('can:viewAny,App\Models\User');
```

Ces **routes** sont utilisées pour gérer les utilisateurs (ex. liste des utilisateurs, suppression, modification).

La route est protégée par le middleware **`can:viewAny,App\Models\User`**, qui garantit que seuls les utilisateurs ayant la permission appropriée peuvent y accéder.

13. Gestion des places de parking (Admin seulement)

```php
Route::post('/parking/{parking}/occuper', [ParkingController::class, 'marquerOccu
Route::post('/parking/{parking}/liberer', [ParkingController::class, 'marquerLibr
```

Ces routes permettent aux administrateurs de marquer les places de parking comme occupées ou libres. Ces actions sont contrôlées par la méthode **marquerOccupee** et **marquerLibre** du **ParkingController**.

14. Gestion de la liste d'attente

```
Route::post('/attente/{id}/update-position', [AttenteController::class, 'updatePo
    ->name('attente.updatePosition');
```

Cette **route** permet de mettre à jour la position d'un utilisateur dans la liste d'attente.

15. Historique des attributions

```
Route::resource('historique', HistoriqueController::class);
```

Ces **routes** permettent de consulter et gérer l'historique des actions concernant les parkings et les réservations.

16. Réservations

```
Route::post('/reservation', [ReservationController::class, 'store'])->name
```

Cette **route** permet à un utilisateur de faire une nouvelle réservation pour un parking.

17. Routes pour l'inscription (inscription de nouveaux utilisateurs)

```
Route::get('/register', [RegisteredUserController::class, 'create'])->name('regis
Route::post('/register', [RegisteredUserController::class, 'store']);
```

Route `GET` **`/register`** **:** Affiche le formulaire d'inscription, géré par la méthode `create` du **`RegisteredUserController`.**

Route `POST` **`/register`** : Envoie les informations saisies pour créer un nouvel utilisateur, géré par la méthode `store` du même contrôleur.

Explication Controller

3.1.1 Contenue de la page : Controller/ControllerParking.php:

1.Méthode index()

```php
public function index()
{
    $places = Parking::all();
    return view('admin.parking.index', comp
}
```

- Récupère toutes les places de parking avec Parking::all().
- Les stocke dans $places.
- Retourne une vue Blade : admin/parking/index.blade.php, en lui passant les données $places.

3. Méthode create()

```php
public function create()
{
    return view('admin.parking.places');
}
```

- Affiche le formulaire de création d'une nouvelle place.
- Vue ciblée : admin/parking/places.blade.php.

4. Méthode store()

```php
public function store(Request $request)
{
    $request->validate([
        'numero_place' => 'required|unique
        'notes' => 'nullable|string|max:5
    ]);
```

Valide les données du formulaire :

• numero_place : obligatoire, unique, entier.

• notes : optionnel, chaîne, max 500 caractères.

• Crée une nouvelle place avec les valeurs envoyées via $request.

• Redirige vers la liste (route('parking.index')) avec un message flash.

5. Méthode edit()

```
ublic function edit(Parking $parking)
{
    return view('admin.parking.edit', c
}
```

Laravel injecte automatiquement l'objet $parking grâce au model binding.

• Envoie cette donnée à la vue d'édition : admin/parking/edit.blade.php.

6. Méthode update()

```
public function update(Request $request, Par
{
    $request->validate([
        'numero_place' => 'required|integer|
        'notes' => 'nullable|string|max:500'
    ]);

    $parking->update([
        'numero_place' => $request->numero_p
        'notes' => $request->notes,
    ]);

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

```
uest, Parking $parking)

|integer|unique:parking,numero_place,' . $pa
|max:500',

->numero_place,

ng.index')->with('success', 'Place mise à jo
```

Valide les champs, tout en autorisant le numéro existant pour la place actuelle (unique:parking,numero_place,$parking->id).

• Met à jour l'enregistrement dans la BDD avec update().

• Redirige vers l'index avec un message.

7. Méthode destroy()

```
public function destroy(Parking $parking)
{
    $parking->delete();
    return redirect()->route('parking.index'
```

• Supprime la place de la base de données.

• Redirige avec un message de confirmation.

8. Méthode marquerOccupee()

```
public function marquerOccupee(Parking $parking)
{
    $parking->marquerOccupee(auth()->id());
    return redirect()->route('parking.index')->wit
}
```

• Appelle une méthode personnalisée marquerOccupee() sur le modèle Parking.

• Lui passe l'ID de l'utilisateur connecté avec auth()->id().

• Redirige avec message.

9. Méthode marquerLibre()

```
public function marquerLibre(Parking $parking)
{
    $parking->marquerLibre();
    return redirect()->route('parking.index')->with
}
```

• Appelle une autre méthode du modèle pour marquer la place comme libre.

• Puis redirige avec un message.

3.1.2 Contenue de la page : Controller/ControllerProfil.php:

1. Authentification de l'utilisateur connecté

```
$user = auth()->user();
```

• Récupère l'utilisateur connecté grâce au système d'authentification Laravel.

