Développer en back-end

Gestion et interaction avec une base de données

Formatrice : Elidrissi Asmae

OFPPT
La Voie de l'Avenir

Plan du cour

Introduction

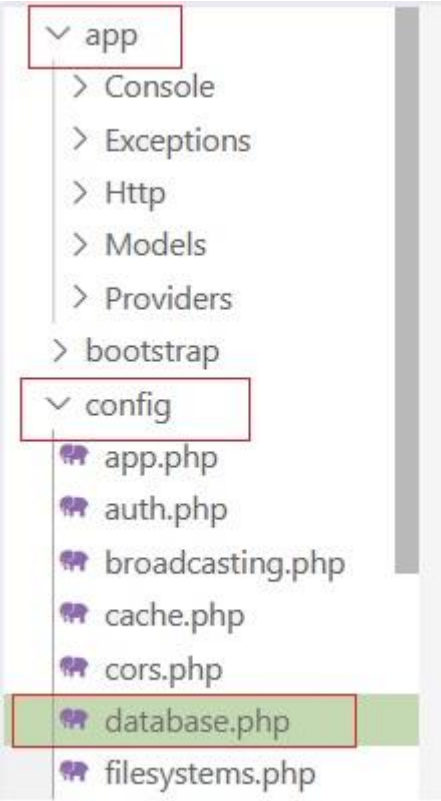
- Laravel rend l'interaction avec les bases de données extrêmement simple sur une variété de bases de données prises en charge à l'aide de SQL brut, d'un générateur de requêtes fluide (Query Builder) et de l' ORM Eloquent .
- Actuellement, Laravel fournit un support de première partie pour cinq bases de données :
  1. MariaDB 10.3+
  2. MySQL 5.7+
  3. PostgreSQL 10.0+
  4. SQLite 3.8.8+
  5. SQL Server 2017+

- Dans ce cours, nous utiliserons MySQL, qui est l'une des plateformes les plus populaires et gratuites pour développement.

Configuration

Pour configurer la base de données, allons dans **config/**, ouvrez le fichier **database.php** et jetez un œil :

```
.....
    'mysql' => [
                'driver' => 'mysql',
                'url' => env('DATABASE_URL'),
                'host' => env('DB_HOST', '127.0.0.1'),
                'port' => env('DB_PORT', '3306'),
                'database' => env('DB_DATABASE', 'forge'),
                'username' => env('DB_USERNAME', 'forge'),
                'password' => env('DB_PASSWORD', ''),
                'unix_socket' => env('DB_SOCKET', ''),
                'charset' => 'utf8mb4',
                'collation' => 'utf8mb4_unicode_ci',
                'prefix' => '',
                'prefix_indexes' => true,
                'strict' => true,
                'engine' => null,
                'options' => extension_loaded('pdo_mysql') ? array_filter([
                    PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
                ]) : [],
            ],
    …..
```

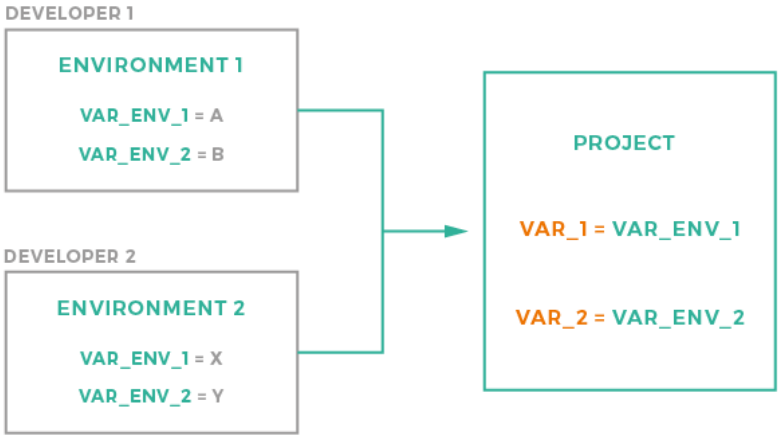


En ouvrant le fichier **database.php** vous remarquez qu'il s'agit en réalité d'un énorme tableau. Via ce tableau Laravel vous permet de vous connecter à divers types de bases de données (sqlite, mysql, pgsql, sglsrv).
Vous voyez également que certaines valeurs font appel à une fonction global **env**() et le premier paramètre passé dans cette fonction correspondant à une constante définit dans le fichier .**env** qui correspond à votre environnement.

