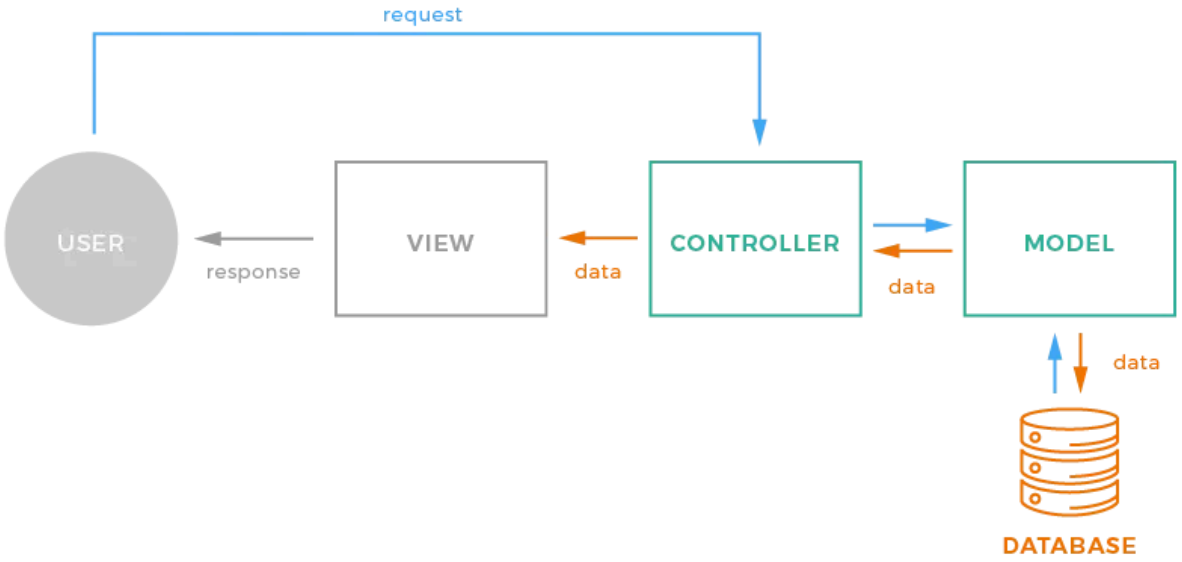
S22 : Intéragir avec la base de données

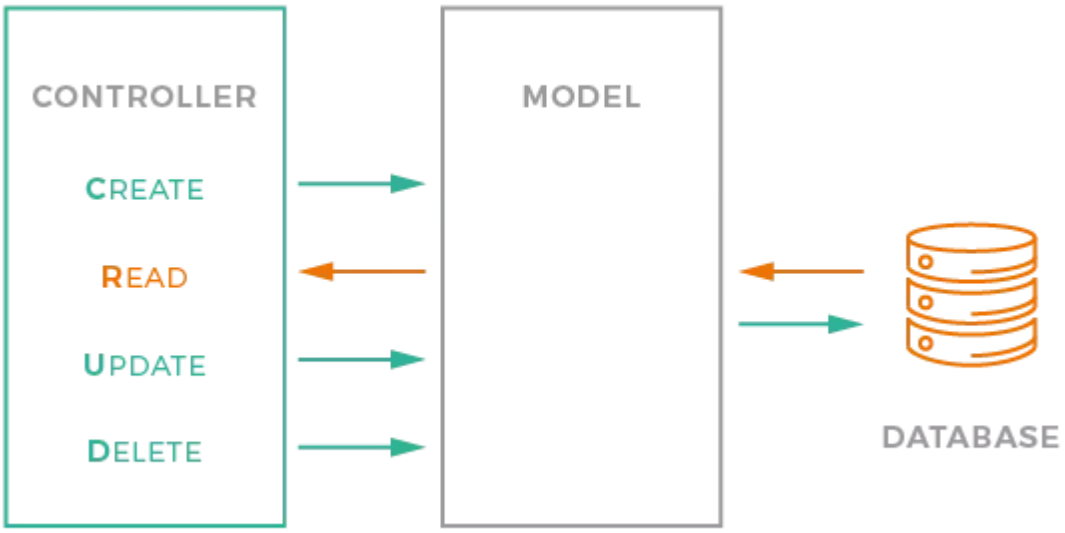
Eloquent

Dans ce chapitre nous allons donc voir comment interagir avec notre base de données. Si vous vous rappelez bien les chapitres précédents nous avons vu que Laravel utilise une architecture en MVC (Model View Controller). Dans cette architecture c'est le Model qui fait office d'intermédiaire entre le controller et la base de données.



