Chapitre 5

Modèles et ORM

Les modèles dans Laravel sont utilisés pour interagir avec la base de données en fournissant des méthodes pour récupérer, stocker, modifier et supprimer les enregistrements de chaque table. On dirait que les modèles combinent les fonctionnalités d'un objet et d'un enregistrement de base de données en une seule entité. Ils peuvent également inclure des relations avec d'autres tables pour permettre des requêtes et des manipulations de données plus complexes. Ainsi, lors de la création d'un modèle dans Laravel, il est souvent recommandé de créer une migration pour la table correspondante. le modèle peut être utilisé pour interagir avec cette table. De même, lorsque des modifications sont apportées à la structure de la base de données à l'aide de migrations, les modèles doivent souvent être mis à jour pour refléter ces modifications.

5.1 Créer des modèles

Dans Laravel, nous créons des modèle à l'aide de la commande `make:model` dans la CLI `artisan`. L'exemple suivant montre comment créer un modèle appelé `Product`.

```
php artisan make:model Product
```

Cette commande permet de créer un fichier `Product.php` dans le répertoire `app\models`. Le contenu initial du fichier est le suivant :

```php
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
class Product extends Model {
    use HasFactory;
}
```

L'observation la plus importante ici est que la classe `Product` hérite de la classe `Model`. cette dernière fournit de nombreuses fonctionnalités utiles couramment utilisés lors de la manipulation de bases de données. En héritant de la classe `Model`, les modèles Laravel héritent de nombreuses méthodes et propriétés utiles qui simplifient l'interaction avec la base de données. Par exemple, la classe `Model` fournit des méthodes pour définir des relations entre modèles, interroger la base de données ... etc. En héritant de cette classe, les modèles Laravel héritent automatiquement toutes ces fonctionnalités, ce qui permet de travailler plus rapidement et plus facilement avec les bases de données dans Laravel.

L'instruction `use HasFactory` va être supprimée et ignoré dans le reste de ce cours. En pratique, elle permet de générer des données de manière aléatoire pour pour remplir la base de données avec des données de test.

5.1.1 La table correspondante

Tous les modèles Laravel sont automatiquement liés à des tables dans la base de données. Par convention, le pluriel du nom de la classe du modèle sera utilisé pour la table, sauf si une autre table a été explicitement spécifiée. Le format `snake_case` est utilisé pour nommer les table et pas `CamelCase`. L'exemple ci-dessous montre le nom de table correspondant par défaut à chaque nom de modèle.

```
Modèle      | Table
Product     | products
UserProfile | user_profiles
```

```
Factory     | factories
```

Si pour n'importe quelle raison vous voulez que le modèle soit liée à une table différente, vous devez simplement créer une propriété dans la classe du modèle appelée `$table` et y mettre le nom de la table de votre choix. La même chose pour la colonne de la clé primaire qui est automatiquement supposée `id`, mais pourrait être changée en déclarant une variable nommée `$primaryKey` contenant le nom de la colonne de la clé primaire. L'exemple suivant lie le modèle `Product` à la table `produits` ayant la clé primaire `code_produit` :

```
<?php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;
class Product extends Model {
    protected $table = "produits";
    protected $primaryKey = "code_produit";
}
```

5.1.2 Migrations, contrôleurs et ressources

Vu que les modèles sont toujours associés à une table dans la base de données, et que les opérations sur le même modèle sont généralement isolées dans un contrôleur qui lui est aussi associé, Laravel offre la possibilité de créer en une seule commande le fichier de migration et le contrôleur qui lui sont associés. Le fichier de migration est créé à l'aide de l'option `--migration` alors que le contrôleur pourrait être créé à l'aide de l'option `--conroller`. Si nous voulons créer un contrôleur de ressource, l'option `--resource` est aussi ajoutée.

```
php artisan make:model Product --migration --controller --resource
```

A l'issue de la commande ci-dessus, trois fichiers seront créés, un fichier `Product.php` dans le répertoire `app/Models`, un fichier `ProductController.php` dans le répertoire `app/Http/Controllers`, et finalement un fichier daté portant le nom `create_products_table.php` dans le réper-

toire `database/migrations`.

