Configuring Express Multer Middleware and Checking File Information


John Au-Yeung Follow
Mar 16, 2020 · 5 min read ★

Photo by Maksym Kaharlytskyi on Unsplash

Multer is a middleware that lets us process file uploads with our Express app.

In this article, we'll look at the settings that we can change to upload files our way, and also check the file information.

Checking File Information

We can check the uploaded files' information by looking at the `req.file` object for single file upload and the array entries of `req.files` for multiple file uploads.

These fields are available for each file:

- `fieldname` — field name specified in the form
- `originalname` — name of the file on the user's computer
- `encoding` — encoding type of the file
- `mimetype` — MIME-type of the file
- `size` — size of the file in bytes
- `destination` — the folder where the file was saved on the server
- `filename` — name of the file stored in the `destination`
- `path` — the full path of the uploaded file
- `buffer` — `Buffer` object of the whole file

Options for File Upload

The `multer` function takes an `options` object that takes a variety of options.

We can do things like set the destination of the files and rename the files.

The following properties can be in the `options` object:

- `dest` or `storage` — where to store the files
- `fileFilter` — controls which files are accepted
- `limits` — limits of the uploaded data
- `preservePath` — keep the full path of the files instead of just the base name

Storage Options

DiskStorage

We can store files on disk by using the d `iskStorage` method.

There're 2 options available, `destination` and `filename` . They're both functions that determine the destination where the file is saved and what to rename the file to respectively.

Each function takes the requestion object, file object and callback function. The callback function is called at the end of each function with the first argument being `null` .

The second argument is the destination that we want to save the file for the `destination` function and the filename that we want to rename the file to for the `filename` function.

For example, we can rename a file by keeping the original name and adding a timestamp to the end of the file as follows:

```
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, './uploads/')
  },
  filename: (req, file, cb) => {
    cb(null, `${file.originalname}-${+Date.now()}`)
  }
})

const upload = multer({ storage });
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.sendFile('public/index.html');
});

app.post('/upload', upload.single('upload'), (req, res) => {
  res.send('file uploaded')
});

app.listen(3000, () => console.log('server started'));
```

MemoryStorage

`memoryStorage` stores files in memory as a `Buffer` object and doesn't take any option.

For example, we can use it as follows:

```
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const storage = multer.memoryStorage();
const upload = multer({ storage })
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));
```

```
app.get('/', (req, res) => {
  res.sendFile('public/index.html');
});

app.post('/upload', upload.single('upload'), (req, res) => {
  console.log(req.file);
  res.send('uploaded');
});

app.listen(3000, () => console.log('server started'));
```

Then we get the `buffer` property in `req.file` with the content of the upload file as the value of it.

Photo by Ludovic Charlet on Unsplash

Upload Limits

The `limits` object specifies the size limits of the following optional properties:

- `fieldNameSize` — maximum field name size. Defaults to 100 bytes
- `fieldSize` — maximum field value size. Defaults to 1MB
- `fields` — the maximum number of non-file fields. Defaults to `Infinity`
- `fileSize` — maximum file size in bytes. Defaults to `Infinity`
- `files` — maximum of file fields. Defaults to `Infinity`
- `parts` — the maximum number of parts (fields and files). Defaults to `Infinity`
- `headerPairs` — the maximum number of header key-value pairs to parse. Defaults to 2000.

This is useful for preventing denial of service attacks.

We can set the limits as follows:

```
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const upload = multer({
  limits: {
    fieldSize: 1024 * 512,
    fieldNameSize: 200
  },
  dest: './uploads/'
});
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.sendFile('public/index.html');
});

app.post('/upload', upload.single('upload'), (req, res) => {
  res.send('file uploaded')
});
```

```
app.listen(3000, () => console.log('server started'));
```

We set the field size limit to 512 KB and field name size to 200 bytes in the code above.

Controlling the Files to Process

The `fileFilter` field is a function that lets us control which files should be uploaded and which should be skipped.

For example, we can throw an error if the file uploaded doesn't have the MIME-type `image/png` and then handle the error in our route as follows:

```
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const upload = multer({
  fileFilter: (req, file, cb) => {
    if (file.mimetype === 'image/png') {
      cb(null, true);
    }
    else {
      cb(new multer.MulterError('not a PNG'));
    }
  },
  dest: './uploads/'
})
  .single('upload')
const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.sendFile('public/index.html');
});

app.post('/upload', (req, res) => {
  upload(req, res, (err) => {
    if (err instanceof multer.MulterError) {
      res.send('file not uploaded since it\'s not a PNG');
    }
    else {
      res.send('file uploaded');
    }
  })
});

app.listen(3000, () => console.log('server started'));
```

First, we have the file type check by setting a function with the `fileFilter` function:

```
const upload = multer({
  fileFilter: (req, file, cb) => {
    if (file.mimetype === 'image/png') {
      cb(null, true);
    }
    else {
      cb(new multer.MulterError('not a PNG'));
    }
  },
  dest: './uploads/'
})
  .single('upload')
```

Then in our `/upload` route, we have:

```
app.post('/upload', (req, res) => {
  upload(req, res, (err) => {
    if (err instanceof multer.MulterError) {
      res.send('file not uploaded since it\'s not a PNG');
    }
    else {
      res.send('file uploaded');
    }
  })
});
```

to catch the `MulterError` and respond accordingly. If the file is a PNG, then it's uploaded. Otherwise, an error is thrown and the error won't be uploaded.

In either case, we send a response indicating if the file was uploaded.

Conclusion

Multer has lots of options for us to control how file upload is done. We can check the file type, set the destination, control how much we can upload, etc.

Also, we can catch errors in our routes by using the `upload` function inside our route handler instead of using it as a middleware.

We can also check for file information with `req.file` for single file upload and the `req.files` array for multiple file upload.

Finally, we can change where files are stored by changing the destination and storage type.

JavaScript Programming Software Development Web Development Technology