File Upload With Multer in Node.js and Express

Esther Vaati
May 25, 2022 • 7 min read

JavaScript Node HTML/CSS HTML

When a web client uploads a file to a server, it is generally submitted through a form and encoded as `multipart/form-data`. Multer is Express middleware used to handle this `multipart/form-data` when your users upload files.

In this tutorial, I'll show you how to use the Multer library to handle different file upload situations in Node.

How Does Multer Work?

As I said above, Multer is Express middleware. Middleware is a piece of software that connects different applications or software components. In Express, middleware processes and transforms incoming requests to the server. In our case, Multer acts as a helper when uploading files.

Project Setup

We will be using the Node Express framework for this project. Of course, yo need to have Node installed.

Create a directory for our project, navigate into the directory, and run `npm init -y` to create a **package.json** file that manages all the dependencies for our application. The `-y` suffix—also known as the `yes` flag—is used to accept the default values that come from `npm init` prompts automatically.

```
mkdir upload-express
cd upload-express
npm init -y
```

Next, we will install Multer, Express, and the other dependencies necessary to bootstrap an Express app.

```
npm install express multer --save
```

We will create a **server.js** file:

```
touch server.js
```

In the **server.js** file, we will initialise all the modules, create an Express app, and create a server for connecting to browsers.

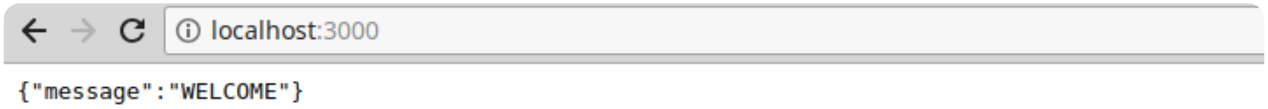
```
// require the installed packages
const express = require('express')
const multer = require('multer');

//CREATE EXPRESS APP
const app = express();

//ROUTES WILL GO HERE
app.get('/', function(req, res) {
    res.json({ message: 'WELCOME' });
});

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

After requiring packages and listening to our server on `port:3000`, we can run the snippet `node server.js` as this runs our server locally. Navigate to `localhost:3000` on your browser and you should see the following message.



Create the Client Code

When we are sure our server is running fine, the next thing will be to create an **index.html** file to write all the code that will be served to the client.

```
touch index.html
```

This file will contain the different forms that we will use for uploading our different file types.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>MY APP</title>
</head>
<body>


 <!-- SINGLE FILE -->
<form action="/uploadfile" enctype="multipart/form-data" method="POST
   <input type="file" name="myFile" />
   <input type="submit" value="Upload a file"/>
</form>


<!-- MULTIPLE FILES -->

<form action="/uploadmultiple"  enctype="multipart/form-data" met
  Select images: <input type="file" name="myFiles" multiple>
```

```
    Select images: <input type="file" name="myFiles" multiple>
    <input type="submit" value="Upload your files"/>
  </form>

    <!-- PHOTO-->

  <form action="/uploadphoto" enctype="multipart/form-data" method="POS
    <input type="file" name="myImage" accept="image/*" />
    <input type="submit" value="Upload Photo"/>
  </form>



  </body>
  </html>

```

Open the **server.js** file and write a GET route that renders the **index.html** file instead of the **"WELCOME"** message.

```
// ROUTES
app.get('/',function(req,res){
  res.sendFile(__dirname + '/index.html');
});
```

Multer Storage

The next thing will be to define a storage location for our files. Multer gives the option of storing files either in memory (`multer.memoryStorage`) or to disk (`memory.diskStorage`). For this project, we will store the files to disk.

For the disk storage, we have two objects: destination and filename. `destination` is used to tell Multer where to upload the files, and `filename` is used to name the file within the destination.

