Drop Zone

# Create a drag-and-drop component with the HTML Drag and Drop API

Creating a custom code for the Drag & Drop file features for the Auction page. The reason for this is there are many pre-build ones but due to the lack of either documentation or customization they were not used.

Our drag-and-drop component will do seven things:

1. Listen for drag and drop
2. Detect when a file is dropped on the drop zone
3. Display image name and file type
4. Validate dropped images
5. Delete images with unsupported file types
6. Preview images with valid file types
7. Upload images

This component uses the [HTML Drag and Drop API](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API). The interface enables applications to use drag-and-drop features in browsers.

## **Getting started**

We’ll use React for the UI and Axios to make a post request when uploading the images. I’ll assume Node.js is already installed on your computer.

Let's start from Scratch.

Open the terminal, navigate to the directory where you want to add your project, and type the following.

npx create-react-app react-file-dropzone

Remove the contents of the return inside the App function, then remove all imports except for React and App.css imports. Your App.js component should look as follows.

import React from 'react';

import './App.css';

function App() {

return (

);

}

export default App;

Add div elements to the return of the App function.

return (

<div>

<p className="title">React Drag and Drop Image Upload</p>

<div className="content">

</div>

</div>

);

Remove the contents of the App.css file and add the following CSS code.

.title {

font-size: 2rem;

text-align: center !important;

margin-top: 10%;

color: #4aa1f3;

font-weight: bold;

}

.content {

background-color: white;

min-height: 100vh;

display: flex;

flex-direction: column;

align-items: center;

justify-content: center;

}

## **Create the Dropzone component**

To create a Dropzone component, create a folder called dropzone inside the src folder and add two files: DropZone.js and DropZone.css.

Add an arrow function called DropZone inside the DropZone.js file and export function as default. Set the parent element to empty tags.

import React from 'react';

const DropZone = () => {

return (

<>

</>

)

}

export default DropZone;

Next, import the DropZone component into the App.js file.

import DropZone from "./dropzone/DropZone";

Add the component as a child of the div element with class name content.

return (

<div>

<p className="title">React Drag and Drop Image Upload</p>

<div className="content">

<DropZone />

</div>

</div>

);

Now back to the DropZone component. Add a div element with class name container. Inside the container component, add a div with class name drop-container.

<div className="container">

<div className="drop-container">

</div>

</div>

Add this CSS style into the DropZone.css and import the file inside the component.

.container {

transform: translateY(-100%);

}

.container p {

color: red;

text-align: center;

}

.drop-container {

display: flex;

align-items: center;

justify-content: center;

margin: 0;

width: 800px;

height: 200px;

border: 4px dashed #4aa1f3;

}

Inside the div with class name drop-container, add the following elements.

<div className="drop-message">

<div className="upload-icon"></div>

Drag & Drop files here or click to upload

</div>

.upload-icon {

width: 50px;

height: 50px;

background: url(../images/upload.png) no-repeat center center;

background-size: 100%;

text-align: center;

margin: 0 auto;

padding-top: 30px;

}

.drop-message {

text-align: center;

color: #4aa1f3;

font-family: Arial;

font-size: 20px;

}

You can get the icon images used in the guide from the “libertys\src\Images\assets”.

## **Drag events**

[HTML Drag-and-Drop](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API) uses the DOM event model and drag events inherited from mouse events.

A drag operation begins when a user selects an item from the OS, drags the item to a droppable element, then releases the dragged item. During drag operations, several events are fired. Some events might fire multiple times.

The drag-and-drop API defines eight events: four events for the draggable element and four events for the droppable element. For this guide, we’ll only need the four events for the droppable element. Each drag event type has an associated event handler. The events are:

* dragenter — A dragged item enters a valid drop target (ondragenter)
* dragleave — A dragged item leaves a valid drop target (ondragleave)
* dragover — A dragged item is dragged over a valid drop target every few hundred milliseconds (ondragover)
* drop — an item is dropped on a valid drop target (ondrop)

## **Drag-and-drop functionality**

So far, we’ve specified the drop region for our files, but there’s no valid region in which to drop them. When a file is dragged into a browser from the OS, the browser will attempt to open and display it by default. If you want to allow a drop, you must prevent the default handling of the event handlers.

On the div element with class name drop-container, add four event methods.

onDragOver={dragOver}

onDragEnter={dragEnter}

onDragLeave={dragLeave}

onDrop={fileDrop}

The LHS is the event handler method and the RHS are the methods to handle event handlers. The div element should look as follows.

