DAY 11

We want to implement an Image upload system in our application. So we have a system or API to upload an image for some data.

Let's suppose we are storing Student Details

Student Record

Field Name	value	Type
name	prince	String
age	26	Number
email	test@gmail.com	String
phone	+91 4589	String
address	Xyz colony, JH	String
photo	/uploads/pr_photo.jpg	String

Multer Package

We can use a Popular package named "Multer" to handle image uploads.

```
npm install multer
```

Multer is a Node.js middleware primarily used for handling **multipart/form-data**, which is the encoding type required for file uploads. It is commonly used with **Express.js** or **Node.js** applications to upload files such as images, PDFs, and others.

How Multer Works

1. **Middleware Integration**: Multer integrates with Express or Node.js routes and acts as middleware. It intercepts incoming file data (from forms) and processes it, then passes the file data to your route handler.
2. **Storage**: Multer allows you to specify how and where to store the uploaded files (locally on disk, in memory, or in cloud storage).

3. **File Handling**: It parses incoming files in the multipart form-data request and makes them accessible via `req.file` or `req.files`.
4. **Limits and Filters**: Multer allows setting file size limits, filtering files based on type (e.g., only images), and handling errors gracefully.

multipart/form-data

`multipart/form-data` is a content type used to send files or large data in HTTP requests, typically used when submitting forms that include file uploads.

It allows you to send **binary data** (like images, videos, and other file types) along with **text data** in a single HTTP request, making it ideal for forms with both types of input.

Standard Form Data (Text Data Submission)

When you submit a simple form that only contains **text fields** (like username, password, and email), the browser uses the **`application/x-www-form-urlencoded`** format by default to encode and send the form data. This type of encoding is suitable for sending **text** but not for sending **binary data** like images or files.

How `application/x-www-form-urlencoded` Works

- When you submit a form, the data in the form (e.g., input fields like name, email, etc.) is encoded and sent to the server.
- In **`application/x-www-form-urlencoded`**, the form data is converted into a query string format.
- A **query string** looks like this:

```
key1=value1&key2=value2&key3=value3
```

Here:

- Each **input field** is encoded as a key-value pair (`key=value`).
- Multiple key-value pairs are joined by an `&` symbol.

- Special characters (spaces, symbols) are **URL-encoded**, meaning they are converted into a specific format (e.g., space becomes `%20`).

Example:

- **HTML Form:**

```
<form action="/submit" method="POST">
  <input type="text" name="username" value="John Doe">
  <input type="text" name="email" value="john@example.com">
  <button type="submit">Submit</button>
</form>
```

When the form is submitted, it sends the data like this:

```
POST /submit HTTP/1.1
Content-Type: application/x-www-form-urlencoded

username=John%20Doe&email=john%40example.com
```

Inefficient for Large Data: Encoding files (like an image) in this format would result in a very inefficient and bloated request, making the process slow and error-prone.

Aspect	`application/x-www-form-urlencoded`	`application/json`
Format	Key-value pairs in URL-encoded form	JSON object with structured data
Data Type Handling	Typically text data, needs URL encoding for special characters	Can handle structured data types like objects and arrays
Readability	Less readable (URL encoded)	More readable (JSON format)
Content-Type Header	`application/x-www-form-urlencoded`	`application/json`
Typical Use Case	Submitting HTML form data	Sending data in REST APIs
Complex Data	Difficult to represent complex/nested structures	Well-suited for nested/complex data

How does an Image get converted into Binary data?

When you upload a JPEG image during a file upload process, the image is converted into **binary data** by the **browser** before sending it to the server. Here's a detailed breakdown of the steps and how it works:

Step-by-Step Process:

1. User Selects the Image in the Browser

- A user selects a file (e.g., a JPEG image) through an HTML form element with `<input type="file">`.
- The file on the user's local system is stored in binary format. Every file on a computer, whether an image, document, or video, is essentially binary data on disk.

2. Browser Reads the File

- Once the user selects the file and submits the form, the **browser** takes the selected file (in this case, a JPEG image).
- The browser doesn't need to "convert" the image in the traditional sense, as files are already stored as **binary data** on your computer.
- The file is read from the computer's file system in its raw binary form, which contains all the bytes (1s and 0s) that make up the JPEG image.

3. Browser Prepares the HTTP Request

- When the form is submitted, the browser prepares an **HTTP request** to send the form data to the server.
- If the form uses `enctype="multipart/form-data"`, the browser knows that the form data can include binary files (like images).
- The browser includes the binary data of the image in the request, **along with other form fields** (like text inputs).