Configuration

C’est quoi un environnement ? Pourquoi ne pas directement mettre mes valeurs dans le fichier database.php ?

- Un environnement est un environnement de travail, où vous avez vos propres mécanismes. Cela est très utile lorsque vous travaillez à plusieurs sur un même projet. En effet, pour reprendre l’exemple de la connexion à la base de données vous pouvez très bien avoir vos propres identifiants, appeler votre base de données différemment etc… que votre collègue.



Donc, pour vous connecter à notre base de données, il suffit de configurer le fichier .**env** (se trouvant dans la racine de votre projet)
-> Indiquez ici les valeurs de connexion à votre base de données:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=ma_base
DB_USERNAME=root
DB_PASSWORD=
```

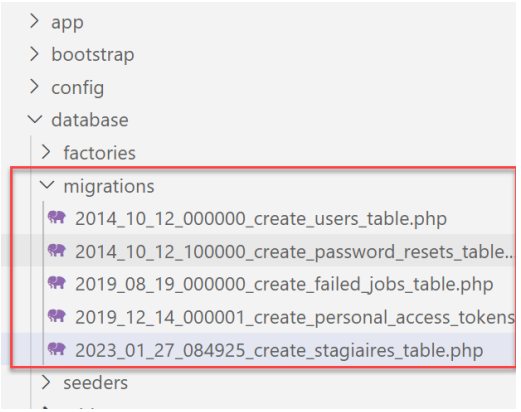
Les migrations : Définition

- Une migration permet de créer et de mettre à jour un schéma de base de données. Autrement dit, vous pouvez créer des tables, des colonnes dans ces tables, ou supprimer, créer des index… Tout ce qui concerne la maintenance de vos tables peut être pris en charge par cet outil. Vous avez ainsi un suivi de vos modifications.
- Vous pouvez utiliser la commande Artisan **make:migration** pour générer une migration de base de données.
- La nouvelle migration sera placée dans le répertoire **database/migrations.**
- Chaque nom de fichier de migration contient la date de sa création qui permettra à Laravel de déterminer l'ordre des migrations.
- Pour créer une migration, il suffit d'exécuter cette commande:

```
php artisan make:migration create_nomTable_table
```

- Exemple :

```
php artisan make:migration create_stagiaires_table
```



- Laravel utilisera le nom de la migration pour tenter de deviner le nom de la table et si la migration créera ou non une nouvelle table.
- Si Laravel est capable de déterminer le nom de la table à partir du nom de la migration, Laravel préremplira le fichier de migration généré avec la table spécifiée. Sinon, vous pouvez simplement spécifier manuellement la table dans le fichier de migration.
- Vous pouvez également indiquer explicitement le nom de la table à créer en utilisant l'option *create*:

```
php artisan make:migration nom_fichier --create=nom_table
```

Exemple: `php artisan make:migration create_xxx --create=stagiaires`

Les migrations : Structure de la classe

- Une classe de migration contient deux méthodes : **up** et **down**.
- La méthode **up** est utilisée pour ajouter de nouvelles tables, colonnes ou index à votre base de données,
- La méthode **down** qui permet de faire un "retour en arrière" de toutes les opérations effectuées par la méthode **up**
- Dans ces deux méthodes, vous pouvez utiliser le générateur de schémas Laravel pour créer et modifier des tables de manière expressive.

- la façade **Schema** permet de gérer le schéma de notre base de données. *(Les **façades** fournissent une interface "statique" aux classes disponibles dans le conteneur de services de l'application LARAVEL.)*

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
      Schema::create('stagiaires', function (Blueprint $table)
{
            $table->id();
            $table->timestamps();
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('stagiaires');
    }
};
```

