Laravel-Mediable Documentation

Release 5.0.0

Sean Fraser <sean@plankdesign.com>

Sep 12, 2023

Getting Started

Laravel-Mediable is a package for easily uploading and attaching media files to models with Laravel 5.

CHAPTER 1

Features

- Filesystem-driven approach is easily configurable to allow any number of upload directories with different accessibility. Easily restrict uploads by MIME type, extension and/or aggregate type (e.g. `image` for JPEG, PNG or GIF).
- Many-to-many polymorphic relationships allow any number of media to be assigned to any number of other models without any need to modify their schema.
- Attach media to models with tags, in order to set and retrieve media for specific purposes, such as `'thumbnail'`, `'featured image'`, `'gallery'` or `'download'`.
- Integrated support for integration/image for manipulating image files to create variants for different use cases.

1.1 Installation

Add the package to your Laravel app using composer.

```
$ composer require plank/laravel-mediable
```

Register the package's service provider in *config/app.php*. In Laravel versions 5.5 and beyond, this step can be skipped if package auto-discovery is enabled.

```
'providers' => [
    //...
    Plank\Mediable\MediableServiceProvider::class,
    //...
];
```

The package comes with a Facade for the image uploader, which you can optionally register as well. In Laravel versions 5.5 and beyond, this step can be skipped if package auto-discovery is enabled.

```
'aliases' => [
    //...
    'MediaUploader' => Plank\Mediable\Facades\MediaUploader::class,
```

(continues on next page)

(continued from previous page)

```
    //...
]
```

Publish the config file (`config/mediable.php`) and migration file (`database/migrations/####_##_##_######_create_mediable_tables.php`) of the package using artisan.

```
$ php artisan vendor:publish --provider="Plank\Mediable\MediableServiceProvider"
```

Run the migrations to add the required tables to your database.

```
$ php artisan migrate
```

1.2 Configuration

1.2.1 Disks

Laravel-Mediable is built on top of Laravel's Filesystem component. Before you use the package, you will need to configure the filesystem disks where you would like files to be stored in `config/filesystems.php`. Learn more about filesystem disk.

```
<?php
//...
'disks' => [
    'local' => [
        'driver' => 'local',
        'root'   => storage_path('app'),
        'url' => 'https://example.com/storage/app',
        'visibility' => 'public'
    ],

    'uploads' => [
        'driver' => 'local',
        'root'   => public_path('uploads'),
        'url' => 'https://example.com/uploads',
        'visibility' => 'public'
    ],
]
//...
```

Once you have set up as many disks as you need, edit `config/mediable.php` to authorize the package to use the disks you have created.

```
<?php
//...
/*
 * Filesystem disk to use if none is specified
 */
'default_disk' => 'uploads',

/*
 * Filesystems that can be used for media storage
 */
'allowed_disks' => [
```

(continues on next page)

(continued from previous page)

```
    'local',
    'uploads',
],
//...
```

1.2.2 Validation

The *config/mediable.php* offers a number of options for configuring how media uploads are validated. These values serve as defaults, which can be overridden on a case-by-case basis for each `MediaUploader` instance.

```
<?php
//...
/*
 * The maximum file size in bytes for a single uploaded file
 */
'max_size' => 1024 * 1024 * 10,

/*
 * What to do if a duplicate file is uploaded. Options include:
 *
 * * 'increment': the new file's name is given an incrementing suffix
 * * 'replace' : the old file and media model is deleted
 * * 'error': an Exception is thrown
 *
 */
'on_duplicate' => Plank\Mediable\MediaUploader::ON_DUPLICATE_INCREMENT,

/*
 * Reject files unless both their mime and extension are recognized and both match a␣
↪single aggregate type
 */
'strict_type_checking' => false,

/*
 * Reject files whose mime type or extension is not recognized
 * if true, files will be given a type of ``'other'``
 */
'allow_unrecognized_types' => false,

/*
 * Only allow files with specific MIME type(s) to be uploaded
 */
'allowed_mime_types' => [],

/*
 * Only allow files with specific file extension(s) to be uploaded
 */
'allowed_extensions' => [],

/*
 * Only allow files matching specific aggregate type(s) to be uploaded
 */
'allowed_aggregate_types' => [],
//...
```

1.2.3 Aggregate Types

Laravel-Mediable provides functionality for handling multiple kinds of files under a shared aggregate type. This is intended to make it easy to find similar media without needing to constantly juggle multiple MIME types or file extensions.

