Cours Laravel 5.3 – les bases – injection de dépendance, conteneur et façades

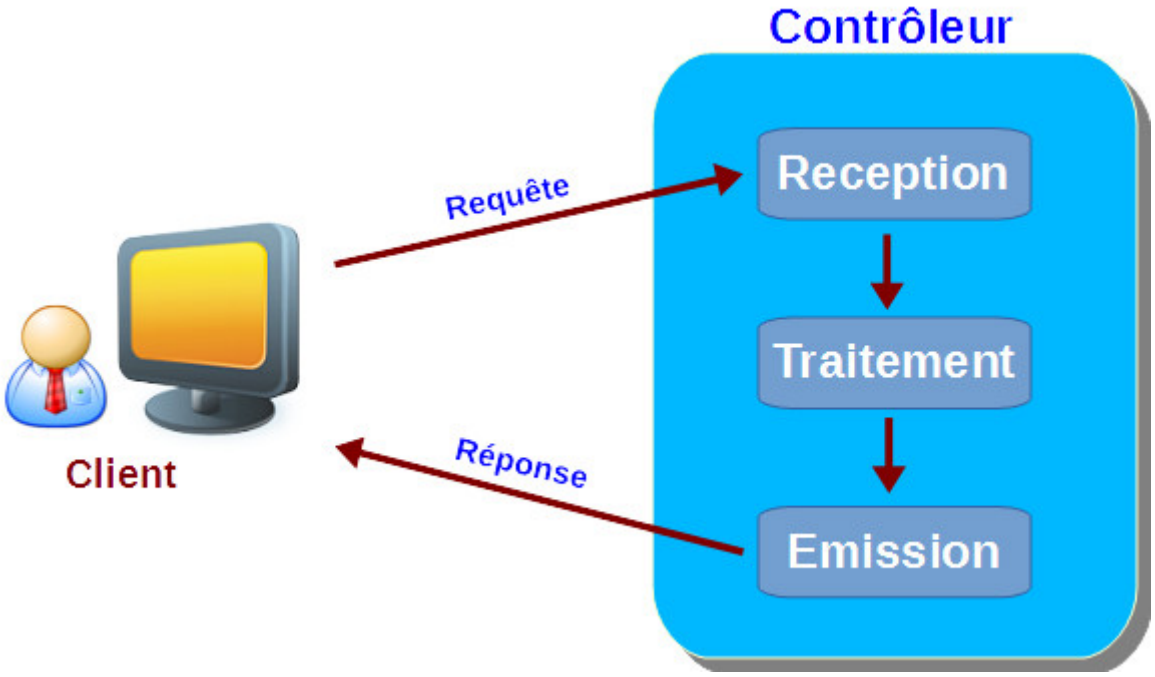
Dans ce chapitre nous allons reprendre l'exemple précédent de l'envoi de photos en nous posant des questions d'organisation du code. Laravel ce n'est pas seulement un framework pratique, c'est aussi un style de programmation. Il vaut mieux évoquer ce style le plus tôt possible dans l'apprentissage pour prendre rapidement les bonnes habitudes.

Vous pouvez très bien créer un site complet dans le fichier des routes, vous pouvez aussi vous contenter de contrôleurs pour effectuer tous les traitements nécessaires. Je vous propose une autre approche, plus en accord avec ce que nous offre Laravel.

Le problème et sa solution

Le problème

Je vous ai déjà dit qu'un contrôleur a pour mission de réceptionner les requêtes et d'envoyer les réponses. Entre les deux il y a évidemment du traitement à effectuer, la réponse doit se construire, parfois c'est très simple, parfois plus long et délicat. Mais globalement nous avons pour un contrôleur ce fonctionnement :



Reprenons la méthode **store** de notre contrôleur **PhotoController** du précédent chapitre :

```
public function store(ImagesRequest $request)
{
    $request->image->store(config('images.path'), 'public'));
    return view('photo_ok');
}
```

Qu'avons-nous comme traitement ? On récupère l'image transmise et on enregistre cette image.