Les migrations : Edition du fichier de migration

- Pour définir une table, nous devons éditer le fichier de migration.
- La façade Schema offre trois principales méthodes statiques :
  - **create()** : pour créer une table
  - **dropIfExists (ou drop())** : pour supprimer une table
  - **table()** : pour modifier une table

La méthode up :

- Pour créer une nouvelle table de base de données, on utilise la méthode **create** de la façade **Schema**.
- La méthode **create** accepte deux arguments :
  1. le premier est **le nom de la table**,
  2. le second est une *closure* qui reçoit un objet **Blueprint** pouvant être utilisé pour définir la table

```
public function up()
{
Schema::create('stagiaires', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
}
```

La méthode **up**(), propose de créer une table 'stagiaires' avec les colonnes suivantes :

- **id** qui sera en auto incrémentation et clé primaire
- ***created_at*** et ***updated_at :*** deux champs crées grâce à la fonction **timestamps**()

Les migrations : Edition du fichier de migration

Types de colonnes :

Pour définir d'autres types de colonnes, un large choix de fonctions est offert par la façade Schema, vous pouvez consulter la liste ici;
Voici quelques un :

Méthode	Description	Exemple
id()	La méthode id est l'équivalent de la méthode bigIncrements. Par défaut cette méthode créera une colonne avec le nom "id" qui s'auto-incrémente et qui est clé primaire de sa table. On peut attribuer un autre nom à la colonne en le passant en paramètre à la méthode.	`$table->id();` `$table->id('stagiaire_id');`
decimal()	Crée une colonne de type DECIMAL avec les precisions fournis: le total des chiffres et le nombre de chiffres décimaux)	`$table->decimal('amount', $precision = 8, $scale = 2);`
enum()	Crée une colonne de type ENUM, en indiquant la liste des valeurs à accepter.	`$table->enum('difficulty', ['easy', 'hard']);`
integer()	Crée une colonne de type INTEGER	`$table->integer('votes');`
string()	Crée une colonne de type VARCHAR en indiquant le nombre de caractère maximal (par défaut 255)	`$table->string('name', 100);`
text()	Crée une colonne de type TEXT	`$table->text('description');`
char()	Crée une colonne de type CHAR avec la longueur indiquée	`$table->char('name', 100);`
boolean()	Crée une colonne de type BOOLEAN	`$table->boolean('confirmed');`

Les migrations : Edition du fichier de migration

Modificateurs de colonnes :

- Il est possible d'ajouter un modificateur de colonne. Parmi les plus utilisé, notons :
  - unique() : on peut définir des colonnes composées uniques ex. : $table->unique([‘col1’, col2’]:
  - nullable()
  - unsigned()
  - default($value) : *la colonne doit être définie nullable*

Clés étrangères et contraintes d'intégrité référentielle

- Notez: Le nom des clés étrangères doit être au format nomtable_id, c'est-à-dire le nom de la table contenant la clé primaire, sans le s final, tout en lettres minuscule, suivi de _id (ex : categorie_id).

Il y a deux techniques pour créer une clé étrangère et sa contrainte d'intégrité référentielle

Méthode 1 :

```
$table->unsignedBigInteger(‘groupe_id’);
$table->foreign('groupe_id')->references('id')->on(‘groupes’);
```

Méthode 2 :

On peut faire l'équivalent en une seule ligne :

```
$table->foreignId(‘groupe_id')->constrained(‘groupes’);
```

Si le nom de la table référencée par la contrainte respecte les règles de nomenclature, il est possible de raccourcir encore plus l'instruction :

```
$table->foreignId(‘groupe_id')->constrained();
```

Les migrations : Edition du fichier de migration

Exemples:

Extrait de a classe : create_filieres_table

```
public function up(){
Schema::create('filieres', function (Blueprint $table)
{
            $table->id();
            $table->string('titre', 100);
            $table->string('description', 300);
            $table->timestamps();
        });
}
```

Exrait de la classe : create_groupes_table

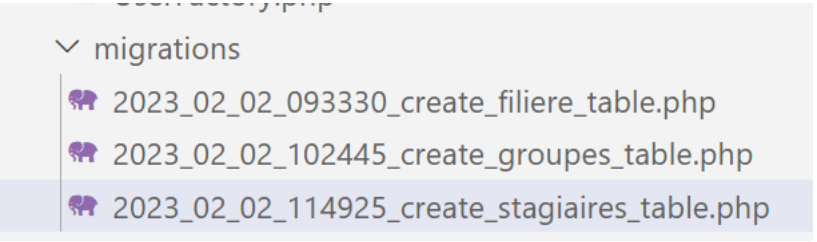
```
public function up()
    {
        Schema::create('groupes', function (Blueprint $table)
     {
            $table->id();
            $table->string("libelle", 25);
            $table->foreignId('filiere_id')->constrained();
            $table->timestamps();
        });
    }
```

Extrait de la classe : create_stagiaires_table

```
public function up()
    {
        Schema::create('stagiaires', function (Blueprint $table) {
            $table->id();
            $table->string("nom_complet", 300);
            $table->enum("genre", ["M", "F"])->default("M");
            $table->date("date_naissance")->nullable();
            $table->decimal("note", 4, 2);
            $table->foreignId('groupe_id')->constrained();
            $table->timestamps();
        });
    }
```

Les migrations : Exécution des fichiers de migration

La console de commande en ligne Artisan permet de lancer les migrations des fichiers crées dans le dossier migration en exécutant la commande suivante : **`php artisan migrate`**



`php artisan migrate`

```
INFO  Running migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 44ms DONE
2023_02_02_093330_create_filiere_table ..... 29ms DONE
2023_02_02_102445_create_groupes_table ..... 75ms DONE
2023_02_02_114925_create_stagiaires_table ..... 52ms DONE
```

Suite à l'exécution de la commande **php artisan migrate**, 5 tables ont été créées dans la base de données;
La table « *migrations* » est une table créée par Laravel pour gérer les migrations et leurs historiques.

	Table	Action	Lignes	Type	Interclassement	Taille
☐	filieres	Parcourir Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8mb4_unicode_ci	32,0 k:
☐	groupes	Parcourir Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8mb4_unicode_ci	32,0 k:
☐	migrations	Parcourir Structure Rechercher Insérer Vider Supprimer	4	InnoDB	utf8mb4_unicode_ci	16,0 k:
☐	personal_access_tokens	Parcourir Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8mb4_unicode_ci	48,0 k:
☐	stagiaires	Parcourir Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8mb4_unicode_ci	32,0 k:
	5 tables	Somme	4	InnoDB	utf8mb4_general_ci	160, k:

Les migrations : La commande migrate

Voici les commandes artisan offertes pour manipuler les migrations:

```
migrate
  migrate:fresh          Drop all tables and re-run all migrations (supprime toutes les tables et relance la migration)
  migrate:install        Create the migration repository (crée une table « migrations » dans la base de données où seront stockés toutes les références des migrations créées)
  migrate:refresh        Reset and re-run all migrations (réinitialise et relance les migrations)
  migrate:reset          Rollback all database migrations (annule toutes les migrations)
  migrate:rollback       Rollback the last database migration (annule la dernière migration)
  migrate:status         Show the status of each migration ( affiche des informations sur les migrations)
```

Les migrations : Modification d'une table à l'aide d'un fichier de migration

Une fois les tables de la base de données créées, il est possible de les modifier. L'utilisation de fichiers de migration est la solution privilégiée.

Si vous avez déjà effectué une migration et que vous devez modifier la structure d'une table, il est préférable de créer un fichier de migration qui modifie la structure d'une table plutôt que de modifier un fichier de migration existant.

