UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Licence 2 Informatique
INFO0303 – 2019/2020
***Resp. :* F. Alin**

Travaux Pratiques n° 5

Introduction à Laravel

1 Les modèles

L'objectif d'un modèle Laravel, est d'assurer de manière transparente les échanges entre l'application et la base de données. Pour cela Laravel utilise le paradigme ORM (object relational mapper). Ainsi, au lieu de dialoguer directement avec la base de données, l'application passe par cette classe qui fait office d'interface et prend en charge tous les aspects techniques de la gestion de bases de données.

Avant d'utiliser une base de données avec Laravel, il faut s'assurer que l'application est convenablement configurée. Pour cela on dispose de deux fichiers :

— `config\database.php` : Ce fichier permet entre autre de définir le type de base de donnée utilisée (Sql, Postgre, ...). Il n'est en général pas nécessaire de modifier ce fichier ;

— `.env` : Ce fichier se trouve dans la racine du site. C'est ici que nous allons définir les paramètres d'accès à la base de données (nom de la base, host, mot de passe, login, etc ...)

```
 ....
DB_CONNECTION=mysql
DB_HOST=monsite.test
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
....
```

Nous allons maintenant décrire les différentes étapes de création d'un modèle :

1.1 Création d'un modèle avec artisan

Syntaxe :

```
php artisan make:model Boat --migration
```

Cette commande va créer le fichier `app\Boats` que nous pourrons modifier plus tard. L'option –migration génère en même temps un fichier dans le répertoire `database\migration\`. Cette option n'est pas obligatoire et le fichier de migration peut être généré individuellement plus tard.

1.2 Création de la migration

Pour utiliser le modèle précédent il faut disposer dans la base de données d'une table portant le même nom que le modèle. Dans l'exemple précédent nous avons créé simultanément le modèle et

la migration grâce à l'option –migration. Il est possible de générer la migration seule à l'aide de la commande :

Syntaxe :

```
php artisan make:migration boats
```

Ce fichier contient deux méthodes :

```
class CreateBoatsTable extends Migration
{
    public function up()
    {
        Schema::create('boats', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('boats');
    }
}
```

On utilise la méthode `up()` pour définir les colonnes de la table :

```
$table->string('nom',25);
$table->string('modele');
$table->decimal('longueur',4,2);
$table->decimal('largeur',4,2);
$table->date('date_construction');
```

Une fois la structure de la table correctement définie dans le fichier il faut exécuter la migration pour créer effectivement la table. Pour cela on dispose de la commande `php artisan migrate`.

Remarque :

`artisan` propose de nombreuses options de gestion de l'application. Si par exemple vous avez fait une erreur dans la structure de votre table, il est possible d'annuler la dernière migration avec la commande `php artisan migration:rollback`. Pour plus d'informations vous pouvez vous reporter à l'adresse http ://laravel.com/docs/master/migrations.

1.2.1 Peuplement de la base de données

Il est souvent nécessaire, lorsque l'on travaille avec une base de données, de peupler cette base soit avec des données de productions soit avec des données aléatoires afin de tester l'application. Laravel dispose d'un mécanisme permettant d'automatiser cette tâche. Le peuplement de la base se fait à l'aide de la commande `artisan` :

```
php artisan db:seed
```

Cette commande lance l'exécution du fichier `DataSeeder.php` situé dans le répertoire `database/seeds`

```php
<?php

use Illuminate\Database\Seeder;

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

Le fichier `UserTableSeeder` n'existe pas. Il faut le créer. Pour cela on utilise une fois encore `artisan` :

```
php artisan make:seed BoatTableSeeder
```

Cette commande a pour effet la création dans le répertoire `database/seeds` du fichier `BoatTableSeeder.php` :

```php
<?php

use App\Boat;
use Illuminate\Database\Seeder;

class BoatTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('boats')->truncate();
        Boat::create([
            'nom' =>'teBoat',
            'modele' => 'Benetaux Oceanis 35',
            'longueur'=> 9.5,
            'largeur' =>3.5,
            'date_construction' =>'10/10/2010'
        ]);
        Boat::create([...]);
        ......
    }
}
```

La méthode `run()` commence par vider la table des données qui pourraient déjà être présentes en appelant la méthode `truncate()` puis créer les nouveau enregistrements en appelant la méthode `create()` de la classe `Boat`. Il reste encore à modifier le fichier `DatabaseSeeder.php` en insérant la ligne suivante :

