

Chapter 21: File Uploads & Cloud Integration

21.1 File Upload Architecture

21.1.1 Introduction

- * File uploads allow users to send files from the frontend to the backend and store them securely.
- * Cloud integration helps in storing and managing these files in scalable and reliable cloud storage services.
- * This architecture ensures smooth file handling, security and high availability.
- * It is widely used in web applications like social media, e-commerce, and document management systems.



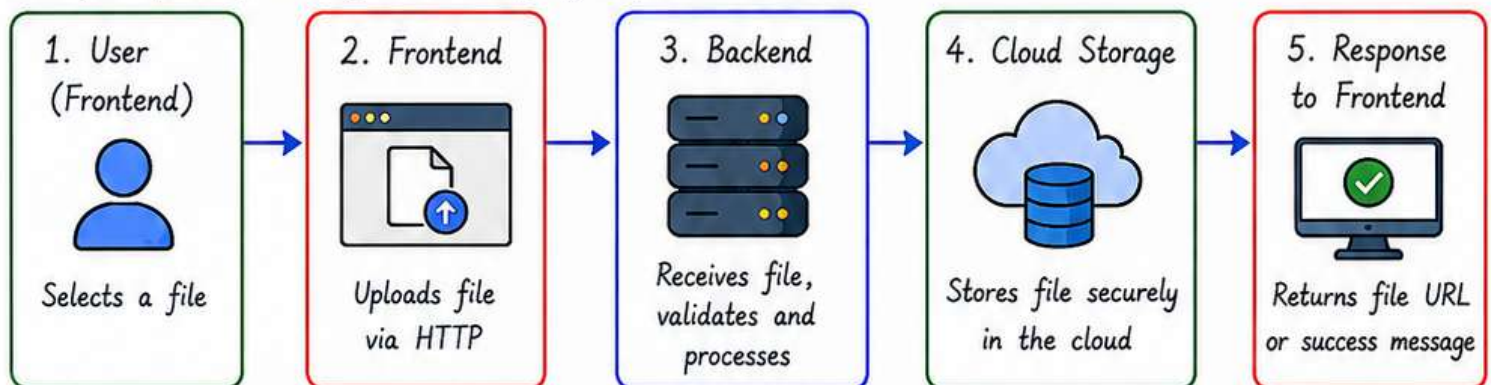
Key Points:

- ✓ File uploads connect the frontend, backend and cloud storage.
- ✓ Cloud storage provides scalability, reliability and global accessibility.
- ✓ Proper validation and security are important during file uploads.
- ✓ This architecture improves performance and user experience.



21.1.2 Architecture Overview

The file upload process follows a step-by-step architecture:



21.1.3 Components

- * **Frontend**: Handles file selection and sends file to backend.
- * **Backend**: Handles file reception, validation and processing.
- * **Cloud Storage**: Stores files securely (e.g., AWS S3, Google Cloud Storage).
- * **Database (Optional)**: Stores file metadata like name, type, size, URL.



Key Takeaway:



File upload architecture connects users, applications and cloud storage to store and manage files **securely**, **efficiently** and at **scale**.



