File Uploads Deep: Saving Pictures Safely with Multer

Forms can send files, not just text: Multer receives them, size limits and file-type checks keep the server safe, and safe names stop overwrites.

Introduction

Every form so far sent **text**. But profile pictures, product photos and assignment PDFs are **files**, and files travel differently: the form must use a special encoding called `multipart/form-data`, and plain body middleware cannot unpack it. The package that does this job in Express is **Multer** — middleware that catches the uploaded file, saves it where you say, and hands your route a neat report about it.

Uploads are also where beginners get hurt. A file can be enormous, it can be the wrong kind entirely, and two visitors can both upload a file named `photo.jpg`. This lesson builds the safe version from the start: a **size limit**, a **file-type check**, and **safe generated names** — plus the final safety habit: when a record is deleted or its picture replaced, the old file is removed too, so the disk does not fill with orphans. *(Roman Urdu: File upload mein teen hifazatein lazmi hain — size ki hadd, qism ki jaanch, aur mehfooz naam.)*

Key points

- **Forms need the multipart encoding:** `<form enctype="multipart/form-data">` with an `<input type="file" name="photo">`. Without that encoding, only the file's *name* travels — never its contents.
- **Multer is middleware for files:** `upload.single("photo")` runs before your route, receives the one file sent in the field named `photo`, saves it, and fills `req.file` with the report: original name, size, and where it was saved. Text fields still arrive in `req.body`.
- **Destination and name are your decision:** Multer's `diskStorage` lets you choose the folder (`uploads/`) and the saved **filename**. Never keep the visitor's original name as-is.
  (Roman Urdu: Aayi hui file ko apna naam do — visitor ke naam par bharosa mat karo.)
- **Safe names prevent overwrites:** build the saved name from something unique — the current time plus a random number, keeping the original ending: `1720000000000-483920.jpg`. Two `photo.jpg` uploads can now never destroy each other.
- **Size limits are a wall, not a wish:** `limits: { fileSize: 2 * 1024 * 1024 }` refuses anything over 2 MB. Multer raises an error your error handler answers honestly (lesson N.9 makes this graceful).
- **Type checks look at the file, not the claim:** the `fileFilter` function inspects the file's type (its MIME type) and accepts only what you allow — pictures for a photo field — refusing the rest before a single byte is saved.
- **Old files must die with their records:** replacing a picture or deleting a student should delete the old file from `uploads/` in the same request, using `fs.unlink`. Otherwise the disk quietly fills with files nobody points at.

Code examples

Example 1 — Receiving one picture, safely named

```
// npm install express multer
const express = require("express");
const multer = require("multer");

const app = express();

const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, "uploads/"),
  filename: (req, file, cb) => {
    const unique = Date.now() + "-" + Math.round(Math.random() * 1e9);
    const ending = file.originalname.substring(file.originalname.lastIndexOf("."));
    cb(null, unique + ending);
  },
});

const upload = multer({ storage });

app.post("/students/:id/photo", upload.single("photo"), (req, res) => {
  res.json({
    savedAs: req.file.filename,
    sizeKB: Math.round(req.file.size / 1024),
    studentId: req.params.id,
  });
});

app.listen(3000, () => console.log("Server running on port 3000"));
```

What it does: The `diskStorage` rules answer the two questions every upload raises: *where* (the `uploads/` folder — create it first) and *under what name* (a unique time-and-random name that keeps only the original ending like `.jpg`). `upload.single("photo")` then stands in front of the route like the middleware you know: by the time the route runs, the file is already saved and `req.file` holds its report.

What you will see: posting a picture for student 1 answers something like `{"savedAs":"1720000000000-483920.jpg","sizeKB":84,"studentId":"1"}`, and that exact file now sits inside the `uploads/` folder — open it and it is your picture.

Example 2 — The two walls: size limit and type check

```
const uploadSafe = multer({
  storage,
  limits: { fileSize: 2 * 1024 * 1024 }, // 2 MB, no more
  fileFilter: (req, file, cb) => {
    if (file.mimetype.startsWith("image/")) {
      cb(null, true);
    } else {
      cb(new Error("Only picture files are allowed"));
    }
  },
});

app.post("/students/:id/photo", uploadSafe.single("photo"), (req, res) => {
  res.json({ savedAs: req.file.filename });
});
```

What it does: Two guards join the storage rules. The **limit** stops the upload the moment it passes 2 MB — a phone video cannot eat your server's disk. The **filter** checks the file's real type before saving: anything claiming to be an image passes; a `.exe`, a script or a PDF is refused with an error. Honest note: a MIME type is a strong first wall, not a laboratory test — real systems also re-check the file's contents, but these two walls stop the everyday accidents and attacks.

