Advisory boards aren't only for executives. Join the LogRocket Content Advisory Board today →

LogRocket
Frontend Analytics

Mar 10, 2022 · 6 min read

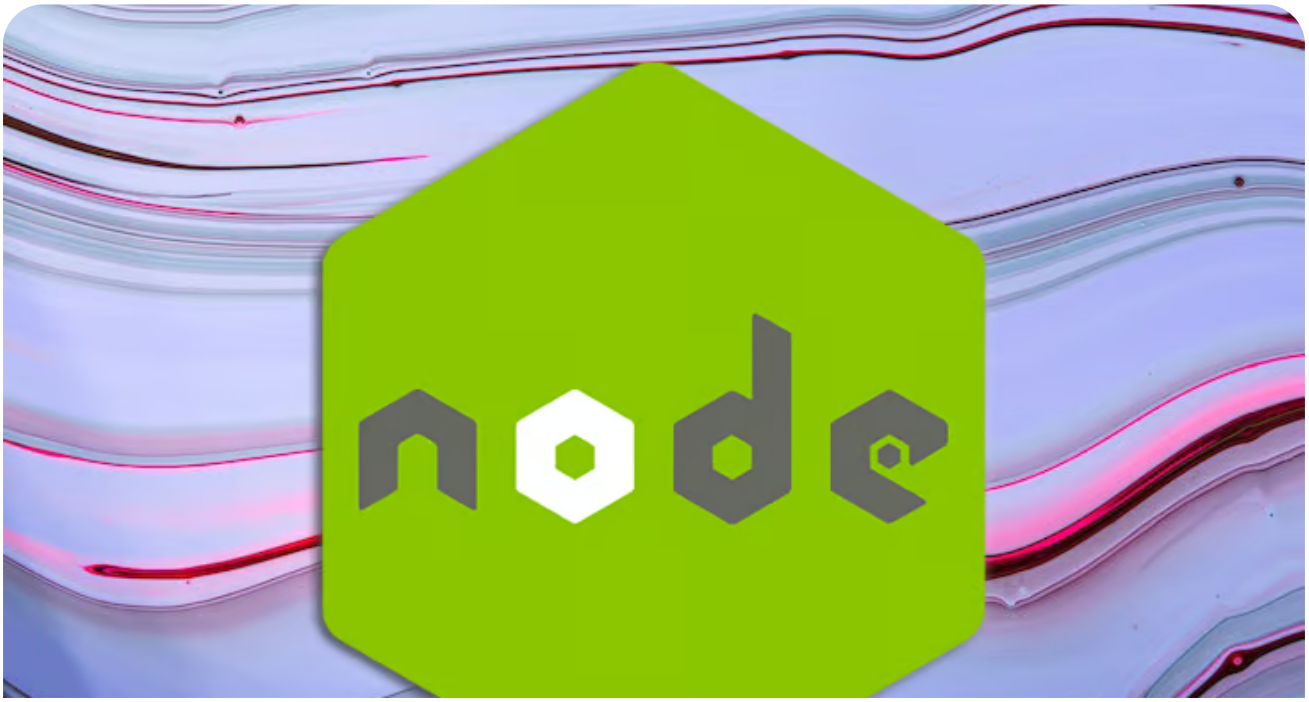
Multer: Easily upload files with Node.js and Express



Dillion Megida
I'm a frontend engineer and technical writer based in Nigeria.

Table of contents

***Editor's note**: This article was last updated 24 March 2022 to reflect updates to Node.js and the `body-parser` library.*





Multer is a Node.js middleware for handling `multipart/form-data` that makes the otherwise painstaking process of uploading files in Node.js much easier. In this article, we'll learn the purpose of Multer in handling files in submitted forms. We'll also explore Multer by building a mini app with a frontend and backend to test uploading a file. Let's get started!

Table of contents

- Managing user inputs in forms
- Encoding and uploading forms with Multer
- Multer: an overview
  - Building an app with Multer support
  - Creating our frontend
  - Install and configure Multer
- Conclusion

Managing user inputs in forms

Web applications receive all different types of input from users, including text, graphical controls like checkboxes or radio buttons, and files, like images, videos, and other media.

In forms, each of these inputs are submitted to a server that processes the inputs, uses them in some way, perhaps saving them somewhere else, then gives the frontend a `success` or `failed` response.