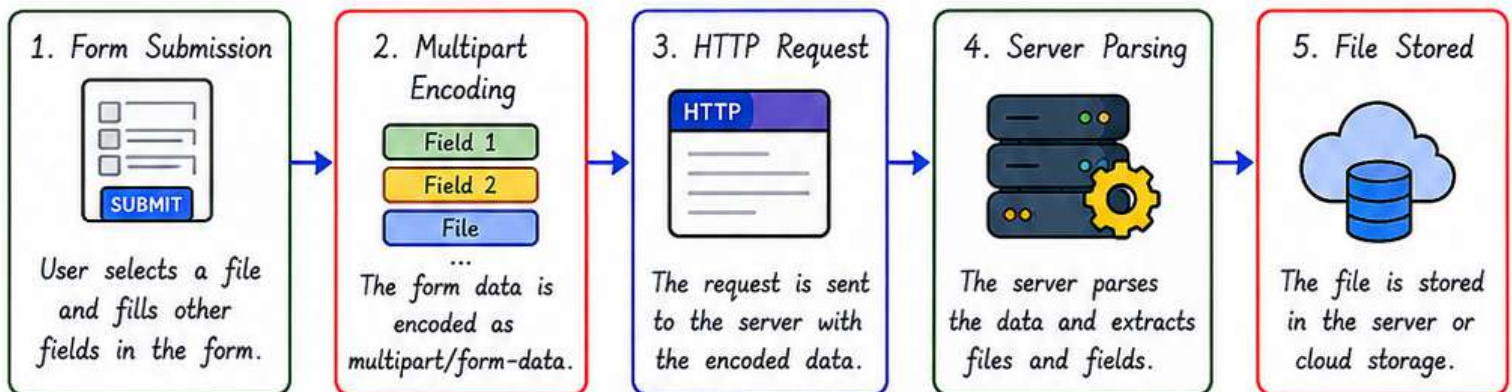
21.2 Multipart/Form Data

21.2.1 What is Multipart/Form Data?

- * Multipart/Form Data is an **encoding type** used to send **files** along with other data in an HTTP request.
- * It is commonly used in **file upload forms**.
- * Each part in the request represents a **field** (text or file).
- * The data is sent in parts separated by a **boundary**.



21.2.2 How Multipart/Form Data Works?



21.2.3 Example (Using FormData in JavaScript)

```
const formData = new FormData();
formData.append('file', document.getElementById('fileInput').files[0]);
formData.append('name', 'Bhavitha');
formData.append('email', 'bhavitha@example.com');

fetch('https://your-api.com/upload', {
  method: 'POST',
  body: formData
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.log('Error:', error));
```

Explanation:

- ✓ **FormData()** creates a new form data object.
- ✓ **append()** adds fields and files to the form data.
- ✓ The file is taken from the file input element.
- ✓ The data is sent using **fetch()** with **POST** method.

21.2.4 Request Structure (Multipart)

```
POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary12345
-----WebKitFormBoundary12345
Content-Disposition: form-data; name="name"
Bhavitha
-----WebKitFormBoundary12345
Content-Disposition: form-data; name="file"; filename="resume.pdf"
Content-Type: application/pdf
<binary file data here>
-----WebKitFormBoundary12345--
```

Each part is separated by a boundary.
It can contain text fields or files.



Key Takeaway:



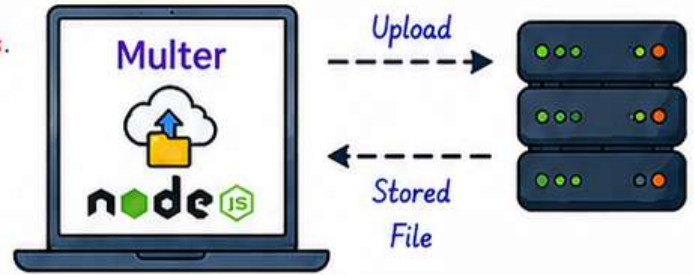
Multipart/Form Data allows sending both **files** and other **data** together in a single request, making it ideal for file upload forms.



21.3 Multer

21.3.1 What is Multer?

- * Multer is a **Node.js** middleware used for handling multipart/form-data, which is primarily used for **file uploads**.
- * It is built on top of **busboy** for maximum efficiency.
- * Multer makes it easy to handle **files**, store them and access other form data.
- * It is widely used in **Express.js** applications.



