Understanding and Working with Files in Laravel

File uploads is one the most commonly used features on the web. From uploading avatars to family pictures to sending documents via email, we can't do without files on the web.

In today's article will cover all the ways to handle files in Laravel. If you are new to Laravel, browse the courses or navigate to the tutorials section. After reading the article, If we left something out please let us know in the comments and we'll update the post accordingly.

Handling of files is another thing Laravel has simplified in its ecosystem. Before we get started, we'll need a few things. First, a Laravel project. There are a few ways to create a new Laravel project, but let's stick to composer for now.

Table of Contents


```
composer create-project --prefer-dist laravel/laravel files
```

Where `files` is the name of our project. After installing the app, we'll need a few packages installed, so, let's get them out of the way. You should note that these packages are only necessary if you intend to save images to Amazon's s3 or manipulate images like cropping, filters etc.

```
composer require league/flysystem-aws-s3-v3:~1.0 intervention/image:~2.4
```

After installing the dependencies, the final one is Mailtrap. Mailtrap is a fake SMTP server for development teams to test, view and share emails sent from the development and staging environments without spamming real customers. So head over to Mailtrap and create a new inbox for testing.

Then, in `welcome.blade.php` update the head tag to:

```
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>File uploads</title>
<style>
  * {
    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto,
        "Helvetica Neue", Arial, sans-serif, "Apple Color Emoji",
```

```
        "Segoe UI Emoji", "Segoe UI Symbol";
  }
</style>
```

Modify the body contents to:

```
<form action="/process" enctype="multipart/form-data" method="POST">
    <p>
        <label for="photo">
            <input type="file" name="photo" id="photo">
        </label>
    </p>
    <button>Upload</button>
    {{ csrf_field() }}
</form>
```

For the file upload form, the `enctype="multipart/form-data"` and `method="POST"` are extremely important as the browser will know how to properly format the request. `{{ csrf_field() }}` is Laravel specific and will generate a hidden input field with a token that Laravel can use to verify the form submission is legit.

If the CSRF token does not exist on the page, Laravel will show “The page has expired due to inactivity” page.

Now that we have our dependencies out of the way, let's get started.

Understanding How Laravel Handles Files

Development as we know it in 2018 is growing fast, and in most cases there are many solutions to one problem. Take file hosting for example, now we have so many options to store files, the sheer number of solutions ranging from self hosted to FTP to cloud storage to GFS and many others.

Since Laravel is framework that encourages flexibility, it has a native way to handle the many file structures. Be it local, Amazon's s3, Google's Cloud, Laravel has you covered.

Laravel's solution to this problem is to call them disks. Makes sense, any file storage system you can think of can be labeled as a disk in Laravel. To this regard, Laravel comes with native support for some providers (disks). We have: local, public, s3, rackspace, FTP etc. All this is possible because of Flysystem.

If you open `config/filesystems.php` you’ll see the available disks and their respected configuration.

File Uploads in Laravel

From the introduction section above, we have a form with a file input ready to be processed. We can see that the form is pointed to /process. In routes/web.php, we define a new POST /process route.

```
use Illuminate\Http\Request;

Route::post('process', function (Request $request) {
    $path = $request->file('photo')->store('photos');

    dd($path);
});
```

What the above code does is grab the photo field from the request and save it to the photos folder. dd() is a Laravel function that kills the running script and dumps the argument to the page. For me, the file was saved to "photos/3hcX8yrOs2NYhpadt4Eacq4TFtpVYUCw6VTRJhfn.png". To find this file on the file system, navigate to storage/app and you'll find the uploaded file.

If you don't like the default naming pattern provided by Laravel, you can provide yours using the storeAs method.

```
Route::post('process', function (Request $request) {
    // cache the file
    $file = $request->file('photo');

    // generate a new filename. getClientOriginalExtension() for the file
extension
    $filename = 'profile-photo-' . time() . '.' . $file-
>getClientOriginalExtension();

    // save to storage/app/photos as the new $filename
    $path = $file->storeAs('photos', $filename);

    dd($path);
});
```

After running the above code, I got "photos/profile-photo-1517311378.png".

Difference Between Local and Public Disks

In config/filesystems.php you can see the disks local and public defined. By default, Laravel uses the local disk configuration. The major difference between local and public disk is that local is private and cannot be accessed from the browser while public can be accessed from the browser.

Since the public disk is in storage/app/public and Laravel's server root is in public you need to link storage/app/public to Laravel's public folder. We can do that with our trusty artisan by running php artisan storage:link.