The package defines a number of common file types in the config file (`config/mediable.php`). Feel free to modify the default types provided by the package or add your own. Each aggregate type requires a key used to identify the type and a list of MIME types and file extensions that should be recognized as belonging to that aggregate type. For example, if you wanted to add an aggregate type for different types of markup, you could do the following.

```
<?php
//...
'aggregate_types' => [
    //...
    'markup' => [
        'mime_types' => [
            'text/markdown',
            'text/html',
            'text/xml',
            'application/xml',
            'application/xhtml+xml',
        ],
        'extensions' => [
            'md',
            'html',
            'htm',
            'xhtml',
            'xml'
        ]
    ],
    //...
]
//...
```

Note: a MIME type or extension could be present in more than one aggregate type's definitions (the system will try to find the best match), but each Media record can only have one aggregate type.

1.2.4 Extending functionality

The `config/mediable.php` file lets you specify a number of classes to be use for internal behaviour. This is to allow for extending some of the the default classes used by the package or to cover additional use cases.

```
<?php
//...
/*
 * FQCN of the model to use for media
 *
 * Should extend Plank\Mediable\Media::class
 */
'model' => Plank\Mediable\Media::class,

/*
 * List of adapters to use for various source inputs
 *
 * Adapters can map either to a class or a pattern (regex)
```

(continues on next page)

(continued from previous page)

```
 */
'source_adapters' => [
    'class' => [
        Symfony\Component\HttpFoundation\File\UploadedFile::class =>␣
↪Plank\Mediable\SourceAdapters\UploadedFileAdapter::class,
        Symfony\Component\HttpFoundation\File\File::class =>␣
↪Plank\Mediable\SourceAdapters\FileAdapter::class,
        Psr\Http\Message\StreamInterface::class =>␣
↪Plank\Mediable\SourceAdapters\StreamAdapter::class,
    ],
    'pattern' => [
        '^https?://' => Plank\Mediable\SourceAdapters\RemoteUrlAdapter::class,
        '^/' => Plank\Mediable\SourceAdapters\LocalPathAdapter::class
    ],
],

/*
 * List of URL Generators to use for handling various filesystem disks
 */
'url_generators' => [
    'local' => Plank\Mediable\UrlGenerators\LocalUrlGenerator::class,
    's3' => Plank\Mediable\UrlGenerators\S3UrlGenerator::class,
],
```

It is also possible to define the connection name that Laravel-Mediable will use to access the database.

```
<?php
//...
/*
 * The database connection name to use
 *
 * Set to null in order to use the default database connection
 */
'connection_name' => null,
//...
```

1.3 Uploading Files

The easiest way to upload media to your server is with the `MediaUploader` class, which handles validating the file, moving it to its destination and creating a `Media` record to reference it. You can get an instance of the MediaUploader using the Facade and configure it with a fluent interface.

To upload a file to the root of the default disk (set in `config/mediable.php`), all you need to do is the following:

```
<?php
use MediaUploader; //use the facade
$media = MediaUploader::fromSource($request->file('thumbnail'))->upload();
```

1.3.1 Source Files

The `fromSource()` method will accept any of the following:

- an instance of `Symfony\Component\HttpFoundation\UploadedFile`, which is returned by `$request->file()`.

- an instance of `Symfony\Component\HttpFoundation\File`.
- an instance of `Psr\Http\Message\StreamInterface`, which is returned by libraries using PSR-7 HTTP message interfaces, like Guzzle.
- a stream resource handle.
- a URL as a string, beginning with `http://` or `https://`.
- an absolute path as a string, beginning with `/`.

1.3.2 Specifying Destination

By default, the uploader will place the file in the root of the default disk specified in `config/mediable.php`. You can customize where the uploader will put the file on your server before you invoke the `upload()` method.

```
<?php
$uploader = MediaUploader::fromSource($request->file('thumbnail'))

// specify a disk to use instead of the default
->toDisk('s3');

// place the file in a directory relative to the disk root
->toDirectory('user/john/profile')

// alternatively, specify both the disk and directory at once
->toDestination('s3', 'user/john/profile')

->upload();
```

1.3.3 Specifying Filename

By default, the uploader will copy the source file while maintaining its original filename. You can override this behaviour by providing a custom filename.

```
<?php
MediaUploader::fromSource(...)
    ->useFilename('profile')
    ->upload();
```

You can also tell the uploader to generate a filename based on the MD5 hash of the file's contents.

```
<?php
MediaUploader::fromSource(...)
    ->useHashForFilename()
    ->upload();
```

You can restore the default behaviour with `useOriginalFilename()`.

1.3.4 Handling Duplicates

Occasionally, a file with a matching name might already exist at the destination you would like to upload to. The uploader allows you to configure how it should respond to this scenario. There are three possible behaviours:

```php
<?php

// keep both, append incrementing counter to new file name
$uploader->onDuplicateIncrement();

// replace old file with new one, update existing Media record, maintain associations
$uploader->onDuplicateUpdate();

// replace old file and media record with new ones, break associations
$uploader->onDuplicateReplace();

// replace old file and media record with new ones, break associations
// will also delete any existing variants of the replaced media record
$uploader->onDuplicateReplaceWithVariants();

// cancel upload, throw an exception
$uploader->onDuplicateError();
```

1.3.5 Validation

The `MediaUpload` will perform a number of validation checks on the source file. If any of the checks fail, a `Plank\Mediable\MediaUploadException` will be thrown with a message indicating why the file was rejected.