• Stocke l'objet User dans la variable $user.

2. Récupération des attributions actives

```
$attributionsActives = HistoriqueAttribution::where
    ->whereNull('expiration_at')
    ->with('parking')
    ->orderBy('date_attribution', 'desc')
    ->get();
```

• Recherche dans la table historique_attributions les lignes où :

• L'utilisateur est celui connecté (user_id = $user->id)

• L'attribution n'a pas encore expiré (expiration_at est NULL)

• ->with('parking') : charge aussi les infos de la place de parking liée (relation définie dans le modèle).

• Trie les résultats par date d'attribution (du plus récent au plus ancien).

• Renvoie tous les résultats avec get().

3. Récupération de l'historique complet

```
$attributions = HistoriqueAttribution::where('
    ->with('parking')
    ->orderBy('date_attribution', 'desc')
    ->get();
```

• Même principe que ci-dessus mais sans filtrer sur expiration_at.

• Donc ici on récupère tout l'historique (places passées et actuelles).

4. Position dans la liste d'attente et Vérifie s'il y a des places libres

```
$position = ListAttente::where('user_id', $user->id)
$parkingLibre = Parking::whereNull('user_id')->exist
```

• Cherche dans la table list_attente la ligne correspondant à l'utilisateur.

• Récupère uniquement la valeur de la colonne position.

• Cherche s'il existe au moins une place de parking non attribuée (user_id est NULL).

• ->exists() retourne true ou false.

3.1.3 Contenue de la page : Controller/ReservationController.php:

1. Méthode `store` (Gestion de la demande de réservation)

```php
public function store(Request $request)
{
    $user = Auth::user();
```

`public function store(Request $request)` : La méthode `store` est utilisée pour traiter une demande de réservation de place de parking. Elle prend un objet `Request` en paramètre, qui contient les données envoyées par l'utilisateur.

`$user = Auth::user();` : Récupère l'utilisateur actuellement connecté via le système d'authentification de Laravel.

2.

```php
// Vérifier s'il y a une place disponible
$parkingLibre = Parking::whereNull('user_id')->first();
```

`$parkingLibre = Parking::whereNull('user_id')->first();` : Cherche une place de parking qui n'est pas encore attribuée à un utilisateur (`user_id` est `null`). La méthode `first()` retourne la première place libre trouvée.

3.

```php
if ($parkingLibre) {
    // Attribuer la place
    $parkingLibre->marquerOccupee($user->id);
```

- **`if ($parkingLibre)`** : Si une place de parking libre a été trouvée (`$parkingLibre` n'est pas `null`), l'utilisateur sera attribué à cette place.
- **`$parkingLibre->marquerOccupee($user->id);`** : Appelle la méthode `marquerOccupee` sur l'objet `Parking` pour marquer la place comme occupée par l'utilisateur. Le paramètre `$user->id` est l'identifiant de l'utilisateur.

4.

```php
// Enregistrer l'attribution dans l'historique
HistoriqueAttribution::create([
    'user_id' => $user->id,
    'parking_id' => $parkingLibre->id,
    'date_attribution' => now(),
]);
```

- Enregistre l'attribution dans l'historique avec l'heure actuelle.

5.

```php
return redirect()->route('dashboard')->with('success', 'Place attribuée avec succès.');
```

- Redirige avec un message de succès.

2.Méthode Cancel (Annuler une réservation de place de parking)

```php
public function cancel()
{
    $user = Auth::user();

    // Récupérer la place de parking attribuée à l'utilisateur
    $parking = Parking::where('user_id', $user->id)->first();
```

- Récupère la place de parking actuellement attribuée à l'utilisateur.

Si l'utilisateur a une place :

```php
$parking->marquerLibre();
```

Sa appelle une méthode personnalisée pour libérer la place (`user_id` remis à `null`, sûrement).

```php
HistoriqueAttribution::where('user_id', $user->id)
    ->whereNull('expiration_at')
    ->update(['expiration_at' => now()]);

return redirect()->route('dashboard')->with('success', 'Votre réservation a été annulée avec succès.');
```

Met à jour l'entrée d'historique en ajoutant la **date de fin d'utilisation** (`expiration_at`).

3.1.3 Contenue de la page : Controller/UserController.php:

1. Méthode `index`

```php
public function index()
{
    $users = User::all();
    return view('admin.Users.index', compact('users'));
}
```

- Récupère tous les utilisateurs (`User::all()`).
- Envoie la variable `$users` à la vue `admin.Users.index`.

2. Méthode `create`

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

- Affiche le formulaire de création d'un utilisateur.

3. Méthode `store(Request $request)`

```php
public function store(Request $request)
```

- Reçoit les données du formulaire de création.