La question est : est-ce qu'un contrôleur doit savoir comment s'effectue ce traitement ? Si vous avez plusieurs contrôleurs dans votre application qui doivent effectuer le même traitement vous allez multiplier cette mise en place. Imaginez que vous ayez ensuite envie de modifier l'enregistrement des images, par exemple en les mettant sur le cloud, vous allez devoir retoucher le code de tous vos contrôleurs ! La répétition de code n'est jamais une bonne chose, une saine règle de programmation veut qu'on commence à se poser des questions sur l'organisation du code dès qu'on fait des copies.

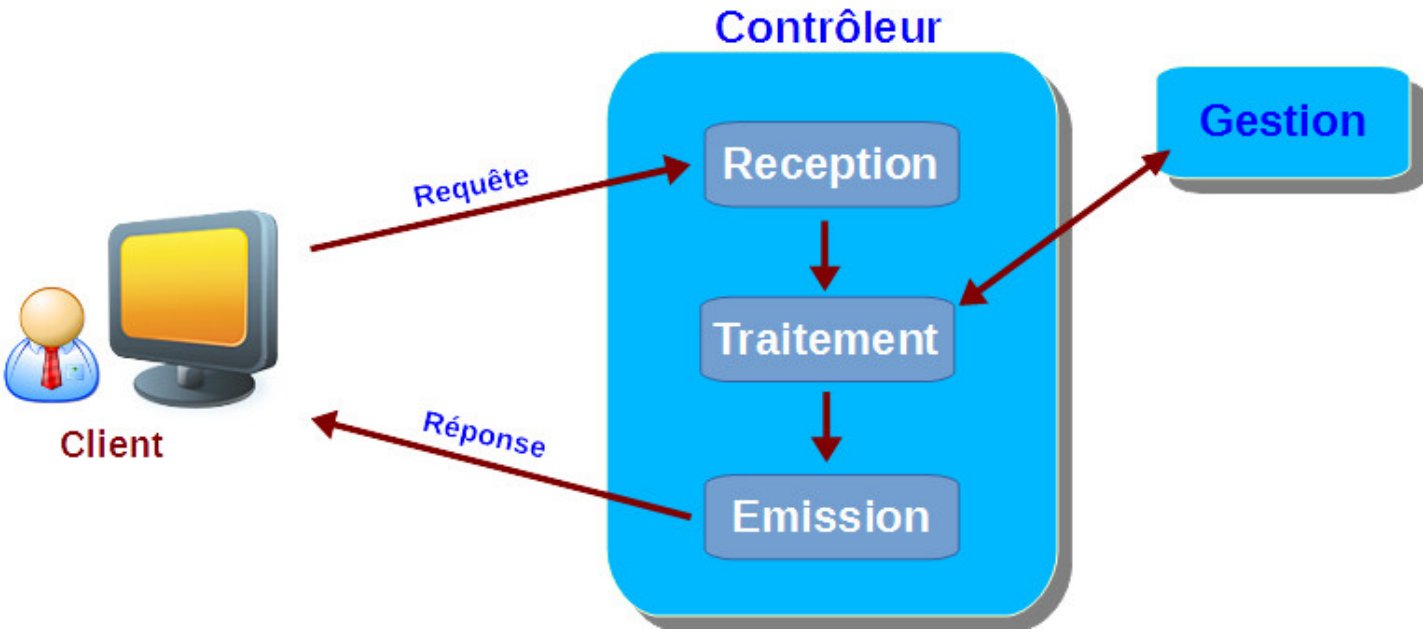
*Cette préconisation est connue sous l'acronyme **DRY** (« Don't Repeat Yourself »).*

Un autre élément à prendre en compte aussi est la testabilité des classes. Nous verrons cet aspect important du développement trop

souvent négligé. Pour qu'une classe soit testable il faut que sa mission soit simple et parfaitement identifiée et il ne faut pas qu'elle soit étroitement liée avec une autre classe. En effet cette dépendance rend les tests plus difficiles.