You can override the most validation configuration values set in `config/mediable.php` on a case-by-case basis using the same fluent interface.

```php
<?php
$media = MediaUploader::fromSource($request->file('image'))

    // model class to use
    ->setModelClass(MediaSubclass::class)

    // maximum filesize in bytes
    ->setMaximumSize(99999)

    // whether the aggregate type must match both the MIME type and extension
    ->setStrictTypeChecking(true)

    // whether to allow the 'other' aggregate type
    ->setAllowUnrecognizedTypes(true)

    // only allow files of specific MIME types
    ->setAllowedMimeTypes(['image/jpeg'])

    // only allow files of specific extensions
    ->setAllowedExtensions(['jpg', 'jpeg'])

    // only allow files of specific aggregate types
    ->setAllowedAggregateTypes(['image'])

    ->upload();
```

You can also validate the file without uploading it by calling the `verifyFile` method. If the file does not pass validation, an instance of `Plank\Mediable\MediaUploadException` will be thrown

```php
<?php
$media = MediaUploader::fromSource($request->file('image'))

    // model class to use
    ->setModelClass(MediaSubclass::class)

    // maximum filesize in bytes
    ->setMaximumSize(99999)

    // only allow files of specific MIME types
    ->setAllowedMimeTypes(['image/jpeg'])

    ->verifyFile()
```

1.3.6 Alter Model before upload

You can manipulate the model before it's saved by passing a callable to the `beforeSave` method. The callback takes two params, `$model` an instance of `Plank\Mediable\Media` the current model and `$source` an instance of `Plank\Mediable\SourceAdapters\SourceAdapterInterface` the current source.

```php
<?php
$media = MediaUploader::fromSource($request->file('image'))

    // model class to use
    ->setModelClass(CustomMediaClass::class)

    // pass the callable
    ->beforeSave(function (Media $model, SourceAdapterInterface $source) {
        $model->setAttribute('customAttribute', 'value')
    })

    ->upload()
```

1.3.7 Visibility

In addition to setting visibility on Disks as a whole, you can also specify whether a file should be publicly viewable on a file by file basic

```php
<?php
MediaUploader::fromSource($request->file('image'))
    ->makePrivate() // Disable public access
    ->makePublic() // Default behaviour
    ->upload()
```

1.3.8 Options

You can also specify additional option flags to be passed to the underlying filesystem adapter. This is particularly useful when dealing with cloud storage such as S3.

```php
<?php
MediaUploader::fromSource($request->file('image'))
```

(continues on next page)

(continued from previous page)

```
    ->withOptions(['Cache-Control' => 'max-age=3600'])
    ->upload();
```

1.3.9 Handling Exceptions

If you want to return more granular HTTP status codes when a `Plank\Mediable\MediaUploadException` is thrown, you can use the `Plank\Mediable\HandlesMediaUploadExceptions` trait in your app's *ExceptionsHandler* or in your controller. For example, if you have set a maximum file size, an 413 HTTP response code (Request Entity Too Large) will be returned instead of a 500.

Call the `transformMediaUploadException` method as part of the `render` method of the exception handler, and a `HttpException` with the appropriate status code will be returned. Take a look at the `HandlesMediaExceptions` source code for the table of associated status codes and exceptions.

```php
<?php

namespace App\Exceptions;

use Plank\Mediable\HandlesMediaUploadExceptions;

class Handler
{
    use HandlesMediaUploadExceptions;

    public function render($request, $e)
    {
        $e = $this->transformMediaUploadException($e);

        return parent::render($request, $e);
    }
}
```

If you only want some actions to throw an `HttpException`, you can apply the trait to the controller instead.

```php
<?php

class ExampleController extends Controller
{
    use HandlesMediaUploadExceptions;

    public function upload(Request $request)
    {
        try{
            MediaUploader::fromSource($request->file('file'))
                ->toDestination(...)
                ->upload();
        }catch(MediaUploadException $e){
            throw $this->transformMediaUploadException($e);
        }
    }
}
```

1.3.10 Importing Files

If you need to create a media record for a file that is already in place on the desired filesystem disk, you can use one the import methods instead.

```
<?php
$media = MediaUploader::import($disk, $directory, $filename, $extension);
// or
$media = MediaUploader::importPath($disk, $path);
```

If you have string file data, you can import it using the *fromString* method.

```
<?php
// Encoded image converted to string
$jpg = Image::make('https://www.plankdesign.com/externaluse/plank.png')->encode('jpg
↪');

MediaUploader::fromString($jpg)
    ->toDestination(...)
    ->upload();
```

