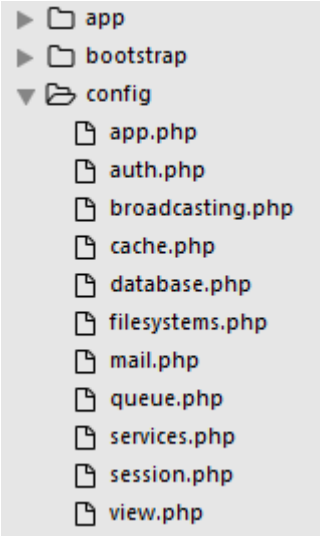
Cours Laravel 5.5 – les bases – configuration, session et gestion de fichiers

Dans ce chapitre nous verrons la configuration, la gestion des sessions et des fichiers avec un exemple simple d'envoi et d'enregistrement de fichiers images dans un dossier à partir d'un formulaire.

La configuration

Tout ce qui concerne la configuration de Laravel se trouve dans le dossier **config** :



De nombreuses valeurs de configuration sont définies dans le fichier **.env**.

Les fichiers de configuration contiennent en fait juste un tableau avec des clés et des valeurs. Par exemple pour les vues (**view.php**) :

```
<?php

return [

    'paths' => [
```

```
        resource_path('views'),
    ],

    ...

];
```

On a la clé **paths** et la valeur : un tableau avec . Pour récupérer une valeur il suffit d'utiliser sa clé avec l'helper config :

```
config('view.paths');
```

On utilise le nom du fichier (**view**) et le nom de la clé (**paths**) séparés par un point.

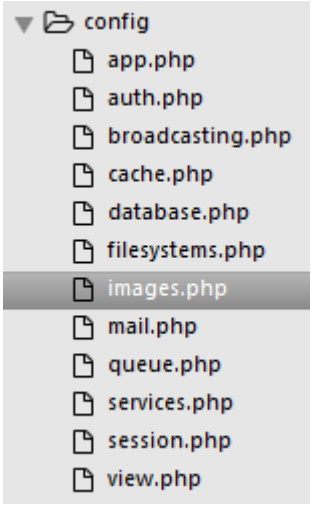
On peut aussi changer une valeur :

```
config(['view.paths' => resource_path().'/mes_vues']);
```

Si je fais effectivement cela mes vues, au lieu d'être cherchées dans le dossier **resources/views** seront cherchées dans le dossier **resources/mes_vues**.

Lorsqu'on change ainsi une valeur de configuration ce n'est valable que pour la requête en cours.

Vous pouvez évidemment créer vos propres fichiers de configuration. Pour l'exemple de ce chapitre on va avoir besoin justement d'utiliser une configuration. Comme notre application doit enregistrer des fichiers d'images dans un dossier il faut définir l'emplacement et le nom de ce dossier de destination. On va donc créer un fichier **images.php** :



Dans ce fichier on va définir le nom du dossier :

```
return ['path' => 'uploads'];
```

*En production pour gagner en performances il est conseillé de mettre toute la configuration en cache dans un seul fichier avec la commande **php artisan config:cache**.*

Les sessions

Étant donné que les requêtes HTTP sont fugitives et ne laissent aucune trace il est important de disposer d'un système qui permet de mémoriser des informations entre deux requêtes. C'est justement l'objet des sessions.

La configuration des sessions se trouve dans le fichier de configuration **session.php**. On trouve dans le fichier **.env** la définition du driver :

```
SESSION_DRIVER=file
```

Par défaut c'est un fichier (dans **storage/framework/sessions**) qui mémorise les informations des sessions mais on peut aussi utiliser : les cookies, la base de données, apc, memcached, redis…

Quel que soit le driver utilisé l'helper **session** de Laravel permet une gestion simplifiée des sessions. Vous pouvez ainsi créer une variable de session :

```
session(['clé' => 'valeur']);
```

Vous pouvez aussi récupérer une valeur à partir de sa clé :

```
$valeur = session('clef');
```

Vous pouvez prévoir une valeur par défaut :

```
$valeur = session('clef', 'valeur par défaut');
```

Il est souvent utile (ça sera le cas pour notre exemple) de savoir si une certaine clé est présente en session :

```
if (session()->has('error')) ...
```

Ces informations demeurent pour le même client à travers ses requêtes. Laravel s'occupe de ces informations, on se contente de lui indiquer un couple clé-valeur et il s'occupe de tout.