La solution

Alors quelle est la solution ? L'injection de dépendance ! Voyons de quoi il s'agit. Regardez ce schéma :



Une nouvelle classe entre en jeu pour la gestion, c'est elle qui est effectivement chargée du traitement, le contrôleur fait juste appel à ses méthodes. Mais comment cette classe est-elle injectée dans le contrôleur ? Voici le code du contrôleur modifié :

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Repositories\PhotosRepository;

class PhotoController extends Controller
{
    public function create()
    {
        return view('photo');
    }
```

```
    public function store(ImagesRequest $request, PhotosRepository
$photosRepository)
    {
        $photosRepository->save($request->image);
        return view('photo_ok');
    }
}
```

Vous remarquez qu'au niveau de la méthode **store** il y a un nouveau paramètre de type **App\Repositories\PhotosRepository**. On utilise la méthode **save** de la classe ainsi injectée pour faire le traitement.

De cette façon le contrôleur ignore totalement comment se fait la gestion, il sait juste que la classe **PhotosRepository** sait la faire. Il se contente d'utiliser la méthode de cette classe qui est « injectée ».

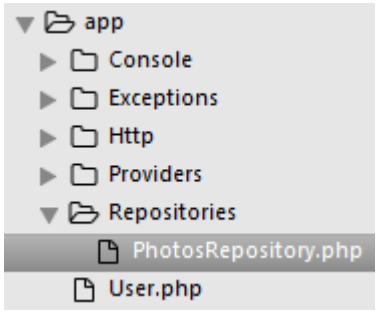
Maintenant vous vous demandez sans doute comment cette classe est injectée là, en d'autres termes comment et où est créée cette instance. Eh bien Laravel est assez malin pour le faire lui-même.

PHP est très tolérant sur les types des variables. Lorsque vous en déclarez une vous n'êtes pas obligé de préciser que c'est un string ou un array. PHP devine le type selon la valeur affectée. Il en est de même pour les paramètres des fonctions. Mais personne ne vous empêche de déclarer un type comme je l'ai fait ici pour le paramètre de la méthode (malheureusement pour le moment PHP ne reconnaît que les tableaux et les classes). C'est même indispensable pour que Laravel sache quelle classe est concernée. Étant donné que je déclare le type, Laravel est capable de créer une instance de ce type et de l'injecter dans le contrôleur.

Pour trouver la classe Laravel utilise l'introspection (**reflexion** en anglais) de PHP qui permet d'inspecter le code en cours d'exécution. Elle permet aussi de manipuler du code et donc de créer par exemple un objet d'une certaine classe. Vous pouvez trouver tous les renseignements dans le manuel PHP.

La gestion

Maintenant qu'on a dit au contrôleur qu'une classe s'occupe de la gestion il nous faut la créer. Pour bien organiser notre application on crée un nouveau dossier et on place notre classe dedans :



Le codage ne pose aucun problème parce qu'il est identique à ce qu'on avait dans le contrôleur :

```
<?php

namespace App\Repositories;

use Illuminate\Http\UploadedFile;

class PhotosRepository
{
    public function save(UploadedFile $image)
    {
        $image->store(config('images.path'), 'public');
    }
}
```

Attention à ne pas oublier les espaces de noms !

Maintenant notre code est parfaitement organisé et facile à maintenir et à tester.

Mais allons un peu plus loin, créons une interface pour notre classe :