When submitting forms that contain text inputs, the server, Node.js in our case, has less work to do. Using Express, you can easily grab all the inputs entered in the `req.body` object. However, submitting forms with files is a bit more complex because they require more processing, which is where Multer comes in.

Encoding and uploading forms with Multer

All forms include an `enctype` attribute, which specifies how data should be encoded by the browser before sending it 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.

There 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>
```

The code above 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`, which is built into Express middleware for form submissions. But, Multer differs in that it supports multipart data, only processing `multipart/form-data` forms.

Multer does the work of `body-parser` by attaching the values of text fields in the `req.body` object. Multer also creates a new object for multiple files, either `req.file` or `req.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'll build a small sample app to show how a frontend app can send three different files at once in a form, and how Multer is able to process the files on the backend, making them available for further use.

Building an app with Multer support

We'll start by building the frontend using vanilla HTML, CSS, and JavaScript. Of course, you can easily use any framework to follow along.

Creating our frontend

First, create a folder called `file-upload-example`, then create another folder called `frontend` inside. In the frontend folder, we'll have three standard files, `index.html`, `styles.css`, and `script.js`:

```
&lt;!-- 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 yc
            </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</buttc
        </form>
    </div>
    <script src='./script.js'></script>
</body>
```

Notice that we've created a label and input for `Your Name` as well as `Select Files`. We also added an `Upload` button.

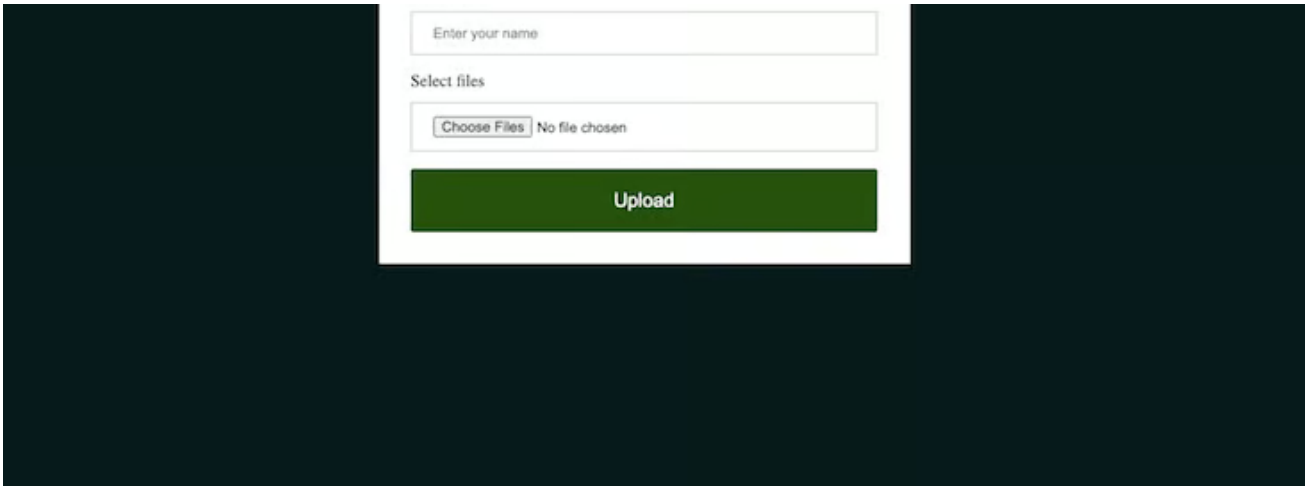
Next, we'll add the CSS for styling:

```
/* 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 {
    margin-bottom: 15px;
}
form label {
    display: block;
    margin-bottom: 10px;
}
form input {
    padding: 12px 20px;
    width: 100%;
    border: 1px solid #ccc;
}
.submit-btn {
    width: 100%;
    border: none;
    background: rgb(37, 83, 3);
    font-size: 18px;
    color: white;
    border-radius: 3px;
    padding: 20px;
    text-align: center;
}
```

Below is a screenshot of the webpage so far:





File upload webpage screenshot with CSS

As you can see, the form we created takes two inputs, `name` and `files`. The `multiple` attribute specified in the `files` input enables us to select multiple files.

Next, we'll 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,
        headers: {
          "Content-Type": "multipart/form-data"
```

```
        }
    })
        .then((res) => console.log(res))
        .catch((err) => ("Error occured", err));
}
```

There are several important things that must happen when we use `script.js`. First, we 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'll append the value of the name input using a key of `name` to the `formData`. Then, we dynamically add the multiple files we selected to the `formData` using a key of `files`.



