Bases de données, Eloquent et modèles

https://github.com/heig-vd-devprodmed-course/heig-vd-devprodmed-course

Visualiser le contenu complet sur GitHub à cette adresse.

L. Delafontaine, avec l'aide de GitHub Copilot.

Plus de détails sur GitHub

Cette présentation est un résumé du contenu complet disponible sur GitHub.

Pour plus de détails, consulter le contenu complet sur GitHub ou en cliquant sur l'en-tête de ce document.

Objectifs (1)

- Décrire comment Laravel peut interagir avec plusieurs types de bases de données.
- Décrire le concept de migrations avec Laravel.
- Décrire le concept d'un ORM tel qu'Eloquent.
- Décrire le concept de "query builder" de Laravel.

Objectifs (2)

- Décrire le concept de "seeders" dans Laravel.
- Décrire la partie "modèle" du patron de conception MVC.
- Implémenter ces concepts avec Laravel pour réaliser le petit réseau social du mini-projet.

Laravel et les bases de données (1)

Laravel supporte plusieurs systèmes de gestion de bases de données (SGBD) :

- MySQL / MariaDB.
- PostgreSQL.
- SQLite.
- Et bien d'autres.

Cette intégration est facilitée par l'utilisation de pilotes de base de données PHP et du fichier de configuration `.env`.

Laravel et les bases de données (2)

Le fichier `.env` permet de définir les paramètres de connexion à la base de données de manière simple et sécurisée.

Pour SQLite :

```
DB_CONNECTION=sqlite
```

Pour MySQL :

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=nom_de_la_base
DB_USERNAME=utilisateur
DB_PASSWORD=mot_de_passe
```

Laravel et les bases de données (3)

Avantages de cette approche :

- Changement de SGBD sans modifier le code.
- Séparation de la configuration et du code.
- Sécurité : **le fichier `.env` ne doit jamais être partagé publiquement**.
- Abstraction : Laravel gère les différences entre les SGBD.

Plus besoin de fichiers de configuration INI comme en ProgServ2.

Migrations

Les migrations sont des fichiers qui décrivent la structure de la base de données ainsi que son évolution au fil du temps.

Elles permettent de gérer la base de données de manière versionnée, comme on gère le code avec Git.

Plus facile de maintenir et d'évoluer la base de données de façon indépendante du code applicatif.

Structure d'une migration (1)

Chaque migration contient deux méthodes principales :

1. `up()` : définit les modifications à apporter à la base de données (création/modification de tables, colonnes, index, etc.).
2. `down()` : définit comment annuler les modifications effectuées par `up()`.

Utilisation des classes `Schema` et `Blueprint` de Laravel pour définir les tables et leurs colonnes.

Structure d'une migration (2)

`up()` : modifications à apporter à la base de données.

```php
public function up(): void
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password');
        $table->timestamps();
    });
}
```

Structure d'une migration (3)

`down()` : comment annuler les modifications effectuées par `up()`.

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

Créer une nouvelle migration (1)

Pour créer une nouvelle migration :

```
php artisan make:migration create_demo_table
```

Résultat :

```
INFO  Migration [database/migrations/2026_01_28_144030_create_demo_table.php]
created successfully.
```

Le fichier est créé dans `database/migrations/`.

Créer une nouvelle migration (2)

Laravel génère automatiquement un fichier avec un horodatage pour garantir l'ordre d'exécution des migrations.

Format du nom : `YYYY_MM_DD_HHMMSS_nom_de_la_migration.php`

Les migrations sont exécutées dans l'ordre de leur horodatage, ce qui permet de gérer les dépendances entre les différentes migrations.

Modifier une migration

Modifiez les méthodes `up()` et `down()` pour correspondre à la structure de la table souhaitée.

Ne modifiez jamais une migration qui a déjà été appliquée en production. Créez une nouvelle migration si besoin.

Ceci peut entraîner des incohérences et des problèmes de maintenance qui sont difficiles à résoudre.

Appliquer les migrations

Pour appliquer les migrations à la base de données :

```
php artisan migrate
```

Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés.

Annuler les migrations

Pour annuler la dernière migration appliquée :