<div className="drop-container"

onDragOver={dragOver}

onDragEnter={dragEnter}

onDragLeave={dragLeave}

onDrop={fileDrop}

>

...

</div>

You can define the methods for handling the events.

const dragOver = (e) => {

e.preventDefault();

}

const dragEnter = (e) => {

e.preventDefault();

}

const dragLeave = (e) => {

e.preventDefault();

}

const fileDrop = (e) => {

e.preventDefault();

const files = e.dataTransfer.files;

console.log(files);

}

The e.dataTransfer is a Data Transfer object that holds the data that is being dragged during a drag-and-drop operation. It may hold one or more data items. e.dataTransfer.files contains the dragged local files as a FileList.

Next, we need to handle the files from the FileList by validating the file type, checking the file size, and display the file name.

## **Validate and check file sizes**

In this guide, we’re only adding image files. That’s because the storage platform that we’ll use later to store the uploaded files only allows mostly image files. Let’s validate the files.

First, create a method called handleFiles and add it to fileDrop with files as a parameter.

const handleFiles = (files) => {

}

Add to fileDrop

const fileDrop = (e) => {

e.preventDefault();

const files = e.dataTransfer.files;

if (files.length) {

handleFiles(files);

}

}

Create a new method called validateFile with a parameter called file. The method will return a boolean value.

const validateFile = (file) => {

const validTypes = ['image/jpeg', 'image/jpg', 'image/png', 'image/gif', 'image/x-icon'];

if (validTypes.indexOf(file.type) === -1) {

return false;

}

return true;

}

Here we have an array with the [file types](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types). You can add or remove any type. The file parameter from the FileList contains a type property. When using JavaScript’s indexOf method, if the type is not found in the array, it returns -1; otherwise, it returns the index of the value in the array. It returns false if the type is not found or true if it is found.

Let’s use the validateFile method inside handleFiles. The FileList is an array, and we can loop through the array.

for(let i = 0; i < files.length; i++) {

if (validateFile(files[i])) {

// add to an array so we can display the name of file

} else {

// add a new property called invalid

// add to the same array so we can display the name of the file

// set error message

}

}

Import the useState hook from react.

import React, { useState } from 'react';

Add two state variables inside the DropZone function.

const [selectedFiles, setSelectedFiles] = useState([]);

const [errorMessage, setErrorMessage] = useState('');

selectedFiles has a type of array while errorMessage has a type of string.

Inside the else part of the handleFiles method, add a new variable to the file object so it can be used to identify invalid files.

// add a new property called invalid

files\[i\]['invalid'] = true;

Next, update the selectedFiles array with the new object that contains an invalid property.

// add to the same array so we can display the name of the file

setSelectedFiles(prevArray => [...prevArray, files[i]]);

Here we’re using a callback inside the setSelectedFiles method so we can get a quick update to the array.

Add the error message.

// set error message

setErrorMessage('File type not permitted');

Before we can try this out, we need to add a div to display the files inside the selectedFiles array. Add the following after the div with class name drop-container.

<div className="file-display-container">

<div className="file-status-bar">

<div>

<div className="file-type-logo"></div>

<div className="file-type">png</div>

<span className="file-name">test-file.png</span>

<span className="file-size">(20.5 KB)</span> {<span className='file-error-message'>(File type not permitted)</span>}

</div>

<div className="file-remove">X</div>

</div>

</div>

Add these styles to the CSS file for DropZone.

.file-display-container {

position: fixed;

width: 805px;

}

.file-status-bar{

width: 100%;

vertical-align:top;

margin-top: 10px;

margin-bottom: 20px;

position: relative;

line-height: 50px;

height: 50px;

}

.file-status-bar > div {

overflow: hidden;

}

.file-type {

display: inline-block!important;

position: absolute;

font-size: 12px;

font-weight: 700;

line-height: 13px;

margin-top: 25px;

padding: 0 4px;

border-radius: 2px;

box-shadow: 1px 1px 2px #abc;

color: #fff;

background: #0080c8;

text-transform: uppercase;

}

.file-name {

display: inline-block;

vertical-align:top;

margin-left: 50px;

color: #4aa1f3;

}

.file-error {

display: inline-block;

vertical-align: top;

margin-left: 50px;

color: #9aa9bb;

}

.file-error-message {

color: red;

}

.file-type-logo {

width: 50px;

height: 50px;

background: url(../images/generic.png) no-repeat center center;

background-size: 100%;

position: absolute;

}

.file-size {

display:inline-block;

vertical-align:top;

color:#30693D;

margin-left:10px;

margin-right:5px;

margin-left: 10px;

color: #444242;

font-weight: 700;

font-size: 14px;

}

.file-remove {

position: absolute;

top: 20px;

right: 10px;

line-height: 15px;

cursor: pointer;

color: red;

margin-right: -10px;

}

You can see how the files will be displayed. The file type will be extracted and displayed, as will the file name and size. The error message will only be displayed for invalid files.