Uploading Multiple Files

Since Laravel doesn't provide a function to upload multiple files, we need to do that ourselves. It's not much different from what we've been doing so far, we just need a loop.

First, let's update our file upload input to accept multiple files.

```
<input type="file" name="photos[]" id="photo" multiple>
```

When we try to process this $request->file('photos'), it's now an array of UploadedFile instances so we need to loop through the array and save each file.

```
Route::post('process', function (Request $request) {
    $photos = $request->file('photos');
    $paths  = [];

    foreach ($photos as $photo) {
        $extension = $photo->getClientOriginalExtension();
        $filename  = 'profile-photo-' . time() . '.' . $extension;
        $paths[]   = $photo->storeAs('photos', $filename);
    }
```

```
    dd($paths);
});
```

After running this, I got the following array, since I uploaded a GIF and a PNG:

```
array:2 [▼
  0 => "photos/profile-photo-1517315875.gif"
  1 => "photos/profile-photo-1517315875.png"
]
```

Validating File Uploads

Validation for file uploads is extremely important. Apart from preventing users from uploading the wrong file types, it's also for security. Let me give an example regarding security. There's a PHP configuration option cgi.fix_pathinfo=1. What this does is when it encounters a file like https://site.com/images/evil.jpg/nonexistent.php, PHP will assume nonexistent.php is a PHP file and it will try to run it. When it discovers that nonexistent.php doesn't exists, PHP will be like "I need to fix this ASAP" and try to execute evil.jpg (a PHP file disguised as a JPEG). Because evil.jpg wasn't validated when it was uploaded, a hacker now has a script they can freely run live on your server… Not… good.

To validate files in Laravel, there are so many ways, but let's stick to controller validation.

```
Route::post('process', function (Request $request) {
    // validate the uploaded file
    $validation = $request->validate([
        'photo' => 'required|file|image|mimes:jpeg,png,gif,webp|max:2048'
        // for multiple file uploads
        // 'photo.*' => 'required|file|image|mimes:jpeg,png,gif,webp|max:2048'
    ]);
    $file      = $validation['photo']; // get the validated file
    $extension = $file->getClientOriginalExtension();
    $filename  = 'profile-photo-' . time() . '.' . $extension;
    $path      = $file->storeAs('photos', $filename);

    dd($path);
});
```

For the above snippet, we told Laravel to make sure the field with a name of photo is required, a successfully uploaded file, it's an image, it has one of the defined mime types, and it's a max of 2048 kilobytes ~~ 2 megabytes.

Now, when a malicious user uploads a disguised file, the file will fail validation and if for some weird reason you leave cgi.fix_pathinfo on, this is not a means by which you can get PWNED!!!

If you head over to Laravel's validation page you'll see a whole bunch of validation rules.

Moving Files to the Cloud

Okay, your site is now an adult, it has many visitors and you decide it's time to move to the cloud. Or maybe from the beginning, you decided your files will live on separate server. The good news is Laravel comes with support for many cloud providers, but, for this tutorial, let's stick with Amazon.

Earlier we installed league/flysystem-aws-s3-v3 through composer. Laravel will automatically look for it if you choose to use Amazon S3 or throw an exception.

To upload files to the cloud, just use:

```
$request->file('photo')->store('photos', 's3');
```

For multiple file uploads:

```
foreach ($photos as $photo) {
    $extension = $photo->getClientOriginalExtension();
    $filename  = 'profile-photo-' . time() . '.' . $extension;
    $paths[]   = $photo->storeAs('photos', $filename, 's3');
}
```

Users may have already uploaded files before you decide to switch to a cloud provider, you can check the upcoming sections for what to do when files already exist.

Note: you'll have to configure your Amazon s3 credentials in config/filesystems.php.**

Sending Files as Email Attachments

Before we do this, let's quickly configure our mail environment. In .env file you will see this section

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

We need a username and password which we can get at Mailtrap.io. Mailtrap is really good for testing emails during development as you don't have to crowd your email with spam. You can also share inboxes with team members or create separate inboxes.

First, create an account and login:

1. Create a new inbox
2. Click to open inbox
3. Copy username and password under SMTP section

After copying credentials, we can modify .env to:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=8a1d546090493b
MAIL_PASSWORD=328dd2af5aefc3
MAIL_ENCRYPTION=null
```

Don't bother using mine, I deleted it.

Create your mailable

```
php artisan make:mail FileDownloaded
```

Then, edit its build method and change it to:

```
public function build()
{
    return $this->from('files@mailtrap.io')
        ->view('emails.files_downloaded')
```

```
        ->attach(storage_path('app/file.txt'), [
            'as' => 'secret.txt'
        ]);
}
```

As you can see from the method above, we pass the absolute file path to the attach() method and pass an optional array where we can change the name of the attachment or even add custom headers. Next we need to create our email view.

Create a new view file in resources/views/emails/files_downloaded.blade.php and place the content below.

