VERSION
9.x

Search

File Storage

# Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3. Even better, it's amazingly simple to switch between these storage options between your local development machine and production server as the API remains the same for each system.

# Configuration

Laravel's filesystem configuration file is located at `config/filesystems.php`. Within this file, you may configure all of your filesystem "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file so you can modify the configuration to reflect your storage preferences and credentials.

The `local` driver interacts with files stored locally on the server running the Laravel application while the `s3` driver is used to write to Amazon's S3 cloud storage service.

You may configure as many disks as you like and may even have multiple disks that use the same driver.

# The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory defined in your `filesystems` configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would write to `storage/app/example.txt`:

```
use Illuminate\Support\Facades\Storage;

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

# The Public Disk

The `public` disk included in your application's `filesystems` configuration file is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores its files in `storage/app/public`.

To make these files accessible from the web, you should create a symbolic link from `public/storage` to `storage/app/public`. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer.

To create the symbolic link, you may use the `storage:link` Artisan command:

```
php artisan storage:link
```

Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the `asset` helper:

```
echo asset('storage/file.txt');
```

You may configure additional symbolic links in your `filesystems` configuration file. Each of the configured links will be created when you run the `storage:link` command:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

# Driver Prerequisites

# S3 Driver Configuration

Before using the S3 driver, you will need to install the Flysystem S3 package via the Composer package manager:

```
composer require league/flysystem-aws-s3-v3 "^3.0"
```

The S3 driver configuration information is located in your `config/filesystems.php` configuration file. This file contains an example configuration array for an S3 driver. You are free to modify this array with your own S3 configuration and credentials. For convenience, these environment variables match the naming convention used by the AWS CLI.

# FTP Driver Configuration

Before using the FTP driver, you will need to install the Flysystem FTP package via the Composer package manager:

```
composer require league/flysystem-ftp "^3.0"
```

Laravel's Flysystem integrations work great with FTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an FTP filesystem, you may use the configuration example below:

```
'ftp' => [
    'driver' => 'ftp',
    'host' => env('FTP_HOST'),
    'username' => env('FTP_USERNAME'),
    'password' => env('FTP_PASSWORD'),

    // Optional FTP Settings...
    // 'port' => env('FTP_PORT', 21),
    // 'root' => env('FTP_ROOT'),
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

# SFTP Driver Configuration

Before using the SFTP driver, you will need to install the Flysystem SFTP package via the Composer package manager:

```
composer require league/flysystem-sftp-v3 "^3.0"
```

Laravel's Flysystem integrations work great with SFTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an SFTP filesystem, you may use the configuration example below:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => env('SFTP_HOST'),

    // Settings for basic authentication...
    'username' => env('SFTP_USERNAME'),
    'password' => env('SFTP_PASSWORD'),

    // Settings for SSH key based authentication with encryption password..
    'privateKey' => env('SFTP_PRIVATE_KEY'),
    'password' => env('SFTP_PASSWORD'),

    // Optional SFTP Settings...
    // 'hostFingerprint' => env('SFTP_HOST_FINGERPRINT'),
    // 'maxTries' => 4,
    // 'passphrase' => env('SFTP_PASSPHRASE'),
    // 'port' => env('SFTP_PORT', 22),
    // 'root' => env('SFTP_ROOT', ''),
    // 'timeout' => 30,
    // 'useAgent' => true,
],
```

# Amazon S3 Compatible Filesystems

By default, your application's `filesystems` configuration file contains a disk configuration for the `s3` disk. In addition to using this disk to interact with Amazon S3, you may use it to interact with any S3 compatible file storage service such as MinIO or DigitalOcean Spaces.

Typically, after updating the disk's credentials to match the credentials of the service you are planning to use, you only need to update the value of the `url` configuration option. This option's value is typically defined via the `AWS_ENDPOINT` environment variable:

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

# Obtaining Disk Instances

The `Storage` facade may be used to interact with any of your configured disks. For example, you may use the `put` method on the facade to store an avatar on the default disk. If you call methods on the `Storage` facade without first calling the `disk` method, the method will automatically be passed to the default disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $content);
```

If your application interacts with multiple disks, you may use the `disk` method on the `Storage` facade to work with files on a particular disk:

```
Storage::disk('s3')->put('avatars/1', $content);
```

# On-Demand Disks

Sometimes you may wish to create a disk at runtime using a given configuration without that configuration actually being present in your application's `filesystems` configuration file. To accomplish this, you may pass a configuration array to the `Storage` facade's `build` method:

```
use Illuminate\Support\Facades\Storage;

$disk = Storage::build([
    'driver' => 'local',
    'root' => '/path/to/root',
]);

$disk->put('image.jpg', $content);
```

# Retrieving Files

The `get` method may be used to retrieve the contents of a file. The raw string contents of the file will be returned by the method. Remember, all file paths should be specified relative to the disk's "root" location:

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

The `exists` method may be used to determine if a file exists on the disk:

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

The `missing` method may be used to determine if a file is missing from the disk:

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

# Downloading Files

The `download` method may be used to generate a response that forces the user's browser to download the file at the given path. The `download` method accepts a filename as the second argument to the method, which will determine the filename that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

# File URLs

You may use the `url` method to get the URL for a given file. If you are using the `local` driver, this will typically just prepend `/storage` to the given path and return a relative URL to the file. If you are using the `s3` driver, the fully qualified remote URL will be returned:

```php
use Illuminate\Support\Facades\Storage;

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

When using the `local` driver, all files that should be publicly accessible should be placed in the `storage/app/public` directory. Furthermore, you should create a symbolic link at `public/storage` which points to the `storage/app/public` directory.

When using the `local` driver, the return value of `url` is not URL encoded. For this reason, we recommend always storing your files using names that will create valid URLs.

# Temporary URLs

Using the `temporaryUrl` method, you may create temporary URLs to files stored using the `s3` driver. This method accepts a path and a `DateTime` instance specifying when the URL should expire:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

If you need to specify additional S3 request parameters, you may pass the array of request parameters as the third argument to the `temporaryUrl` method:

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment; filename=file2.jpg',
    ]
);
```

If you need to customize how temporary URLs are created for a specific storage disk, you can use the `buildTemporaryUrlsUsing` method. For example, this can be useful if you have a controller that allows you to download files stored via a disk that doesn't typically support temporary URLs. Usually, this method should be called from the `boot` method of a service provider:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\URL;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::disk('local')->buildTemporaryUrlsUsing(function ($path, $e
            return URL::temporarySignedRoute(
                'files.download',
                $expiration,
                array_merge($options, ['path' => $path])
            );
        });
    }
}
```

URL Host Customization

If you would like to pre-define the host for URLs generated using the `Storage` facade, you may add a `url` option to the disk's configuration array:

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

# File Metadata

In addition to reading and writing files, Laravel can also provide information about the files themselves. For example, the `size` method may be used to get the size of a file in bytes:

```
use Illuminate\Support\Facades\Storage;

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

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

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

# File Paths

You may use the `path` method to get the path for a given file. If you are using the `local` driver, this will return the absolute path to the file. If you are using the `s3` driver, this method will return the relative path to the file in the S3 bucket:

```
use Illuminate\Support\Facades\Storage;

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

# Storing Files

The `put` method may be used to store file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Remember, all file paths should be specified relative to the "root" location configured for the disk:

```
use Illuminate\Support\Facades\Storage;

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

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

# Failed Writes

If the `put` method (or other "write" operations) is unable to write the file to disk, `false` will be returned:

```
if (! Storage::put('file.jpg', $contents)) {
    // The file could not be written to disk...
}
```

If you wish, you may define the `throw` option within your filesystem disk's configuration array. When this option is defined as `true`, "write" methods such as `put` will throw an instance of `League\Flysystem\UnableToWriteFile` when write operations fail:

```
'public' => [
    'driver' => 'local',
    // ...
    'throw' => true,
],
```

# Prepending & Appending To Files

The `prepend` and `append` methods allow you to write to the beginning or end of a file:

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

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

# Copying & Moving Files

The `copy` method may be used to copy an existing file to a new location on the disk, while the `move` method may be used to rename or move an existing file to a new location:

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

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

# Automatic Streaming

Streaming files to storage offers significantly reduced memory usage. If you would like Laravel to automatically manage streaming a given file to your storage location, you may use the `putFile` or `putFileAs` method. This method accepts either an `Illuminate\Http\File` or `Illuminate\Http\UploadedFile` instance and will automatically stream the file to your desired location:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;
```

```
// Automatically generate a unique ID for filename...
$path = Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a filename...
$path = Storage::putFileAs('photos', new File('/path/to/photo'), 'photo.jpg
```

There are a few important things to note about the `putFile` method. Note that we only specified a directory name and not a filename. By default, the `putFile` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `putFile` method so you can store the path, including the generated filename, in your database.

The `putFile` and `putFileAs` methods also accept an argument to specify the "visibility" of the stored file. This is particularly useful if you are storing the file on a cloud disk such as Amazon S3 and would like the file to be publicly accessible via generated URLs:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

# File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as photos and documents. Laravel makes it very easy to store uploaded files using the `store` method on an uploaded file instance. Call the `store` method with the path at which you wish to store the uploaded file:

```
<?php
```

```php
namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

There are a few important things to note about this example. Note that we only specified a directory name, not a filename. By default, the `store` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `store` method so you can store the path, including the generated filename, in your database.