Ce ne sont là que les méthodes de base pour les sessions, vous trouverez tous les renseignements complémentaires dans la documentation.

La gestion des fichiers

Laravel utilise Flysystem comme composant de gestion de fichiers. Il en propose une API bien pensée et facile à utiliser. On peut ainsi manipuler fichiers et dossiers de la même manière en local ou à distance, que ce soit en FTP ou sur le cloud.

La configuration du système se trouve dans le fichier **config/filesystem.php**. Par défaut on est en local :

```
'default' => env('FILESYSTEM_DRIVER', 'local'),
```

Mais on peut aussi choisir ftp, s3 ou rackspace (pour ces deux derniers il faut installer les composants correspondants). On peut aussi mettre en place un accès à d'autres possibilités et il existe des composants à ajouter, comme par exemple celui-ci pour Dropbox.

On peut définir des « disques » (**disks**) qui sont des cibles combinant un driver, un dossier racine et différents éléments de configuration :

```
'disks' => [

    'local' => [
        'driver' => 'local',
        'root' => storage_path('app'),
    ],

    'public' => [
        'driver' => 'local',
        'root' => storage_path('app/public'),
        'url' => env('APP_URL').'/storage',
```

```
        'visibility' => 'public',
    ],

    's3' => [
        'driver' => 's3',
        'key' => env('AWS_KEY'),
        'secret' => env('AWS_SECRET'),
        'region' => env('AWS_REGION'),
        'bucket' => env('AWS_BUCKET'),
    ],

],
```

Donc par défaut c'est le disque local qui est actif et le dossier racine est **storage/app**.

Autrement dit si j'utilise ce code :

```
Storage::disk('local')->put('recettes.txt', 'Contenu du fichier');
```

Je vais envoyer le fichier **recettes.txt** dans le dossier **storage/app**.

Et si j'utilise ce code :

```
Storage::disk('public')->put('recettes.txt', 'Contenu du fichier');
```

Cette fois j'envoie le fichier dans le dossier **storage/app/public**. Par convention ce dossier doit être accessible depuis le web.

Mais je vous ai dit précédemment que seul le dossier **public** à la racine doit être accessible. Alors ?

Alors pour que ce soit possible il faut créer un lien symbolique **public/storage** qui pointe sur **storage/app/public**. Il y a d'ailleurs une commande d'Artisan pour ça :

```
php artisan storage:link
```

Mais franchement je préfère créer directement un dossier dans **public**. Les motivations avancées (ne pas perdre de fichiers au déploiement) me paraissent trop minces.

Donc rien n'empêche de changer la configuration :

```
'public' => [
    'driver' => 'local',
    'root' => public_path(),
    'visibility' => 'public',
],
```

Ainsi le disque « public » pointe sur le dossier **public**. C'est d'ailleurs ce qu'on va faire pour l'exemple de ce chapitre.

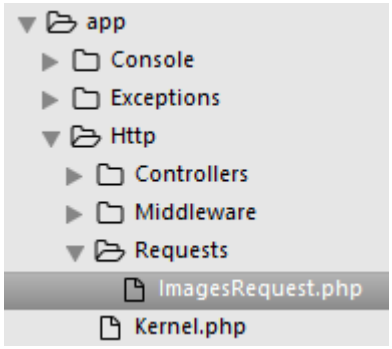
Il y a de nombreuses méthodes pour manipuler dossiers et fichiers, je ne vais pas développer tout ça pour le moment, vous avez le détail dans la documentation. On va surtout utiliser les possibilités de téléchargement des fichiers dans ce chapitre.