```
<h1>Only you can stop forest fires</h1>
<p>Lorem, ipsum dolor sit amet consectetur adipisicing elit. Labore at
reiciendis consequatur, ea culpa molestiae ad minima est quibusdam ducimus
laboriosam dolorem, quasi sequi! Atque dolore ullam nisi accusantium. Tenetur!</
p>
```

Now, in routes/web.php we can create a new route and trigger a mail when we visit it.

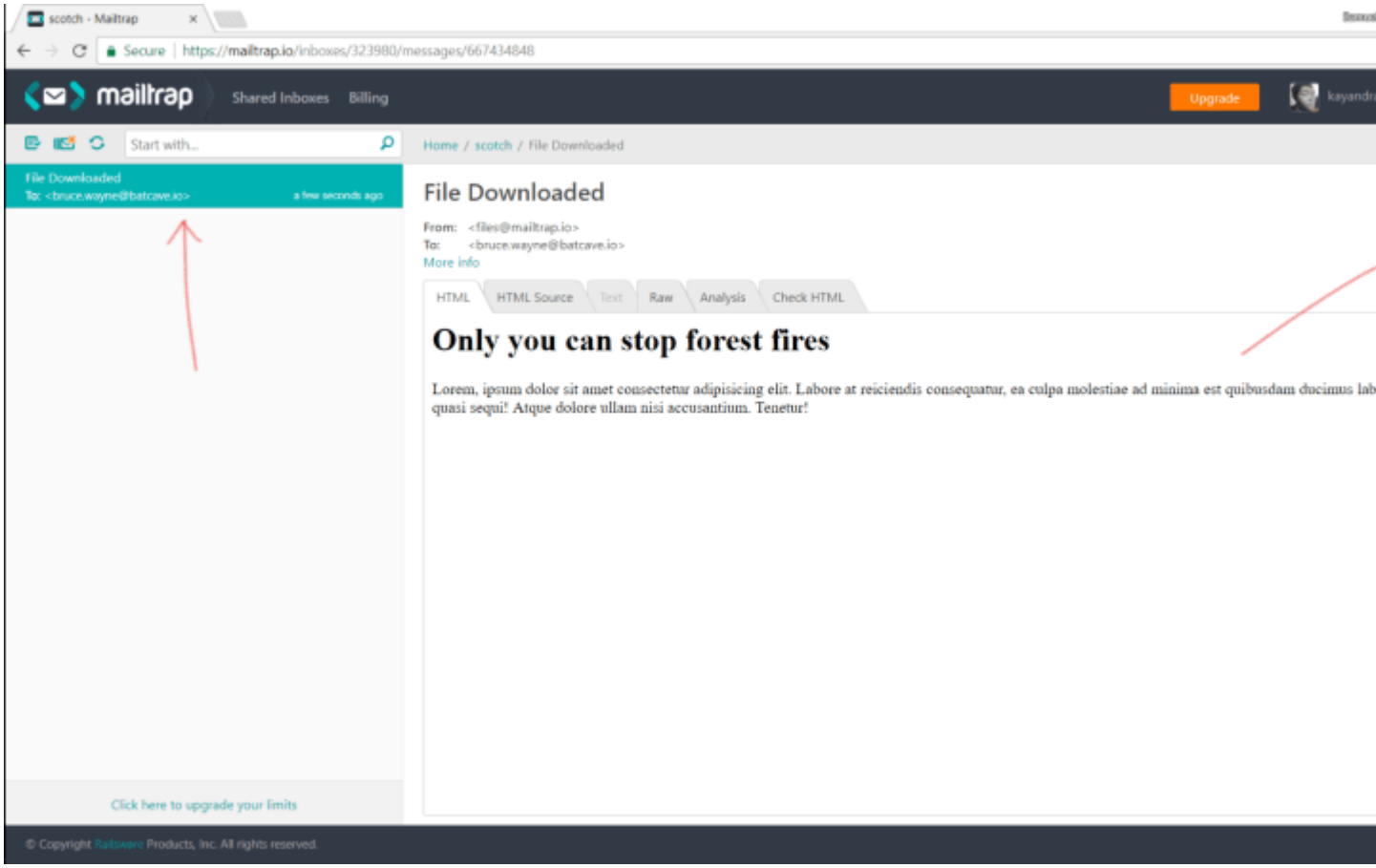
```
use App\Mail\FileDownloaded;