1.3.11 Replacing Files

If you need to swap out the file belonging to a `Media` record, you can use the `replace()` method. This will upload the file and update the existing record while maintaining any attachments to other models.

```
<?php
$media = Media::find($id);

MediaUploader::fromSource($source)
    ->replace($media);
```

1.3.12 Updating Files

If a file has changed on disk, you can re-evaluate its attributes with the `update()` method. This will reassign the media record's `mime_type`, `aggregate_type` and `size` attributes and will save the changes to the database, if any.

```
<?php
MediaUploader::update($media);
```

1.4 Handling Media

Add the `Mediable` trait to any Eloquent models that you would like to be able to attach media to.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Plank\Mediable\Mediable;
```

(continues on next page)

(continued from previous page)

```
class Post extends Model
{
    use Mediable;

    // ...
}
```

1.4.1 Attaching Media

You can attach media to your `Mediable` model using the `attachMedia()` method. This method takes a second argument, specifying one or more tags which define the relationship between the model and the media. Tags are simply strings; you can use any value you need to describe how the model should use its attached media.

```
<?php
$post = Post::first();
$post->attachMedia($media, 'thumbnail');
```

You can attach multiple media to the same tag with a single call. The `attachMedia()` method accept any of the following for its first parameter:

- a numeric or string id
- an instance of `\Plank\Mediable\Media`
- an array of ids
- an instance of `\Illuminate\Database\Eloquent\Collection`

```
<?php
$post->attachMedia([$media1->getKey(), $media2->getKey()], 'gallery');
```

You can also assign media to multiple tags with a single call.

```
<?php
$post->attachMedia($media, ['gallery', 'featured']);
```

1.4.2 Replacing Media

`Media` and `Mediable` models share a many-to-many relationship, which allows for any number of media to be added to any key. The `attachMedia()` method will add a new association, but will not remove any existing associations to other media. If you want to replace the media previously attached to the specified tag(s) you can use the `syncMedia()` method. This method accepts the same inputs as `attachMedia()`.

```
<?php
$post->syncMedia($media, 'thumbnail');
```

1.4.3 Retrieving Media

You can retrieve media attached to a file by referring to the tag to which it was previously assigned.

```
<?php
$media = $post->getMedia('thumbnail');
```

This returns a collection of all media assigned to that tag. In cases where you only need one `Media` entity, you can instead use `firstMedia()` or `lastMedia()`.

```
<?php
$media = $post->firstMedia('thumbnail');
// shorthand for
$media = $post->getMedia('thumbnail')->first();
```

If you specify an array of tags, the method will return media is attached to any of those tags. Set the `$match_all` parameter to `true` to tell the method to only return media that are attached to all of the specified tags.

```
<?php
$post->getMedia(['header', 'footer']); // get media with either tag
$post->getMedia(['header', 'footer'], true); //get media with both tags
$post->getMediaMatchAll(['header', 'footer']); //alias
```

You can also get all media attached to a model, grouped by tag.

```
<?php
$post->getAllMediaByTag();
```

1.4.4 Media Order

The system keeps track of the order in which `Media` are attached to each `Mediable` model's tags and are always returned in the same order.

To change the order of media assigned to a given tag, or to insert a new item at a particular index manipulate the eloquent collection then use the `syncMedia()` method to commit the changes.

```
<?php
$media = $post->getMedia('gallery');
$media = $media->prepend($new_media);
$post->syncMedia($media, 'gallery');
```

An `ORDER BY` clause is automatically applied to all queries run on the `media()` relationship. To disable this default behaviour, use the `unordered()` query scope.

```
<?php
$mediable->media()
    ->unordered()
    ->...
```

1.4.5 Checking for the Presence of Media

You can verify if a model has one or more media assigned to a given tag with the `hasMedia()` method.

```
<?php
if($post->hasMedia('thumbnail')){
    // ...
}
```

You can specify multiple tags when calling either method, which functions similarly to `getMedia()`. The method will return `true` if `getMedia()` passed the same parameters would return any instances.

You also can also perform this check using the query builder.

```php
<?php
$posts = Post::whereHasMedia('thumbnail')->get();
```

1.4.6 Detaching Media

You can remove a media record from a model with the `detachMedia()` method.

```php
<?php
$post->detachMedia($media); // remove media from all tags
$post->detachMedia($media, 'feature'); //remove media from specific tag
$post->detachMedia($media, ['feature', 'thumbnail']); //remove media from multiple␣
↪tags
```

You can also remove all media assigned to one or more tags.