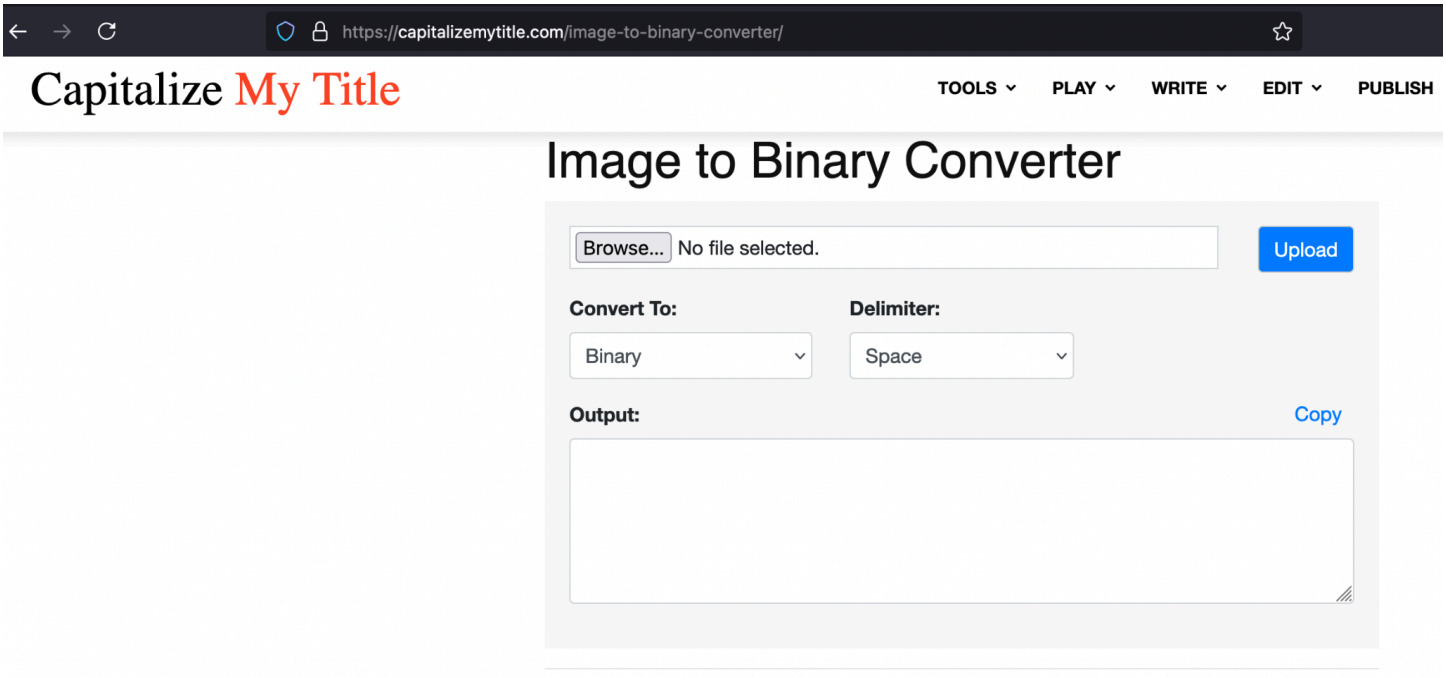
4. Encoding the Form Data (Multipart Encoding)

- The browser creates a **multipart/form-data** request, which separates each part of the form (text inputs, file inputs) with a boundary string.
- For the file, the browser includes the binary content of the image file (the raw bytes) in one of the parts of the request.