Route::get('mail', function () {
    $email = 'bruce.wayne@batcave.io';

    Mail::to($email)->send(new FileDownloaded);

    dd('done');
});
```

If you head over to Mailtrap, you should see this.



Storage Facade for When Files Already Exist

In an application, it’s not every time we process files through uploads. Sometimes, we decide to defer cloud file uploads till a certain user action is complete. Other times we have some files on disk before switching to a cloud provider. For times like this, Laravel provides a convenient Storage facade. For those who don’t know, facades in Laravel are class aliases. So instead of doing something like Symfony\File\Whatever\Long\Namespace\UploadedFile, we can do Storage instead.

Choosing a disk to upload file. If no disk is specified, Laravel looks in config/filesystems.php and use the default disk.

```
Storage::disk('local')->exists('file.txt');
```

use default cloud provider

```
// Storage::disk('cloud')->exists('file.txt'); will not work so do:
Storage::cloud()->exists('file.txt');
```

Create a new file with contents

```
Storage::put('file.txt', 'Contents');
```

Prepend to file

```
Storage::prepend('file.txt', 'Prepended Text');
```

Append to file

```
Storage::append('file.txt', 'Prepended Text');
```

Get file contents

```
Storage::get('file.txt')
```

Check if file exists

```
Storage::exists('file.txt')
```

Force file download

```
Storage::download('file.txt', $name, $headers); // $name and $headers are
optional
```

Generate publicly accessible URL

```
Storage::url('file.txt');
```

Generate a temporary public URL (i.e files that won’t exists after a set time). This will only work for cloud providers as Laravel doesn’t yet know how to handle generation of temporary URLs for local disk.

```
Storage::temporaryUrl('file.txt’, now()->addMinutes(10));
```

Get file size

```
Storage::size('file.txt');
```

Last modified date

```
Storage::lastModified('file.txt')
```

Copy files

```
Storage::copy('file.txt', 'shared/file.txt');
```

Move files

```
Storage::move('file.txt', 'secret/file.txt');
```

Delete files

```
Storage::delete('file.txt');

// to delete multiple files
Storage::delete(['file1.txt', 'file2.txt']);
```

Manipulating files

Resizing images, adding filters etc. This is where Laravel needs external help. Adding this feature natively to Laravel will only bloat the application since not installs need it. We need a package called intervention/image. We already installed this package, but for reference.

```
composer require intervention/image
```

Since Laravel can automatically detect packages, we don't need to register anything. If you are using a version of Laravel lesser than 5.5 read this.

To resize an image

```
$image = Image::make(storage_path('app/public/profile.jpg'))->resize(300, 200);
```

Even Laravel's packages are fluent.

You can head over to their website and see all the fancy effects and filters you can add to your image.

Don’t forget directories

Laravel also provides handy helpers to work with directories. They are all based on PHP iterators so they'll provide the utmost performance.

To get all files:

```
Storage::files
```

To get all files in a directory including files in sub-folders

```
Storage::allFiles($directory_name);
```

To get all directories within a directory

```
Storage::directories($directory_name);
```

To get all directories within a directory including files in sub-directories

```
Storage::allDirectories($directory_name);
```

Make a directory

```
Storage::makeDirectory($directory_name);
```

Delete a directory

```
Storage::deleteDirectory($directory_name);
```

Conclusion

If we left anything out, please let us know down in the comments. Also, checkout Mailtrap, they are really good and they will help you sail through the development phase with regards to debugging emails.

https://scotch.io/tutorials/understanding-and-working-with-files-in-laravel

Sistema de arquivos / Armazenamento na nuvem

Introdução

O Laravel provê uma maravilhosa abstração do sistema de arquivos graças ao pacote PHP Flysystem, do Frank de Jonge. A integração "Laravel Flysystem" possibilita uma forma simples para utilizar drivers para trabalhar com sistemas de arquivos locais, Amazon S3, a Rackspace Cloud Storage. Melhor do que isso, é extremamente simples alternar entre essas opções de armazenamento. A API trata cada sistema de armazenamento da mesma forma!

Configuração

O arquivo de configuração dos sistema de arquivos está localizado em `config/filesystems.php`. Neste arquivo você pode configurar todos os seus "discos". Cada disco representa um driver de armazenamento particular e o local de armazenamento. No arquivo de configuração você encontra exemplos para cada driver suportado. Então, simplesmente modifique a configuração para refletir as suas preferências e credenciais do seu armazenamento!

Antes de tilizar os drivers S3 or Rackspace, voCê precisará instalar os devidos pacotes através do Composer:

- Amazon S3: `league/flysystem-aws-s3-v2 ~1.0`
- Rackspace: `league/flysystem-rackspace ~1.0`

A propósito, você pode configurar quantos discos você quiser, e pode ter múltiplos discos que usem o mesmo driver.

Quando utilizando o driver `local`, observe que todas as operações com arquivos são relativas ao diretório `root` definido em seu arquivo de configuração. Por padrão, este valor é configurado no doretório `storage/app`. Portanto, o seguinte método iria armazenar um arquivo em `storage/app/file.txt`:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Utilização básica

A fachada `Storage` pode ser usada para interagir com qualquer um dos seus discos configurados. Alternativamente, você pode tipar o contrato `Illuminate\Contracts\Filesystem\Factory` em qualquer classe que é resolvida via container de serviços do Laravel.

Obtendo um disco específico