```php
<?php
$post->detachMediaTags('feature');
$post->detachMediaTags(['feature', 'thumbnail']);
```

1.4.7 Loading Media

When dealing with any model relationships, taking care to avoid running into the "N+1 problem" is an important optimization consideration. The N+1 problem can be summed up as a separate query being run for the related content of each record of the parent model. Consider the following example:

```php
<?php
$posts = Post::limit(10)->get();
foreach($posts as $post){
    echo $post->firstMedia('thumbnail')->getUrl();
}
```

Assuming there are at least 10 Post records available, this code will execute 11 queries: one query to load the 10 posts from the database, then another 10 queries to load the media for each of the post records individually. This will slow down the rendering of the page.

There are a couple of approaches that can be taken to preload the attached media in order to avoid this issue.

Eager Loading

The Eloquent query builder's `with()` method is the preferred way to eager load related models. This package also provides an alias.

```php
<?php
$posts = Post::with('media')->get();
// or
$posts = Post::withMedia()->get();
```

You can also load only media attached to specific tags.

```php
<?php
$posts = Post::withMedia(['thumbnail', 'featured'])->get(); // attached to either tags
$posts = Post::withMediaMatchAll(['thumbnail', 'featured'])->get(); // attached to␣
↪both tags
```

Note if using this approach to conditionally preload media by tag, you will not be able to access media with other tags using `getMedia()` without first reloading the media relationship on that record.

If you are using variants, they can also be eager loaded at the same time

```
<?php
Post::withMediaAndVariants($tags)->get();
Post::withMediaAndVariantsMatchAll($tags)->get();
```

Lazy Eager Loading

If you have already loaded models from the database, you can still load relationships with the `load()` method of the Eloquent Collection class. The package also provides an alias.

```
<?php
$posts = Post::all();
// ...

$posts->load('media');
// or
$posts->loadMedia();
```

You can also load only media attached to specific tags.

```
<?php
$posts->loadMedia(['thumbnail', 'featured']); // attached to either tag
$posts->loadMediaMatchAll(['thumbnail', 'featured']); // attached to both tags
```

The same method is available as part of the `Mediable` trait, and can be used directly on a model instance.

```
<?php
$post = Post::first();
$post->loadMedia();
$post->loadMedia(['thumbnail', 'featured']); // attached to either tag
$post->loadMediaMatchAll(['thumbnail', 'featured']); // attached to both tags
```

Any of these methods can be used to reload the media relationship of the model.

Note if using this approach to conditionally preload media by tag, you will not be able to access media with other tags using `getMedia()` without first reloading the media relationship on that record.

Variants can also be eager loaded this way.

```
<?php
// lazy eager load from a collection of Mediables
$posts->loadMediaWithVariants($tags);
$posts->loadMediaWithVariantsMatchAll($tags);

// lazy eager load from a single Mediable model
$post->loadMediaWithVariants($tags);
$post->loadMediaWithVariantsMatchAll($tags);
```

1.4.8 Automatic Rehydration

By default, `Mediable` models will automatically reload their media relationship the next time the media at a given tag is accessed after that tag is modified.

The `attachMedia()`, `syncMedia()`, `detachMedia()`, and `detachMediaTags()` methods will mark any tags passed as being dirty, while the `hasMedia()` `getMedia()`, `firstMedia()`, `lastMedia()`, `getAllMediaByTag()`, and `getTagsForMedia()` methods will execute `loadMedia()` to reload all media if they attempt to read a dirty tag.

For example:

```
<?php
$post->loadMedia();
$post->getMedia('gallery'); // returns an empty collection
$post->getMedia('thumbnail'); // returns an empty collection
$post->attachMedia($media, 'gallery'); // marks the gallery tag as dirty

$post->getMedia('thumbnail'); // still returns an empty collection
$post->getMedia('gallery'); // performs a `loadMedia()`, returns a collection with
↪$media
```

You can enable or disable this behaviour on a class-by-class basis by adding the `$rehydrates_media` property to your `Mediable` model.

```
<?php
// ...

class Post extends Model
{
    use Mediable;

    protected $rehydrates_media = false;

    // ...
}
```

You can also set the application-wide default behaviour in `config/mediable.php`.

```
'rehydrate_media' => true,
```

1.4.9 Deleting Mediables

You can delete mediable model with standard Eloquent model `delete()` method. This will also detach any associated `Mediable` models.

```
<?php
$post->delete();
```

Note: The `delete()` method on the query builder *will not* purge media relationships.

```
<?php
Media::where(...)->delete(); //will not detach relationships
```

Soft Deletes

If your `Mediable` class uses Laravel's `SoftDeletes` trait, the model will only detach its media relationships if `forceDelete()` is used.

You can change the `detach_on_soft_delete` setting to `true` in `config/mediable.php` to have relationships automatically detach when either the `Media` record or `Mediable` model are soft deleted.

1.4.10 Custom Mediables Table

By default the `mediables` table is used to for the media-to-mediables relationship. You can change this in `config/mediable.php`:

```
/*
 * Name to be used for mediables joining table
 */
'mediables_table' => 'prefixed_mediables',
```

1.5 Using Media

1.5.1 Media Paths

`Media` records keep track of the location of their file and are able to generate a number of paths relative to the file. Consider the following example, given a `Media` instance with the following attributes:

```
[
        'disk' => 'uploads',
        'directory' => 'foo/bar',
        'filename' => 'picture',
        'extension' => 'jpg'
        // ...
];
```

The following attributes and methods would be exposed:

```
<?php
$media->getAbsolutePath();
// /var/www/site/public/uploads/foo/bar/picture.jpg

$media->getDiskPath();
// foo/bar/picture.jpg

$media->directory;
// foo/bar

$media->basename;
// picture.jpg

$media->filename;
// picture

$media->extension;
// jpg
```

1.5.2 URLs and Downloads

URLs can be generated for Media stored on a public disk and set to public visibility.