```
<?php

namespace App\Repositories;
```

```
use Illuminate\Http\UploadedFile;

interface PhotosRepositoryInterface
{
  public function save(UploadedFile $image);
}
```

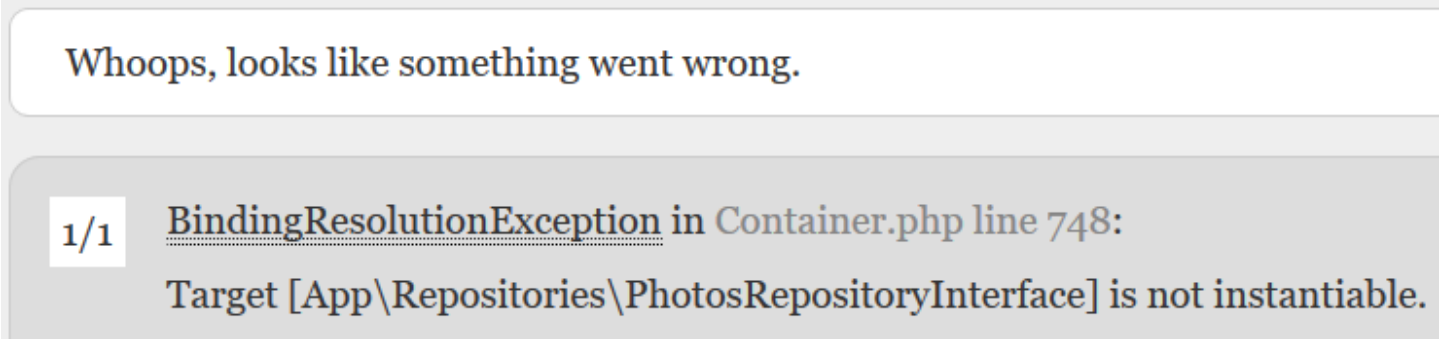
Il suffit ensuite d’en informer la classe **PhotosRepository** :

```
class PhotosRepository implements PhotosRepositoryInterface
```

Ce qui serait bien maintenant serait dans notre contrôleur de référencer l’interface :

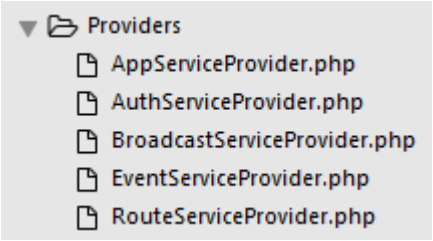
```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;
use App\Repositories\PhotosRepositoryInterface;

class PhotoController extends Controller
{
    public function create()
    {
        return view('photo');
    }

        public function store(ImagesRequest $request,
PhotosRepositoryInterface $photosRepository)
    {
        $photosRepository->save($request->image);
        return view('photo_ok');
    }
}
```

Le souci c’est que Laravel n’arrive pas à deviner la classe à instancier à partir de cette interface :



Comment s'en sortir ?

Lorsque j'ai présenté la structure de Laravel j'ai mentionné la présence de providers :



A quoi sert un provider ? Tout simplement à procéder à des initialisations : événements, middlewares, et surtout des liaisons de dépendance. Laravel possède un conteneur de dépendances qui constitue le cœur de son fonctionnement. C'est grâce à ce conteneur qu'on va pouvoir établir une liaison entre une interface et une classe.

Ouvrez le fichier **app\Providers\AppServiceProvider.php** et ajoutez cette ligne de code :

```
public function register()
{
    $this->app->bind(
        'App\Repositories\PhotosRepositoryInterface',
        'App\Repositories\PhotosRepository'
    );
}
```

La méthode **register** est activée au démarrage de l'application, c'est l'endroit idéal pour notre liaison. Ici on dit à l'application (**app**) d'établir une liaison (**bind**) entre l'interface **App\Repositories\PhotosRepositoryInterface** et la classe **App\Repositories\PhotosRepository**. Ainsi chaque fois qu'on se référera à cette interface dans une injection Laravel saura quelle

classe instancier. Si on veut changer la classe de gestion il suffit de modifier le code du provider.

Maintenant notre application fonctionne .

Si vous obtenez encore un message d'erreur vous disant que l'interface ne peut pas être instanciée lancez la commande :

composer dumpautoload

Les façades

Laravel propose de nombreuses façades pour simplifier la syntaxe. Vous pouvez les trouver toutes déclarées dans le fichier **config/app.php** :