```
$disk = Storage::disk('s3');

$disk = Storage::disk('local');
```

Verificando se um arquivo existe

```
$exists = Storage::disk('s3')->exists('file.jpg');
```

Executando métodos no disco padrão

```
if (Storage::exists('file.jpg'))
{
    //
}
```

Obtendo o conteúdo de um arquivo

```
$contents = Storage::get('file.jpg');
```

Atribuindo o conteúdo a um arquivo

```
Storage::put('file.jpg', $contents);
```

Colocando conteúdo no início de um arquivo (prepend)

```
Storage::prepend('file.log', 'Prepended Text');
```

Colocando conteúdo no fim de um arquivo (append)

```
Storage::append('file.log', 'Appended Text');
```

Excluindo um arquivo

```
Storage::delete('file.jpg');

Storage::delete(['file1.jpg', 'file2.jpg']);
```

Copiando um arquivo para um novo local

```
Storage::copy('old/file1.jpg', 'new/file1.jpg');
```

Movendo um arquivo para um novo local

```
Storage::move('old/file1.jpg', 'new/file1.jpg');
```

Obtendo o tamanho de um arquivo

```
$size = Storage::size('file1.jpg');
```

Obtendo a data da última nodificação (UNIX)

```
$time = Storage::lastModified('file1.jpg');
```

Obtendo todos os arquivos de um diretório

```
$files = Storage::files($directory);
```

```
// Recursivo...
$files = Storage::allFiles($directory);
```

Obtendo todos os diretórios de um diretório

```
$directories = Storage::directories($directory);

// Recursivo...
$directories = Storage::allDirectories($directory);
```

Cria um diretório

```
Storage::makeDirectory($directory);
```

Apaga um diretório

```
Storage::deleteDirectory($directory);
```

Sistemas de arquivos Customizados

A integração do sistema de arquivos do Laravel fornece drivers para vários "drivers" de fora da caixa, contudo, sistemas de arquivos não é limitado a esses e tem adaptadores para vários outros sistemas de armazenamento. Você pode criar um driver customizado se você desejar usar um desses adaptadores adicionais nas suas aplicações Laravel. Não se precupe, não é tão difícil.

A fim de criar o sistema de arquivos você precisará criar um fornecedor de serviços, tais como `DropboxFilesystemServiceProvider`. No método `boot` dos fornecedores, você pode injetar uma instância do contrato `Illuminate\Contracts\Filesystem\Factory` e chamar o método `extend` da instância injetada. Alternativamente, você pode usar o método `extend` da fachada `Disk`.

O primeiro argumento do método `extend` é o nome do driver e o segundo é uma é uma Closure(função anômima declarada dentro do escopo de outra.) que recebe as variáveis `$app`e `$config`. O resolvedor da Closure deve retornar a instância de `League\Flysystem\Filesystem`.

> **Nota** A variável $config já irá conter os valores definidos no arquivo `config/filesystems.php` para o disco especificado.

O Examplo do Dropbox

```
<?php namespace App\Providers;

use Storage;
use League\Flysystem\Filesystem;
use Dropbox\Client as DropboxClient;
use League\Flysystem\Dropbox\DropboxAdapter;
use Illuminate\Support\ServiceProvider;

class DropboxFilesystemServiceProvider extends ServiceProvider {

    public function boot()
    {
        Storage::extend('dropbox', function($app, $config)
```

```
        {
            $client = new DropboxClient($config['accessToken'],
$config['clientIdentifier']);

            return new Filesystem(new DropboxAdapter($client));
        });
    }

    public function register()
    {
        //
    }

}
```

https://laravel-docs-pt-br.readthedocs.io/en/latest/filesystem/