1. Le CRUD avec les Resources

CRUD est un acronyme signifiant **Create, Read, Update and Delete.** Il s'agit des 4 actions de base concernant la persistante de données. D'après la logique de notre architecture Laravel vous aurez compris que c'est dans le controller que ces actions se passe.



Pour nous éviter de devoir réécrire ces fonctions de base à chaque fois que nous en avons besoin Laravel nous permet avec une simple commande de créer un controller dit “resource” avec les méthodes `create()`, `store()`, `show()`, `edit()`, `update()`, `destroy()`. Pour cela vous n’avez qu’à rajouter l’option `--resource` lors de la création d’un nouveau controller :

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

Pour continuer avec notre exemple d’articles (posts) nous allons créer le controller PostController en tant que resource. Mais nous aurons également besoin d’un model Post. Pour cela il existe une commande vous permettant de créer à la fois un controller resource et son model associé :

```
php artisan make:model Post -cr
```

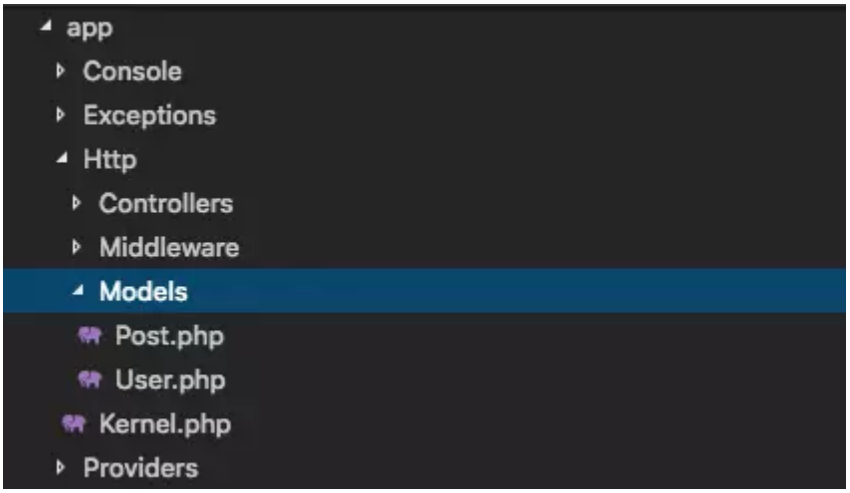
Il vous est même possible de créer d’un coup le model avec la migration et le controller adapté ! (il chôme pas l’artisan !) :

```
php artisan make:model Post -mcr
```

Ici nous avons déjà créé notre fichier de migration donc nous rajouterons seulement l’option `-cr` à la commande de création du model Post. On se retrouve donc avec 2 nouveaux fichiers :

- un controller **PostController.php** situé dans app/Http/Controllers
- un model **Post.php** situé dans app/Http

Personnellement je préfère ranger mes models dans un dossier **Models**. Je trouve cela plus propre.



Afin d'être toujours cohérent entre mon organisation de dossiers et mes namespaces je modifie également le namespace de mon fichier Model Post.php et je précise également dans mon controller qu'il doit cherche le model dans un autre namespace :

```
<?php

namespace App\Http\Models;

use Illuminate\Database\Eloquent\Model;


class Post extends Model
{
    //
}
```

```
<?php

namespace App\Http\Controllers;

use App\Http\Models\Post;
use Illuminate\Http\Request;


class PostController extends Controller
{
    //
}
```

Voilà c'est un peu plus propre ! En parlant de namespace nous allons, comme très souvent, regarder ceux qui sont utilisés dans les fichiers que nous venons de créer afin de soulever cette couche invisible et comprendre plus en détail le framework.

On observe, dans notre fichier model Post.php, que l'on utilise un namespace qui fait appel à la `Class Eloquent` :

```
use Illuminate\Database\Eloquent\Model;
```

Vous pouvez aller voir ce qu'elle fait pour vous donner une idée de ce qu'elle fait. Nous n'allons pas l'analyser en détail, il me faudrait un cours entier pour cela. Mais nous allons voir de quoi il s'agit.

2. Eloquent, l'ORM de Laravel

Commençons tout d'abord par expliquer ce qu'est un ORM pour ceux à qui ce terme ne parlerait pas. Un ORM (Object-Relational Mapping) est un logiciel permettant la conversion des données relationnelles d'une base de données en objets afin de pouvoir les manipuler dans notre application en POO (Programmation Orientée Objet).

