

WF.SDAD.DOMP.23

STEPHAN HOEKSEMA & POLINA KOZLOVA



# Laravel

Hier zijn de stappen om een nieuwe Laravel-applicatie genaamd "F1Registration" te maken voor zowel Windows- als Macintosh-laptops, inclusief MySQL-databaseconfiguratie, bootstrap-UI en authenticatie:

## Voor Windows:

### 1. Installeren van PHP en Composer:

- Download en installeer [PHP voor Windows](#).
- Stel je `php.ini` bestand in (meestal te vinden in de `php` map) om extensions zoals `pdo_mysql` in te schakelen.
- Download en installeer [Composer](#) - een PHP-pakketbeheerder.

### 2. Installeren van Laravel Installer:

```
composer global require laravel/installer
```

### 3. Installeren van MySQL:

- Download en installeer [MySQL voor Windows](#).

### 4. Maak de Laravel App:

```
laravel new F1Registration
```

## Voor Macintosh:

### 1. Installeren van PHP en Composer:

- PHP is meestal al geïnstalleerd op Mac. Controleer met `php -v`.
- Installeer [Composer](#).

### 2. Installeren van Laravel Installer:

```
composer global require laravel/installer
```

### 3. Installeren van MySQL:

- Gebruik [Homebrew](#) (als je dit nog niet hebt, installeer het dan eerst).

```
brew install mysql
```

### 4. Maak de Laravel App:

```
laravel new F1Registration
```

## Algemene vervolgstappen (voor zowel Windows als Mac):

## 5. Navigeer naar de Projectmap met de cmd/terminal (phpStorm):

```
cd F1Registration
```

## 6. Configuratie van de Database:

- Bewerk `.env` in je project root:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=f1registration
DB_USERNAME=je_mysql_username
DB_PASSWORD=je_mysql_password
```

## 7. Installeer Laravel UI en Bootstrap:

```
composer require laravel/ui
php artisan ui bootstrap --auth
```

## 8. Voer Migrations uit:

- Maak eerst een nieuwe database genaamd "f1registration" in MySQL.

```
#inloggen
mysql -u je_mysql_username -p
je_mysql_password

#database maken
create database f1registration;
exit

# in de root van je applicatie
php artisan migrate
```

Nu heb je een nieuwe Laravel-applicatie "F1Registration" met Bootstrap UI en authenticatie. Open je browser en ga naar `http://localhost:8000` (nadat je `php artisan serve` hebt uitgevoerd) om de nieuwe applicatie te bekijken.

# Profile Model -a

Hier zijn de stappen om een nieuw `Profile` model uit te maken met een een-op-een-relatie met de `User` model:

## 1. Genereer het Model, Migratie, Controller, en Factory:

Met de `-a` optie wordt er een migratiebestand, een seeder, een factory en een resource controller worden gegenereerd. Daarnaast worden er nog een aantal andere onderdelen gegenereerd wanneer deze nodig zijn worden ze besproken. Het volgende voeren we uit in de root van onze applicatie:

```
php artisan make:model Profile -a
```

Door de -a te gebruiken bij het maken van de model worden er nog twee request classes gemaakt: StoreProfileRequest en UpdateProfileRequest hierin staat bij beide:

```
public function authorize(): bool
{
    return false;
}
```

Zet deze waarde voorlopig op **true**. Dit ga je gebruiken wanneer je specifieke autorisatie gaat gebruiken op verschillende niveau's.

## 2. Wijzig het Migratiebestand:

Ga naar de `database/migrations` map en zoek het `create_profiles_table` migratiebestand. Voeg de volgende velden toe (het bestand wordt vooraf gegaan aan een key van datum en tijd wanneer het bestand gemaakt is):

```
public function up()
{
    Schema::create('profiles', function (Blueprint $table) {
        $table->id();
        $table->unsignedBigInteger('user_id')->unique(); // Foreign key voor user
        $table->string('firstname');
        $table->string('lastname');
        $table->string('mobile');
        $table->timestamps();

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

Hou het klein en begin met een aantal velden je kan het op een later moment nog uitbreiden. Opnieuw de migratie uitvoeren en met een frisse database beginnen.