Over 200k developers use LogRocket to create better digital experiences

Learn more →

Note: if we're only concerned with a single file, we can append `files.files[0]` .

Finally, we'll add a `POST` request to `http://localhost:5000/upload_files` , which is the API on the backend that we'll build in the next section.

Setting up the server

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

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

```
npm init -y
```

The command above 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 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.use(express.urlencoded({ extended: true }));

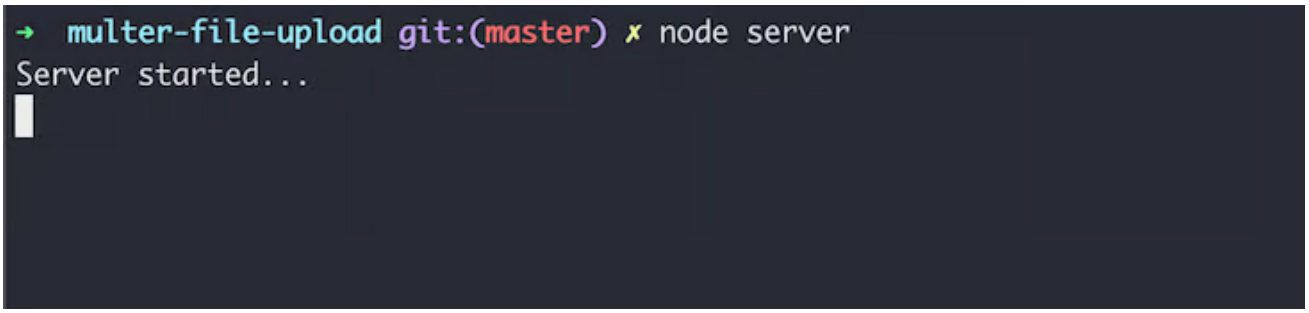
app.post("/upload_files", uploadFiles);
function uploadFiles(req, res) {
    console.log(req.body);
}
app.listen(5000, () => {
    console.log(`Server started...`);
});
```

Express contains the `bodyParser` object, which is a middleware for populating `req.body` with the submitted inputs on 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'll test this out in the example below.

Running `body-parser` in Express

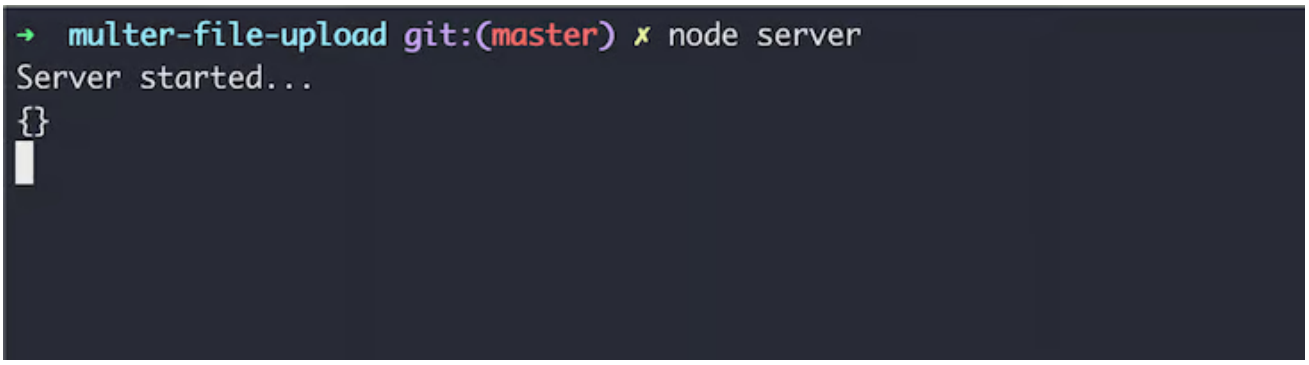
In your terminal, run `node server` to start the server. If done correctly, you'll see the following in your terminal:



Run Node server output start server

You can now open your frontend app in your browser. Fill in both inputs in the frontend, the name and files, then click **submit**. On your backend, you should see

the following:



Backend visual name and file inputs

The code in the image above means that the `req.body` object is empty, which is to be expected. If you'll recall, `body-parser` doesn't support `multipart` data. Instead, we'll use Multer to parse the form.

Install and configure Multer

Install Multer by running the following command in your terminal:

```
npm i multer
```

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

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

Although Multer has many other configuration options, we're only interested in the `dest` property for our project, which specifies the directory where Multer will save the encoded files.

Next, we'll use Multer to intercept incoming requests on our API and parse the inputs to make them available on the `req` object:

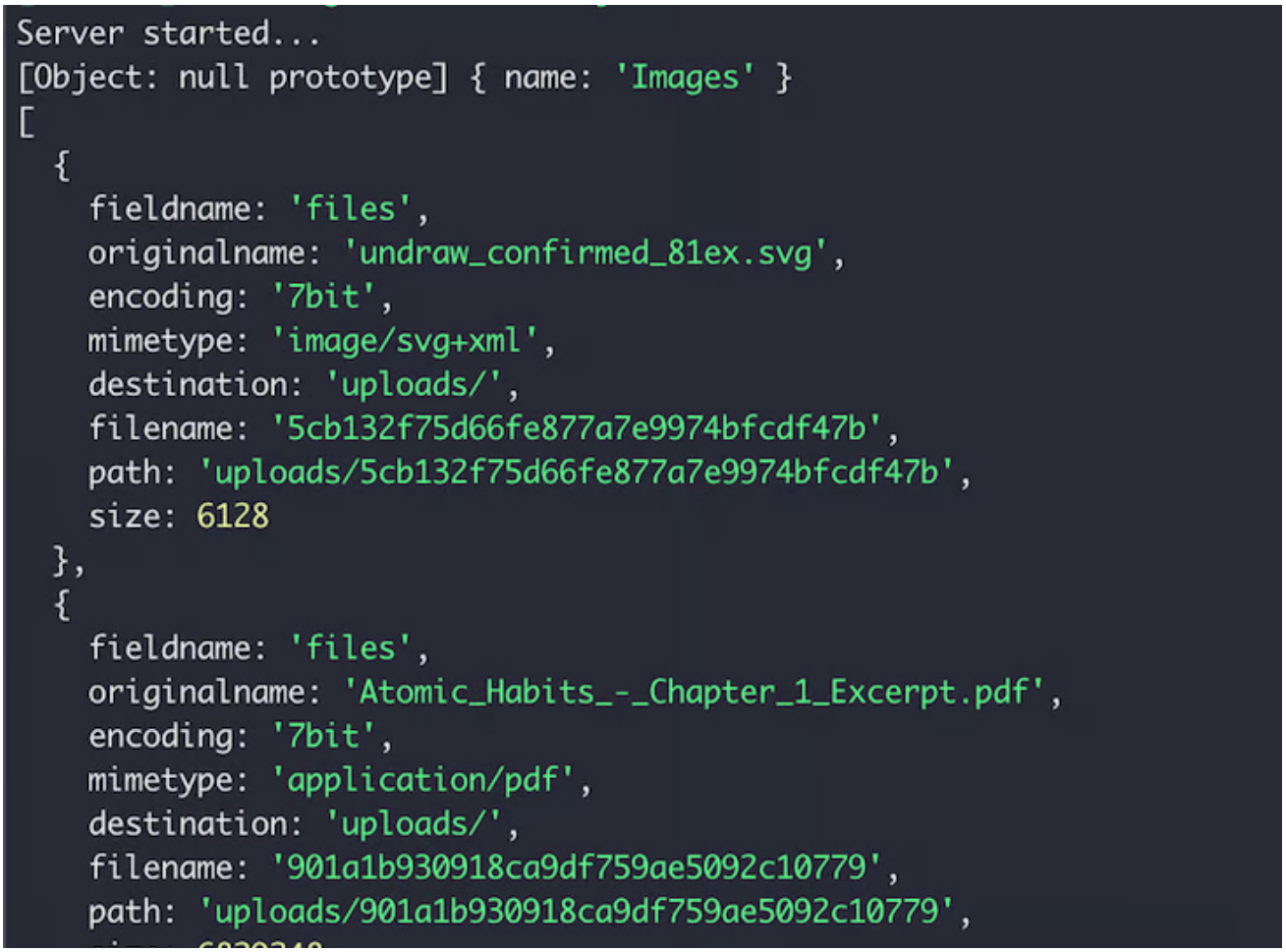
```
app.post("/upload_files", upload.array("files"), uploadFiles);
```

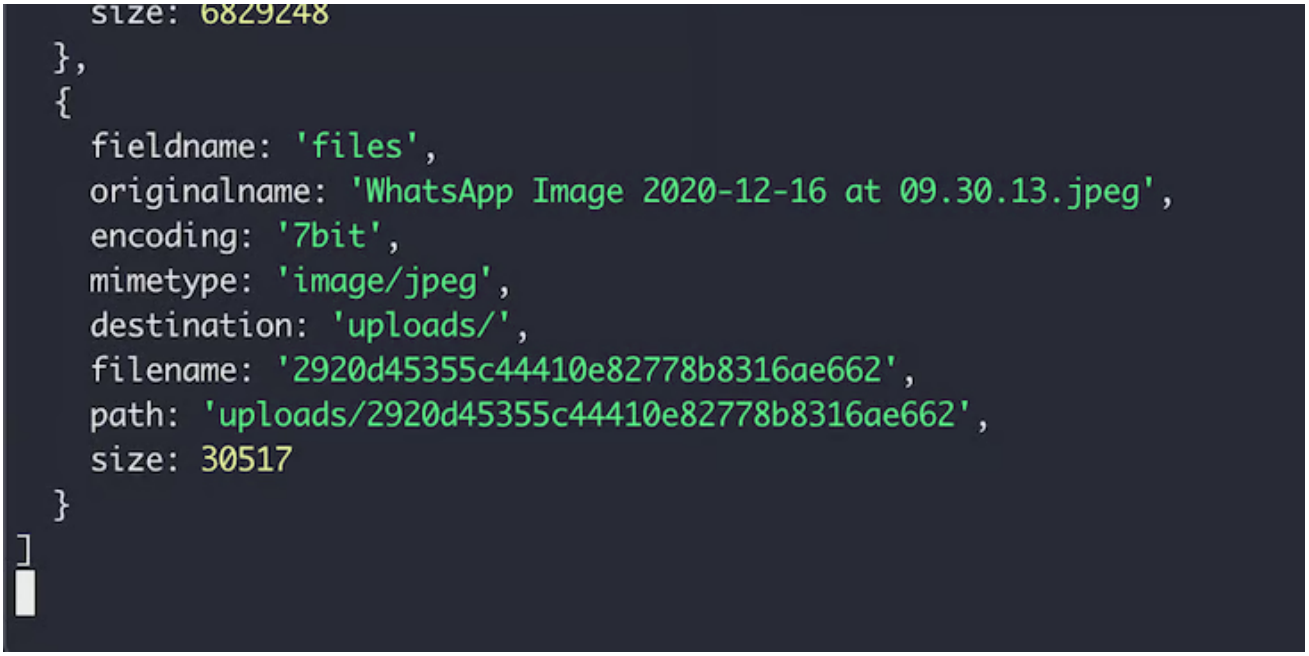
```
function uploadFiles(req, res) {
    console.log(req.body);
    console.log(req.files);
    res.json({ message: "Successfully uploaded files" });
}
```

To handle 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 `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 can see in the example below, I entered `Images` in the `text` input and selected a PDF, an SVG, and a JPEG file. Below is a screenshot of the logged result:





Logged results screenshot images text input

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.

Conclusion

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. Multer comes in handy when forms contain `multipart` data that includes text inputs and files, which the `body-parser` library cannot handle.

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.