Now that we have placeholders for the files to be displayed on the page, let’s use the selectedFiles array inside the template.

Create a method called fileSize. This will take in a size parameter. The file object from FileList contains a size property.

const fileSize = (size) => {

if (size === 0) return '0 Bytes';

const k = 1024;

const sizes = ['Bytes', 'KB', 'MB', 'GB', 'TB'];

const i = Math.floor(Math.log(size) / Math.log(k));

return parseFloat((size / Math.pow(k, i)).toFixed(2)) + ' ' + sizes[i];

}

We return the string 0 Bytes if the size is zero. 1 KB is equivalent to 1024 bytes. We calculate the natural log of the size by dividing by the natural log of bytes value. Math.floor returns an integer. The return value of the function is the size divided by the value of k to the power of i with the sizes value appended.

Add a method for getting the file type from the file name.

const fileType = (fileName) => {

return fileName.substring(fileName.lastIndexOf('.') + 1, fileName.length) || fileName;

}

The substring() method extracts the characters from a string between two specified indices and returns the new substring. The substring() method returns the characters after the . in the file name.

<div className="file-display-container">

{

selectedFiles.map((data, i) =>

<div className="file-status-bar" key={i}>

<div>

<div className="file-type-logo"></div>

<div className="file-type">{fileType(data.name)}</div>

<span className={`file-name ${data.invalid ? 'file-error' : ''}`}>{data.name}</span>

<span className="file-size">({fileSize(data.size)})</span> {data.invalid && <span className='file-error-message'>({errorMessage})</span>}

</div>

<div className="file-remove">X</div>

</div>

)

}

</div>

Loop through the selectedFiles array. Use the fileType method by passing the file name.

<span className={`file-name ${data.invalid ? 'file-error' : ''}`}>{data.name}</span>

Check whether the object contains the invalid property added, which would indicate an invalid file. Add the class name file-error.

<span className="file-size">({fileSize(data.size)})</span> {data.invalid && <span className='file-error-message'>({errorMessage})</span>}

We display the file size by using the fileSize method. The error message is displayed next to it if it is an invalid file.

Adding valid files is simple. Inside the if part of the handleFiles method, add:

setSelectedFiles(prevArray => [...prevArray, files[i]]);

You can even drag and drop multiple files at the same time.

## **Remove duplicate valid files**

One drawback of the selectedFiles array is that you can added a particular file multiple times. We don’t want this behavior.

To remove duplicates from the selectedFiles array, add a new useState variable inside the component.

const [validFiles, setValidFiles] = useState([]);

Import the useEffect hook.

import React, { ..., useEffect } from 'react';

Use the JavaScript reduce, find, and concat methods to remove duplicates and add the individual values into the new array validFiles.

useEffect(() => {

let filteredArray = selectedFiles.reduce((file, current) => {

const x = file.find(item => item.name === current.name);

if (!x) {

return file.concat([current]);

} else {

return file;

}

}, []);

setValidFiles([...filteredArray]);

}, [selectedFiles]);

The result of filteredArray is used to update the validFiles array. Now replace the selectedFiles in the HTML map to validFiles.

<div className="file-display-container">

{

validFiles.map((data, i) =>

...

)

}

</div>

You can check to make sure a particular file can be added only once.

## **Removing files**

Before uploading the images, users should have the option to remove images from the list. We already have a button for removing an item from the list.

Add a method called removeFile with the file name as a parameter on the div element with class name file-remove.