```
$media->getUrl();
// http://localhost/uploads/foo/bar/picture.jpg
```

$media->getUrl() will throw an exception if the file or its disk has its visibility set to private. You can check if it is safe to generate a url for a record with the *$media->isPubliclyAccessible()* method.

For private files stored on an Amazon S3 disk, it is possible to generate a temporary signed URL to allow authorized users the ability to download the file for a specified period of time.

```
<?php
$media->getTemporaryUrl(Carbon::now->addMinutes(5));
```

For private files, it is possible to expose them to authorized users by streaming the file from the server.

```
<?php
return response()->streamDownload(
    function() use ($media) {
        $stream = $media->stream();
        while($bytes = $stream->read(1024)) {
            echo $bytes;
        }
    },
    $media->basename,
    [
        'Content-Type' => $media->mime_type,
        'Content-Length' => $media->size
    ]
);
```

1.5.3 Querying Media

If you need to query the media table directly, rather than through associated models, the Media class exposes a few helpful methods for the query builder.

```
<?php
Media::inDirectory('uploads', 'foo/bar');
Media::inOrUnderDirectory('uploads', 'foo');
Media::forPathOnDisk('uploads', 'foo/bar/picture.jpg');
Media::whereBasename('picture.jpg');
```

1.5.4 Moving Media

You should taking caution if manually changing a media record's attributes, as your record and file could go out of sync.

You can change the location of a media file on disk.

```
    <?php
    $media->move('new/directory');
    $media->move('new/directory', 'new-filename');
    $media->rename('new-filename');
$media->moveToDisk('uploads', 'new/directory', 'new-filename');
```

1.5.5 Copying Media

You can duplicate a media file to a different location on disk with the `copyTo()` method. Doing so will create a new `Media` record for the new file. If a filename is not provided, the new file will copy the original filename.

```php
<?php
$newMedia = $media->copyTo('new/directory');
$newMedia = $media->copyTo('new/directory', 'new-filename');
$newMedia = $media->copyToDisk('uploads', 'new/directory', 'new-filename');
```

Note Both `moveToDisk()` and `copyToDisk()` support passing an additional `$options` argument with flags to be passed to the underlying filesystem adapter of the destination disk.

1.5.6 Deleting Media

You can delete media with standard Eloquent model `delete()` method. This will also delete the file associated with the record and detach any associated `Mediable` models.

```php
<?php
$media->delete();
```

Note: The `delete()` method on the query builder *will not* delete the associated file. It will still purge relationships due to the cascading foreign key.

```php
<?php
Media::where(...)->delete(); //will not delete files
```

Soft Deletes

If you subclass the `Media` class and add Laravel's `SoftDeletes` trait, the media will only delete its associated file and detach its relationship if `forceDelete()` is used.

You can change the `detach_on_soft_delete` setting to `true` in `config/mediable.php` to have relationships automatically detach when either the `Media` record or `Mediable` model are soft deleted.

1.5.7 Setting Visibility

You can update the visibility of a *Media* record's file

```php
<?php
$media->makePublic();
$media->makePrivate();
```

1.6 Aggregate Types

Laravel-Mediable provides functionality for handling multiple kinds of files under a shared aggregate type. This is intended to make it easy to find similar media without needing to constantly juggle multiple MIME types or file extensions. For example, you might want to query for an image, but not care if it is in JPEG, PNG or GIF format.

```php
<?php
Media::where('aggregate_type', Media::TYPE_IMAGE)->get();
```

You can use this functionality to restrict the uploader to only accept certain types of files.

```php
<?php
MediaUploader::fromSource($request->file('thumbnail'))
    ->toDestination('uploads', '')
    ->setAllowedAggregateTypes([Media::TYPE_IMAGE, Media::TYPE_IMAGE_VECTOR])
    ->upload()
```

To customize the aggregate type definitions for your project, see *Configuring Aggregate Types*.

1.7 Image Variants

Laravel-Mediable integrates the intervention/image library to make it easy to manipulate image files and create numerous variations for different purposes.

1.7.1 Configure Intervention/image ImageManager

By default, intervention/image will use the GD library driver. If you intend to use the additional features of the ImageMagick driver, you should make sure that the PHP extension is installed and the correct configuration is bound to the Laravel service container.