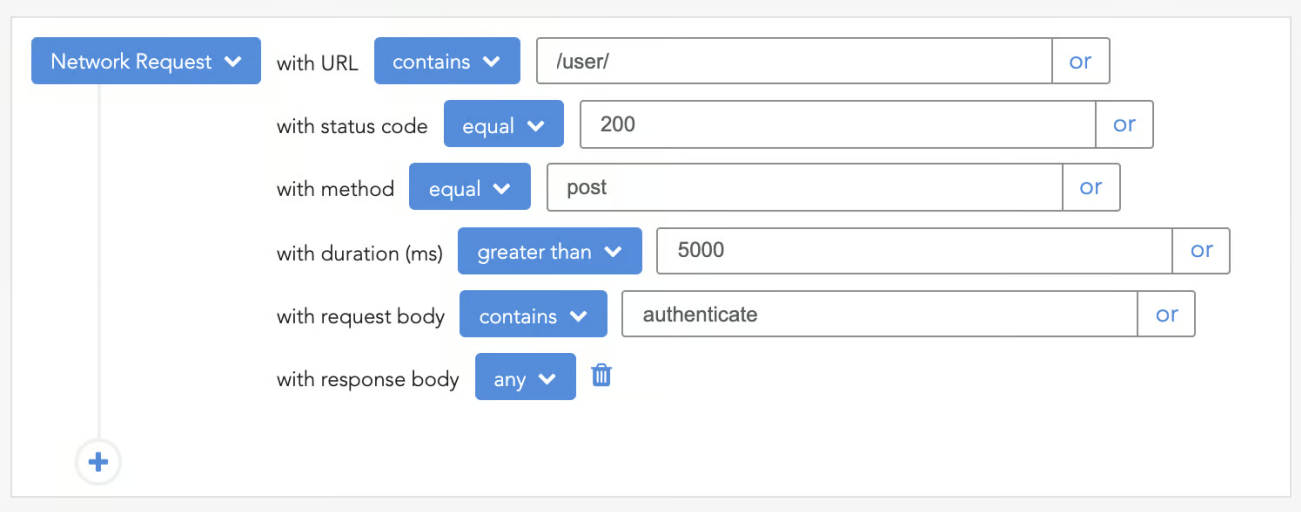
In this article, we've seen a brief of form submissions, the benefits of body parsers on the server and the role that Multer plays in handling form inputs. We also built a small application using Node.js and Multer to see a file upload process.

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

the Upload API Reference. I hope you enjoyed this article! Happy coding!

200s 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.



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

Share this:

#code

ır 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.

Stop guessing about your digital

experience with LogRocket

Get started for free

Recent posts:



Mastering the modulo operator in JavaScript: A complete guide

We investigate how the modulo operator works in JavaScript and explore several real-life applications, like cyclic patterns and animations.



Timonwa Akintokun
Dec 8, 2023 · 7 min read



Leader Spotlight: Building with grit with Ken Pickering

Ken Pickering discusses the challenges associated with uncertainty and shares insights on scaling data structures and engineering teams.



Jessica Srinivas
Dec 8, 2023 · 7 min read



Understanding SolidJS props: A complete guide

Let's see how Solid props work to promote component reusability, exploring basic to advanced concepts for a complete understanding.



Temitope Oyedele