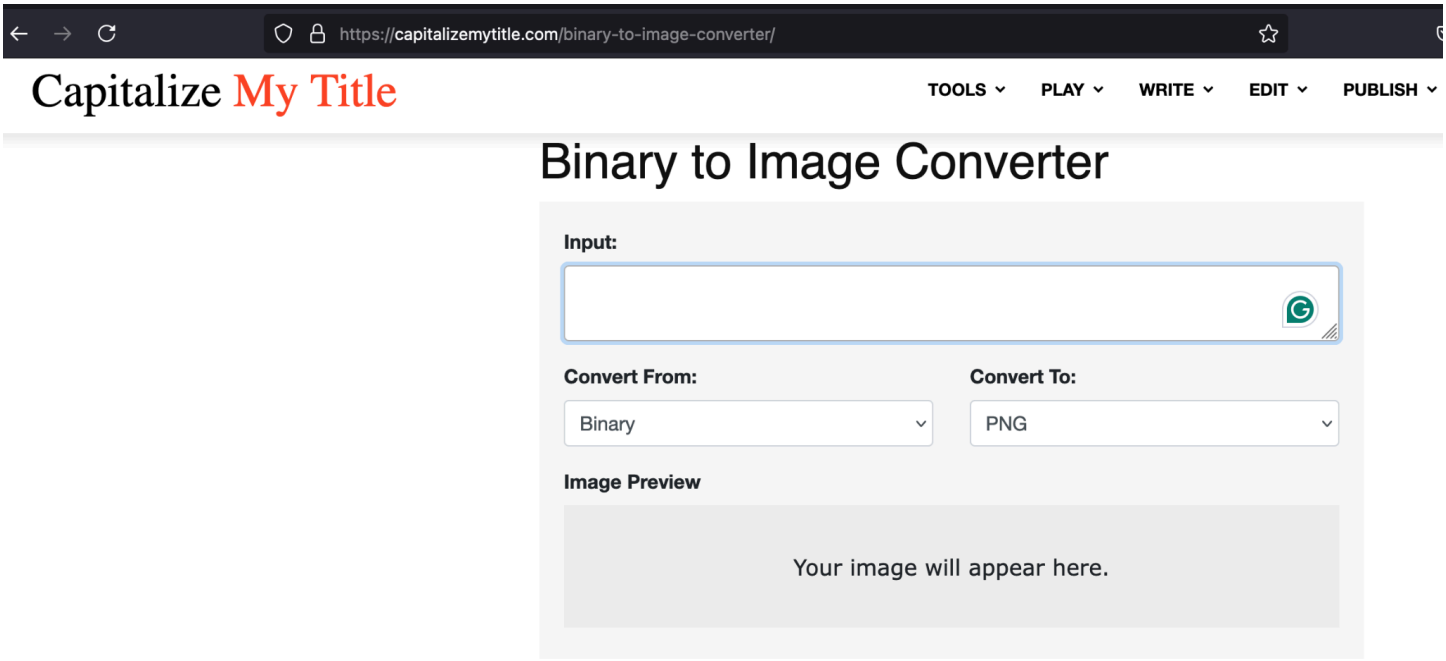
```
POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

------WebKitFormBoundary
Content-Disposition: form-data; name="username"

JohnDoe
------WebKitFormBoundary
Content-Disposition: form-data; name="profileImage";
filename="profile.jpg"
Content-Type: image/jpeg

(binary data of profile.jpg here)
------WebKitFormBoundary--
```

We can see the binary data of the file that is present in our system and also can see those binary data on online portal as well, even we can do interconvert from binary file to Image and vice versa.

Convert Image to Binary



Convert Binary to Image



Check Binary data in the system

If you're comfortable with the command line, you can use commands to display the binary content of a JPEG file.

Here are examples of different operating systems:

On Linux or Mac: You can use the `xxd` command, which creates a hex dump of a file.

```
xxd your-image.jpg | less
```

On Windows: You can use the `CertUtil` command:

```
certutil -encodehex your-image.jpg
```

Create REST APIs for Students Record

Now, we will create a Basic REST API for student record

```
project-folder/
├── uploads/              # Folder to store uploaded images
├── models/
│   └── student.js        # Mongoose model for Student
├── db.js                 # MongoDB connection file
└── server.js             # Main server file
```

How multer middleware works?

Now **`multer` Middleware Automatically Detects It**

On the server side, `multer` detects the `multipart/form-data` content type automatically. When `multer` is used as middleware (like `upload.single('photo')`), it looks for files in incoming requests with this content type.

If the request has `Content-Type: multipart/form-data`, `multer`:

- Parses the incoming data,
- Extracts the file(s),
- Converts them into a `Buffer` (if using in-memory storage), and
- Adds the file data to `req.file` (for a single file) or `req.files` (for multiple files).

Why `"photo"`?

- The `"photo"` parameter in `upload.single('photo')` specifies the **field name** in the form data where the file is expected to be uploaded. For example, if you're sending data through a form or with Postman, this would correspond to the field where you provide the file.

Why `.single`?

- `.single()` is a `multer` function that indicates **only one file** should be uploaded from that specific field. It does two things:
- Ensures that only one file is uploaded in that request.
- Adds the file to `req.file` (if using `upload.single()`) rather than `req.files`, which is used for multiple files.