```
$this->call(BoatTableSeeder::class);
```

Cette méthode est intéressante mais s'avère très vite limitée lorsqu'il s'agit de peupler la base avec une grande quantité de données en particulier à des fins de tests ou de mise au point. Cette fois encore Laravel dispose d'outils qui vont permettre d'automatiser se travail fastidieux. Le code suivant suivant présente une utilisation de la classe `facker\Factory` :

```
public function run()
    {
        DB::table('boats')->truncate();

        $faker = \Faker\Factory::create();

        foreach (range(1,50) as $index) {
            Boat::create([
                'nom' =>$faker->name,
                'modele' => $faker->sentence (2),
                'longueur'=> $faker->randomFloat(2,0,15),
                'largeur' => $faker->randomFloat(2,0,5),
                'date_construction' => $faker->date('Y-m-d')
            ]);
        }
    }
```

1.3 Création d'un contrôleur ressourceful (RESTful)

Laravel propose l'implémentation d'un contrôleur REST (representa1 state transfert) qui permet de mettre simplement en place les différentes tâches généralement nécessaires à l'exploitation d'une base de données. On parle de modèle CRUD (create, read, update and delete). Ce contrôleur est créé à l'aide d'`artisan` :

Syntaxe :

`php artisan make:controller BoatsController --resource` Cette commande crée fichier `BoatsController` dans le `app\\Http\\Controller`. Ce contrôleur contient 7 méthodes publiques :

```php
<?php
namespace App\Http\Controllers;

use App\Boat;
use Illuminate\Http\Request;

class BoatController extends Controller
{
    public function index()
    {...}

    public function create()
    {...}

    public function store(Request $request)
    {...}

    public function show(Boat $boat)
    {...}

    public function edit(Boat $boat)
    {...}

    public function update(Request $request, Boat $boat)
    {...}

    public function destroy(Boat $boat)
    {...}
}
```

C'est en renseignant ces 7 méthodes que nous pourrons définir les comportements nécessaires à la gestion de la base de données.

Il faut enfin définir les routes nécessaires à l'utilisation de toutes ces ressources. Pour cela nous ajoutons dans le fichier `web.php` situé dans le répertoire `routes` la commande suivante :

Syntaxe :

```
ROUTE::resource('boats',BoatsController')
```

et importer la déclaration du `namespace` de la classe `Boats` au début fichier du contrôleur `use App\Boats`.

1.4 Conclusion

Il est possible avec `artisan` de générer tous les fichiers (migration, modèle et contrôleur) simultanément. Pour cela on utilise la commande :

Syntaxe :

```
php artisan make:model Boat --controller --resource
```

Conclusion :

Dans cette partie nous avons appris à construire tous les fichiers nécessaires à la mise en place d'un modèle. Dans la suite nous allons apprendre à paramétrer le modèle ainsi construit.

2 Adaptation du modèle

Laravel permet d'écrire des getters (accessor) ou des setters (mutators) pour accéder aux différents champs d'une table associée à un modèle. Ces modification se font dans le fichier du modèle (dans notre cas dans le fichier `Boat.php` situé dans le répertoire `app\`).

2.1 Les accessors

Syntaxe :

`public function getPropertieAttribute ()` La règle de nommage des accessors est très simple. Leur nom débute nécessairement par get suivi du nom de la propriété dont on écrit la première lettre en majuscule, et se termine par Attribute

Exemples :

```
public function getNomAttribute($value) {
        return strtoupper($value);
    }
```

Dans cette exemple le getters sur la propriété nom retourne une chaîne de caractères ne contenant que des majuscules.

2.2 Les mutators

La syntaxe reprend les mêmes règles que pour les accessors.

Syntaxe :