<div className="file-remove" onClick={() => removeFile(data.name)}>X</div>

We need to remove the selected file from the validFiles and selectedFiles arrays. Let’s find the index of the file by using the JavaScript findIndex method. Then, use the splice method to remove the item from the arrays and update it with the setValidFiles and setSelectedFiles methods.

const removeFile = (name) => {

// find the index of the item

// remove the item from array

const validFileIndex = validFiles.findIndex(e => e.name === name);

validFiles.splice(validFileIndex, 1);

// update validFiles array

setValidFiles([...validFiles]);

const selectedFileIndex = selectedFiles.findIndex(e => e.name === name);

selectedFiles.splice(selectedFileIndex, 1);

// update selectedFiles array

setSelectedFiles([...selectedFiles]);

}

## **Preview image**

You can add a simple modal to preview an image before all images are uploaded. The preview will only be for valid files. You can remove invalid files by clicking on their file name. Add the modal divs after the div with class name container.

<div className="modal">

<div className="overlay"></div>

<span className="close">X</span>

<div className="modal-image"></div>

</div>

The div element with class name modal-image will display the image.

Add the CSS styles. By default, the modal display is set to none. It will only be displayed when an image name is clicked.

.modal{

z-index: 999;

display: none;

overflow: hidden;

}

.modal .overlay{

width: 100%;

height: 100vh;

background: rgba(0,0,0,.66);

position: absolute;

top: 0;

left: 0;

}

.modal .modal-image{

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%,-50%);

overflow: hidden;

object-fit: cover;

width: 100%;

height: 300px;

background-size: contain;

background-repeat: no-repeat;

background-position: center;

}

.close {

position: absolute;

top: 15px;

right: 35px;

color: #f1f1f1;

font-size: 40px;

font-weight: bold;

transition: 0.3s;

}

Add this on the div inside the element with class name file-status-bar.

onClick={!data.invalid ? () => openImageModal(data) : () => removeFile(data.name)}

<div className="file-display-container">

{

validFiles.map((data, i) =>

<div className="file-status-bar" key={i}>

<div onClick={!data.invalid ? () => openImageModal(data) : () => removeFile(data.name)}>

...

</div>

</div>

)

}

</div>

The openImageModal displays valid files while invalid files are removed when clicked on.

Add the method openImageModal.

const openImageModal = (file) => {

}

Import useRef hook from React. The ref will allow us to display the modal, show the image, and close the modal.

import React, { ..., useRef } from 'react';

Add the following refs variables.

const modalImageRef = useRef();

const modalRef = useRef();

On the modal elements, add the respective refs.

<div className="modal" ref={modalRef}>

<div className="overlay"></div>

<span className="close">X</span>

<div className="modal-image" ref={modalImageRef}></div>

</div>

* modalRef is used to display and hide the modal element and its contents
* modalImageRef displays the image

We need to read the content of the file passed as a parameter into the openImageModal.

Add the FileReader constructor.

const reader = new FileReader();

The [FileReader object](https://developer.mozilla.org/en-US/docs/Web/API/FileReader) enables web applications to asynchronously read the contents of files (or raw data buffers) stored on the user’s computer using File or Blob objects to specify the file or data to read.

Next, set the display of the modal to block using the modalRef.

const reader = new FileReader();

modalRef.current.style.display = "block";

We need a way to read the content of the file using readAsDataURL and add an event handler to be triggered once the reading operation is complete.

const reader = new FileReader();

modalRef.current.style.display = "block";

reader.readAsDataURL(file);

reader.onload = function(e) {

modalImageRef.current.style.backgroundImage = `url(${e.target.result})`;

}

The e.target.result attribute contains a data: URL representing the file’s data. We’ll set that as the background image of the div with ref modalImageRef.

In the CSS style for modal-image, we already set the width, height, and some background properties.

Add an onClick method on the span element with class name close.

<span className="close" onClick={(() => closeModal())}>X</span>

Create a method closeModal. When this method is called, the modal display is set to none and the image background is set to none so as to reset.

const closeModal = () => {

modalRef.current.style.display = "none";

modalImageRef.current.style.backgroundImage = 'none';

}

## **Display upload button**

Let’s add an upload button. The button will only be displayed if all files are valid. If there is at least one invalid file, a message will be displayed instead.

Add a button with class name file-upload-btn as the first element inside the container div.

<button className="file-upload-btn">Upload Files</button>

.file-upload-btn {

color: white;

text-transform: uppercase;

outline: none;

background-color: #4aa1f3;

font-weight: bold;

padding: 8px 15px;

margin-bottom: 5px;

}

To hide and display the upload button, add a new useState variable to hold all invalid files. If the array length is zero, the button is displayed; otherwise, the button is hidden.

const [unsupportedFiles, setUnsupportedFiles] = useState([]);

Use the setUnsupportedFiles the same way we used setSelectedFiles. First, add ther following inside the else part of the handleFiles method and the removeFile method.

const handleFiles = (files) => {

for(let i = 0; i < files.length; i++) {

if (validateFile(files[i])) {

...