La requête de formulaire

Nous allons encore avoir besoin d'une requête de formulaire pour la validation. Comme nous l'avons déjà vu nous utilisons la commande d'Artisan pour la créer :

```
php artisan make:request ImagesRequest
```

Vous devez trouver le fichier ici :



Changez le code pour celui-ci :

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ImagesRequest extends FormRequest
{
```

```
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
                                    return    ['image'    =>
'required|image|dimensions:min_width=100,min_height=100'];
    }
}
```

On a seulement 3 règles pour le champ image :

- le champ est obligatoire (**required**),
- ce doit être une image (**image**),
- l'image doit faire au minimum 100 * 100 pixels (**dimensions**).

Maintenant notre validation est prête.

Les routes et le contrôleur

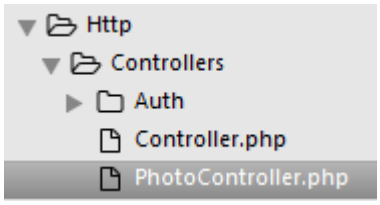
On va avoir besoin de deux routes :

```
Route::get('photo', 'PhotoController@create');
Route::post('photo', 'PhotoController@store');
```

On utilise Artisan pour créer le contrôleur :

```
php artisan make:controller PhotoController
```

Vous devez trouver le fichier ici :



Changez le code pour celui-ci :

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ImagesRequest;

class PhotoController extends Controller
{

    public function create()
    {
        return view('photo');
    }

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

Donc au niveau des urls :

- **http://monsite.fr/photo** avec le verbe **get** pour la demande du formulaire,
- **http://monsite.fr/photo** avec le verbe **post** pour la soumission du formulaire et l'envoi du fichier image associé.

En ce qui concerne le traitement de la soumission, vous remarquez qu'on récupère le chemin du dossier d'enregistrement qu'on a prévu dans la configuration :

```
config('images.path')
```

La partie intéressante se situe avec ce code :

```
$request->image->store(config('images.path'), 'public')
```

On récupère l'image avec **$request->image**. Ce qu'on obtient là est une instance de la classe **Illuminate/Http/UploadedFile**.

Laravel dispose d'une documentation complète de ses classes, par exemple vous trouvez cette classe ici avec la méthode store documentée.

La méthode **store** génère automatiquement un nom de fichier basé sur son contenu (hashage MD5) et elle retourne le chemin complet.

Les vues

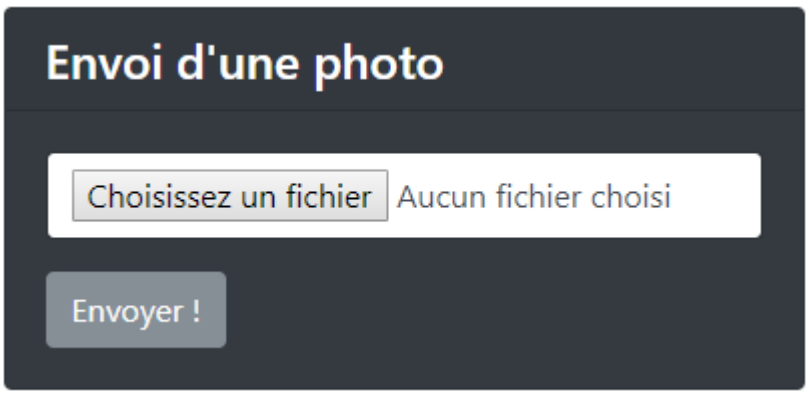
On va utiliser le template des chapitres précédents (resources/views/template.blade.php) :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
          <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
        <title>Mon joli site</title>
                                    <link    rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/boo
tstrap.min.css"                              integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M"
crossorigin="anonymous">
        <style>
            textarea { resize: none; }
            .card { width: 25em; }
        </style>
    </head>
    <body>
        @yield('contenu')
    </body>
</html>
```