## 3. Wijzig het Model:

Voeg in `Profile.php` de relatie toe naar `User` en de velden die toegewezen mogen worden in een query. Er zijn twee manieren van toewijzen **\$fillable**, dit zijn de velden die aangepast mogen worden in een mass-assignment of **\$guarded** dit zijn de velden die niet in een mass-assignment aangepast mogen worden. Het is aan de organisatie waar jij gaat werken wat ze gebruiken. Hier heb ik `$fillable` gebruikt omdat het niet zoveel velden zijn. De volgende onderdelen moeten aangepast worden in de Model file:

```
protected $fillable = ['firstname', 'lastname', 'mobile'];

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

Zie dat hier de relatie wordt aangegeven met **belongsTo**. De profiel behoort bij een gebruiker. De functienaam kun je aanroepen. Dit komt door de autoloading van de classes.

En in model `user.php` voegen we de relatie toe naar `Profile`:

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

Zie dat hier de relatie wordt aangegeven met **hasOne**. De user heeft een profiel. Deze twee classes kunnen we ook benaderen in de Eloquent en de class/method namen gebruiken. Het is dus belangrijk dat je afspreekt met je team hoe je omgaat met die namen.

#### 4. Wijzig de Factory:

Ga naar `database/factories` en open `ProfileFactory.php`. Vul de velden in:

```
public function definition()
{
    return [
        'firstname' => $this->faker->firstName,
        'lastname' => $this->faker->lastName,
        'mobile' => $this->faker->phoneNumber,
    ];
}
```

**Faker** is een method die zorgt voor fake informatie. Ga voor jezelf na wat er allemaal mogelijk is met faker. Dus wat voor soort velden kan allemaal vullen en wat is handig.

Laat bijvoorbeeld zien dat een user 1 of meerdere kinderen kan hebben. We hebben het nu over fake data maar meer dan vijf komt haast niet voor. Hoe zou je dit voor elkaar kunnen krijgen met faker?

#### 5. Maak de Seeder:

In `database/seeders` vind je `ProfileSeeder.php`. Hierin gaan we 10 gebruikers en profielen maken:

```

use App\Models\User;
use App\Models\Profile;

public function run()
{
    User::factory(10)->create()->each(function($user) {
        $user->profile()->save(Profile::factory()->make());
    });
}

```

Tip: Zorg ervoor dat je `UserFactory.php` hebt zodat je gebruikers kunt genereren. Als deze er niet is, maak dan eerst een UserFactory aan met `php artisan make:factory UserFactory --model=User` en vul de velden in.

## 6. Voer de Migratie en Seeder uit:

Migreren:

```
php artisan migrate
```

Seed:

```
php artisan db:seed --class=ProfileSeeder
```

Nu heb je 10 gebruikers met bijbehorende profielen in je database!

# CRUD profile

Laten we een eenvoudige CRUD (Create, Read, Update, Delete) functionaliteit maken voor het `Profile` model in je Laravel applicatie.

## 1. Controller

Je hebt al een `ProfileController` aangemaakt met de `-a` optie. Laten we de nodige functies daarin implementeren.

- **Create:**

```

public function create()
{
    return view('profiles.create');
}

public function store(Request $request)
{
    $request->validate([
        'user_id' => 'required',
        'firstname' => 'required',
        'lastname' => 'required',
    ]);
}

```

```

        'mobile' => 'required',
    ]);

Profile::create($request->all());

return redirect()->route('profiles.index')
    ->with('success', 'Profile created successfully.');
}

```

- **Read:**

```

public function index()
{
    $profiles = Profile::all();
    return view('profiles.index', compact('profiles'));
}

public function show(Profile $profile)
{
    return view('profiles.show', compact('profile'));
}

```

- **Update:**

```

public function edit(Profile $profile)
{
    return view('profiles.edit', compact('profile'));
}

public function update(UpdateProfileRequest $request, Profile $profile)
{
    $request->validate([
        'firstname' => 'required',
        'lastname' => 'required',
        'mobile' => 'required',
    ]);