} else {

...

...

...

setUnsupportedFiles(prevArray => [...prevArray, files[i]]);

}

}

}

Each invalid file dropped by the user will be added to the array.

const removeFile = (name) => {

// find the index of the item

// remove the item from array

...

...

const unsupportedFileIndex = unsupportedFiles.findIndex(e => e.name === name);

if (unsupportedFileIndex !== -1) {

unsupportedFiles.splice(unsupportedFileIndex, 1);

// update unsupportedFiles array

setUnsupportedFiles([...unsupportedFiles]);

}

}

If the index of the element is found, the item is spliced and unsupportedFiles is updated.

Replace the upload button with these two lines:

{unsupportedFiles.length === 0 && validFiles.length ? <button className="file-upload-btn" onClick={() => uploadFiles()}>Upload Files</button> : ''}

{unsupportedFiles.length ? <p>Please remove all unsupported files.</p> : ''}

* The first line checks whether the unsupportedFiles length is zero and validFiles array has at least one value. Then the button is displayed; otherwise, it’s hidden
* The second line displays the message if there is at least one invalid file.

## **Click to select files**

Before we add the upload functionality, let’s add the click to select image functionality. A hidden input field is added with type file, an onChange event method, and a ref property. An onClick method is added to the drop-container so that when any part of the container is clicked, it triggers the hidden input field by using its ref.

Add an onClick to drop-container.

onClick={fileInputClicked}

Add an input field after the drop-message element.

<input

ref={fileInputRef}

className="file-input"

type="file"

multiple

onChange={filesSelected}

/>

const fileInputRef = useRef();

.file-input {

display: none;

}

Add a method called fileInputClicked with fileInputRef.current.clicked.

const fileInputClicked = () => {

fileInputRef.current.click();

}

Add another method for the filesSelected. The selected files can be obtained from fileInputRef.current.files. We just need to pass it into the handleFiles method.

const filesSelected = () => {

if (fileInputRef.current.files.length) {

handleFiles(fileInputRef.current.files);

}

}

With these methods, we can select multiple files.

## **Upload functionality**

It’s time to add the upload functionality to our Dropzone component. For the purpose of this demo, we’ll use a free service called [imgbb](https://imgbb.com/). Create an account and then obtain an API key from <https://api.imgbb.com/>.

### ⁉ NOTE:

This is still implemented within the code, please replace that to your own server.

Add a method called uploadFiles to the upload button.

<button className="file-upload-btn" onClick={() => uploadFiles()}>Upload Files</button>

Add the uploadFiles method.

const uploadFiles = () => {

}

A modal with a progress bar is displayed when the upload button is clicked. Add these upload modal elements.

<div className="upload-modal" ref={uploadModalRef}>

<div className="overlay"></div>

<div className="close" onClick={(() => closeUploadModal())}>X</div>

<div className="progress-container">

<span ref={uploadRef}></span>

<div className="progress">

<div className="progress-bar" ref={progressRef}></div>

</div>

</div>

</div>

.upload-modal {

z-index: 999;

display: none;

overflow: hidden;

}

.upload-modal .overlay{

width: 100%;

height: 100vh;

background: rgba(0,0,0,.66);

position: absolute;

top: 0;

left: 0;

}

.progress-container {

background: white;

width: 500px;

height: 300px;

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%,-50%);

overflow: hidden;

}

.progress-container span {

display: flex;

justify-content: center;

padding-top: 20px;

font-size: 20px;

}

.progress {

width: 90%;

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%,-50%);

background-color: #efefef;

height: 20px;

border-radius: 5px;

}

.progress-bar {

position: absolute;

background-color: #4aa1f3;

height: 20px;

border-radius: 5px;

text-align: center;

color: white;

font-weight: bold;

}

.error {

color: red;

}

The modal has elements with refs. Add the ref variables as well as the closeUploadModal method.

const uploadModalRef = useRef();

const uploadRef = useRef();

const progressRef = useRef();

* uploadModalRef displays and hides the upload modal
* uploadRef shows messages
* progressRef updates the progress bar

Inside the closeUploadModal method, set the display of uploadModalRef to none.

const closeUploadModal = () => {

uploadModalRef.current.style.display = 'none';

}

In the uploadFiles method, first set the display of uploadModalRef to block. Also, add the string File(s) Uploading... to uploadRef innerHTML.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

}

Since we already have the valid files inside the validFiles array, all we need to do is loop through the array, set the right properties using FormData, and then make the request.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

for (let i = 0; i < validFiles.length; i++) {

}

}

If you go through the [imgbb API](https://api.imgbb.com/), you’ll see that to make the request, a key and image properties are required. The key is the API key you obtained and the image is the file to be uploaded. You can convert to a binary file, base64 data, or a URL for an image.