21.3.2 Basic Usage of Multer

```
const express = require('express');
const multer = require('multer');
const app = express();

// Set storage engine
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/'); // Folder where files will be stored
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + '-' + file.originalname); // Unique filename
  }
});

// Initialize upload
const upload = multer({ storage: storage });

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



Multer

Explanation:

- ✓ **diskStorage** is used to store files on the server.
- ✓ **destination** sets the folder path.
- ✓ **filename** sets a unique name for the uploaded file.
- ✓ **upload.single('file')** handles a single file with the field name 'file'.
- ✓ Once uploaded, a success message is returned.

21.3.3 Multer Methods

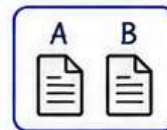
- * **upload.single(fieldname)** - Accepts a single file.
- * **upload.array(fieldname, maxCount)** - Accepts multiple files.
- * **upload.fields(fields)** - Accepts multiple files with different field names.
- * **upload.any()** - Accepts all files (use with caution).



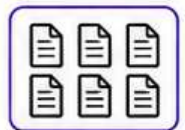
single()



array()



fields()



any()

21.3.4 Multer Configuration Options

Option	Description
storage	Defines where and how the file will be stored.
dest	Specifies the destination folder (used with default storage).
limits	Sets limits like file size, number of files, etc.
fileFilter	Allows filtering files before uploading.

Example (Limits):

```
const upload = multer({
  limits: {
    fileSize: 10 * 1024 * 1024 // 10MB
  }
});
```

Key Takeaway:



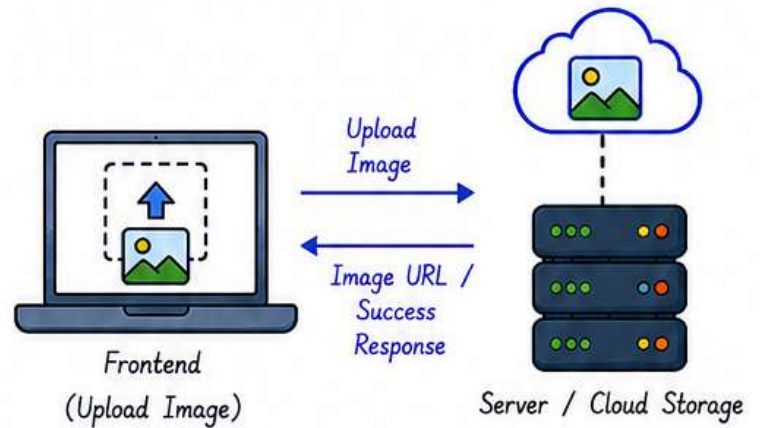
Multer simplifies **file uploads** in Node.js applications by handling **multipart/form-data** requests, storing files **securely** and providing **easy** access to uploaded files.



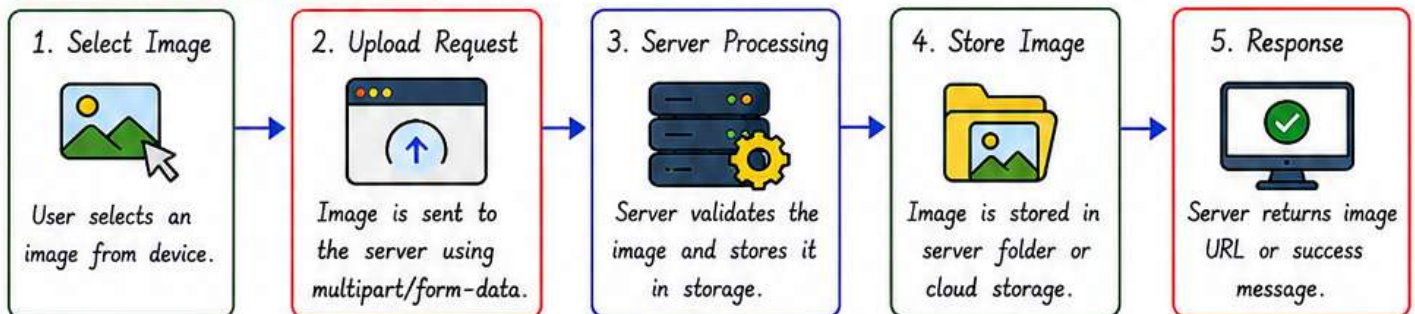
21.4 Image Upload

21.4.1 Introduction

- * Image upload allows users to upload **image files** from the frontend to the server.
- * The images are stored on the server or cloud storage and their **URLs** are saved in the database.
- * It is commonly used in profiles, social media, e-commerce, blogs, etc.
- * Multer is used in **Node.js/Express.js** applications to handle image uploads efficiently.



21.4.2 Image Upload Flow



21.4.3 Example (Image Upload using Multer)

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const app = express();

// Storage configuration
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/images'); // Folder to store images
  },
  filename: function (req, file, cb) {
    const uniqueName = Date.now() + path.extname(file.originalname);
    cb(null, uniqueName); // Unique filename
  }
});

// File filter (only images)
const fileFilter = (req, file, cb) => {
  const allowedTypes = /jpeg|jpg|png|gif/;
  const extname = allowedTypes.test(path.extname(file.originalname).toLowerCase());
  const mimetype = allowedTypes.test(file.mimetype);

  if (extname && mimetype) return cb(null, true);
  cb(new Error('Only image files are allowed!'));
};

const upload = multer({ storage: storage, fileFilter: fileFilter });

// Route for image upload
app.post('/upload-image', upload.single('image'), (req, res) => {
  if (!req.file) return res.status(400).send('No file uploaded!');
  const imageUrl = '/uploads/images/' + req.file.filename;
  res.send({ message: 'Image uploaded successfully!', imageUrl: imageUrl });
});
```