    $profile->update($request->all());

    return redirect()->route('profiles.index')
        ->with('success', 'Profile updated successfully.');
}

```

- **Delete:**

```

public function destroy(Profile $profile)
{
    $profile->delete();

    return redirect()->route('profiles.index')
        ->with('success', 'Profile deleted successfully.');
}

```

## 2. Views

Maak een nieuwe map `profiles` in `resources/views/` en voeg de volgende views toe:

- **index.blade.php** (toont alle profielen):

```

@foreach ($profiles as $profile)
    <p>{{ $profile->firstname }} {{ $profile->lastname }} - {{ $profile->mobile }}
</p>
@endforeach

```

- **create.blade.php** (maakt een nieuw profiel):

```

<form action="{{ route('profiles.store') }}" method="POST">
    @csrf

    <input type="hidden" value="{{ Auth::user()->id }}" name="user_id">

    <label>First Name:</label>
    <input type="text" name="firstname">

    <label>Last Name:</label>
    <input type="text" name="lastname">

    <label>Mobile:</label>
    <input type="text" name="mobile">

    <button type="submit">Create</button>
</form>

```

- **edit.blade.php** (bewerkt een bestaand profiel):

```

<form action="{{ route('profiles.update', $profile->id) }}" method="POST">
    @csrf
    @method('PUT')

    <label>First Name:</label>
    <input type="text" name="firstname" value="{{ $profile->firstname }}>

    <label>Last Name:</label>

```

```
<input type="text" name="lastname" value="{{ $profile->lastname }}>  
  
<label>Mobile:</label>  
<input type="text" name="mobile" value="{{ $profile->mobile }}>  
  
<button type="submit">Update</button>  
</form>
```

- **show.blade.php** (toont details van één profiel):

```
<p>{{ $profile->firstname }} {{ $profile->lastname }} - {{ $profile->mobile }}</p>
```

### 3. Routes

Voeg de nodige routes toe aan `web.php`:

```
Route::resource('profiles', ProfileController::class);
```

Dit voegt automatisch alle benodigde routes toe voor CRUD-functionaliteit, zoals `index`, `create`, `store`, `edit`, `update`, `destroy`, en `show`.

Nu heb je een basis CRUD-functionaliteit voor het `Profile` model. Je kunt dit verder uitbreiden met validatie, flashmeldingen, paginering en andere functies volgens de behoeften van je applicatie.

## Validate profile, restrict

Zeker! Laten we de validatie en authenticatieregels toevoegen:

### 1. Validatie:

Je hebt al enige validatie toegevoegd in de `store` en `update` methoden van je `ProfileController`. Hier is een uitgebreider voorbeeld:

```
$request->validate([  
    'user_id' => 'require'  
    'firstname' => 'required',  
    'lastname' => 'required',  
    'mobile' => 'required',  
]);
```

### 2. Authenticatie:

Om te garanderen dat gebruikers ingelogd zijn voordat ze toegang hebben tot profielroutes, kun je gebruik maken van het `auth` middleware:

In je `ProfileController`'s `__construct` methode:

```
public function __construct()
{
    $this->middleware('auth');
}
```

### 3. Autorisatie:

Om te zorgen dat gebruikers alleen hun eigen profielen kunnen bekijken en bewerken, kun je gebruik maken van policy's of eenvoudige checks in je controller.

Een eenvoudige check kan er zo uitzien:

- In je `edit` en `update` methoden:

```
if ($profile->user_id !== auth()->id()) {
    abort(403, 'Unauthorized action.');
}
```

Een geavanceerdere methode met Policy:

a. Genereer een policy:

```
php artisan make:policy ProfilePolicy --model=Profile
```

b. In `ProfilePolicy.php`, definieer een update methode:

```
public function update(User $user, Profile $profile)
{
    return $user->id === $profile->user_id;
}
```

c. Registreer de policy in `AuthServiceProvider`:

```
protected $policies = [
    Profile::class => ProfilePolicy::class,
];
```

d. Gebruik de policy in je `ProfileController`:

```

public function edit(Profile $profile)
{
    $this->authorize('update', $profile);
    return view('profiles.edit', compact('profile'));
}

public function update(Request $request, Profile $profile)
{
    $this->authorize('update', $profile);