```
public function setNomAttribute($parametre)
```

2.3 Le query scoping

Il est parfois utile de restreindre le champ de recherche des requêtes SQL. Imaginons par exemple que l'on construise un site internet de vente en ligne dont les produits vendus dépendent du pays depuis lequel le consulte l'internaute. Il poura être fastidieux de devoir systématiquement rajouter dans toutes les requêtes que l'on ne veut par exemple uniquement les produits disponibles pour le marché français. On peut pour éviter cela ajouter une méthode spécifique dans le fichier de la

classe modèle afin que cette sélection soit faite automatiquement sans qu'il soit ensuite nécessaire de la refaire systématiquement ensuite pour chaque extraction. Laravel va donc travailler sur une base table restreinte. C'est ce que l'on appelle le "query scoping". On définit le champ d'action des requêtes de notre application. On peut définir un scope globlal (applicable à toutes les requêtes) ou des scopes locaux.

2.3.1 Scope Global

Pour créer un scope global il faut surcharger dans la classe du modèle la méthode `boot`, sans oublier d'appeler la méthode parent de la classe héritée. Il faut également ajouter le `namespace use Illuminate\Database\Eloquent\Builder` en début de fichier avec les déclarations.

```
<?php
...
use Illuminate\Database\Eloquent\Builder;
...

    protected static function boot()
    {
        parent::boot();
        static::addGlobalScope('taille', function (Builder $builder) {
            $builder->where('longueur','<',8);
        });
    }
}
```

Dans cet exemple, toutes les requêtes effectuées ne s'appliqueront qu'à une sous table ne contenant que des bateaux dont la longueur ne dépassera pas 8m.

Si malgré la définition d'un scope local vous souhaitez accéder à toute la base de données, vous pouvez le faire en utilisant la commande suivante qui ignorera alors scope global.

Syntaxe :

```
$boat = Boat::withoutGlobalScopes()->get()
```

2.3.2 Scope Local

Le scope local va constituer avant tout un racourci pour une requête que vous pourriez être amené à effectuer souvent. Pour créer une fonction de scope local dans un modèle il suffit de déclarer une nouvelle fonction publique dont le nom est préfixé par le nom `scope`.

```
public function scopeGrandBateaux($query) {
        return $query->where('longueur','>',8);
}
```

Il suffit alors pour utiliser cette fonction de l'appeler de la façon suivante :

Syntaxe :

```
$bateaux = grandBateaux()->get();
```

Remarque :

Il est possible de créer des scopes paramétrés :

```
public function scopeTailleMax ($query, $taille) {
        return $query->where('longueur','<',$taille);
}
```

Ce scope paramétré pourra alors s'utiliser de la manière suivante :

```
$taille = 5;
$bateau_inf_5m = Boat::tailleMax($taille)->get();
```

3 Création, lecture, mise à jour et suppression de données

Toute application qui interagit avec une base de données doit nécessairement mettre en œuvre des mécanismes de création, lecture, mise à jour et suppression de données. Nous allons voir dans cette section comment compléter le contrôleur de ressource `BoatController` que nous avons défini dans les exemples précédents.

3.1 Création d'un nouvel enregistrement

Pour insérer de nouvelles données dans une table nous pouvons créer explicitement une nouvelle instance de la classe modèle (Boat dans notre exemple).

```
$boat = new Boat;
$boat->nom = "Matimax";
$boat->modele = "Capelli 625";
$boat->longueur = 6.25;
$boat->largeur = 2.35;

$boat->save();
```

Avec cette méthode aucune vérification n'est faite, et si le bateau "Matimax" existe déjà il sera quand même recréé. Laravel propose une autre méthode permettant d'éviter cet inconvénient :

Syntaxe :

```
Boat::firstOrCreate()
```

Cette méthode vérifie si un enregistrement existe déjà avec la ou les même valeurs. Si un tel enregistrement existe il est retourné, sinon une nouvelle instance est créée.