To set these properties, we’ll use the FormData interface. This tool provides a way to easily construct key/value pairs to represent form fields and their values, which can then be easily sent.

Create a new FormData inside the for loop and append a key with the name image and value validFiles[i]. Then, append another key named key. The value should be your API key.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

for (let i = 0; i < validFiles.length; i++) {

const formData = new FormData();

formData.append('image', validFiles[i]);

formData.append('key', 'add your API key here');

}

}

To make the request, we’ll use Axios because it has a method we can use to get the upload progress. From this, the progress bar value can be calculated and displayed.

Run the following command to install.

npm install axios

Once installed, you can import inside the DropZone component.

import axios from 'axios';

A POST request is required to upload the file(s) to imgbb.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

for (let i = 0; i < validFiles.length; i++) {

const formData = new FormData();

formData.append('image', validFiles[i]);

formData.append('key', 'add your API key here');

axios.post('https://api.imgbb.com/1/upload', formData, {})

.catch(() => {

// If error, display a message on the upload modal

uploadRef.current.innerHTML = `<span class="error">Error Uploading File(s)</span>`;

// set progress bar background color to red

progressRef.current.style.backgroundColor = 'red';

});

}

}

Axios post takes three parameters. The URL, data, and last object is where the event for the upload progress is calculated. You can learn more about the Axios request configuration on [GitHub](https://github.com/axios/axios#request-config). We’re interested in the onUploadProgress method, which facilitates the handling of progress events for uploads.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

for (let i = 0; i < validFiles.length; i++) {

const formData = new FormData();

formData.append('image', validFiles[i]);

formData.append('key', 'add your API key here');

axios.post('https://api.imgbb.com/1/upload', formData, {

onUploadProgress: (progressEvent) => {

console.log(progressEvent);

}

})

.catch(() => {

// If error, display a message on the upload modal

uploadRef.current.innerHTML = `<span class="error">Error Uploading File(s)</span>`;

// set progress bar background color to red

progressRef.current.style.backgroundColor = 'red';

});

}

}

The progressEvent contains the loaded and total properties. These properties will be used to calculated the percentage of the upload progress. The percentage will be set as the width of the progress bar.

To calculate the percentage of the upload progress, divide the loaded value by the total and multiply the result by 100. Then, use Math.floor to get an integer.

const uploadFiles = () => {

uploadModalRef.current.style.display = 'block';

uploadRef.current.innerHTML = 'File(s) Uploading...';

for (let i = 0; i < validFiles.length; i++) {

const formData = new FormData();

formData.append('image', validFiles[i]);

formData.append('key', 'add your API key here');

axios.post('https://api.imgbb.com/1/upload', formData, {

onUploadProgress: (progressEvent) => {

const uploadPercentage = Math.floor((progressEvent.loaded / progressEvent.total) \* 100);

progressRef.current.innerHTML = `${uploadPercentage}%`;

progressRef.current.style.width = `${uploadPercentage}%`;

if (uploadPercentage === 100) {

uploadRef.current.innerHTML = 'File(s) Uploaded';

validFiles.length = 0;

setValidFiles([...validFiles]);

setSelectedFiles([...validFiles]);

setUnsupportedFiles([...validFiles]);

}

}

})

.catch(() => {

// If error, display a message on the upload modal

uploadRef.current.innerHTML = `<span class="error">Error Uploading File(s)</span>`;

// set progress bar background color to red

progressRef.current.style.backgroundColor = 'red';

});

}

}

The uploadPercentage value is displayed on the progress bar. The width is updated using the uploadPercentage value. Once the upload is complete, change the string on the modal to File(s) Uploaded and then empty all arrays. This is all you need to calculate and display the upload progress.

The first part we should try out is the upload failure. To see that, disable the validate file functionality and then upload a file that is not an image.

Depending on your internet speed, the upload may be so quick that you won’t be able to see the upload progress. If you upload a large file, it will likely take longer.

### PS.

Make sure drag and drop isn’t hurting performance.