Prologue
Getting Started
Architecture Concepts
The Basics
Frontend
Security
Digging Deeper
Artisan Console
Broadcasting
Cache
Collections
Events
File Storage
Helpers
Mail
Notifications
Package Development
Queues
Task Scheduling
Database
Eloquent ORM
Testing
Official Packages

File Storage

 # Introduction
 # Configuration
The Public Disk
The Local Driver
Driver Prerequisites
Caching
 # Obtaining Disk Instances
 # Retrieving Files
Downloading Files
File URLs
File Metadata
 # Storing Files
File Uploads
File Visibility
 # Deleting Files
 # Directories
 # Custom Filesystems

Introduction

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

Configuration

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

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

The Public Disk

The `public` disk is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores these files in `storage/app/public`. To make them accessible from the web, you should create a symbolic link from `public/storage` to `storage/app/public`. This 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');
```

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 store a file in `storage/app/file.txt`:

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

Permissions

The `public` visibility translates to `0755` for directories and `0644` for files. You can modify the permissions mappings in your `filesystems` configuration file:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0664,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0775,
            'private' => 0700,
        ],
    ],
],
```

Driver Prerequisites

Composer Packages

Before using the SFTP or S3 drivers, you will need to install the appropriate package via Composer:

- SFTP: `league/flysystem-sftp ~1.0`
- Amazon S3: `league/flysystem-aws-s3-v3 ~1.0`

An absolute must for performance is to use a cached adapter. You will need an additional package for this:

- CachedAdapter: `league/flysystem-cached-adapter ~1.0`

S3 Driver Configuration

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

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

```
'ftp' => [
    'driver' => 'ftp',
    'host' => 'ftp.example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Optional FTP Settings...
    // 'port' => 21,
    // 'root' => '',
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

SFTP Driver Configuration

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

```
'sftp' => [
    'driver' => 'sftp',
    'host' => 'example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Settings for SSH key based authentication...
    // 'privateKey' => '/path/to/privateKey',
    // 'password' => 'encryption-password',

    // Optional SFTP Settings...
    // 'port' => 22,
    // 'root' => '',
    // 'timeout' => 30,
],
```

Caching

To enable caching for a given disk, you may add a `cache` directive to the disk's configuration options. The `cache` option should be an array of caching options containing the `disk` name, the `expire` time in seconds, and the cache `prefix`:

```
's3' => [
    'driver' => 's3',

    // Other Disk Options...

    'cache' => [
        'store' => 'memcached',
        'expire' => 600,
        'prefix' => 'cache-prefix',
    ],
],
```

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 call will automatically be passed to the default disk:

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

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

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', $fileContents);
```

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 "root" location configured for the disk:

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

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

```
$exists = Storage::disk('s3')->exists('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 file name as the second argument to the method, which will determine the file name 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 the 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:

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

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

Remember, if you are 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.

Temporary URLs

For files stored using the `s3` you may create a temporary URL to a given file using the `temporaryUrl` method. This methods accepts a path and a `DateTime` instance specifying when the URL should expire:

```
$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']
);
```

Local URL Host Customization

If you would like to pre-define the host for files stored on a disk using the `local` driver, 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 the 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');
```

Storing Files

The `put` method may be used to store raw file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Using streams is greatly recommended when dealing with large files:

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

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

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

Automatic Streaming

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 a `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 file name...
Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a file name...
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, not a file name. By default, the `putFile` method will

generate a unique ID to serve as the file name. 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 file name, 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 S3 and would like the file to be publicly accessible:

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

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');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as profile pictures, 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

namespace App\Http\Controllers;

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

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param  Request  $request
     * @return 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 file name. By default, the `store` method will generate a unique ID to serve as the file name. 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 file name, in your database.

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

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

Specifying A File Name

If you would not like a file name to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the file name, 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 manipulation 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\Util::normalizePath` method.

Specifying A Disk

By default, this 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'
);
```

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 setting 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')
```

Deleting Files

The `delete` method accepts a single filename or an array of files to remove from the disk:

```
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('folder_path/file_name.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 sub-directories, 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 sub-directories:

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

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

Create A Directory

The `makeDirectory` method will create the given directory, including any needed sub-directories:

```
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 drivers 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 set up the 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 should create a service provider such as `DropboxServiceProvider`. In the provider's `boot` method, you may use the `Storage` facade's `extend` method to define the custom driver:

```
<?php

namespace App\Providers;

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

class DropboxServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }

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

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

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 resolver Closure must return an instance of `League\Flysystem\Filesystem`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Next, register the service provider in your `config/app.php` configuration file:

```
'providers' => [
    // ...
    App\Providers\DropboxServiceProvider::class,
];
```

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

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.