5.2 Récupérer les données

Une fois que le modèle est créé, et que la migration de la table correspondante est exécutée, L'ORM (Object-Relation Mapping) utilisé par Laravel va créer un mapping entre les instance du modèle créé et les données dans la base de données. Plusieurs méthodes permettant d'accéder à la base de données sont définies dans la classe `Model`. Nous allons voir dans cette section deux méthodes qui permettent de récupérer les données. La méthode `all()` et la méthode`find()`, les deux sont naturellement des méthodes statiques et donc peuvent être appelées sans créer une instance du modèle en question.

La méthode `all()` permet de récupérer toutes les lignes de la table liée au modèle. L'exemple ci-dessous permet de récupérer tous les produits dans la base de données dans la variable `$products`.

```
$products = Product::all();
```

La méthode all retourne une `Collection`, un objet qui permet de manipuler facilement des ensembles de données. Les collections offrent une variété de méthodes qui facilitent le filtrage, le tri, le regroupement et la transformation de données ... etc. Il est possible de parcourir les éléments d'une collection à l'aide de la boucle `foreach`, mais le plus souvent, cette collection est passée à une vue qui utilise la directive Blade `@foreach` pour la parcourir.

```
<?php
...
use App\Models\Product;
...
class ProductController extends Controller {
    ...
    public function index() {
```

```
        $products = Product::all();
        return view('products.index')->with('products', $products);
    }
    ...
}
```

Dans l'exemple précédent, nous implémentons la méthode `index` dans le contrôleur `ProductController`, cette méthode va récupérer tous les produits de la base de données dans une variable `$products`, puis retourner la vue `index` à partir du répertoire `resources/views/products` avec la variable `$products` comme paramètre. La page de vu va ressembler au code ci-dessous :

```
@extends('layouts.main')
@section('main')
    @foreach($products as $product)
        <x-product-card :product="$product">
    @endforeach
@endsection
```

Ce code suppose bien sûr l'existence du layout principal de votre application et le composant `ProductCard` qui prend une instance du modèle `Product` comme paramètre. Nous pouvons par la suite accéder aux colonnes de la tables `products` comme si elles étaient des attributs dans l'objet `$product`. Le code du composant `product-card` pourrait être similaire au code ci-dessous (en utilisant la classe `card` de bootstrap) :

```
@props['product']
<div class="card">
    <img src="{{$product->image}}" class="card-img-top"/>
    <div class="card-body>
        <h1 class="card-title">{{$product->name}}</h1>
        <p class="card-text">{{$product->details}}</p>
        <a href="{{route('products.show', ['id' => $product->id])}}"
```

```
                    class="card-link">
                {{$product->price}}
            </a>
        </div>
    </div>
```

Dans le cas ou vous voulez récupérer un seul objet à partir de son identifiant, vous devez utiliser la méthode `find()` en passant l'identifiant en question comme paramètre. Dans l'exemple suivant, nous montrons une implémentation possible de la méthode `show` dans le fichier `ProductController`.

```
<?php
...
use App\Models\Product;
...
class ProductController extends Controller {
    ...
    public function show($id) {
        $product = Product::find($id);
        return view('products.show')->with('product', $product);
    }
    ...
}
```

Cette méthode prend l'identifiant du produit comme paramètre à travers la route, trouve le produit en question sur la table correspondante, puis retourne la vue `show.blade.php` en passant le produit récupéré comme paramètre.

5.2.1 findOrFail()

La méthode `find` récupère un enregistrement à partir de la base de données en fonction de son identifiant, si aucun enregistrement n'est trouvé, elle renvoie `null`. La méthode

`findOrFail`, quant à elle, fonctionne de la même manière que la méthode. Cependant, si aucun enregistrement n'est trouvé, elle lève une exception `ModelNotFoundException` qui redirige l'interface web vers une page montrant l'erreur 404 si elle n'est pas captée. La principale différence entre les deux méthodes est la façon dont elles traitent les cas où aucun enregistrement n'est trouvé. La méthode `find` renvoie `null` si aucun enregistrement n'est trouvé, tandis que la méthode `findOrFail` lève une exception si aucun enregistrement n'est trouvé.