Vous l'aurez donc compris, Eloquent est le nom de l'ORM utilisé par Laravel. Celui-ci utilise une implémentation ActiveRecord qui est un design pattern permettant de lire les données d'une base de données. Il fonctionne en encapsulant les attributs d'une table ou d'une vue dans une Class. Les lignes de la table deviennent donc des « tuple » côté PHP.

Vous remarquez que nous ne précisons pas de quelle table il s'agit dans notre model. C'est parce que Laravel utilise des conventions de nommage bien précises pour la conversion de ces tables en objets. Par défaut les tables de la base de données sont donc considérées comme étant au pluriel et en « snake_case ». De même pour la **PRIMARY KEY**, Laravel détermine le nom de cette colonne comme étant `'id'`.

Dans notre exemple le model Post est automatiquement relié à la table "posts" et sa colonne **PRIMARY KEY** est `'id'`.

Si vous êtes un marginal qui n'aime pas les conventions...

...vous avez tout de même la possibilité de nommer vos tables et votre colonne de **PRIMARY KEY** différemment :

```
<?php

namespace App\Http\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
```

```
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_posts';
    protected $primaryKey = 'my_primary_key'
}
```

Si vous vous retrouvez dans un cas particulier où votre **PRIMARY KEY** n'est pas en auto-incrémentation ou n'est pas numérique, vous devez donc passer la valeur de la variable `public $incrementing` à `false`. Si ce n'est pas un integer, vous devez passer la valeur de la variable `protected $keyType` à `'string'`.

```
public $incrementing = false;
protected $keyType = 'string';
```

Concernant les timestamps

Laravel détermine que les colonnes "created_at" et "updated_at" existe dans votre table, si vous ne voulez pas de ces colonnes vous devez le préciser dans le model comme ceci :

```
public $timestamps = false;
```

Pour personnaliser le format des dates de vos timestamps, utilisez la variable `protected $dateFormat` :

```
protected $dateFormat = 'U';
```

Et enfin pour utiliser des noms différents pour ces 2 colonnes, définissez les dans leur constant respective :

```
class Post extends Model
{
    const CREATED_AT = 'date_de_creation';
    const UPDATED_AT = 'derniere_modification';
}
```

> Voici quelques personnalisations possible. Je vous conseillerais cependant d'utiliser au maximum les conventions utilisées car vous pouvez être amené à travailler avec d'autres personnes alors autant tous travailler de la même manière non ? 😉

Pour définir l'accessibilité des colonnes de votre table vous devez les définir dans les tableaux `protected $fillable` et `protected $guarded` afin de sécuriser les

colonnes important du genre `'is_admin'` dans une table “users”. `$fillable` étant une liste blanche et `$guarded` une liste noire.

```
<?php

namespace App\Http\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $guarded = ['is_admin'];
}
```

Vous ne devez en définir qu’un des deux, car en définissant les `$guarded` tous les autres sont `$fillable` et inversement.

Personnellement je trouve plus pratique de définir `$guarded`, car :

- je garde bien en tête ce que je veux protéger
- la liste est souvent plus courte
- si je veux tout définir comme accessible je n’ai qu’à écrire :

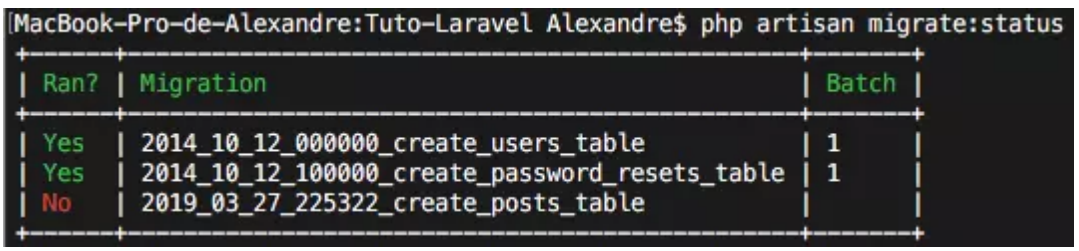
```
protected $guarded = [];
```

Bien, avant de voir comment sélectionner nos données encore faut-il en avoir ! 🙂

Dans le chapitre précédent nous avions créé une table “posts” dans notre base de données. J’espère que vous n’avez pas oublié d’effectuer votre migration… Ok ! Petit rappel :

```
php artisan migrate:status
```