Génération du fichier de migration:

Si vous désirez, par exemple, modifier la table stagiaires et d'ajouter une nouvelle colonne « nomComplet », vous utiliserez la commande :

```
php artisan make:migration ajouter_colonne_ville_to_stagiaires
```

- Laravel peut détecter de quelle table s'agit-t-il en se basant sur le suffixe du nom de fichier; ici « to_stagiaires » concernera la table « stagiaires »
- Le nom du fichier de migration peut être n'importe quoi, en autant qu'il décrive bien ce qui est fait dans ce fichier. Pour indiquer à Laravel de quelle table s'agit-t-il , il suffit d'ajouter à la commande `--table nomTable`

```
php artisan make:migration ajouter_ville --table stagiaires
```

Les migrations : Modification d'une table à l'aide d'un fichier de migration

Édition du fichier de migration

Dans le fichier de migration, vous entrerez le code suivant.

L'ajout de la nouvelle colonne sera faite dans la méthode **up().** L'opération inverse sera codée dans la méthode **down().**

Les ajustements à la table seront faits à l'aide de **Schema::table()**.

```
return new class extends Migration
{    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('stagiaires', function (Blueprint $table) {
            //Ajout de la nouvelle colonne
            $table->string("ville", 70);
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('stagiaires', function (Blueprint $table) {
            //Suppression de la nouvelle colonne
            $table->dropColumn(« ville");
        });
    }
};
```

Avant d'exécuter les fichiers de migration, surtout lors de la modification d'une table, il est conseillé de vérifier les lignes de code SQL qui seront générées avec la commande : **php artisan migrate –pretend**

```
2023_02_05_083437_ajouter_colonne_ville_to_stagiaires ..........................
 ↳ alter table `stagiaires` add `ville` varchar(70) not null
```

Pour appliquer les changements, il suffit de lancer la migration avec la commande : **php artisan migrate**

Les fichiers de seeds pour insérer les données initiales dans la BD

Il est possible d'ajouter des données initiales, aussi appelées données de base, ainsi que des données de test dans les tables à l'aide de fichiers de « seeds ».

Données initiales vs données de test

Quelle est la différence entre **les données initiales** et l**es données de test** ?

- Les données initiales seront livrées avec le site Web. Il s'agit de données qui existeront peu importe à qui le site est vendu (ex : données pour remplir une table de villes, de couleurs, etc.).
- Les données de test, quant à elle, ne seront pas livrées avec le site Web. Il s'agit de données fictives dont le but est de permettre de bien tester le site Web (ex : clients, produits, etc.).

Générer le fichier de seeds

Un fichier de seeds peut être créé à l'aide de la commande :

```
php artisan make:seeder FilieresTableSeeder
```

Le fichier ainsi généré sera placé dans le dossier **database\seeds**, sous le dossier du projet, et contiendra le code ci-après :

```php
<?php
namespace Database\Seeders;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class FilieresTableSeeder extends Seeder
{    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //
    }
}
```

Les fichiers de seeds pour insérer les données initiales dans la BD

Éditer le fichier de seeds

Éditez ce fichier pour y ajouter les données désirées à l'aide de **DB::table()**.

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Str;

class FiliereTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //
        DB::table('filieres')->insert([
            [
                'id'=> 2,
                'titre'=> "DEVOWFS",
                'description'=> "Développement digital-
option web full stack"
            ],
            [
                'id'=> 3,
                'titre'=> "DEVOAM",
                'description'=> "Développement digital-
option applications mobiles"
            ],
            [
                'id'=> 4,
                'titre'=> "DEVORVRA",
                'description'=> "Développement digital-
option réalité virtuelle/réalité augmentée"
            ],
            [
                'id'=> 5,
                'titre'=> Str::random(20),
                'description'=> Str::random(50)
              //La méthode Str::random génère une chaîne
aléatoire de la longueur spécifiée
            ]
        ]);
    }
}
```

Les fichiers de seeds pour insérer les données initiales dans la BD

Exécuter tous les fichiers de seeds

Pour qu'un fichier de seeds soit exécuté, on ajoutera sa référence dans le fichier **database\seeds\DatabaseSeeder.php**. Ce fichier contiendra la liste des fichiers de seeds à exécuter en lot.

Attention : lorsqu'une table contient une clé étrangère, ses données initiales doivent être ajoutées après celles de la table contenant la clé primaire de la relation.

Il peut être intéressant de séparer les données initiales, qui devront être livrées avec l'application, des données de test.