```php
$boat = Boat::firstOrCreate(
        [
                'nom' => 'Matimax'
        ]);
        $boat->modele->'Capelli 625';
        $boat->save();
```

Ce dernier exemple peut être condensé en une seule ligne :

Syntaxe :

```php
$boat = Boat::firstOrCreate(['nom'=>'Matimax'],['modele'=>'Capelli 625']);
```

Enfin si l'on veut uniquement créer une nouvelle instance de l'objet sans l'enregistrer immédiatement dans la base de données on dispose de la méthode `Boat::firstOrNew()`

3.2 Mise à jour d'un enregistrement existant

Si l'on veut mettre à jour un enregistrement dont on connaît la clé primaire on peut utiliser la méthode suivante :

```php
        $boat = Boat::find($id);
        $boat->longueur = 2.25;
        $boat->save();
```

Si l'objectif est de mettre à jour l'enregistrement si il existe ou de le créer dans le cas contraire on peut utiliser la méthode `Boat::updateOrCreate()`.

Syntaxe :

```php
$boat = Boat::updateOrCreate(['nom'=>'Matimax'],['longueur'=>6.25]):
```

3.3 Suppression d'un enregistrement

La suppression d'une entrée dans la base de données dont on connaît la clé primaire s'effectue de la manière suivante :

```php
        $boat = Boat::find($id);
        $boat->delete();
```

Afin de consolider la suppression il peut être utile ensuite d'utiliser la méthode `destroy()` :

Syntaxe :

```php
Boat::destroy($id)
```

3.4 Soft deleting

La méthode de suppression décrite précédemment détruit définitivement l'enregistrement sélectionné. Mais sur un site de production il peut être nécessaire pour des raisons d'historisation par exemple de marquer un enregistrement comme supprimé tout en conservant la trace. Cette méthode s'appelle le "soft deleting". Pour mettre en oeuvre ce mécanisme il faut le préciser au moment de la création de la migration ou comme dans l'exemple suivant créer une nouvelle migration pour modifier la table boats déjà existante.

Syntaxe :

```
php artisan make:migration add_soft_delete_to_boats --table=boats
```

On peut ensuite modifier la migration nouvellement créée :

```
public function up() {
        Schema::table('boats',function (Blueprint $table)
        {
                $table->softDeletes();
        });
}

public function down() {
        Schema::table('boats', function(Blueprint $table) {
                $table->dropColumn('deleted_at);
        });
}
```

Une fois la migration effectuée il faut encore modifier le modèle pour pouvoir profiter du mécanisme de "soft deleting".

```
...
use Illuminate\Database\Eloquent\SoftDeletes;
...

Class Boat extends Model
use SoftDeletes

protected $dates = [
        'created_at',
        'deleted_at',
        'started_at',
        'update_at'
];
```

À partir de maintenant tout appel de la fonction `delete()` de ce modèle enregistre simplement une valeur dans le champ `deleted_at`

Tout enregistrement ayant une valeur `deleted_at` non nulle n'apparaîtra plus dans les requêtes. Cependant il reste possible d'accéder aux données effacées :

Syntaxe :

```
$boat = Boat::withTrashed()->get();
```

4 Ecriture du contrôleur RESTful

Maintenant que nous savons interagir avec la base de données grâce au modèle, nous pouvons compléter le contrôleur afin qu'il soit en mesure de réaliser les actions de création, lecture, mise à jour et suppression des données. Tous les exemples qui suivent font référence au contrôleur `BoatController.php` défini dans les sections précédentes.

4.1 Création d'un nouvel enregistrement

L'appel à la méthode `create()` du contrôleur doit simplement générer l'affichage d'un formulaire de saisie du nouvel enregistrement :

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

Le formulaire utilisé s'appelle `create..blade.php` et se trouve dans le `resources\views\boats\`. Ce formulaire quant à lui pointe sur la méthode `store()` du contrôleur.