5.3 Sauvegarder et modifier les données

Pour enregistrer de nouvelles informations ou modifier des informations existantes à partir du modèle, nous utilisons la méthode `save()`. Cette dernière n'est pas une méthode statique, elle doit donc s'exécuter sur une instance du modèle en question. Les noms de colonnes de la table doivent être utilisés comme des attributs sur l'instance du modèle en question.

```
<?php
...
use App\Models\Product;
use Illuminate\Http\Request;
...
class ProductController extends Controller {
    ...
    public function store(Request $request) {
        $product = new Product();
        $product->name = $request->input('name');
        $product->description = $request->input('description');
        $product->price = $request->input('price');
        $product->save();
        return Redirect::route('products.index')
```

```
            ->with('message', 'produit ajouté avec succès');
    }
    ...
}
```

Le code précédent permet de stocker les informations d'un produit dont les informations ont été envoyées via un formulaire. Nous avons d'abord créé une instance de la classe `Product`, puis nous avons donné à cette instance des attributs qui portent les même noms que les colonnes de la table correspondante. Finalement la méthode `save()` va écrire ces informations dans la table (sans que vous ayez à écrire une requête d'insertion).

Si vous voulez insérer un nouvel enregistrement sans créer une instance du modèle correspondant, il est possible d'utiliser la méthode statique `create`. Cette dernière prend comme paramètre un tableau dont les indices sont les noms de colonnes de la table. L'exemple ci-dessous montre comment l'utiliser et est pratiquement équivalent à l'utilisation de la méthode `save()`.

```
<?php
...
use App\Models\Product;
use Illuminate\Http\Request;
...
class ProductController extends Controller {
    ...
    public function store(Request $request) {
        Product::create([
            'name' => $request->input('name'),
            'description' => $request->input('description'),
            'price' => $request->input('price')
        ]);
        return Redirect::route('products.index')
```

```
            ->with('message', 'produit ajouté avec succès');
    }
    ...
}
```

La méthode `save()` permet également de mettre à jour des enregistrements déjà existants. C'est le cas lorsque cette méthode est exécuté sur une instance qui a été lu à partir de la base de données. Le code suivant permet de mettre à jour un produit existant.

```
<?php
...
use App\Models\Product;
use Illuminate\Http\Request;
...
class ProductController extends Controller {
    ...
    public function update(Request $request, $id$) {
        $product = Product::findOrFail($id);
        $product->name = $request->input('name');
        $product->description = $request->input('description');
        $product->price = $request->input('price');
        $product->save();
        return Redirect::route('products.index')
            ->with('message', 'produit modifié avec succès');
    }
    ...
}
```

Dans ce code, le produit est lu à partir de la base de données à l'aide de la méthode `findOrFail`, modifié puis enregistré avec la méthode `save()`. Aucun nouvel enregistrement n'est créé dans ce cas, mais celui qui existe déjà a été modifié.

5.4 Relations entre les modèles

Laravel permet à travers son ORM de gérer les relations entre les modèles de manière simple et intuitive. Les relations disponibles dans Eloquent incluent :

— **One-to-One :** une relation où un modèle est associé à un seul autre modèle.

— **One-to-Many :** une relation où un modèle est associé à plusieurs autres modèles.

— **Many-to-Many :** une relation où plusieurs modèles sont associés à plusieurs autres modèles.

— **Has-Many-Through :** une relation où un modèle possède une relation `many-to-many` à travers un autre modèle.

— **Relations Polymorphiques :** une relation où un modèle peut être associé à plusieurs autres modèles de différents types.

Pour définir une relation entre deux modèles dans Laravel, nous utilisons des méthodes telles que `belongsTo`, `hasMany`, `belongsToMany` ... etc., selon le type de relation que nous souhaitons établir. Nous pouvons également personnaliser les noms des clés étrangères utilisées pour établir les relations et spécifier des conditions de jointure.

5.4.1 Les relations "one-to-one"

Pour définir une relation "one-to-one" entre deux modèles dans Laravel, nous pouvons utiliser les méthodes `hasOne` et `belongsTo`. Supposons que nous avons deux modèles `User` et `Profile` où chaque utilisateur a un seul profil et chaque profil appartient à un seul utilisateur.