Voici la vue pour le formulaire (**resources/views/photo.blade.php**) :

```
@extends('template')

@section('contenu')
    <br>
    <div class="container">
        <div class="row card text-white bg-dark">
            <h4 class="card-header">Envoi d'une photo</h4>
            <div class="card-body">
                 <form action="{{ url('photo') }}" method="POST"
enctype="multipart/form-data">
                    {{ csrf_field() }}
                    <div class="form-group">
                        <input type="file" class="form-control {{
$errors->has('image') ? 'is-invalid' : '' }}" name="image"
id="image"  value="{{ old('image') }}">
                            {!! $errors->first('image', '<div
class="invalid-feedback">:message</div>') !!}
                    </div>
                       <button type="submit" class="btn btn-
secondary">Envoyer !</button>
                </form>
            </div>
        </div>
    </div>
@endsection
```

Avec cet aspect :



Remarquez comment est créé le formulaire :

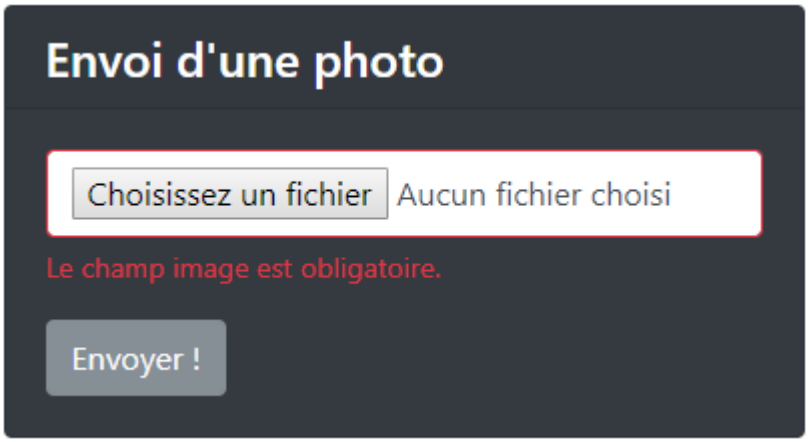
```
<form  action="{{  url('photo')  }}"  method="POST"
enctype="multipart/form-data">
```

On a prévu le type **mime** nécessaire pour associer un fichier lors

de la soumission :

enctype="multipart/form-data"

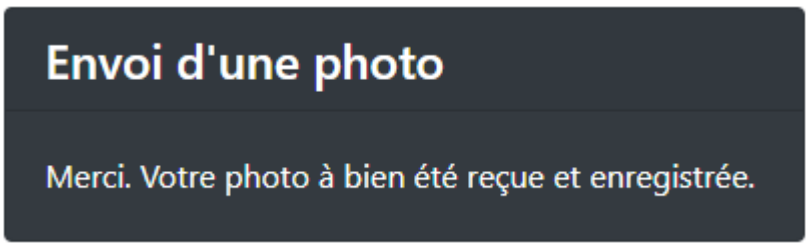
En cas d'erreur de validation le message est affiché et la bordure du champ devient rouge :



Et voici la vue pour la confirmation en retour (**app/views/photo_ok.blade.php**) :