    // ... rest van de code ...
}

```

Met deze stappen kunnen gebruikers alleen hun eigen profielen bewerken en wordt er een 403-foutmelding weergegeven als ze proberen een ander profiel te bewerken.

## Race model / CRUD

Natuurlijk! Laten we stap voor stap een `Race` model met CRUD-functionaliteit maken.

### 1. Model, Migratie, Controller en Factory Aanmaken:

```
php artisan make:model Race -a
```

### 2. Migratie Bewerken:

Wijzig het aangemaakte migratiebestand in `database/migrations` voor de races-tabel:

```

public function up()
{
    Schema::create('races', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('location');
        $table->float('length_circuit'); // Lengte in kilometers bijvoorbeeld
        $table->timestamps();
    });
}

```

### 3. Factory Bewerken:

Wijzig `database/factories/RaceFactory.php`:

```
public function definition()
{
    return [
        'name' => $this->faker->word(),
        'location' => $this->faker->city(),
        'length_circuit' => $this->faker->randomFloat(2, 3, 7), // Random waarde tussen
3 en 7 met 2 decimalen
    ];
}
```

#### 4. Seeder Maken:

Nu, om de tabel met dummy-gegevens te vullen, maken we een seeder:

```
php artisan make:seeder RaceSeeder
```

Vul `RaceSeeder.php` in:

```
use App\Models\Race;

public function run()
{
    Race::factory(10)->create(); // Maakt 10 races
}
```

Om de seeder te gebruiken:

```
php artisan db:seed --class=RaceSeeder
```

#### 5. Controller Bewerken:

In `RaceController.php`, maak de basis CRUD-methoden aan (zoals eerder uitgelegd voor [Profile](#)).

#### 6. Routes Toevoegen:

In `web.php`:

```
Route::resource('races', RaceController::class);
```

#### 7. Views Maken:

Maak een nieuwe map `races` in `resources/views/` en voeg de CRUD-views toe: `index.blade.php`, `create.blade.php`, `edit.blade.php`, `show.blade.php`.

Ik zal een voorbeeld geven voor `create.blade.php`, de anderen zijn vergelijkbaar:

```

<form action="{{ route('races.store') }}" method="POST">
    @csrf
    <label>Name:</label>
    <input type="text" name="name">

    <label>Location:</label>
    <input type="text" name="location">

    <label>Length of Circuit (in km):</label>
    <input type="text" name="length_circuit">

    <button type="submit">Create</button>
</form>

```

Nu heb je een `Race` model met CRUD-functionaliteit in je Laravel applicatie. Dit is een basisimplementatie en kan verder worden aangepast en uitgebreid volgens de specifieke behoeften van je applicatie.

## Racereults

---

Natuurlijk! Het `RaceResult` model dat je voorstelt zou een relatie hebben met zowel de `User` als de `Race` modellen. Laten we dit stap voor stap opbouwen:

### 1. Model en Migratie Aanmaken:

Maak het `RaceResult` model met een migratie:

```
php artisan make:model RaceResult -m
```

### 2. Migratie Bewerken:

In `database/migrations/[timestamp]_create_race_results_table.php`, wijzig de up-methode:

```

public function up()
{
    Schema::create('race_results', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained()->onDelete('cascade');
        $table->foreignId('race_id')->constrained()->onDelete('cascade');
        $table->time('laptimes'); // Dit slaat de ronde tijd op in het HH:MM:SS formaat
        $table->timestamps();
    });
}

```

### 3. Models Aanpassen:

- **RaceResult:**

In `RaceResult.php`:

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

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

- **User:**

In `User.php`:

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

- **Race:**

In `Race.php`:

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

#### 4. Controller Aanmaken:

```
php artisan make:controller RaceResultsController --resource
```

In `RaceResultsController.php`, vul de CRUD-methoden in (vergelijkbaar met de eerder beschreven methoden voor `Profile` en `Race`).

#### 5. Routes Toevoegen:

In `web.php`:

```
Route::resource('race-results', RaceResultsController::class);
```

#### 6. Views Maken:

Maak een nieuwe map `race-results` in `resources/views/` en voeg de CRUD-views toe.

- Voor `create.blade.php`:

```
<form action="{{ route('race-results.store') }}" method="POST">
```

```

@csrf
<label>User:</label>
<select name="user_id">
    @foreach ($users as $user)
        <option value="{{ $user->id }}>{{ $user->name }}</option>
    @endforeach
</select>

<label>Race:</label>
<select name="race_id">
    @foreach ($races as $race)
        <option value="{{ $race->id }}>{{ $race->name }}</option>
    @endforeach
</select>

<label>Laptime:</label>
<input type="time" name="laptime" step="1"> <!-- 'step' zorgt ervoor dat seonden kunnen worden ingevoerd -->

<button type="submit">Submit</button>
</form>

```

**Opmerking:** De `create` methode van `RaceResultsController` moet de lijsten van gebruikers en races naar deze view sturen:

```

public function create()
{
    $users = User::all();
    $races = Race::all();
    return view('race-results.create', compact('users', 'races'));
}

```

Je kunt vervolgens vergelijkbare views maken voor `index`, `edit`, en `show`.

Nu heb je een `RaceResult` model met CRUD-functionaliteit. Je kunt dit verder aanpassen en uitbreiden volgens de behoeften van je applicatie.

## authenticatie en validatieproblemen aanpakken:

### 1 . Authenticatie:

Zorg ervoor dat alleen ingelogde gebruikers toegang hebben tot de CRUD-routes van `RaceResult`.

In de constructor van `RaceResultsController`:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Met het bovenstaande zullen alle methoden in `RaceResultController` vereisen dat de gebruiker is ingelogd.

## 2. Validatie met Request Classes:

Als je problemen hebt met `Request` classes voor het opslaan en bijwerken van gegevens, kunnen we die aanpakken door specifieke `FormRequest` classes te maken voor validatie.

### a. Maak Request Classes:

Voor het opslaan en bijwerken van `RaceResult`:

```
php artisan make:request StoreRaceResultRequest
php artisan make:request UpdateRaceResultRequest
```

### b. Definieer Validatieregels:

In `StoreRaceResultRequest.php`:

```
public function rules()
{
    return [
        'user_id' => 'required|exists:users,id',
        'race_id' => 'required|exists:races,id',
        'laptimes' => 'required|date_format:H:i:s',
    ];
}
```

In `UpdateRaceResultRequest.php`, je kunt dezelfde regels gebruiken (of aangepaste regels indien nodig).

### c. Gebruik deze in je Controller:

In `RaceResultController.php`, update de `store` en `update` methoden om de nieuw gecreëerde requests te gebruiken:

```
use App\Http\Requests\StoreRaceResultRequest;
use App\Http\Requests\UpdateRaceResultRequest;

...

public function store(StoreRaceResultRequest $request)
{
    // $request is nu al gevalideerd!
```

```

    $raceResult = new RaceResult($request->validated());
    $raceResult->save();
    return redirect()->route('race-results.index');
}

public function update(UpdateRaceResultRequest $request, RaceResult $raceResult)
{
    // $request is nu al gevalideerd!
    $raceResult->fill($request->validated());
    $raceResult->save();
    return redirect()->route('race-results.index');
}

```

## Belangrijke Opmerkingen:

1. Als je een 403-fout krijgt bij het gebruik van de `FormRequest` voor validatie, komt dit omdat de `authorize` methode standaard `false` retourneert. Je kunt dit veranderen in `true` in je `FormRequest` classes als je geen specifieke autorisatielogica nodig hebt.
2. Zorg ervoor dat je de relevante `use` statements bovenaan je bestanden hebt om klassen correct te refereren.
3. Aangezien we de `date_format:H:i:s` validatiergel gebruiken voor `lapttime`, wordt ervan uitgegaan dat de lapttime wordt ingevoerd in het formaat HH:MM:SS. Als je een ander formaat gebruikt, pas de validatiergel dan dienovereenkomstig aan.

# LAPTIME

Om de lapttime in het formaat `minuten:seconden:milliseconden` (bijv. `02:34:567`) op te slaan en te valideren, moet je enkele aanpassingen doorvoeren:

## 1. Aanpassen van de Migratie:

In je migratie voor de `race_results` tabel, zou je de `lapttime` kolom van het type `time` naar `string` kunnen veranderen, zodat het geschikt is om milliseconden op te slaan:

```
$table->string('lapttime', 9); // Lengte is 9 om ruimte te maken voor "MM:SS:MMM"
```

Voer vervolgens een migratie uit om deze wijziging door te voeren:

```
php artisan migrate:fresh
```

Let op: `migrate:fresh` zal alle tabellen in de database verwijderen en ze opnieuw aanmaken. Dit betekent dat je alle gegevens in de database verliest.

Als je de gegevens niet wilt verliezen, maak dan een nieuwe migratie om de kolom te wijzigen en voer `php artisan migrate` uit.

## 2. Aanpassen van de Validatieregels:

In je `StoreRaceResultRequest` en `UpdateRaceResultRequest`:

```
'laptimes' => 'required|regex:/^([0-5]?[0-9]:[0-5][0-9]:[0-9]{3})$/'
```

Deze regex valideert de input in het formaat `MM:SS:MMM`.

## 3. Werken met deze Data in je Applicatie:

Wanneer je met dit gegevensformaat in je applicatie werkt (bijvoorbeeld om laptimes te vergelijken), zul je waarschijnlijk willen converteren tussen het `MM:SS:MMM` formaat en een puur aantal milliseconden. Dit zal berekeningen en vergelijkingen veel eenvoudiger maken.

Bijvoorbeeld, om te converteren van `MM:SS:MMM` naar milliseconden:

```
list($minutes, $seconds, $milliseconds) = explode(':', $laptimes);
$totalMilliseconds = ($minutes * 60 * 1000) + ($seconds * 1000) + $milliseconds;
```

En vice versa:

```
$minutes = floor($totalMilliseconds / (60 * 1000));
$seconds = floor(($totalMilliseconds - $minutes * 60 * 1000) / 1000);
$milliseconds = $totalMilliseconds - ($minutes * 60 * 1000) - ($seconds * 1000);

$laptimes = sprintf('%02d:%02d:%03d', $minutes, $seconds, $milliseconds);
```

Met deze aanpassingen kun je laptimes in het gewenste formaat opslaan, valideren en ermee werken in je Laravel applicatie.

Om een leaderboard te implementeren voor de beste laptimes per race, zou je in principe kunnen voortbouwen op de bestaande modellen en structuren die je al hebt. Een specifiek "Leaderboard"-model is niet per se nodig; in plaats daarvan kun je berekeningen en sorteringen uitvoeren op basis van de `RaceResult` data die je al hebt.

Laten we stapsgewijs de noodzakelijke implementatie doorlopen:

### 1. Race Aanpakken:

In de veronderstelling dat er een attribuut is dat aangeeft of een race aankomend is (bijv. een `date` kolom in de `Race` model), kun je dit gebruiken om de aankomende race te identificeren.

### 2. Controller Aanmaken:

Maak een controller aan die de leaderboard-logica zal hanteren:

```
php artisan make:controller LeaderboardController
```

### 3. Methode voor het Ophalen van de Leaderboard Data:

In `LeaderboardController.php`, voeg een methode toe voor het ophalen van de leaderboard data voor de aankomende race:

```
use App\Models\Race;
use App\Models\RaceResult;

public function upcomingRaceLeaderboard()
{
    $upcomingRace = Race::where('date', '>', now())->orderBy('date', 'asc')->first();
    // Aannemend dat 'date' het veld is dat de datum van de race bevat

    if (!$upcomingRace) {
        return view('no_race'); // Een view die een bericht toont als er geen
    aankomende race is
    }

    $leaderboard = RaceResult::where('race_id', $upcomingRace->id)
        ->orderBy('lapttime', 'asc') // Snelste tijd eerst
        ->with('user') // Laad gebruikersdata voor elk resultaat
        ->get();

    return view('leaderboard.upcoming', ['leaderboard' => $leaderboard, 'race' =>
    $upcomingRace]);
}
```

### 4. Routes Toevoegen:

In `web.php`, voeg de route toe:

```
Route::get('/leaderboard/upcoming', [LeaderboardController::class,
    'upcomingRaceLeaderboard'])->name('leaderboard.upcoming');
```

### 5. De Leaderboard View:

Maak een nieuw Blade-bestand `upcoming.blade.php` in een map `leaderboard` onder `resources/views`.

In `resources/views/leaderboard/upcoming.blade.php`:

```
<h1>Leaderboard voor {{ $race->name }}</h1>

<table>
    <thead>
        <tr>
            <th>Positie</th>
            <th>Gebruiker</th>
            <th>Ronde Tijd</th>
```

```

        </tr>
    </thead>
    <tbody>
        @foreach($leaderboard as $index => $result)
        <tr>
            <td>{{ $index + 1 }}</td>
            <td>{{ $result->user->name }}</td>
            <td>{{ $result->laptimes }}</td>
        </tr>
    @endforeach
    </tbody>
</table>

```

## 6. Toon het Leaderboard op de Startpagina:

Wanneer een gebruiker naar de startpagina gaat of inlogt, wil je misschien het leaderboard tonen.

In de betreffende controller-methode (bijv. `HomeController@index` als je een aparte HomeController hebt):

```

public function index()
{
    // Haal het leaderboard op zoals in de LeaderboardController
    // ...

    return view('home', ['leaderboard' => $leaderboard, 'race' => $upcomingRace]);
}

```

Vervolgens kun je in de `home.blade.php` (of waar je ook de startpagina's view hebt) het leaderboard integreren, vergelijkbaar met het `upcoming.blade.php` voorbeeld.

Dit zou een basisimplementatie moeten bieden voor een leaderboard-functionaliteit in je Laravel-app. Je kunt het verder uitbreiden en aanpassen aan de specifieke behoeften en ontwerpeisen van je applicatie.

# API Integratie

Om dit in je applicatie te integreren, zou je een aantal stappen moeten volgen:

### 1. API Integratie:

Eerst moet je een manier hebben om de API aan te roepen en gegevens op te halen over de races van het komende seizoen.

Laravel ondersteunt HTTP-client functionaliteit vanaf versie 7.x. Hiermee kun je API-aanvragen doen.

```

use Illuminate\Support\Facades\Http;

$response = Http::get('http://ergast.com/api/f1/current.json');
$races = $response->json()['MRData']['RaceTable']['Races'];

```

## 2. Database Opslag:

Wanneer je de gegevens hebt, wil je deze waarschijnlijk opslaan in je `Races` tabel. Dit zorgt ervoor dat je niet bij elke aanvraag de externe API moet aanspreken.

```
foreach ($races as $raceData) {
    Race::create([
        'name' => $raceData['raceName'],
        'location' => $raceData['Circuit']['Location']['locality'],
        'date' => $raceData['date'],
        // Voeg eventueel andere relevante velden toe
    ]);
}
```

## 3. Logica voor het Tonen van Aankomende Races:

Wanneer een gebruiker zijn/haar laptime wil invullen, wil je alleen de aankomende races tonen die nog niet gereden zijn:

```
$upcomingRaces = Race::where('date', '>', now())->orderBy('date', 'asc')->get();
```

## 4. Cronjob of Taakplanner:

Om ervoor te zorgen dat je gegevens over races actueel blijven, kun je een geplande taak maken die regelmatig de API aanroeft, bijvoorbeeld eens per dag of week, om te controleren op nieuwe races of wijzigingen.

In Laravel kan dit worden gedaan met de Task Scheduler. In `App\Console\Kernel.php`:

```
protected function schedule(Schedule $schedule)
{
    $schedule->call(function () {
        // Jouw logica voor het ophalen en opslaan van race data
    })->daily(); // of weekly(), afhankelijk van jouw behoefte
}
```

Vergeet niet om de Laravel task scheduler op je server in te stellen door de `schedule:run` commando iedere minuut te draaien.

## 5. Foutafhandeling:

Het is altijd een goed idee om foutafhandeling toe te voegen wanneer je externe API's aanroeft. Wat als de API tijdelijk niet beschikbaar is? Of als er wijzigingen in de structuur van de data zijn? Een eenvoudige manier is om de HTTP-statuscode te controleren en foutmeldingen af te handelen.

## 6. Caching:

Om de prestaties te optimaliseren, kan het nuttig zijn om de opgehaalde racegegevens tijdelijk op te slaan (te cachen). Laravel biedt krachtige caching-mogelijkheden die je kunt gebruiken om de API-reacties tijdelijk op te slaan, zodat je niet bij elke verzoek opnieuw gegevens hoeft op te halen.

## 7. Gebruikersinterface:

Zorg voor een intuïtieve gebruikersinterface waar gebruikers de aankomende races kunnen zien en hun laptimes kunnen invoeren.

Met deze stappen zou je de data van de ergast API in je Laravel-applicatie moeten kunnen integreren en gebruiken om de aankomende races te beheren.

# Diagrammen

---

Een goed ontworpen applicatie begint vaak met duidelijke planning en visualisatie. Hieronder beschrijf ik de relevante diagrammen voor je applicatie:

## 1. ERD (Entity Relationship Diagram):

Dit is een diagram dat de relaties tussen de verschillende entiteiten (meestal tabellen in een database) in je applicatie laat zien.

Voor jouw applicatie:

- `User`: Elk gebruiker heeft één `Profile` en kan meerdere `RaceResult` records hebben.
- `Profile`: Behoort tot een `User`.
- `Race`: Kan meerdere `RaceResult` records hebben.
- `RaceResult`: Behoort tot een `Race` en een `User`.

Op basis van bovenstaande:

- Een één-op-één relatie tussen `User` en `Profile`.
- Een één-op-veel relatie tussen `User` en `RaceResult`.
- Een één-op-veel relatie tussen `Race` en `RaceResult`.

## 2. Flow Diagrammen:

Hiermee kun je processen visualiseren.

- **User Registration Flow**: Toont stappen die een gebruiker neemt om te registreren.
- **Race Creation Flow**: Hoe een race wordt gemaakt en opgeslagen.
- **Race Result Entry Flow**: Hoe een gebruiker een lapttime invoert voor een bepaalde race.

## 3. UML Diagrammen:

Dit zijn diagrammen die de structuur, het gedrag en de interacties van het systeem illustreren.

- **Use Case Diagram**: Laat zien hoe verschillende gebruikers (actoren) interactie hebben met het systeem. Bijvoorbeeld: gebruiker registreert, admin maakt race aan, gebruiker voegt lapttime toe, etc.
- **Sequence Diagram**: Beschrijft de interactie tussen objecten in een sequentiële volgorde. Bijvoorbeeld: de stappen die plaatsvinden wanneer een gebruiker een lapttime toevoegt.

## 4. Class Diagrammen:

Dit diagram toont de verschillende klassen in je applicatie, hun attributen, methoden en de relaties ertussen.

Voor jouw applicatie:

- **User Class:** Met attributen zoals `id`, `email`, en methoden gerelateerd aan authenticatie.
- **Profile Class:** Attributen zoals `firstname`, `lastname`, `mobile` en hun relaties.
- **Race Class:** Attributen zoals `name`, `location`, `length_circuit`.
- **RaceResult Class:** Attributen zoals `lapttime` en de relaties met `User` en `Race`.

De relaties tussen deze klassen worden aangegeven met associatie, aggregatie, en compositie lijnen.

## Samenvatting:

- **ERD:** Voor database design.
- **Flow Diagram:** Voor het visualiseren van processen.
- **UML Diagrams:** Voor het vastleggen van de structuur en het gedrag van het systeem.
- **Class Diagram:** Voor het visualiseren van klassen, hun attributen, methoden, en relaties.

Het maken van deze diagrammen helpt niet alleen bij het plannen, maar ook bij het communiceren van je ideeën met teamleden, stakeholders, en zelfs toekomstige ontwikkelaars. Het kan handig zijn om tools zoals Lucidchart, Draw.io, of Microsoft Visio te gebruiken om deze diagrammen te maken.