Explanation:

- ✓ **diskStorage()** defines where and how the image will be stored.
- ✓ **filename()** creates a unique filename to prevent conflicts.
- ✓ **fileFilter** ensures that only image files (jpg, jpeg, png, gif) are allowed.
- ✓ **upload.single('image')** handles a single image upload with the field name 'image'.
- ✓ On success, the server returns the image URL.



21.4.4 File Validation

- * Check file type (image MIME types).
- * Check file extension.
- * Check file size (set a limit).
- * Reject invalid files with an error message.



Key Takeaway:



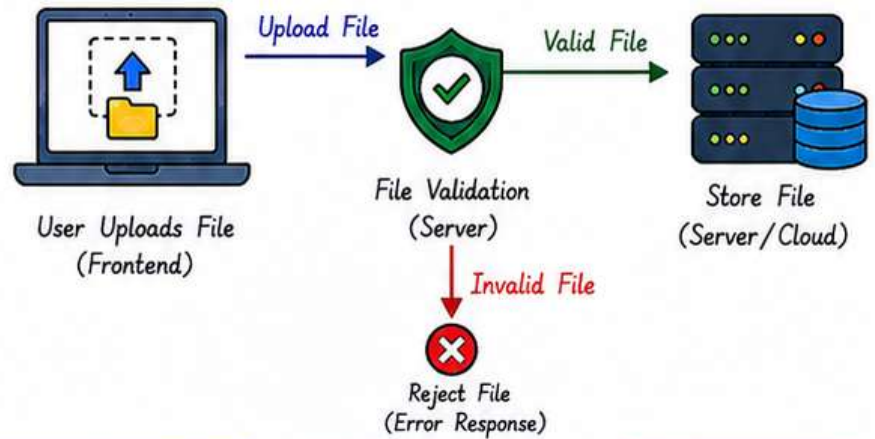
Image upload allows users to upload images securely. Using Multer, we can validate, store and manage images efficiently in Node.js applications.