To create our destination directory, run the code snippet below to create a new folder called `uploads`:

```
mkdir uploads
```

Here, we'll add the following code snippets to the **server.js** file.

```
//server.js

// SET STORAGE
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads')
  },
  filename: function (req, file, cb) {
    cb(null, file.fieldname + '-' + Date.now())
  }
})

var upload = multer({ storage: storage })
```

Handling File Uploads

Uploading a Single File

In the **index.html** file, we defined an action attribute that performs a POST request. Now we need to create an endpoint in the Express application. Open the **server.js** file and add the following code snippet:

```
app.post('/uploadfile', upload.single('myFile'), (req, res, next) =>
  const file = req.file
  if (!file) {
    const error = new Error('Please upload a file')
    error.httpStatusCode = 400
    return next(error)
  }
    res.send(file)

})
```

Note that the name of the file input should be the same as the `myFile` argument passed to the `upload.single` function.

Uploading Multiple Files

Uploading multiple files with Multer is similar to a single file upload, but with a few changes. The `upload.array` method accepts two parameters, which are the required field name `myFiles` and the maximum count of files `12`.

```
//Uploading multiple files
app.post('/uploadmultiple', upload.array('myFiles', 12), (req, res, r
  const files = req.files
  if (!files) {
    const error = new Error('Please choose files')
    error.httpStatusCode = 400
    return next(error)
  }
    res.send(files)
})
```

Uploading Images

Instead of saving uploaded images to the file system, we'll store them in a MongoDB database so that we can retrieve them later as needed. But first, let's install MongoDB.

```
npm install mongodb --save
```

We will then connect to MongoDB through the `Mongo.client` method and then add the MongoDB URL to that method. You can use a cloud service like mLab, which offers a free plan, or simply use the locally available connection. In this tutorial, we'll use the locally available connection, as shown below. We'll include the code snippets below in the **server.js** file:

```
const MongoClient = require('mongodb').MongoClient
const myurl = 'mongodb://localhost:27017';

MongoClient.connect(myurl, (err, client) => {
  if (err) return console.log(err)
  db = client.db('test')
  app.listen(3000, () => {
    console.log('Database connected successfully')
    console.log('Server started on port 3000')
  })
})
```

We need to read the file path of our request. In Node.js, we can use the `file-system` (`fs`) module to read the content of a file. We have to install the `file-system` dependency:

```
npm install file-system --save
```

The `file-system` module is a built-in module which has the methods `readFile()` and `readFileSync()`. In this tutorial, we'll use the `readFileSync()` method. The `readFileSync()` method is used to read the content of a file synchronously, which means that the method has to be finished before the program execution continues. This method accepts the parameter `path`, which is the relative path of the file you want to read.

Open **server.js,** require the `file-system` dependency, and define a POST request that enables the saving of images to the database.

```
const fs = require('file-system');

app.post('/uploadphoto', upload.single('myImage'), (req, res) => {
    var img = fs.readFileSync(req.file.path);
    var encode_image = img.toString('base64');
    // Define a JSONobject for the image attributes for saving to dat

    var finalImg = {
        contentType: req.file.mimetype,
        image: Buffer.from(encode_image, 'base64')
    };
     db.collection('myCollection').insertOne(finalImg, (err, result)
        console.log(result)
        if (err) return console.log(err)
        console.log('saved to database')
        res.redirect('/')
    })
})
```

In the above code, we first encode the image to a base64 string, construct a new buffer from the base64 encoded string, and then save it to our database collection in JSON format.

We then display a success message and redirect the user to the index page.

Retrieving Stored Images

To retrieve the stored images, we perform a MongoDB search using the `find` method and return an array of results. We then obtain the `_id` attributes of the images and return them to the user.