Dans le modèle User, nous définirons la relation en utilisant la méthode `hasOne` de la manière suivante :

```
...
class User extends Model {
    ...
```

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

Ici, nous souhaitons établissons une relation avec le modèle `Profile`. Laravel va automatiquement chercher une colonne `user_id` dans la table `profiles` pour récupérer le profil associé à l'utilisateur.

Dans le modèle `Profile`, nous définirons la relation inverse en utilisant la méthode `belongsTo` de la manière suivante :

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

Dans ce cas, Laravel va utiliser la colonne `user_id` dans la table des `profiles` pour déterminer l'utilisateur associé au profil.

Maintenant, nous pouvons facilement accéder au profil associé à un utilisateur et à l'utilisateur associé à un profil. Par exemple, pour récupérer le profil associé à l'utilisateur ayant l'identifiant 1, nous pouvons utiliser :

```
$user = User::find(1);
$profile = $user->profile;
```

De même, pour récupérer l'utilisateur associé à un profil ayant l'identifiant 1, nous pouvons utiliser :

```
$profile = Profile::find(1);
$user = $profile->user;
```

5.4.2 Les relations "one-to-many"

Les relations "one-to-many" (ou les relations père-fils) sont très courantes dans les applications Laravel, où un modèle principal peut avoir plusieurs instances associées dans un autre modèle. Pour définir une relation "one-to-many" entre deux modèles, nous pouvons utiliser la `hasMany` (d'une façon similaire à l'utilisation de la méthode `hasOne`) dans le modèle principal et la méthode `belongsTo` dans le modèle associé.

Supposons que nous avons deux modèles `Product` et `Category`, où chaque catégorie peut avoir plusieurs produits, mais chaque produit appartient à une catégorie seulement. Dans le modèle `Catégorie`, nous définirons la relation en utilisant la méthode `hasMany` de la manière suivante :

```
...
class Category extends Model {
    ...
    public function products() {
        return $this->hasMany(Product::class);
    }
    ...
}
```

Dans cet exemple, Laravel va utiliser la colonne `category_id` dans la table `products` pour récupérer tous les produits associés à une catégorie. Dans le modèle `Product`, nous définirons la relation inverse en utilisant la méthode `belongsTo` de la manière suivante :

```
...
class Product extends Model {
    ...
    public function category() {
```

```
        return $this->belongsTo(Category::class);
    }
    ...
}
```

Ici, Laravel va utiliser la colonne `category_id` dans la table `products` pour reconaitre les produits qui appartiennent à une catégorie donnée.

Dans l'exemple ci-dessous, nous montrons comment accéder à tous les produits appartenant à une catégorie ayant l'identifiant 1 :

```
$category = Category::find(1);
$products = $category->products;
```

De même, pour récupérer la catégorie à la quelle appartient un produit ayant l'identifiant 1, nous pouvons utiliser :

```
$product = Product::find(1);
$category = $product->category;
```

Nom de la clé étrangère

Comme nous l'avons expliqué dans la section 4.1.3, le nom de la clé étrangère est reconnu automatiquement s'il suit certaines conventions. Si la colonne de la clé étrangère ne porte pas le nom attendu, Laravel fournit un moyen de spécifier le nom de la colonne de la clé étrangère dans les arguments de la méthode de relation. Nous pouvons passer un deuxième argument aux méthodes `hasOne`, `hasMany` ou `belongsTo` pour spécifier le nom de la colonne de la clé étrangère.

Dans l'exemple précédent, si la clé étrangère est enregistrée sur la colonne `code_categorie` au lieu de `categorie_id` le code suivant doit être écrit :

```
...
class Category extends Model {
    ...
```

```
    public function products() {
        return $this->hasMany(Product::class, 'category_id');
    }
    ...
}
```

5.4.3 Les relations "many-to-many" :

Pour définir une relation "many-to-many" entre deux modèles, nous devons utiliser une table pivot qui relie les deux modèles en utilisant leurs clés primaires. La table pivot contient des colonnes pour les clés étrangères des deux modèles ainsi que toutes les autres données spécifiques à la relation.

