File Upload — Tasveerein Aur Files Sambhalna

Photos the safe way: the file input and enctype, image validation, random stored names, storage:link, and cleaning up old files on replace and delete.

Introduction

Students have faces. Products have photos. Every real app eventually asks: *upload your picture.* File upload looks like one input on a form — and hides four separate dangers: the form forgetting how to carry files, a stranger uploading something evil, two visitors' files overwriting each other with the same name, and orphaned files piling up when records change.

Laravel's upload system answers each danger with one habit: the form declares `enctype="multipart/form-data"`, validation checks the file like any other field (`image`, `mimes`, `max`), the framework stores it under a **random name** on a chosen disk, the database remembers only the **path**, and `storage:link` makes the public folder able to show it. This lesson wires the full story onto the Students app — upload, display, replace, delete — with cleanup at every step.

(Roman Urdu: File upload aik mehman-nawazi hai jisme hifazat pehle aati hai — jaanch, naya naam, mehfooz jagah, aur purani safai.)

Key points

- **The form must declare its cargo:** `<form method="POST" enctype="multipart/form-data">` — without that one attribute the file never leaves the browser and `$request->file('photo')` arrives empty. The input itself is plain: `<input type="file" name="photo">`.
- **Files are fields too:** the uploaded file rides inside the same `$request`, validated by the same `validate()` — `'photo' => 'required|image|mimes:jpg,jpeg,png|max:2048'` (image only, those types, at most 2048 KB = 2 MB). Never skip file validation because "the input only accepts images" — the browser's hint is not a guard (LV.5's law).
- **Never keep the visitor's file name:** two `photo.jpg` uploads would collide, and a crafted name can be worse. `$request->file('photo')->store('students', 'public')` saves the file under a long random name inside `storage/app/public/students/` and returns its path.
- **The database stores the path, not the file:** one string column (`photo`) holding `students/xK9...f3.jpg`. Files live on disks; databases hold addresses.
  (Roman Urdu: File almari mein, pata register mein — database sirf rasta yaad rakhta hai, samaan nahi.)
- **`php artisan storage:link` builds the bridge:** it creates a shortcut `public/storage` → `storage/app/public`, so the browser can reach the file at `asset('storage/' . $student->photo)`. Without the link, every photo 404s while existing happily on disk.
- **Replace and delete must clean up:** a new photo deletes the old file; a deleted student deletes his photo. Skipped cleanup is how disks fill with ghosts of records long gone.

- `Storage::download()` **hands files back properly:** for certificates, reports and receipts, it serves the file as a download with the right headers instead of a fragile direct link.

Code examples

Example 1 — The form and the gate: input + validation

```
{{-- students/create.blade.php — the LV.5 form grows one field --}}
<form method="POST" action="{{ route('students.store') }}" enctype="multipart/form-data">
    @csrf

    <label>Photo</label>
    <input type="file" name="photo">
    @error('photo')
        <p class="error">{{ $message }}</p>
    @enderror
    {{-- name, city, marks fields as before --}}
</form>
```

```
// StudentController@store — the file passes the same gate as text
$validated = $request->validate([
    'name'  => 'required|string|max:50',
    'city'  => 'required|string|max:50',
    'marks' => 'required|integer|min:0|max:100',
    'photo' => 'required|image|mimes:jpg,jpeg,png|max:2048',
]);
```

What it does: Two changes carry the whole feature: the form announces it carries files, and the photo joins the validation rules. A `.exe` renamed to `.jpg`, a 9 MB scan, a missing photo — each bounces with its own message under the photo field, beside LV.5's familiar red lines.

What you will see: submit a text file renamed `photo.jpg` and the form bounces: `The photo field must be an image.` Submit a real photo and the cycle proceeds to Example 2.

Example 2 — Store under a random name, save the path, show the face

```
public function store(Request $request)
{
    $validated = $request->validate([ /* rules from Example 1 */ ]);

    $path = $request->file('photo')->store('students', 'public');

    Student::create([
        'name'  => $validated['name'],
        'city'  => $validated['city'],
        'marks' => $validated['marks'],
        'photo' => $path,
    ]);

    return redirect()->route('students.index')
        ->with('success', 'Student added with photo!');
}
```

```
php artisan storage:link
```

```
{{-- students/index.blade.php — the face in the table --}}
<img src="{{ asset('storage/' . $student->photo) }}" alt="{{ $student->name }}" width="48">
```

What it does: `store()` does three jobs in one breath: invents a collision-proof random name, writes the file to the `public` disk under `students/`, and returns the path — which is all the database keeps. (The students migration gains a nullable `photo` string column; `photo` joins the model's `$fillable` so `create()` may carry it.) `storage:link`, run once per project (and once per server, LV.8), bridges the storage folder into the public one so `asset()` can serve the picture.