```
'aliases' => [

    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    'Blade' => Illuminate\Support\Facades\Blade::class,
    'Cache' => Illuminate\Support\Facades\Cache::class,
    'Config' => Illuminate\Support\Facades\Config::class,
    'Cookie' => Illuminate\Support\Facades\Cookie::class,
    'Crypt' => Illuminate\Support\Facades\Crypt::class,
    'DB' => Illuminate\Support\Facades\DB::class,
    'Eloquent' => Illuminate\Database\Eloquent\Model::class,
    'Event' => Illuminate\Support\Facades\Event::class,
    'File' => Illuminate\Support\Facades\File::class,
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
    'Mail' => Illuminate\Support\Facades\Mail::class,
                                    'Notification'        =>
Illuminate\Support\Facades\Notification::class,
    'Password' => Illuminate\Support\Facades\Password::class,
    'Queue' => Illuminate\Support\Facades\Queue::class,
    'Redirect' => Illuminate\Support\Facades\Redirect::class,
    'Redis' => Illuminate\Support\Facades\Redis::class,
```

```
    'Request' => Illuminate\Support\Facades\Request::class,
    'Response' => Illuminate\Support\Facades\Response::class,
    'Route' => Illuminate\Support\Facades\Route::class,
    'Schema' => Illuminate\Support\Facades\Schema::class,
    'Session' => Illuminate\Support\Facades\Session::class,
    'Storage' => Illuminate\Support\Facades\Storage::class,
    'URL' => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View' => Illuminate\Support\Facades\View::class,
],
```

Vous trouvez dans ce tableau le nom de la façade et la classe qui met en place cette façade. Par exemple pour les routes on a la façade **Route** qui correspond à la classe **Illuminate\Support\Facades\Route**. Regardons cette classe :

```
<?php

namespace Illuminate\Support\Facades;

/**
 * @see \Illuminate\Routing\Router
 */
class Route extends Facade
{
    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor()
    {
        return 'router';
    }
}
```

On se contente de retourner ‘router’. Il faut aller voir dans le fichier **Illuminate\Routing\RoutingServiceProvider** pour trouver l’enregistrement du router :

```
protected function registerRouter()
{
    $this->app['router'] = $this->app->share(function ($app) {
        return new Router($app['events'], $app);
```

```
    });
}
```

Les providers permettent d'enregistrer des composants dans le conteneur de Laravel. Ici on déclare 'router' et on voit qu'on crée une instance de la classe **Router** (new Router…). Le nom complet est **Illuminate\Routing\Router**. Si vous allez voir cette classe vous trouverez les méthodes qu'on a utilisées dans ce chapitre, par exemple **get** :

```
public function get($uri, $action = null)
{
    return $this->addRoute(['GET', 'HEAD'], $uri, $action);
}
```

Autrement dit si j'écris en utilisant la façade :

```
Route::get('/', function() { return 'Coucou'; });
```

J'obtiens le même résultat que si j'écris en allant chercher le routeur dans le conteneur :

```
$this->app['router']->get('/', function() { return 'Coucou'; });
```

Ou encore en utilisant un helper :

```
app('router')->get('/', function() { return 'Coucou'; });
```

La différence est que la première syntaxe est plus simple et intuitive mais certains n'aiment pas trop ce genre d'appel statique.

En résumé

- Un contrôleur doit déléguer toute tâche qui ne relève pas de sa compétence.
- L'injection de dépendance permet de bien séparer les tâches, de simplifier la maintenance du code et les tests unitaires.
- Les providers permettent de faire des initialisations, en particulier des liaisons de dépendance entre interfaces et classes.

- Laravel est équipé de nombreuses façades qui simplifient la syntaxe.
- Il existe aussi des helpers pour simplifier la syntaxe.