```php
<?php

use Intervention\Image\ImageManager;

class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            ImageManager::class,
            function() {
                return new ImageManager(['driver' => 'imagick']);
            }
        );
    }
}
```

1.7.2 Defining Variant Manipulations

Image Manipulation Callback

Before variants can be created, the manipulations to be applied to the images need to be defined. This should be done as part of the application boot step.

```php
<?php
use Plank\Mediable\Facades\ImageManipulator;
use Plank\Mediable\ImageManipulation;
use Intervention\Image\Image;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
```

(continues on next page)

(continued from previous page)

```
    {
        ImageManipulator::defineVariant(
            'thumb',
            ImageManipulation::make(function (Image $image, Media $originalMedia) {
                $image->fit(32, 32);
            })->toPngFormat()
        );

        ImageManipulator::defineVariant(
            'bw-square',
            ImageManipulation::make(function (Image $image, Media $originalMedia) {
                $image->fit(128, 128)->greyscale();
            })
        );
    }
}
```

Each variant definition must contain a name and an instance of the `ImageManipulation` class, which contains the instructions for converting the image into the desired derivative form.

First and foremost, each manipulation requires a callback which contains instructions on how the image should be modified. The callback will be passed an instance of `Intervention\Image\Image` and the original `Media` record, and may use any of the methods available to the library to change its form. See the intervention/image documentation for available methods.

Output Formats

The ImageManipulation class also offers a fluent interface for defining how the modified file should be output. If not specified, will attempt to use the same format as the original file, based on the `mime_type` and `extension` attributes of the original Media record.

```
<?php
$manipulation->outputJpegFormat();
$manipulation->outputPngFormat();
$manipulation->outputGifFormat();
$manipulation->outputTiffFormat();
$manipulation->outputBmpFormat();
$manipulation->outputWebpFormat();
$manipulation->setOutputFormat($format);
```

If outputting to JPEG format, it is also possible to set the desired level of lossy compression, from 0 (low quality, smaller file size) to 100 (high quality, larger file size). Defaults to 90. This value is ignored by other formats.

```
<?php
$manipulation->outputJpegFormat()->setOutputQuality(50);
```

Note: Intervention/image requires different dependency libraries to be installed in order to output different format. Review the intervention image documentation for more details.

Output Destination

By default, variants will be created in the same disk and directory as the original file, with a filename that includes the variant name as as suffix. You can choose to customize the output disk, directory and filename.

```php
<?php
$manipulation->toDisk('uploads');
$manipulation->toDirectory('files/variants');

// shorthand for the above
$manipulation->toDestination('uploads', 'files/variants');

$manipulation->useFilename('my-custom-filename');
$manipulation->useHashForFilename();
$manipulation->useOriginalFilename(); //restore default behaviour
```

If another file exists at the output destination, the ImageManipulator will attempt to find a unique filename by appending an incrementing number. This can be configured to throw an exception instead if a conflict is discovered.

```php
<?php
$manipulation->onDuplicateIncrement(); // default behaviour
$manipulation->onDuplicateError();
```

File Visibility

By default, newly created variants will use the default filesystem visibility of the destination filesystem disk. To modify this, you may use one of the following methods.

```php
<?php
$manipulation->makePrivate();
$manipulation->makePublic();
// to copy the visibility of the original media file
$manipulation->matchOriginalVisibility();
```

Before Save Callback

You can specify a callback which will be invoked after the image manipulation is processed, but before the file is written to disk and a `Media` record is written to the database. The callback will be passed the populated `Media` record, which can be modified. This can also be used to set additional fields.

```php
<?php
$manipulation->beforeSave(function(Media $media) {
    $media->directory = 'thumbnails';
    $media->someOtherField = 'potato';
});
```

Note: Modifying the disk, directory, filename, or extension fields will cause the output destination to be changed accordingly. Duplicates will be checked again against the new location.

1.7.3 Creating Variants

Variants can be created from the `ImageManipulator` class. This will create a new file derived from applying the manipulation to the original. A new Media record will be create to represent the new file.

```php
<?php
use Plank\Mediable\Facades\ImageManipulator;

$variantMedia = ImageManipulator::createImageVariant($originalMedia, 'thumbnail');
```

Depending on the size of the files and the nature of the manipulations, creating variants may be a time consuming operation. As such, it may be more beneficial to perform the operation asynchronously. The `CreateImageVariants` job can be used to easily queue variants to be processed. A collection of `Media` records and multiple variant names can be provided in order to process the creation of several variants as part of the same worker process.