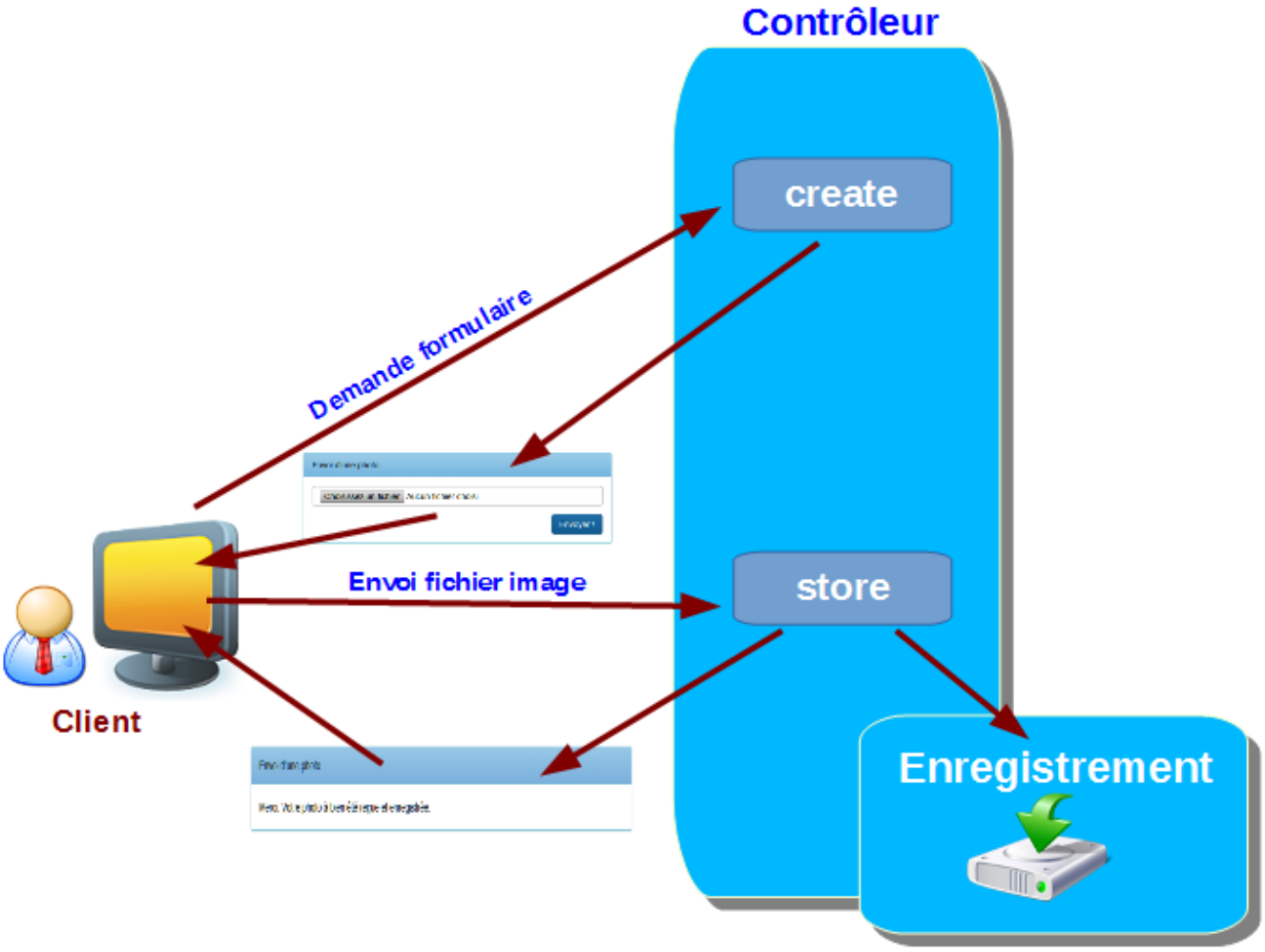
```
@extends('template')

@section('contenu')
    <br>
    <div class="container">
        <div class="row card text-white bg-dark">
            <h4 class="card-header">Envoi d'une photo</h4>
            <div class="card-body">
                <p class="card-text">Merci. Votre photo à bien été reçue et enregistrée.</p>
            </div>
        </div>
    </div>
@endsection
```

Avec cet aspect :



On retrouve normalement le fichier bien rangé dans le dossier prévu :

Voici le schéma de fonctionnement :



En résumé

- Les fichiers de configuration permettent de mémoriser facilement des ensembles clé-valeur et sont gérés par l'helper **config**.
- Les sessions permettent de mémoriser des informations concernant un client et sont facilement manipulables avec l'helper **session**.
- Laravel comporte un système complet de gestion de fichiers en local, à distance ou sur le cloud avec une API commune. Il est facile de créer un système de téléchargement de fichier.