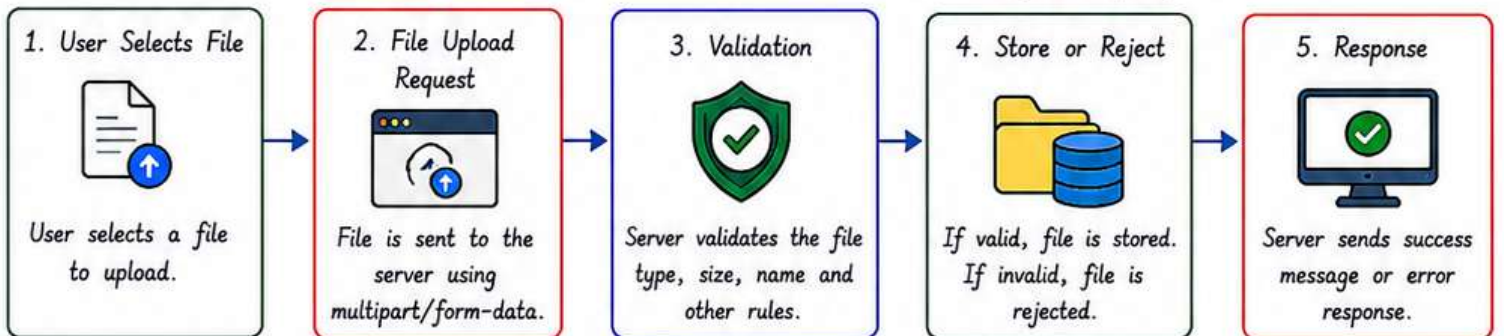
21.5 File Validation

21.5.1 Introduction

- * File validation ensures that only **valid files** are uploaded to the server.
- * It helps in maintaining **data security**, **consistency** and **performance**.
- * Validation includes checking **file type**, **size**, **name**, and **content**.
- * Multer provides built-in and custom methods to validate files.



21.5.2 Validation Flow



21.5.3 Example (File Validation using Multer)

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const app = express();

// File validation function
const fileFilter = (req, file, cb) => {
  const allowedTypes = /jpeg|jpg|png|pdf/;
  const allowedMimeTypes = ['image/jpeg', 'image/png', 'application/pdf'];
  const extname = allowedTypes.test(path.extname(file.originalname).toLowerCase());
  const mimetype = allowedMimeTypes.includes(file.mimetype);

  if (extname && mimetype) {
    return cb(null, true); // Accept file
  } else {
    return cb(new Error('Invalid file type! Only JPG, PNG or PDF allowed.'), false);
  }
};

// Storage configuration
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/'); // Folder to store files
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + '-' + file.originalname); // Unique filename
  }
});

// Initialize multer with validation
const upload = multer({
  storage: storage,
  fileFilter: fileFilter,
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB limit
});