```
    public function run()
    {
        // données initiales
        $this->call([
                    PaysTableSeeder::class,
                    ProvincesTableSeeder::class,
                    VillesTableSeeder::class
                   ]);

        // données de test
        $this->call([
                        CategoriesTableSeeder::class,
                        ProduitsTableSeeder::class
                    ]);
    }
```

Les fichiers de seeds pour insérer les données initiales dans la BD

Exécuter tous les fichiers de seeds

Le fichier **DatabaseSeeder.php** sera exécuté par la commande suivante :

```
php artisan db:seed
```

Si une table contenait déjà des données, le fichier de seeds ajoutera les nouvelles données à la suite des données existantes.
Si vous exécutez cette commande plus d'une fois, les données seront dédoublées.

Exemple:

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(FiliereTableSeeder::class);
    }
}
```

Si on exécute la commande : **php artisan db:seed**
On aura le résultat ci-après:

	id	titre	description	created_at	updated_at
Éditer Copier Supprimer	2	DEVOWFS	Développement digital- option web full stack	NULL	NULL
Éditer Copier Supprimer	3	DEVOAM	Développement digital- option applications mobiles	NULL	NULL
Éditer Copier Supprimer	4	DEVORVRA	Développement digital- option réalité virtuelle/ré...	NULL	NULL
Éditer Copier Supprimer	5	aoNCarNHouzrvEsbhtrK	NqHjJ0EcfozTuf5myQ4uBnbOmayleb9lhUsQRthdafiN36Ig8m	NULL	NULL

Manipuler les données avec Query Builder

- Le générateur de requêtes de base de données de Laravel fournit une interface pratique et fluide pour créer et exécuter des requêtes de base de données.
- Il peut être utilisé **pour effectuer la plupart des opérations de base de données** dans votre application et fonctionne parfaitement avec tous les systèmes de base de données pris en charge par Laravel.

Méthode	Description	Exemple
Sélection des résultats		
get()	La méthode get() permet de récupérer un groupe de résultat. Elle retourne une Collection (de la Class Illuminate\Support\Collection) contenant chaque résultats obtenu par la requête.	`$stagiaires1 = DB::table('stagiaires')->get();` `$stagiaires2 = DB::table('stagiaires')-> get(['id', 'nom_complet']);`
where()	La méthode where() permet de poser une condition à la sélection de résultat	`$redoublants = DB::table('stagiaires')-> where( 'note', '<', 10)->get();`
first()	La méthode first() a un fonctionnement similaire à get() mais ne récupère que le premier résultat correspondant à la requête. S'il ne trouve aucun valeur first() retourne la valeur null.	`$stagiaire = DB::table('stagiaires')->first();`
find()	La méthode find() permet de sélectionner un résultat d'après un id que l'on passe en paramètre :	`$stagiaire = DB::table('stagiaires')->find(1);`

Manipuler les données avec Query Builder

Rangement et groupement		
orderBy()	Permet d'organiser les résultats en les rangeant dans un order ascendant (ASC) ou descendant (DESC) d'après une colonne de la table	`$stagiaires = DB::table('stagiaires')->orderBy('note', 'DESC')->get();`
groupBy()	La méthode groupBy permet de grouper les résultats par valeur(s) d'une ou plusieurs colonnes.	`$stagiaires=DB::table("stagiaires") ->selectRaw("count('id') as 'nombre_de_stagiaires', genre") ->groupby("genre")->get();`
inRandomOrder()	La méthode inRandomOrder() vous permet tout simplement de sortir vos résultats dans un ordre aléatoire.	`$stagiaires =DB::table("stagiaires")->inRandomOrder()->get();`

