Uploading files using Node.js and Multer

February 19, 2021 · 6 min read

Multer makes the otherwise painstaking process of uploading files in Node much easier. In this article, we'll learn the purpose of Multer in handling files in submitted forms. We will also build a mini-app with a frontend and backend to test uploading a file.

Managing user inputs

Web applications receive all different types of input from users, including text, graphical controls (like checkboxes or radio buttons), and files such as images, videos, and other media. Each of these inputs on forms are submitted to a server that processes the inputs, uses the inputs in some way (perhaps saving it somewhere else), and gives the frontend a "success" or "failed" response.

When submitting forms that contain text inputs, the server (in our case, Node) has less work to do. Using Express, you can seamlessly grab all the entered inputs in the `req.body` object. For files, however, it's a bit more complex. Files require more processing, which is where Multer comes in.

Encoding and uploading forms

All forms include an `enctype` attribute which specifies how data should be encoded by the browser before sending to the server. The default value is `application/x-www-form-urlencoded` , which supports alphanumeric data. The other encoding type is `multipart/form-data` , which involves uploading files through forms.

Here are two ways to upload forms with `multipart/form-data` encoding. The first is by using the `enctype` attribute:

```
<form action='/upload_files' enctype='multipart/form-data'>
...
</form>
```

This sends the form-data to the `/upload_files` path of your application.

The second is by using the `FormData` API. The `FormData` API allows us to build a `multipart/form-data` form with key-value pairs that can be sent to the server. Here's how it's used:

```
const form = new FormData()
form.append('name', "Dillion")
form.append('image', <a file>)
```

On sending such forms, it becomes the server's responsibility to correctly parse the form and execute the final operation on the data.

Multer: an overview

Multer is a middleware designed to handle `multipart/form-data` in forms. It is similar to the popular Node.js `body-parser` middleware for form submissions, but differs in that it supports multipart data.

Multer only processes `multipart/form-data` forms. It does the work of `body-parser` by attaching the values of text fields in the `req.body` object and also creates a new object `req.file` or `req.files` (for multiple files) which holds information about those files. From the file object, you can pick whatever information is required to post the file to a media management API like Cloudinary.

Now that we understand the importance of Multer, we will build a small app that shows how a frontend app sends three different files at once in a form, and how Multer is able to process the files on the backend to make them available for further use.

Building an app with Multer support

We will start by building the frontend using vanilla HTML, CSS and JS. Of course, you can easily use any framework to do the same.

Creating our frontend

To follow along, first create a folder called `file-upload-example`. Within this file, create another folder called `frontend`. In the frontend folder, we will have three standard files: `index.html`, `styles.css` and `script.js`.

Here are the codes:

```
<!-- index.html -->
<body>
    <div class="container">
        <h1>File Upload</h1>
        <form id='form'>
            <div class="input-group">
                <label for='name'>Your name</label>
                <input name='name' id='name' placeholder="Enter your name"
/>
            </div>
            <div class="input-group">
                <label for='files'>Select files</label>
                <input id='files' type="file" multiple>
            </div>
            <button class="submit-btn" type='submit'>Upload</button>
        </form>
    </div>
    <script src='./script.js'></script>
</body>
```

In the section above, notice that we have created a label and input for “Your Name” as well as “Select Files” (we also added an “Upload” button).

Next, we will add the CSS:

```
/* style.css */
body {
    background-color: rgb(6, 26, 27);
}
* {
    box-sizing: border-box;
}
.container {
    max-width: 500px;
    margin: 60px auto;
}
.container h1 {
    text-align: center;
    color: white;
}
form {
    background-color: white;
    padding: 30px;
}
form .input-group {
```

Here's a screenshot of the webpage:

As you can see, the form we have created takes two inputs: `name` and `files` . The `multiple` attribute specified in the `files` input enables multiple selection of files.

Next, we will send the form to the server, using the code below:

```
// script.js
const form = document.getElementById("form");

form.addEventListener("submit", submitForm);

function submitForm(e) {
    e.preventDefault();
    const name = document.getElementById("name");
    const files = document.getElementById("files");
    const formData = new FormData();
    formData.append("name", name.value);
        for(let i =0; i < files.files.length; i++) {
            formData.append("files", files.files[i]);
    }
    fetch("http://localhost:5000/upload_files", {
        method: 'post',
        body: formData
    })
        .then((res) => console.log(res))
        .catch((err) => ("Error occured", err));
```

When we use `script.js` , there are several very important things that must happen. First, we will get the form element from the DOM and add a submit event to it. Upon submitting, we use `preventDefault` to prevent the default action that the browser would take when a form is submitted (which would normally be redirecting to the value of the `action` attribute). Next, we get the `name` and `files` input element from the DOM and create `formData.`

From here, we will append the value of the name input using a key of `name` to the `formData` . Then, we dynamically add the multiple files selected to the `formData` using a key of `files` (Note: if we are only concerned with a single file, we can append only this: `files.files[0`). Finally, we will add a POST request to `http://localhost:5000/upload_files` which is the API on the backend that we will build in the next section.