// Route for file upload
app.post('upload', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).send({ message: 'File upload failed!', error: req.fileValidationError });
  }
  res.send({ message: 'File uploaded successfully!', filename: req.file.filename });
});
```

Explanation:

- ✓ **fileFilter()** checks the file extension and MIME type.
- ✓ Only **JPG, PNG** and **PDF** files are allowed.
- ✓ File size is limited to **5MB**.
- ✓ Invalid files are rejected with an error message.
- ✓ Valid files are stored in the **'uploads/'** folder.



21.5.4 Common Validation Rules

- * Check file type (extension).
- * Check MIME type.
- * Check file size.
- * Check file name (allowed pattern).
- * Check file content (if required).
- * Limit number of files.

Key Takeaway:

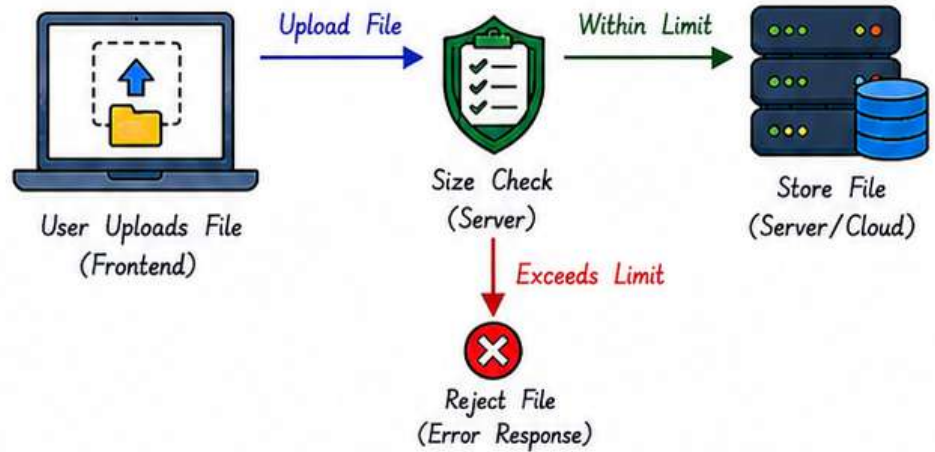
File validation is essential for **secure** file uploads. It prevents **invalid** or **malicious** files from being stored and ensures that only **valid** files are accepted.



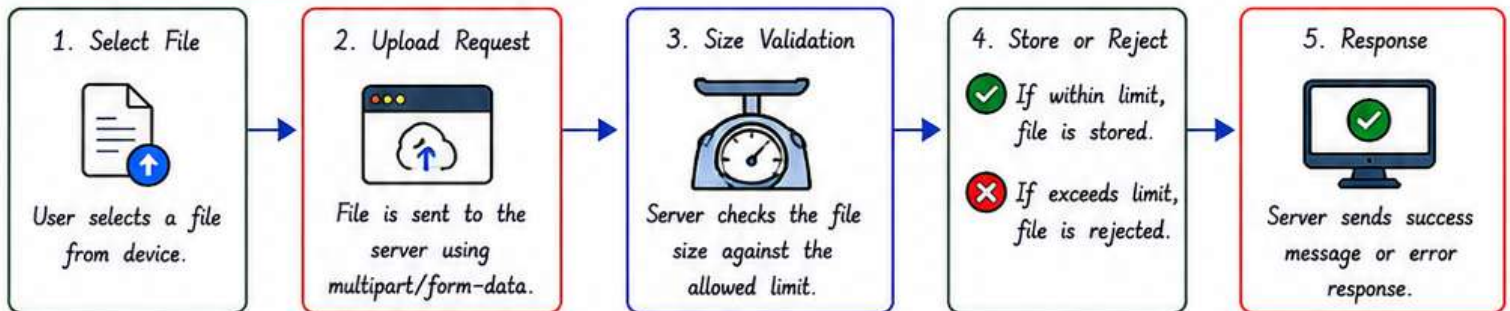
21.6 File Size Restrictions

21.6.1 Introduction

- * File size restriction ensures that users cannot upload files larger than a **defined limit**.
- * It helps in **saving server storage** and **improving performance**.
- * Multer allows us to set file size limits using the **limits** option.
- * If the file size exceeds the limit, the upload is **rejected** with an **error**.



21.6.2 File Size Restriction Flow



21.6.3 Example (File Size Restriction using Multer)

```
const express = require('express');
const multer = require('multer');
const app = express();

// Define storage configuration
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/'); // Folder to store files
  },
  filename: function (req, file, cb) {
    cb(null, Date.now() + '-' + file.originalname); // Unique filename
  }
});

// File size limit (5MB)
const upload = multer({
  storage: storage,
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB limit
});

// Route for file upload
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) return res.status(400).send('No file uploaded!');
  res.send({
    message: 'File uploaded successfully!',
    filename: req.file.filename,
    size: req.file.size // size in bytes
  });
});