```php
<?php
use Plank\Mediable\Jobs\CreateImageVariants;
use Illuminate\Database\Eloquent\Collection;

// will produce one variant
CreateImageVariants::dispatch($media, ['square']);

// will produce 4 variants (2 of each media)
CreateImageVariants::dispatch(
    new Collection([$media1, $media2]),
    ['square', 'bw-square']
);
```

Recreating Variants

If a variant with the requested variant name already exists for the provided media, the `ImageManipulator` will skip over it. If you need to regenerate a variant (e.g. because the manipulations changed), you can tell the `ImageManipulator` to recreate the variant by passing an additional `$forceRecreate` parameter.

```php
<?php
$variantMedia = ImageManipulator::createImageVariant($originalMedia, 'thumbnail',␣
↪true);
CreateImageVariants::dispatch($media, ['square', 'bw-square'], true);
```

Doing so will cause the original file to be deleted, and a new one created at the specified output destination. The variant record will retain its primary key and any associations, but its attributes will be updated as necessary.

Tagging Variants

When defining variants, it is possible to pass one or more "tags" to group the definitions in order to more easily retrieve all of the ones applicable to a specific purpose.

```php
<?php
use Plank\Mediable\Jobs\CreateImageVariants;

ImageManipulator::defineVariant(
    'avatar-small',
    ImageManipulation::make(/* ... */),
    ['avatar']
);

ImageManipulator::defineVariant(
    'avatar-large',
    ImageManipulation::make(/* ... */),
```

(continues on next page)

(continued from previous page)

```
    ['avatar']
);

// generate all 'avatar' variants
CreateImageVariants::dispatch(
    $mediaCollection,
    ImageManipulator::getVariantNamesByTag('avatar')
);
```

1.7.4 Using Variants

For all intents and purposes, variants are fully functional `Media` records. They can be attached to `Mediable` models, output paths and URLs, be moved and copied, etc.

However, variants also remember the name of the variant definition and the original `Media` record from which they were created. This information can be used to find the right file for a given context. This package takes an un-opinionated approach to how your application should use the variants that you create. You can either attach variants directly to your models, or attach the original and then navigate to the appropriate variant.

```
<?php
$src = $post->getMedia('feature')
    ->findVariant('thumbnail')
    ->getUrl()
```

Original vs. Variants

An "original" `Media` record is one the one that was initially uploaded to the server. A variant is the derivative that was created by manipulating the original. You can distinguish them with these methods:

```
<?php
// check if the Media is an original
$media->isOriginal();

// check if the Media is any kind of variant
$media->isVariant();

// check if the Media is a specific kind of variant
$media->isVariant('thumbnail');

// read the kind of the variant, will be `null` for originals
$media->variant_name
```

Navigating between variants

From any instance of a Media, you can jump to any other in the same variant family using the following methods. If you are already dealing with the variant that you are requesting, it will return itself.

```
<?php
$original = $media->findOriginal();
$variant = $media->findVariant('thumbnail');
$bool = $media->hasVariant('thumbnail');
```

Warning: Avoid chaining find calls from one `Media` to the next. To avoid unnecessary database calls, it is best to always start from the same initial node.

List All Variants

You can also list out all of the variants and the original of a variant family as a keyed dictionary.

```
<?php

// excluding the current model
$collection = $media->getAllVariants();

// including the current model
$collection = $media->getAllVariantsAndSelf();

/* outputs
[
    'original' => Media{},
    'thumbnail' => Media{},
    'large' => Media{}
    etc.
]
*/
```

Manual Adjustments

If necessary, you can also promote a variant to become an original. Doing so clears its variant name and detaches it from the rest of its former variant family.

```
<?php
$variant->makeOriginal()->save();
```

To manually indicate that one `Media` record is a variant of another

```
<?php
$media->makeVariantOf($otherMedia, 'small')->save();
$media->makeVariantOf($otherMediaId, 'small')->save();
```

Note: A variant family is a set, not a tree. If a variant is created from or associated to another variant, they will share the same original Media.

Eager Loading

When accessing media variants from a collection of Mediable records, be sure to eager load them when possible to avoid the N+1 query problem.

```
<?php
// eager load
$posts = Post::withMediaAndVariants($tags)->get();
$posts = Post::withMediaAndVariantsMatchAll($tags)->get();
```

(continues on next page)

(continued from previous page)

```
// lazy eager load from a collection of Mediables
$posts->loadMediaAndVariants($tags);
$posts->loadMediaAndVariantsMatchAll($tags);

// lazy eager load from a single Mediable model
$post->loadMediaAndVariants($tags);
$post->loadMediaAndVariantsMatchAll($tags);
```

1.8 Artisan Commands

This package provides a handful of artisan commands to help keep you filesystem and database in sync.

Create a media record in the database for any files on the disk that do not already have a record. This will apply any type restrictions in the mediable configuration file.

```
$ php artisan media:import [disk]
```

Delete any media records representing a file that no longer exists on the disk.

```
$ php artisan media:prune [disk]
```

To perform both commands together, you can use:

```
$ php artisan media:sync [disk]
```