What you will see: the terminal confirms `The [public/storage] link has been connected to [storage/app/public]`; the students list now shows a small round face beside each name; and the table's photo cell for Ali holds a path like `students/9fK2...a1.jpg` — a name no visitor chose and no collision can touch.

Example 3 — Replace + delete: no ghosts left behind

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

public function update(Request $request, Student $student)
{
    $validated = $request->validate([
        'name'  => 'required|string|max:50',
        'marks' => 'required|integer|min:0|max:100',
        'photo' => 'nullable|image|mimes:jpg,jpeg,png|max:2048',
    ]);

    if ($request->hasFile('photo')) {
        Storage::disk('public')->delete($student->photo);
        $student->photo = $request->file('photo')->store('students', 'public');
    }

    $student->name  = $validated['name'];
    $student->marks = $validated['marks'];
    $student->save();

    return redirect()->route('students.index')
        ->with('success', 'Student updated!');
}

public function destroy(Student $student)
{
    Storage::disk('public')->delete($student->photo);
    $student->delete();

    return redirect()->route('students.index')
        ->with('success', 'Student deleted!');
}
```

What it does: The update's photo rule is `nullable` — keep the old face unless a new one arrives. When one does arrive, the OLD file is deleted from the disk before the new path is stored. `destroy()` deletes the file first, then the row. The disk and the database now tell the same story, forever.

What you will see: replace Hira's photo and count the files in `storage/app/public/students/` before and after — the count does not grow; the old file is simply gone. Delete a practice student and his photo vanishes with his row.

Quick recap

- Upload needs three habits: `enctype` on the form, the file validated like a field, and storage under a random name.
- The database holds the **path**; the disk holds the file; `storage:link` bridges storage into public view.
- Replace deletes the old file; destroy deletes the file with the row — no ghosts.
- `Storage::download()` serves files back as proper downloads.

Common mistakes

1. **Forgetting `enctype="multipart/form-data"`** — the silent classic: the form submits, `$request->file('photo')` is null, and no error explains why. Fix: the attribute goes on EVERY form that carries a file, the moment the file input is added.
2. **Keeping the visitor's original file name** — collisions overwrite strangers' photos, and crafted names attack the server. Fix: `store()` and its random names, always; the original name is metadata, never the address.
3. **Skipping `storage:link`** — uploads succeed, paths save, and every image 404s. Fix: run it once per project (and once on the server at deploy time) before blaming the code.
4. **Never deleting replaced/deleted files** — the disk becomes a museum of removed records. Fix: cleanup lives inside `update()` and `destroy()` themselves, so no path through the app can orphan a file.

Practice

1. Make the photo optional at create time too (`nullable` + a default avatar path shown when `photo` is empty). Register one student with a photo and one without; the list must show both faces correctly.
2. Add a "Download class list" button that builds a `students.txt` summary on the fly, saves it to storage, and returns it with `Storage::download()`. Open the downloaded file and check the five names are inside.

Practice quiz

1. Without which form attribute does an uploaded file never leave the browser?

- method="POST" alone
- enctype="multipart/form-data"; without it $request->file('photo') arrives empty
- action="/upload"
- name="photo"

Answer: enctype="multipart/form-data"; without it $request->file('photo') arrives empty

2. What does the database store for an uploaded photo, and under what name is the file saved?

- The file itself, under the visitor's original name

- One string column holding the path, like students/xK9...f3.jpg; store() saves under a long random name, because two photo.jpg uploads would collide
- Nothing, files are not recorded
- The file's size only

Answer: One string column holding the path, like students/xK9...f3.jpg; store() saves under a long random name, because two photo.jpg uploads would collide

3. Without php artisan storage:link, what happens to uploaded photos?

- They are deleted
- Every photo 404s while existing happily on disk, because the public/storage shortcut to storage/app/public was never built
- They upload to the wrong folder
- Nothing changes, the link is optional

Answer: Every photo 404s while existing happily on disk, because the public/storage shortcut to storage/app/public was never built

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/laravel-file-upload · More free tools and notes at https://scopetreasure.com