```
php artisan migrate:rollback
```

Cela permet de revenir en arrière en annulant les modifications apportées par la/les méthode(s) `up` du/des fichier(s) de migrations qui a/ont été appliqué(s) en dernier.

Annuler une migration en production peut entraîner des pertes de données. Soyez très prudent.e.

Le concept d'ORM

Un ORM (Object-Relational Mapping) permet de lier les tables de la base de données à des classes et les enregistrements à des objets.

C'est un pont entre le monde relationnel (base de données) et le monde objet (programmation orientée objet).

Au lieu d'écrire du SQL :

```sql
SELECT * FROM users WHERE id = 1;
UPDATE users SET name = 'Alice' WHERE id = 1;
DELETE FROM users WHERE id = 1;
```

On utilise des objets :

```php
$user = User::find(1);
$user->name = 'Alice';
$user->save();
$user->delete();
```

Avantages d'un ORM

Les avantages d'un ORM :

- **Abstraction** : pas besoin de connaître SQL en détail.
- **Sécurité** : protection contre les injections SQL.
- **Maintenabilité** : code plus lisible et facile à modifier.
- **Portabilité** : changer de SGBD ne nécessite pas de modifier le code.

Inconvénients d'un ORM

Les inconvénients d'un ORM :

- **Performance** : peut être moins performant que des requêtes SQL optimisées.
- **Abstraction** : peut masquer les détails de la base de données, ce qui peut être problématique pour le débogage.
- **Spécificité** : peut être spécifique à un framework ou un langage, limitant la portabilité du code.

Quand utiliser un ORM ?

Un ORM est recommandé pour la plupart des applications car il permet de développer plus rapidement et de manière plus sécurisée.

Pour des cas très spécifiques où la performance est critique, il peut être nécessaire d'écrire des requêtes SQL personnalisées.

Eloquent : l'ORM de Laravel (1)

Eloquent est l'ORM inclus dans Laravel. Un modèle Eloquent représente une table de la base de données et chaque instance de ce modèle représente une ligne de cette table.

Un modèle Eloquent contient généralement (entre autres) :

- La logique d'accès aux données.
- Les relations entre les modèles.
- Les méthodes métier.

Eloquent : l'ORM de Laravel (2)

Par convention, Laravel suppose que :

- Le modèle `User` correspond à la table `users` (pluriel en minuscules).
- Les clés primaires s'appellent `id`.
- Les timestamps `created_at` et `updated_at` sont gérés automatiquement.

Créer un modèle (1)

Pour créer un modèle Eloquent :

```
php artisan make:model User
```

Cela crée un fichier `app/Models/User.php` :

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    //
}
```

Créer un modèle (2)

Le modèle peut contenir des méthodes métier :

```php
class User extends Model
{
    public function isAdmin(): bool
    {
        return $this->role === 'admin';
    }

    public function getFullName(): string
    {
        return $this->first_name . ' ' . $this->last_name;
    }
}
```

Utiliser un modèle

Pour créer un nouvel enregistrement dans la table `users` :

```php
$user = new User();

$user->name = 'Alice';
$user->email = 'alice@example.com';

$user->save();
```

Opérations CRUD avec Eloquent

Eloquent facilite les opérations CRUD sur les modèles :

- **Create** : créer un nouvel enregistrement.
- **Read** : lire des enregistrements existants.
- **Update** : mettre à jour des enregistrements existants.
- **Delete** : supprimer des enregistrements existants.

Créer

```php
$user = new User();

$user->name = 'Alice';
$user->email = 'alice@example.com';

$user->save();
```

Lire

```php
// Récupérer un.e utilisateur.trice par ID
$user = User::find(1);

// Récupérer tou.tes les utilisateur.trices
$users = User::all();

// Récupérer avec une condition
$users = User::where('email', 'like', '%example.com')->get();
```

Mettre à jour

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

$user->name = 'Alice Updated';

$user->save();
```

Supprimer

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

$user->delete();

// Ou directement
User::destroy(1);
```

Gérer les relations entre modèles (1)

Eloquent permet de définir des relations entre les modèles :

- **One-to-One** : un.e utilisateur.trice a un profil.
- **One-to-Many** : un.e utilisateur.trice a plusieurs posts.
- **Many-to-Many** : un.e utilisateur.trice peut avoir plusieurs rôles.

Ces relations facilitent l'accès aux données liées.

Prenons un exemple de relation One-to-Many entre `User` et `Post`.