Dec 7, 2023 · 11 min read

View all posts

6 Replies to "Multer: Easily upload files with Node.js and Express"

Brandon says: Reply

May 10, 2022 at 10:27 am

Using this tutorial, I am seeing that my images files are correctly coming through in the request body within an array "images" (created on front end with each file from the file input's filelist). However, Multer is returning an empty array for within req.files

Josh Brown says: Reply

August 4, 2022 at 7:14 pm

You must remove the "Content-Type" header or else the request fails with a 500. And

you have to have the response body without a comma at the end:
{
method: 'POST',
body: formData
}



Kennedy Chidi says: Reply

September 1, 2022 at 11:14 am

Hello I'm getting an error at the server saying 'unexpected end of form" please help

Eleventy vs. Next.js for static site generation

phamquoctrung says: Reply

September 29, 2022 at 4:17 am

When I open the frontend app in browser using URL=http://localhost:5000 /upload_files, there is error Cannot GET /upload_files. Can you pls help?

Hinata says: Reply

March 14, 2023 at 7:18 am

i don't know why suddenly my multer stops working, i use single upload in server and form data from front end, any suggestion ?

hinata says: Reply

March 14, 2023 at 7:20 am

thank you for this useful tutorial, it really helps me upload picture to server, hope to find more interesting thing here

Leave a Reply