Structure of `req.file`

When an image is uploaded and received in `req.file`, it contains a lot of useful metadata about the file. Here's an example of what `req.file` might look like:

```
{
    "fieldname": "photo",
    "originalname": "profile-pic.jpg",
    "encoding": "7bit",
    "mimetype": "image/jpeg",
    "size": 2048,
    "buffer": "<Buffer 89 50 4e 47 0d 0a ... >",
    "destination": "uploads/",
    "filename": "1234567890-profile-pic.jpg",
    "path": "uploads/1234567890-profile-pic.jpg"
}
```

`buffer`: This is the actual image data in a `Buffer` format (a type of data used to represent binary data in Node.js

If you want to store the image in upload/ folder

```
// Set up multer to store files in /uploads folder
const storage = multer.diskStorage({
    destination: (req, file, cb) => {
        cb(null, 'uploads/');
    },
    filename: (req, file, cb) => {
        const uniqueSuffix = Date.now() + '-' +
Math.round(Math.random() * 1E9);
        cb(null, uniqueSuffix + '-' + file.originalname);
    }
});
```

The `destination` function defines the folder where the uploaded file should be stored.

- The first argument `req` is the request object.
- The second argument `file` is the file object that Multer creates for the uploaded file.
- The third argument `cb` (callback) is used to pass the folder path where the file should be saved.

The `filename` function specifies the name for the uploaded file once it's saved to disk.

If you want to store the image in the Database

```
// Configure multer to store files in memory as Buffer
const storage = multer.memoryStorage();
```

- `file.originalname` contains the original name of the uploaded file, so appending a unique suffix ensures that each file has a unique name, avoiding overwrites.
- The `uniqueSuffix` is generated here using the current timestamp (`Date.now()`) and a random number.

- When you configure `multer` with `memoryStorage`, it means that the uploaded files are stored in memory.
- Even we can check the stored base64 image data online that image is correct or not.

For Notes,

Comment on this video:

I enjoyed the Nodejs series, Email: youremail@gmail.com, Thanks and subscribed.

Assignment Question

Question 1: Profile Picture Upload

A client wants users to upload a profile picture during registration. The image should be saved on the server, and the URL should be stored in the database (for now, simulate the database by logging the URL to the console).

Requirements:

1. **Create an Express route** `/uploadProfilePicture` to handle the profile picture upload.
2. Configure `multer` to:
    - Store the uploaded images in an `uploads/profile-pics` directory.
    - Only accept images (JPEG, PNG).
    - Limit the file size to 1 MB.

3. After the file is successfully uploaded:
    - Save the file with a unique name using a timestamp prefix (e.g., `1671023498000-profile.jpg`).
    - Log the image path (e.g., `/uploads/profile-pics/1671023498000-profile.jpg`) to the console.
4. **Test the route** by uploading a sample profile picture using Postman or a frontend form.

Hints:

- Use `multer`'s `diskStorage` to configure file destination and filename.
- Use a regex or `mimetype` filter to restrict file types to images only.
- Use error handling for cases where the file is too large or the wrong type is uploaded.

Question 2: Product Image Gallery

A client running an e-commerce website wants to allow vendors to upload multiple images of their products. You need to implement an endpoint that lets users upload up to 5 images per product, which will be saved in a specific folder on the server.

Requirements:

1. **Create an Express route** `/uploadProductImages` to handle multiple file uploads.
2. Configure `multer` to:
    - Store images in the `uploads/product-images` directory.
    - Accept up to 5 images in one request.
    - Restrict file types to images (JPEG, PNG).
3. After the files are uploaded, return a JSON response with the URLs of the uploaded images.
4. **Test the route** by uploading multiple images for a product using Postman or a frontend form.

Hints:

- Use `upload.array('images', 5)` for handling multiple file uploads.

- In your JSON response, include the list of URLs like `[{ "url": "/uploads/product-images/1671023498000-image1.jpg" }]`.

Testing and Submission:

For each question:

1. **Run the server** and test each route using Postman or a simple HTML form with `enctype="multipart/form-data"`.
2. Verify that files are saved in the correct directories and that images can be retrieved using the saved paths.
3. Submit your code along with screenshots showing successful file uploads and console output/logs of the uploaded paths.

Example Solution Outline (For Reference Only):

1. For **Question 1**, create a basic Express route, configure `multer` with `diskStorage`, and add filters and limits.
2. For **Question 2**, adjust the route to handle an array of files and output the URLs as JSON.