Gérer les relations entre modèles (2)

```php
// app/Models/User.php
class User extends Model {
    public function posts() {
        return $this->hasMany(Post::class);
    }
}

// app/Models/Post.php
class Post extends Model {
    public function user() {
        return $this->belongsTo(User::class);
    }
}
```

Gérer les relations entre modèles (3)

```php
// Récupérer un.e utilisateur.trice avec l'ID 1
$user = User::find(1);

// Récupérer les posts de l'utilisateur.trice
$posts = $user->posts;

// Récupérer un post avec l'ID 1
$post = Post::find(1);

// Récupérer l'utilisateur.trice du post
$user = $post->user;
```

Requêtes et query builder

Eloquent utilise un *"query builder"* pour construire les requêtes SQL de manière fluide (= chaîner plusieurs méthodes) et orientée objet.

Permet d'interagir avec la base de données sans écrire de SQL brut.

```php
// Chaîner plusieurs conditions
$users = User::where('active', true)
    ->where('created_at', '>', now()->subDays(7))
    ->orderBy('name')
    ->get();
```

La documentation officielle est **très** exhaustive.

Seeders (1)

Les seeders sont des classes qui permettent de remplir la base de données avec des données prédéfinies. Très utiles pour :

- Tester les fonctionnalités de l'application avec des données factices.
- Insérer des données de référence (rôles, catégories).
- Créer des comptes administrateurs.
- Préparer des environnements de développement ou de démonstration.

Seeders (2)

Pour créer un seeder :

```
php artisan make:seeder UserSeeder
```

Cela crée un fichier `database/seeders/UserSeeder.php` :

```php
class UserSeeder extends Seeder
{
    public function run(): void
    {
        //
    }
}
```

Seeders (3)

Remplir le seeder :

```php
public function run(): void
{
    DB::table('users')->insert([
        'name' => 'Alice',
        'email' => 'alice@example.com',
    ]);
}
```

Il est aussi possible d'utiliser les modèles Eloquent plutôt que passer par l'objet `DB`.

Seeders (4)

Pour exécuter les seeders, vous pouvez utiliser la commande suivante :

```
php artisan db:seed
```

Ou encore cette commande pour réinitialiser la base de données et exécuter les seeders :

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

Le modèle dans le patron de conception MVC (1)

Un patron de conception (*"design pattern"*) est une solution réutilisable à un problème de conception courant.

Le patron **MVC** (Model-View-Controller) est largement utilisé dans le développement d'applications web pour séparer les préoccupations :

- **Modèle** : représente les données et la logique métier.
- **Vue** : affiche les données à l'utilisateur.trice (interface).
- **Contrôleur** : gère les requêtes et coordonne le modèle et la vue.

Le modèle dans le patron de conception MVC (2)

Le modèle est responsable de :

- La gestion des données.
- Les règles métier.
- L'interaction avec la base de données.
- Les relations entre les entités.

C'est la partie centrale du patron MVC, car elle encapsule les données de l'application.

Pourquoi commencer par le modèle ?

Il est recommandé de commencer par le modèle lors du développement d'une application, car cela permet de définir clairement les données et la logique métier **avant** de se préoccuper de l'affichage ou de la gestion des requêtes.

Dans toute application, les technologies peuvent évoluer mais les **données** restent au cœur de l'application.

Une approche centrée sur le modèle permet de construire une base solide qui peut évoluer au fil du temps sans compromettre la structure des données.

Utiliser Artisan pour gérer les modèles

Laravel propose des commandes Artisan pour créer et gérer les modèles Eloquent de manière rapide et efficace.

Artisan est l'interface en ligne de commande de Laravel.

Pour plus d'informations, consulter la documentation officielle de Laravel : https://laravel.com/docs/12.x/eloquent#generating-model-classes.

Conclusion

Les migrations, Eloquent et Artisan sont des outils puissants pour gérer les bases de données dans Laravel.

Ils permettent de travailler avec les données de manière orientée objet, tout en gardant une abstraction sur le SGBD utilisé.

Questions

Est-ce que vous avez des questions ?

À vous de jouer !

- (Re)lire le contenu de cours.
- Faire les exercices.
- Faire le mini-projet.
- Poser des questions si nécessaire.

➡️ Visualiser le contenu complet sur GitHub.

N'hésitez pas à vous entraidez si vous avez des difficultés !

Sources

- Illustration principale par Richard Jacobs sur Unsplash.
- Illustration par Aline de Nadai sur Unsplash.
- Illustration par Nikita Kachanovsky sur Unsplash.