```
@extends('layouts.app')
@section ('content')
    <div class="row">
        <div class="col">
            {!! Form::open(['route'=>'boats.store'],['class'=>'form']) !!}
            <div class="form-group">
                {!! Form::label('nom','Nom du bateau : ',
                    ['class'=>'control-label'])
                !!}
                {!! Form::text('nom',null,
                    [
                        'class'=>'form-control input-lg',
                        'placeholder'=>'mon bateau'
                    ])
                !!}
            </div>
            <div class="form-group">
                {!! Form::label('modele','Marque et type : ',
                    ['class'=>'control-label'])
                !!}
                {!! Form::text('modele',null,
                    [
                        'class'=>'form-control input-lg',
                        'placeholder'=>'modéle du bateau'
                    ])
```

```
                !!}
            </div>
            ....
            {!! Form::close() !!}
        </div>
    </div>
@endsection
```

Ce formulaire est accessible à l'adresse `monsite.test\boats\create`. Il ne reste plus qu'à mettre en oeuvre l'enregistrement effectif des données dans la base. Pour cela il faut compléter la méthode `store()` du contrôleur :

```
public function store(Request $request)
   {
       $boat = Boat::create(
           $request->input()
       );
       flash('Nouveau bateau enregistré !')->success();
       return redirect()->route('boats.show',['boat'=>$boat]);
   }
```

Ici, après l'enregistrement, le contrôleur nous renvoie vers la vue `show` et affiche un message indiquant que tout s'est bien passé.

Remarque :

Cette méthode d'enregistrement par lot de l'ensemble des champs ne fonctionne que si l'on a défini dans la classe modèle `Boat` la variable `$fillable` :

```
        protected $fillable = [
                'nom',
                'modele',
                'longueur',
                'largeur',
                'date_construction'
    ];}
```

Si l'on n'a pas défini la variable `$fillable` il reste toujours possible d'utiliser l'alternative suivante :

```
$boat = new Boat;
$boat->nom = $request->nom;
$boat->modele->$request->modele;
....
```

Remarque :

Pour pouvoir utiliser la méthode `flash()` qui n'est pas installée par défaut nous allons faire appel à `composer`.

Syntaxe :

```
composer require laracast/flash
```

puis dans le fichier `config\app.php` on ajoute dans le tableau `Provider array` : `Laracasts\Flash\FlashServiceProvider::class`

4.2 Mise à jour d'un enregistrement

La mise à jour d'un enregistrement passe par la méthode `update()` qui se limite à appeler la vue qui affichera l'enregistrement à modifier :

```
    public function edit(Boat $boat)
    {
        return view('boats.edit')->with('boat',$boat);
    }
```

Création du formulaire d'édition d'un enregistrement `edit.balde.php` :

```
@extends('layouts.app')

@section ('content')
    <div class="row">
        <div class="col">
            {!! Form::model($boat,
            [
                'method'=>'put',
                'route'=>['boats.update', $boat->id],
                'class'=>'form'
            ]) !!}

            ......

            {!! Form::close() !!}
        </div>
    </div>
@endsection
```

Pour la création de ce formulaire nous utilisons cette fois la classe `Form::modele()`. Par ailleurs le formulaire utilise maintenant la méthode `PUT`.

Il faut encore mettre à jour la méthode `update()` du contrôleur.

```
    public function update(Request $request, Boat $boat)
    {
        $boat->update(
            $request->input()
        );
        return redirect()
            ->route ('boats.edit',$boat)
            ->with('message','Bateau correctement mis à jour');
    }
```

4.3 Suppression d'un enregistrement

Il faut créer un formulaire qui utilise cette fois la méthode `DELETE`.

```
...
{!! Form::open (
    [
        'route'=>['boats.destroy',$boat],
        'method'=>'delete'
    ]) !!}
    {!!  Form::submit('Effacer bateau',['class'=>'btn btn-danger']) !!}
    {!! Form::close() !!}
...
```

et mettre à jour la méthode `destroy()` du contrôleur :

```
public function destroy(Boat $boat)
    {
        $boat->delete();
        redirect()
            ->route('boats.index')
            ->with('message','Le bateau a été correctement effacé');
    }
```

Ici le contrôleur retourne sur la page d'accueil après avoir effectué la suppression de l'enregistrement sélectionné.

Conclusion :

Dans cette partie nous avons apris à interagir avec une base de données à partir d'un modèle pour effectuer toutes les opérations de base de gestion d'une base de données, puis nous nous avons complété le contrôleur de ressource afin de mettre en œuvre tous ces mécanismes pour la gestion de la table `Boat`.