4.

```php
$request->validate([
    'name' => ['required', 'string', 'max:255'],
    'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],
    'password' => ['required', 'confirmed', Password::defaults()],
    'admin' => ['boolean'],
]);
```

Valide les champs :

- `name` : requis, texte, max 255 caractères.
- `email` : requis, email valide, unique.
- `password` : requis, confirmé (`password_confirmation` doit correspondre), règles par défaut Laravel.
- `admin` : booléen (facultatif).

```php
User::create([
    'name' => $request->name,
    'email' => $request->email,
    'password' => Hash::make($request->password),
    'admin' => $request->admin ?? false,
]);
```

- Crée un nouvel utilisateur avec les données validées.
- Le mot de passe est **haché** pour la sécurité.

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

- Redirige vers la liste des utilisateurs avec un message .

3. Méthode `show(User $user)`

```php
public function show(User $user)
{
    return view('admin.Users.show', compact('user'));
}
```

- Affiche les détails d'un utilisateur spécifique.

4. Méthode `edit(User $user)`

```php
public function edit(User $user)
{
    return view('admin.Users.edit', compact('user'));
}
```

- Affiche le formulaire pour modifier un utilisateur.

5. Méthode Update

```php
public function update(Request $request, User $user)
{
    // Règles de validation
    $request->validate([
        'name' => ['required', 'string', 'max:255'],
        'email' => ['required', 'string', 'email', 'max:255', 'unique:users,email,' . $user->id],
    ]);

    $updateData = [
        'name' => $request->name,
        'email' => $request->email,
        'admin' => $request->has('admin') ? 1 : 0,
    ];
```

- Reçoit les données de modification.
- Puis Valide les données, autorise l'email existant de l'utilisateur .

```php
$updateData = [
    'name' => $request->name,
    'email' => $request->email,
    'admin' => $request->has('admin') ? 1 : 0,
];
```

- Prépare les données à mettre à jour.
- Si la case admin est cochée, valeur 1, sinon 0.

```php
    try {
        $user->update($updateData);
        return redirect()->route('admin.index')->with('success', 'Utilisateur mis à jour avec succès.');
    } catch (\Exception $e) {
        \Log::error('Erreur lors de la mise à jour de l\'utilisateur: ' . $e->getMessage());
        return back()->withErrors(['error' => 'Une erreur est survenue lors de la mise à jour. Veuillez réessayer.'])->
    }
}
```

- Tente de mettre à jour les données.
- En cas d'erreur, log l'exception et retourne vers le formulaire avec un message d'erreur.

5. Méthode destroy(user $user)

```php
public function destroy(User $user)
{
    $user->delete();

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

- Supprime un utilisateur de la base de données.
- Redirige avec un message de confirmation.

3.1.4 Contenue de la page : Controller/adminController.php:

1.Méthode (index)

```php
public function index()
{
    if (Auth::user()->can('viewAny', User::class)) {
        $users = User::paginate(10); // Ajout de la pagination
        return view('admin.main', compact('users')); // ✅ Correction ici
    }
    return redirect('/');
}
```

- Affiche la **liste paginée des utilisateurs** (10 par page).
- Vérifie que l'utilisateur connecté a la permission `viewAny` sur la classe `User` (via une **policy Laravel**).
- Si autorisé ➜ charge la vue `admin.main` avec les utilisateurs.
- Sinon ➜ redirige vers l'accueil `/`.

2.Méthode (create)

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

- Enregistre un **nouvel utilisateur** à partir du formulaire.

```php
$request->validate([
    'name' => 'required|string|max:255',
    'email' => 'required|email|unique:users,email',
    'password' => 'required|string|min:6',
]);
```

- Le nom est requis, max 255 caractères.
- L'email doit être unique.

- Le mot de passe doit faire au moins 6 caractères.

Création :

```php
User::create([
    'name' => $request->name,
    'email' => $request->email,
    'password' => bcrypt($request->password), // Hachage du mot de passe
    'admin' => $request->has('admin') ? 1 : 0,
]);
```

- Enregistre l'utilisateur avec le mot de passe **haché**.
- Le champ `admin` est coché ? ➜ valeur 1 (sinon 0).

Méthode (édit):

```php
public function edit($id)
{
    $user = User::findOrFail($id);
    return view('admin.edit', compact('user'));
}
```

- Récupère un utilisateur par son ID.
- Charge la vue `admin.edit` avec ses données pour modification.

Méthode (Update):

```php
public function update(Request $request, $id)
{
    $user = User::findOrFail($id);

    $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|email|unique:users,email,' . $user->id,
    ]);

    $user->update([
        'name' => $request->name,
        'email' => $request->email,
        'role' => $request->role,
    ]);

    return redirect()->route('admin.index')->with('success', 'Utilisateur mis à jour avec succès.');
}
```

- Met à jour le nom, l'email et le rôle (le champ `role` n'est pas validé ici, à sécuriser si besoin).
- Tout en récupérant et validant les requête données

Méthode (Destroy):

```php
public function destroy($id)
{
    $user = User::findOrFail($id);
    $user->delete();

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

- Supprime un utilisateur existant.
- Redirige avec un message de confirmation.