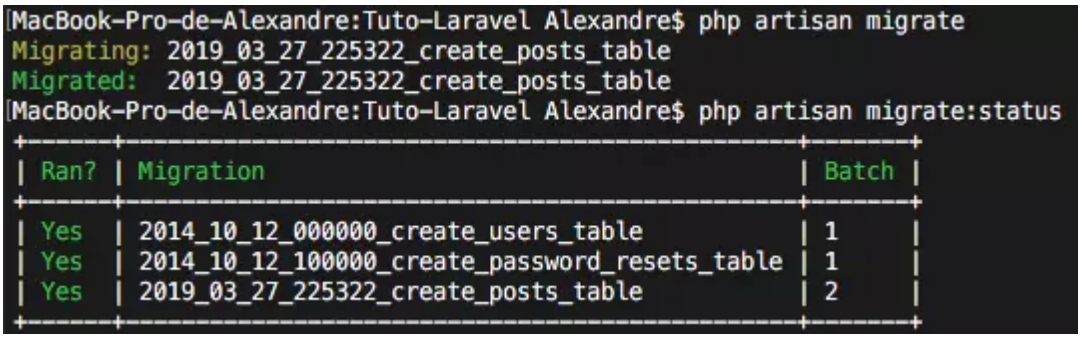
Avec cette commande vous pouvez checker le status de vos migrations. Ici la migration **create_posts_table** n’a pas été envoyée dans la base de données.



Donc ! Un peu coup de :

```
php artisan migrate
```

Et nous voici avec toutes nos migrations “bashed” :



Voilà, notre table posts a bien été créée, je vous laisse vérifier vous-même dans votre base de données, il ne s’agit que de rappels déjà vu.

Maintenant nous avons besoin d’ajouter dans notre base de données un utilisateur et des articles qui lui sont associés car nous avons définie une clé étrangère ‘user_id’ dans la migration de notre table “posts” souvenez vous.

Pour l’exemple nous allons les créer directement depuis le controller WelcomeController comme ceci :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Str;
use App\Http\Models\Post;
use App\Http\Models\User;


class WelcomeController extends Controller
{
    public function index() {
        User::create([
            'name'     => 'user-1',
            'email'    => 'test@mail.com',
            'password' => bcrypt('user')
        ]);
        for ($i=0; $i < 10; $i++) {
            Post::create([
                'title'       => Str::random(10),
                'description' => Str::random(50),
                'content'     => Str::random(1000),
                'user_id'     => '1'
```

```
        ]);
    }
    return view('welcome');
  }
}
```

Rendez-vous sur la page d'accueil et rafraîchissez une fois la page pour exécuter le code.

> Nous verrons dans le chapitre suivant comment générer de manière plus simple et plus propre des données pour remplir notre base de données et effectuer nos tests. Ici j'utilise une alternative un peu brouillon car je ne veux pas me disperser dans mes explications.

3. Builder une Collection !

Nous voici avec nos données dans notre table de "posts" côté mysql. Maintenant, côté PHP, pour récupérer les données qui nous sont nécessaires nous devons les rassembler dans une collection.

Les Collections de Laravel sont des supers tableaux regroupant les différentes données souhaitées et d'autres informations relatives au Model en question.

Pour voir plus clairement comment une collection est faite ajoutez un petit `dd(Post::al())` dans notre controller WelcomeController qui nous retournera l'ensemble des posts de notre base de données. Et n'oubliez pas de supprimer le code écrit précédemment qui nous a permis d'entrer nos nouvelles données dans la base :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Models\Post;


class WelcomeController extends Controller
{
  public function index() {
    dd(Post::all());
    return view('welcome');
  }
}
```

> dd() est une fonction Laravel (qui signifie Dump and Die) qui vous permet d'afficher une variable et d'arrêter le programme juste après.
>
> Si vous ne souhaitez pas arrêter le programme et laissez le code continuer ce qu'il a à faire après vous pouvez utiliser la fonction `dump()`.

Le `dd()` de `Post::all()` vous permet de voir que cela vous retourne bien un objet de la Class Collection et qu'il contient un tableau avec les différents objets de la Class Post.

L’une des utilités de la Collection est de permettre de ne faire qu’une requête sql, puis de rassembler les infos dans une collection. Ensuite lorsque nous voudrons manipuler telle ou telle donnée plus précisément nous n’aurons qu’à effectuer les actions sur notre collection sans avoir à faire une nouvelle requête sql.

Ok mais comment on la récupère cette Collection ?