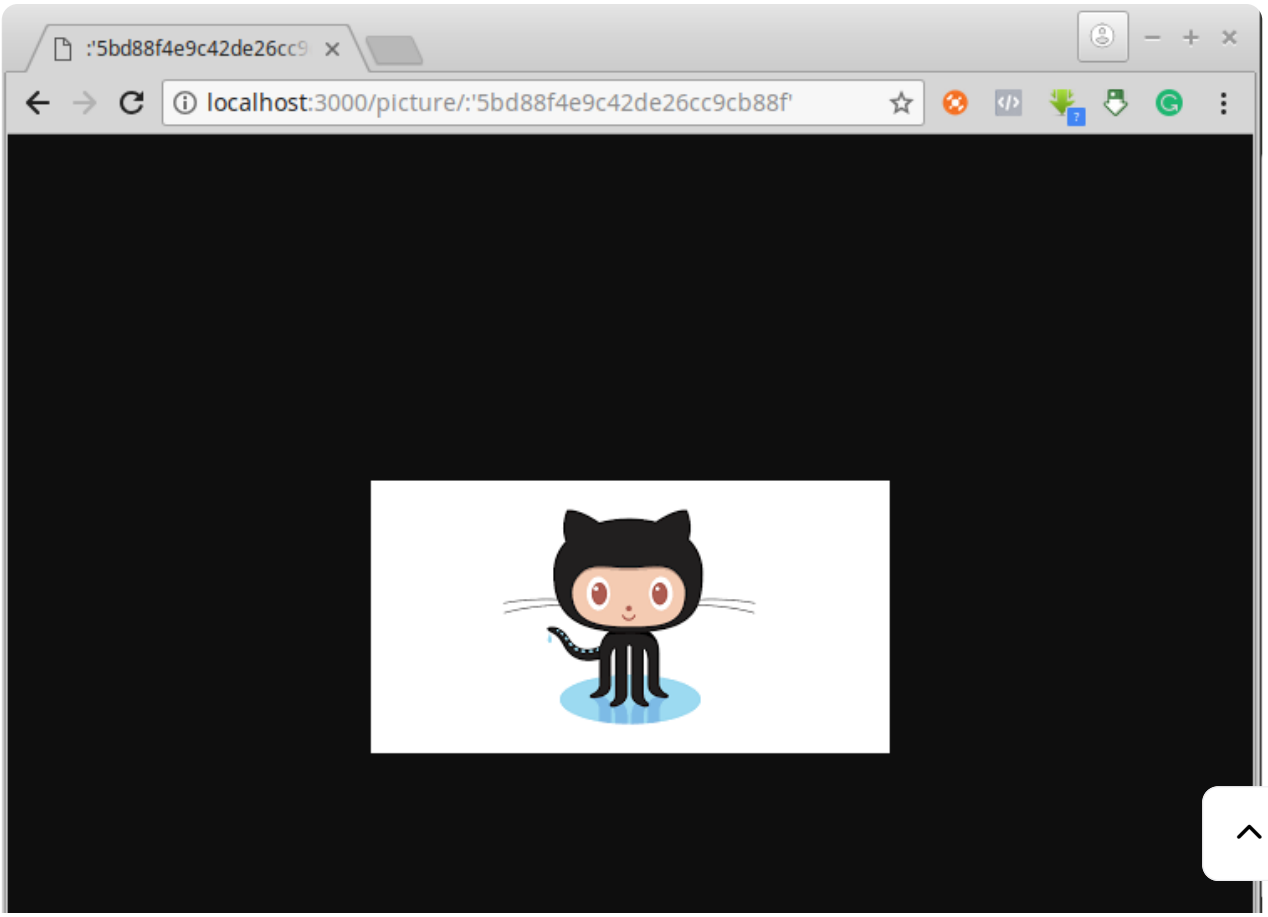
```
app.get('/photos', (req, res) => {
    db.collection('myCollection').find().toArray((err, result) => {
        const imgArray = result.map(element => element._id);
        console.log(imgArray);
        if (err) return console.log(err)
        res.send(imgArray)

    })
});
```

Since we already know the ids of the images, we can view an image by passing its id in the browser, as illustrated below.

```
const ObjectId = require('mongodb').ObjectId;

app.get('/photo/:id', (req, res) => {
    var filename = req.params.id;
    db.collection('myCollection').findOne({ '_id': ObjectId(filename)
        if (err) return console.log(err)
        res.contentType('image/jpeg');
        res.send(result.image.buffer)
    })
})
```



Conclusion

I hope you found this tutorial helpful. File upload can be an intimidating topic, but it doesn't have to be hard to implement. With Express and Multer, handling `multipart/form-data` is quite straightforward.

You can find the full source code for the file upload example in our GitHub repo.

This post has been updated with contributions from Mary Okosun. Mary is a software developer based in Lagos, Nigeria, with expertise in Node.js, JavaScript, MySQL, and NoSQL technologies.

Esther Vaati
Software developer

Software developer and Educator. Student of Life

vaatiesther_

View on GitHub