



## Multer – Armazenamento de fotos

Bom, vou ser logo direto, não armazenamos a foto em si, não é recomendado por ocupar espaço, e sim, o seu caminho e fazemos uma foreign key da foto referenciando ao usuário.

Usaremos o multer para armazenar a foto em uma pasta do disco do nosso servidor, após o usuário enviar. Então vamos criar um arquivo de config do multer (multerConfig.js)

```
src > config >  multerConfig.js >  salvarDb
1  import { fileURLToPath } from 'url';
2  import multer from 'multer';
3  import path from 'path';
4  import fs from 'fs';
5  import fotoDb from '../models/dbFotoModel.js';
6
7  const filename = fileURLToPath(import.meta.url);
8  const dirname = path.dirname(filename);
9  const randomNumber = Math.floor(Math.random() * 1000 + 1000);
10
11  const storage = multer.diskStorage({
12    destination: (req, file, cb) => {
13      const userFolder = path.resolve(dirname, '..', '..', 'uploads', String(req.id));
14
15      // Criar a pasta do usuário se não existir
16      if (!fs.existsSync(userFolder)) {
17        fs.mkdirSync(userFolder, { recursive: true }); // o recursive serve para criar o caminho tood
18      }
19
20      cb(null, userFolder); //antes dele chegar aqui, ele já criou a pasta
21    },
22    filename: (req, file, cb) => {
23      const nameFile = `${Date.now()} ${randomNumber}${path.extname(file.originalname)}`;
24      req.savedFilename = nameFile; // Salva o nome do arquivo para usar depois no banco
25      cb(null, nameFile);
26    },
27  });
28
29  const fileFilter = (req, file, cb) => {
30    if (file.mimetype !== 'image/png' && file.mimetype !== 'image/jpeg') {
31      return cb(new multer.MulterError('Arquivo inválido'));
32    }
33    cb(null, true);
34  };
```

Em .diskStorage estamos apenas configurando o caminho e o nome do arquivo. Usamos o id do usuário para criar a pasta, para isso usamos o fs e as funções .existsSync() para ver se a pasta existe e o mkdirSync() para criar a pasta, dentro dessa função usamos o recursive: true que permite que, caso não tenha as outras pastas de caminho, ele crie, caso seja falso, ele irá dar erro. Por fim, usamos o cb(null, true); o null é para tratar erros, o true é para prosseguir, como fosse um next

```

36 // Middleware para salvar no banco após o upload
37 const salvarDb = async (req, res, next) => {
38   if (!req.file) {
39     return next(new Error('Nenhum arquivo foi enviado.'));
40   }
41
42   try {
43     const wayDb = `/uploads/${req.id}/${req.savedFilename}`;
44     const foto = new FotoDb();
45     await foto.add(req.id, wayDb);
46
47     console.log('Imagem salva no banco:', wayDb);
48     next();
49   } catch (e) {
50     console.log('Erro ao salvar no banco:', e);
51     next(e);
52   }
53 };
54
55 const upload = multer({ storage, fileFilter }); // a ordem de execução do multer é o fileFilter sendo o primeiro, se ele permitir, ele executa o storage
56
57 export { upload, salvarDb };

```

A continuação do código. Uma função para armazenar o caminho no banco de dados

```

src > models > JS dbFotoModel.js > ...
1  import db from './model.js';
2
3  class AdicionarFoto{
4    async add(id_user, way){
5      try{
6        const [ user ] = await db.execute(`
7          INSERT INTO foto (id_user, photo_way) VALUES (?, ?)
8          `, [id_user, way]);
9        return user[0];
10     }catch(e){
11       console.log('Erro em armazenar caminho da foto: ', e);
12     }
13   }
14 }
15 export default AdicionarFoto;
16

```

OBS: o multer tem uma sequência de prioridade para executar, por exemplo: Nota-se que criamos uma função para filtrar a foto para ver se ela é .jpeg ou .png. Então o multer irá executar dessa maneira:

**Mas e a ordem das propriedades?**

A ordem no objeto { storage, fileFilter } não importa porque o Multer tem uma lógica interna:

Ele sempre executa fileFilter primeiro.

Se fileFilter passar, ele chama storage.

Isso é um comportamento interno do Multer, não baseado na ordem dos argumentos.

O Multer não se baseia no nome das variáveis em si, mas sim nas propriedades de configuração que você passa ao inicializar o Multer, como storage, fileFilter, limits, etc.

Ou seja, o Multer precisa dessas propriedades com nomes específicos para que ele saiba o que fazer.

## Agora executamos nossa função:

```
src > controllers > fotoControll.js > ...
1 import { upload, salvarDb } from "../config/multerConfig.js";
2
3 class FotoController {
4   async store(req, res) {
5     upload.single("foto")(req, res, async (erro) => { //o upload retorna uma função de callback, e ai a função que está ao lado fica como argumento
6       if (erro) {
7         return res.status(400).json({ erro: erro.code });
8       }
9       try {
10        await salvarDb(req, res, () => {}); // Chama a função para salvar no banco
11        res.json({
12          mensagem: "Upload realizado com sucesso!",
13          caminho: `~/uploads/${req.id}/${req.file.filename}`,
14        });
15      } catch (error) {
16        res.status(500).json({ erro: "Erro ao salvar no banco." });
17      }
18    });
19  }
20 }
21
22 export default new FotoController();
```

`await salvarDb(req, res, () => {});`

está chamando a função `salvarDb`, que provavelmente salva os dados da foto no banco de dados. Agora, a parte `() => {}` no final pode parecer estranha, mas eu vou te explicar o que está acontecendo.

### Por que tem um callback vazio `() => {}`?

Em muitos middlewares (como Multer, Express, etc.), funções do tipo middleware aceitam um terceiro argumento, que geralmente é o `next()`.

O formato típico de um middleware é: `(req, res, next) => { ... }`

`req` → Objeto da requisição

`res` → Objeto da resposta

`next` → Chama o próximo middleware na sequência

Se `salvarDb` estiver sendo usada como um middleware, mas não precisa de um `next()`, então estamos passando um callback vazio `() => {}` só para preencher o espaço, porque algumas funções podem esperar que um terceiro argumento seja passado.