Et bien c'est le fameux "query builder" qui, comme son nom vous laisse le deviner, permet de "builder" une requête de la base de données. Pour les curieux débrouillards vous retrouverez facilement le chemin vers cette Class en remontant les namespaces utilisés dans les class de votre model. Un petit coup de pouce pour les autres 😉 :

```
use Illuminate\Database\Eloquent\Model; // dans votre class Post
```

```
use Illuminate\Database\Query\Builder as QueryBuilder; // dans cette class Model
```

> Le "as" est un alias créé avec l'opérateur "use". Il a diverses utilités que je vous invite à découvrir ici . Ici l'utilité de faire un alias d'un nom absolu et de permettre de pouvoir instancier à la fois notre class avec son nom absolu ainsi que d'instancier la class avec l'alias (cf exemple 3 et 4 php.net)

Le query builder de Laravel utilise les *"Bindings Parameters"* de PDO afin de protéger votre application des injections SQL. Ce builder vous permet donc d'envoyer les requêtes en SQL à votre base de données.

Dans cette class située dans le namespace Illuminate\Database\Query\Builder vous pouvez remarquer qu'un autre builder est appelé :

```
use Illuminate\Database\Eloquent\Builder as EloquentBuilder;
```

Eh oui ! En réalité Laravel utilise 2 query builders. Voyez ce deuxième builder (EloquentBuilder) comme la couche supérieure de cette gros mécanisme qu'est le query building.

Pour faire simple :

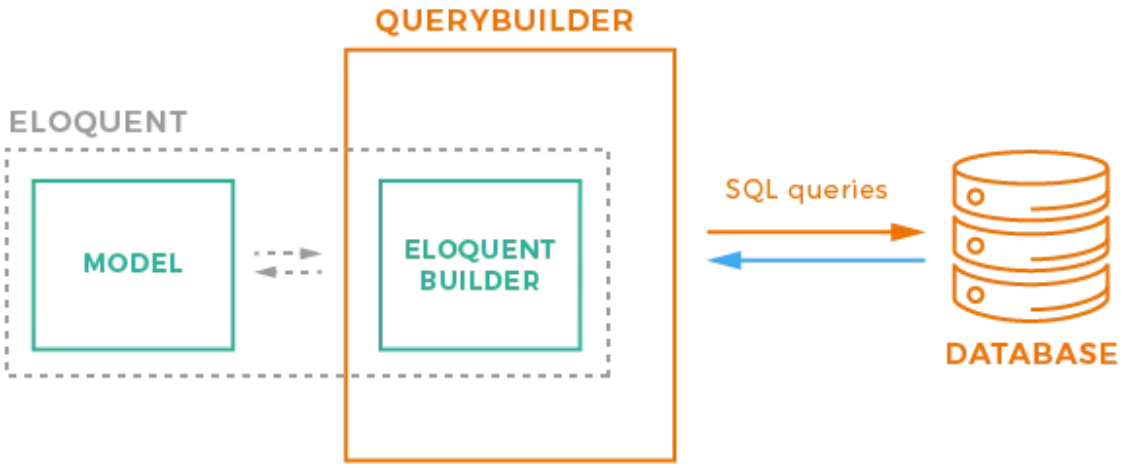
- le « **QueryBuilder** » sert à créer des requêtes SQL. Il fait le lien direct entre notre framework PHP Laravel et notre base de données SQL.
- le « **EloquentBuilder** » est une sorte de Librairie, de Helper SQL. Il étend donc la fonctionnalité du QueryBuilder et est là pour vous simplifier la vie sur pas mal de requêtes que vous aurez à faire.

Ok ! Donc notre `Post::all()` vient de QueryBuilder ou EloquentBuilder ?

Hmmm... Aucun des deux ! Haha !...

Oui en fait il y a une troisième class qui va aussi vous permettre de récupérer vos données. Cette Class vous venez de la voir un peu plus haut, juste avant les 2 builders. Il s'agit de la class Model de Eloquent !

Voici un petit schéma pour vous illustrer un peu comment cela marche :



Nous avons vu que le Model êtes celui qui interagit avec la base de données. Dites-vous simplement qu'Eloquent nous permet de faciliter l'écriture de nos requêtes SQL.

Petite précision concernant la class QueryBuilder. Celle-ci vous permettant d'écrire des expressions brut (en "raw"), elle vous laisse vulnérable à d'éventuelles failles dans votre sécurité d'injection SQL donc prenez garde si vous êtes amené à vous en servir.

Tout cela peut vous sembler un peu barbare voir pas vraiment utile à connaitre mais j'insiste à nouveau sur le faite que rien n'est magique dans Laravel et que toute méthode utilisée en "surface"a une origine très facilement compréhensible une fois que l'on si penche dessus.

Maintenant que nous avons vu sommairement comment fonctionne l'interaction avec la Base de Données nous allons voir plus en détaillant les méthodes nous permettant de "builder" nos collections.

Mais avant cela il serait plus intéressant d'avoir un peu plus d'enregistrement dans notre base et nous n'allons certainement pas le faire à la main ! 😉

Chapitre précédent