Supposons que nous avons deux modèles User et `Order` (commande) et `Product`. Ou chaque commande peut contenir plusieurs produits, et chaque produit peut figurer sur plusieurs commandes. Dans le modèle `Order`, nous utilisons la méthode `hasMany` de la manière suivante :

```
...
class Order extends Model {
    ...
    public function products() {
        return $this->hasMany(Product::class);
    }
    ...
}
```

Dans l'autre sens, nous allons utiliser la méthode `belongsToMany` sur le modèle `Product` pour compléter la définition de la relation.

```
...
class Product extends Model {
    ...
```

```
    public function orders() {
        return $this->belongsToMany(Order::class);
    }
    ...
}
```

Dans ce cas, laravel va chercher la table pivot nommée `order_product` pour récupérer tous les produits dans une commande, ou toutes les commandes sur lesquelles un produit apparaît. Pour récupérer tous les produits sur la commande ayant l'identifiant 1, nous écrivons :

```
$oder = Order::find(1);
$products = $order->products;
```

Pour récupérer toutes les commandes sur lesquelles apparaît le produit ayant l'identifiant 1, les instructions suivantes sont utilisées :

```
$product = Product::find(1);
$orders = $product->orders;
```

Nom de la table pivot

Le nom de la table pivot utilisé pour définir une relation "many-to-many" est déterminé automatiquement par Laravel en combinant les noms des modèles associés de manière alphabétique et en les séparant par un soulignement (_). Dans l'exemple précédent, la table pivot est nommée `order_product` car alphabétiquement le nom `order` est inférieur au nom `product`.

Toutefois, il est possible de spécifier un nom de table personnalisé en passant le nom de la table comme deuxième argument de la méthode `belongsToMany` dans chaque modèle.

Données supplémentaires

Lorsque nous utilisons une relation "many-to-many", il est souvent nécessaire d'ajouter des données supplémentaires (attributs de relation) à la table pivot. Par exemple, nous pourrions vouloir stocker une le quantité de chaque produit sur chaque commande. Pour ajouter des colonnes supplémentaires à la table pivot, nous pouvons utiliser la méthode `withPivot` lors de la définition de la relation dans chaque modèle. La méthode `withPivot` prend les noms de colonnes qui doivent être ajoutées à la table pivot.

Voici un exemple où nous ajoutons une colonnes `quantity` à la table pivot entre les modèles `Product` et `Commande` :

```
...
class Product extends Model {
    ...
    public function orders() {
        return $this->belongsToMany(Order::class)
                    ->withPivot('quantity');
    }
    ...
}
```

Par la suite, pour récupérer les valeurs de ces colonnes supplémentaires, nous pouvons utiliser la propriété pivot du modèle. Par exemple :

```
$order = Order::find(1);
$product = $order->products()->first();
echo $product->pivot->quantity;
```

5.4.4 Autres relations

Nous avons vu les relations les plus utilisées, mais Laravel supporte certains autres types de relations que nous n'allons pas présenter en détails dans ce cours, mais qui peuvent être très utiles pour des projets un peu plus avancés.

Relations polymorphiques

Les relations polymorphiques [1] permettent à un modèle d'appartenir à plus d'un autre modèle sur une seule association. Par exemple, un commentaire (ou review) peut appartenir à la fois à un produit, ou à un magasin.

Les relations polymorphiques se divisent aux même trois catégories vues précédement :

— One to one.

— One to many.

— Many to many.

Cependant, dans les trois types, le modèle enfant peut appartenir à plus d'un type de modèle en utilisant une seule association. Par exemple, un produit de blog et un commentaite peuvent partager une relation polymorphique avec un magazin.

Les relations "à travers" un autre modèle

Une relation à travers un autre modèle est une relation entre deux modèles qui utilise un troisième comme pivot. Cette relation fournit un moyen pratique d'accéder aux relations éloignées via une relation intermédiaire. Par exemple, nous pourrions avoir une relation `buy` entre un modèle `Client`, et un modèle `Product`, cette relation peut passer à travers le modèle `Order` même s'il n'y a aucune relation directe entre le client et le produit. Laravel supporte deux types de relations "à travers" un modèle :

— La relation `hasOneThrough` [2].

— La relation `hasManyThrough` [3].

1. https://laravel.com/docs/10.x/eloquent-relationships#polymorphic-relationships
2. https://laravel.com/docs/10.x/eloquent-relationships#has-one-through
3. https://laravel.com/docs/10.x/eloquent-relationships#has-many-through