Setting up the server

For our demo, we will build our backend using Node.js and Express. We will set up a simple API at `upload_files` and start our server on `localhost:5000`. The API will receive a POST request which contains the inputs from the submitted form.

To use Node for our server, we'll need to setup a basic Node project. In the root directory of the project in the terminal (at `file-upload-example`), run the following:

```
npm init -y
```

This creates a basic `package.json` with some information about your app.

Next, we'll install the required dependency, which for our purposes is `express`:

```
npm i --save express
```

Next, create a `server.js` file and add the following code:

```
// server.js
const express = require("express");

const app = express();
app.use(express.json());

app.post("/upload_files", uploadFiles);
function uploadFiles(req, res) {
    console.log(req.body);
}
app.listen(5000, () => {
    console.log(`Server started...`);
});
```

`express` contains the `bodyParser` object which, as we mentioned earlier, is a middleware for populating `req.body` with the submitted inputs of a form. Calling `app.use(express.json())` executes the middleware on every request made to our server.

The API is set up with `app.post('/upload_files', uploadFiles)` ; `uploadFiles` is the API controller. As seen above, we are only logging out `req.body` , which should be populated by `epxress.json()` . We will test this out in the example below.

Running `body-parser` in Express

In your terminal, run `node server` to start the server. If done correctly, you will see the following in your terminal:

If all looks correct, you can now open your frontend app in your browser. Fill in both inputs in the frontend (name and files) and click submit. On your backend, you should see the following:

This means that the `req.body` object is empty. This is to be expected because, if you'll recall, `body-parser` does not support `multipart` data. Instead, we'll use Multer to parse the form.

Install and configure Multer

Install `multer` by running the following in your terminal:

```
npm i --save multer
```

To configure, add the following to the top of `server.js` :

```
const multer = require("multer");
const upload = multer({ dest: "uploads/" });
...
```

Although `multer` has many other configurations, the only one we are interested in for our purposes is the `dest` property, which specifies the directory where `multer` will save the encoded files.

Next, we will use `multer` to intercept incoming requests on our API and parse the inputs to make them available on the `req` object. To do this, run the following:

```
app.post("/upload_files", upload.array("files"), uploadFiles);

function uploadFiles(req, res) {
    console.log(req.body);
    console.log(req.files);
    res.json({ message: "Successfully uploaded files" });
}
```

For handling multiple files, use `upload.array`. For a single file, use `upload.single`. Note that the "files" argument depends on the name of the input specified in the `formData`.

`multer` will add the text inputs to `req.body` and add the files sent to the `req.files` array. To see this at work in the terminal, enter text and select multiple images on the frontend, then submit and check the logged results in your terminal.

As you will see in the example below, I entered "Images" in the text input and selected a PDF, an SVG, and a JPEG file. Here's a screenshot of the logged result:

For reference, if you want to upload to a storage service like Cloudinary, you will have have to send the file directly from the uploads folder. The `path` property shows the path to the file.

For all of the source code used in this example, visit this repository.

Recap

For text inputs alone, the `bodyParser` object used inside of `express` is enough to parse those inputs. They make the inputs available as a key value pair in the `req.body` object. Where `multer` comes in is when forms contain `multipart` data which includes text inputs and files. The body parser library cannot handle such forms.

With `multer` , you can handle single or multiple files in addition to text inputs sent through a form. Remember that you should only use `multer` when you're sending files through forms, because `multer` cannot handle any form that isn't multipart.

In this article, we've seen a brief of form submissions, the benefits of body parsers on the server and the role that multer plays in handling form inputs. We also built a small application using Node.js and Multer to see a file upload process.

For next steps, you can look at uploading to Cloudinary from your server using the Upload API Reference.

200's only Monitor failed and slow network requests in production

Deploying a Node-based web app or website is the easy part. Making sure your Node instance continues to serve resources to your app is where things get tougher. If you're interested in ensuring requests to the backend or third party services are successful, try LogRocket. https://logrocket.com/signup/

LogRocket is like a DVR for web apps, recording literally everything that happens on your site. Instead of guessing why problems happen, you can aggregate and report on problematic network requests to quickly understand the root cause.

LogRocket instruments your app to record baseline performance timings such as page load time, time to first byte, slow network requests, and also logs Redux, NgRx, and Vuex actions/state. Start monitoring for free.

Share this:

Twitter Reddit LinkedIn Facebook

Dillion Megida Follow

I'm a Frontend Engineer and Technical Writer based in Nigeria.

#node

Leave a Reply

Enter your comment here...