What you will see: a 1 MB picture uploads normally; a 5 MB file fails with Multer's "File too large" error; a `.txt` file is refused with `Only picture files are allowed` — and in both failures, nothing new appears in `uploads/`.

Example 3 — Replacing a picture: the old file dies

```
const fs = require("fs");

// A tiny record keeper for this lesson.
const photos = {}; // studentId -> saved filename

app.post("/students/:id/photo", uploadSafe.single("photo"), (req, res) => {
  const id = req.params.id;
  const old = photos[id];
  if (old) {
    fs.unlink("uploads/" + old, () => {});
  }
  photos[id] = req.file.filename;
  res.json({ savedAs: req.file.filename, replaced: old || null });
});
```

What it does: Before remembering the new filename, the route looks up the student's previous picture and **deletes that file** with `fs.unlink` — Node's own file system module from lesson N.1's world of server powers. The record now points only at the new picture, and the disk holds exactly one photo per student, forever. The same pattern runs when a record itself is deleted: remove the database row *and* its file, together.

What you will see: uploading a second picture for student 1 answers with the new `savedAs` and the `replaced` name of the first — and checking `uploads/` shows the first file gone, only the replacement remaining.

Quick recap

- File forms use `multipart/form-data`; **Multer** middleware receives the file and fills `req.file`.
- **You** choose the folder and the saved name — unique names, original ending, never the visitor's filename.
- Two walls on every upload: a **size limit** and a **file-type filter**.
- Replacing or deleting a record **deletes its old file** in the same request.

Common mistakes

1. **Keeping the visitor's original filename** — two `photo.jpg` uploads overwrite each other, and a crafted name can even try to escape the folder. The fix: generate your own unique saved name every time, as in Example 1; the original name is only a report detail.
2. **No size limit "for now"** — one huge upload fills the disk or eats the memory and the whole server suffers for every user. The fix: set `limits.fileSize` on day one; a refused upload is a small error, a full disk is an outage.
3. **Saving the file but losing the link** — the file lands in `uploads/` but its filename is never stored on the record, so no page can ever show the picture. The fix: the moment an upload succeeds, save `req.file.filename` onto the record (database field like `photo`) in the same request.

Practice

1. Build an uploads page for the students API: the photo route from Example 2, plus `express.static("uploads")` mounted so saved pictures can be viewed at `http://localhost:3000/<saved-name>`. Upload two different pictures for two students and open both in the browser.
2. Add a second field for a PDF assignment (`upload.fields`) with its own rule: PDFs only, up to 5 MB, saved into `uploads/assignments/`. Test one honest PDF and one renamed non-PDF, and record what each answer says.

Practice quiz

1. What does upload.single("photo") do before your route runs?

- It validates the text fields

- It receives the one file sent in the field named photo, saves it, and fills req.file with the report: original name, size, and where it was saved; text fields still arrive in req.body
- It deletes old files
- It resizes the photo

Answer: It receives the one file sent in the field named photo, saves it, and fills req.file with the report: original name, size, and where it was saved; text fields still arrive in req.body

2. How do you build a safe saved filename with Multer?

- Keep the visitor's original name as-is
- From something unique, the current time plus a random number, keeping the original ending, like 1720000000000-483920.jpg, so two photo.jpg uploads can never destroy each other
- From the user's name
- Multer names files itself, you cannot choose

Answer: From something unique, the current time plus a random number, keeping the original ending, like 1720000000000-483920.jpg, so two photo.jpg uploads can never destroy each other

3. What are the two walls on every Multer upload, besides the form encoding?

- A password and a captcha
- A size limit, like limits: { fileSize: 2 * 1024 * 1024 } refusing anything over 2 MB, and a fileFilter type check that inspects the file's MIME type and accepts only what you allow
- Two file inputs
- A virus scan and a backup

Answer: A size limit, like limits: { fileSize: 2 * 1024 * 1024 } refusing anything over 2 MB, and a fileFilter type check that inspects the file's MIME type and accepts only what you allow

---

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/node-file-uploads-multer · More free tools and notes at https://scopetreasure.com