Les jointures		
join()	La méthode join() permet de faire une jointure 'INNER JOIN'.	`$result= DB::table('groupes')-> join('filieres', 'groupes.filiere_id', '=', 'filieres.id')->get(["groupes.id", "groupes.libelle", "filieres.description", "groupes.created_at"] ); return dd($result);`
leftJoin() et rightJoin()	Les méthodes leftJoin() et rightJoin() permettent d'effectuer les commandes SQL de jointure **'LEFT JOIN'** et '**RIGHT JOIN**'.	

Manipuler les données avec Query Builder

Mise à jour des données		
insert()	La méthode insert() permet d'insérer une ligne dans une table.	`DB::table("filieres")->insert([ "titre"=>"AA", "Description" => "Assistant Administratif" ]);`
insertGetId()	La méthode insertGetId() fait le même job que insert() mais en plus elle permet de récupérer l'id créé pour cette enregistrement. Bien évidement insertGetId() ne marche qu'avec des tables ayant un id en auto-incrémentation.	`$id= DB::table("filieres")->insertGetId([ "titre"=>"AA", "Description" => "Assistant Administratif" ]); dd($id); //ça affiche 7`
update()	La méthode update() permet de mettre à jour des données.	`DB::table('filieres')->where('id', 3)->update([ 'titre' => 'DEV-OAM' ]);`

Manipuler les données avec Query Builder

<table>
<tr><th colspan="3">Mise à jour des données</th></tr>
<tr><td>updateOrInsert()</td><td>La méthode updateOrInsert() permet de chercher un enregistrement pour le mettre à jour, mais si celui-ci n’existe pas et bien elle le crée.
La méthode prend en paramètre 2 tableaux, le premier sert de conditions pour trouver l’enregistrement en question, le second permet de préciser les données à mettre à jour.</td><td><pre>
DB::table('stagiaires')->updateOrInsert(
    ["nom_complet"=>"Ahmed Alami",
"groupe_id"=>"1", "ville"=>"fès"],
    ["note"=>15]
);
//Si le stagiaire Ahmed Alami du groupe 1 de
la ville fès n’existe pas dans la table
Stagiaire, on lui insère une nouvelle ligne
dans ladite table avec une note 15.
</pre></td></tr>
<tr><td>delete()</td><td>La méthode delete() permet de supprimer un enregistrement</td><td><pre>
DB::table('groupes')->where('id', 2)-
>delete();
</pre></td></tr>
<tr><th colspan="3">Opérations numérique</th></tr>
<tr><td>max(),
min(),
avg(),
sum()</td><td>Vous pouvez également appliquer des opérations numériques existantes dans les commandes SQL :</td><td><pre>
$max_note = DB::table('stagiaires')->max('note');
$min_note = DB::table('stagiaires')->min('note');
$moyenne = DB::table('stagiaires')->avg('note');
$somme = DB::table('stagiaires')->sum('note');

return dd($max_note,$min_note, $moyenne, $somme);
</pre></td></tr>
</table>

Manipuler les données avec Query Builder

Exemple:

Créons une classe appelée « **FiliereController** », et y ajoutons une méthode index permettant de récupérer toutes les filières enregistrées dans la base de données :

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class FiliereController extends Controller
{
    public function index(){
        $filieres=DB::table("filieres")->get();
        dd($filieres);
    }
}
```

- Ajoutons, par la suite, dans le fichier web.php, une route évoquant la méthode ci-dessus:

```
Route::get("/filieres", [FiliereController::class, "index"]);
```

En démarrant le serveur et en visitant l'url : http://localhost:8000/filières
On peut avoir le résultat suivant :