You may also call the `putFile` method on the `Storage` facade to perform the same file storage operation as the example above:

```php
$path = Storage::putFile('avatars', $request->file('avatar'));
```

# Specifying A File Name

If you do not want a filename to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the filename, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

You may also use the `putFileAs` method on the `Storage` facade, which will perform the same file storage operation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

Unprintable and invalid unicode characters will automatically be removed from file paths. Therefore, you may wish to sanitize your file paths before passing them to Laravel's file storage methods. File paths are normalized using the `League\Flysystem\WhitespacePathNormalizer::normalizePath` method.

# Specifying A Disk

By default, this uploaded file's `store` method will use your default disk. If you would like to specify another disk, pass the disk name as the second argument to the `store` method:

```
$path = $request->file('avatar')->store(
    'avatars/'.$request->user()->id, 's3'
);
```

If you are using the `storeAs` method, you may pass the disk name as the third argument to the method:

```
$path = $request->file('avatar')->storeAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

# Other Uploaded File Information

If you would like to get the original name and extension of the uploaded file, you may do so using the `getClientOriginalName` and `getClientOriginalExtension` methods:

```
$file = $request->file('avatar');

$name = $file->getClientOriginalName();
$extension = $file->getClientOriginalExtension();
```

However, keep in mind that the `getClientOriginalName` and `getClientOriginalExtension` methods are considered unsafe, as the file name and extension may be tampered with by a malicious user. For this reason, you should typically prefer the `hashName` and `extension` methods to get a name and an extension for the given file upload:

```
$file = $request->file('avatar');

$name = $file->hashName(); // Generate a unique, random name...
$extension = $file->extension(); // Determine the file's extension based on
```

# File Visibility

In Laravel's Flysystem integration, "visibility" is an abstraction of file permissions across multiple platforms. Files may either be declared `public` or `private`. When a file is declared `public`, you are indicating that the file should generally be accessible to others. For example, when using the S3 driver, you may retrieve URLs for `public` files.

You can set the visibility when writing the file via the `put` method:

```
use Illuminate\Support\Facades\Storage;

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

If the file has already been stored, its visibility can be retrieved and set via the `getVisibility` and `setVisibility` methods:

```
$visibility = Storage::getVisibility('file.jpg');
```

```
Storage::setVisibility('file.jpg', 'public');
```

When interacting with uploaded files, you may use the `storePublicly` and `storePubliclyAs` methods to store the uploaded file with `public` visibility:

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

# Local Files & Visibility

When using the `local` driver, `public` visibility translates to `0755` permissions for directories and `0644` permissions for files. You can modify the permissions mappings in your application's `filesystems` configuration file:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0644,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0755,
            'private' => 0700,
        ],
    ],
],
```

Deleting Files

The `delete` method accepts a single filename or an array of files to delete:

```
use Illuminate\Support\Facades\Storage;

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

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

If necessary, you may specify the disk that the file should be deleted from:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('path/file.jpg');
```

Directories

Get All Files Within A Directory

The `files` method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all subdirectories, you may use the `allFiles` method:

```
use Illuminate\Support\Facades\Storage;
```

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

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

# Get All Directories Within A Directory

The `directories` method returns an array of all the directories within a given directory. Additionally, you may use the `allDirectories` method to get a list of all directories within a given directory and all of its subdirectories:

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

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

# Create A Directory

The `makeDirectory` method will create the given directory, including any needed subdirectories:

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

# Delete A Directory

Finally, the `deleteDirectory` method may be used to remove a directory and all of its files:

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

# Custom Filesystems

Laravel's Flysystem integration provides support for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application.

In order to define a custom filesystem you will need a Flysystem adapter. Let's add a community maintained Dropbox adapter to our project:

```
composer require spatie/flysystem-dropbox
```

Next, you can register the driver within the `boot` method of one of your application's service providers. To accomplish this, you should use the `extend` method of the `Storage` facade:

```
<?php

namespace App\Providers;

use Illuminate\Filesystem\FilesystemAdapter;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
```

```
         * @return void
         */
        public function register()
        {
            //
        }

        /**
         * Bootstrap any application services.
         *
         * @return void
         */
        public function boot()
        {
            Storage::extend('dropbox', function ($app, $config) {
                $adapter = new DropboxAdapter(new DropboxClient(
                    $config['authorization_token']
                ));

                return new FilesystemAdapter(
                    new Filesystem($adapter, $config),
                    $adapter,
                    $config
                );
            });
        }
    }
```

The first argument of the `extend` method is the name of the driver and the second is a closure that receives the `$app` and `$config` variables. The closure must return an instance of `Illuminate\Filesystem\FilesystemAdapter`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Once you have created and registered the extension's service provider, you may use the `dropbox` driver in your `config/filesystems.php` configuration file.

64 Robots	Echo
Kirschbaum	Envoyer
Curotec	Forge
Jump24	Homestead
A2 Design	Horizon
ABOUT YOU	Mix
Byte 5	Nova
Cubet	Passport
Cyber-Duck	Scout
DevSquad	Socialite
Ideil	Spark
Romega Software	Telescope
Worksome	Valet
WebReinvent	Vapor