// Error handling for file size limit
app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    if (err.code === 'LIMIT_FILE_SIZE') {
      return res.status(400).send('File size is too large! Maximum 5MB allowed.');
```

Explanation:

- ✓ **limits: { fileSize: 5 * 1024 * 1024 }** sets the maximum file size to **5MB**.
- ✓ If the uploaded file size exceeds 5MB, the upload will be rejected.
- ✓ Multer throws an error with code **LIMIT_FILE_SIZE**.
- ✓ Custom error handler sends a clear error message to the user.

File Size Units

- * 1 KB = 1024 Bytes
- * 1 MB = 1024 KB
- * 1 GB = 1024 MB
- * Example: 5MB = 5 * 1024 * 1024 Bytes

21.6.4 Common Size Limits

Limit	Use Case
1 MB	Small profile images, documents
5 MB	General file uploads
10 MB	Images, PDFs
20 MB	Large files, videos (short)
50 MB+	Videos, backups (use cloud storage)



Key Takeaway:



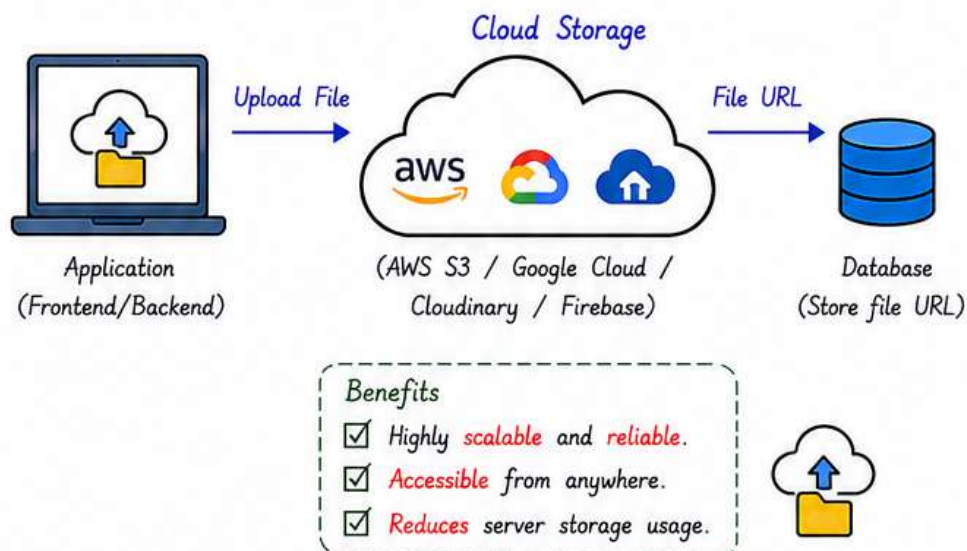
File size restrictions help in **managing server storage**, improving performance and ensuring smooth user experience by allowing only files within **defined limits**.



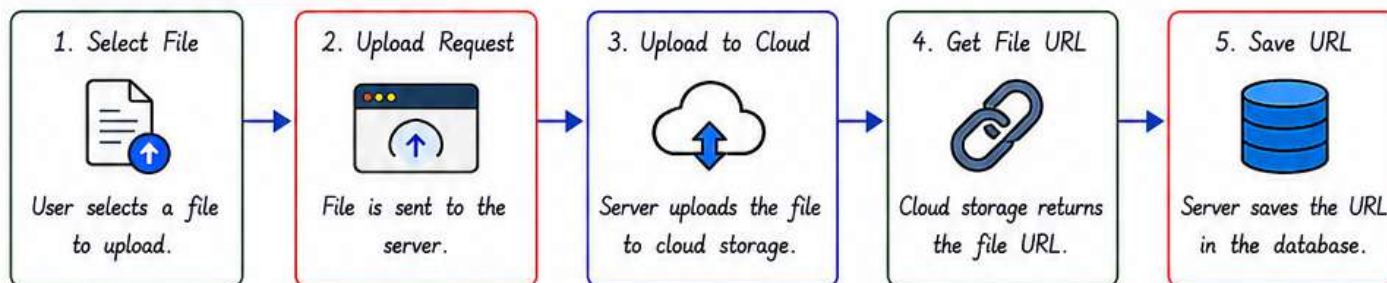
21.7 Cloud Storage

21.7.1 Introduction

- * Cloud storage allows us to store files on **remote servers** managed by third-party providers.
- * It provides **scalability**, **reliability** and **accessibility** from anywhere.
- * Popular cloud storage services are **AWS S3**, **Google Cloud Storage**, **Cloudinary**, **Firebase Storage**, etc.
- * Files are uploaded from the application to the cloud and the **URL** is saved in the database.



21.7.2 Cloud Storage Flow



21.7.3 Example (Upload to AWS S3 using Multer)

```
const express = require('express');
const multer = require('multer');
const AWS = require('aws-sdk');
const multerS3 = require('multer-s3');
const app = express();

// Configure AWS S3
AWS.config.update({
  accessKeyId: 'YOUR_ACCESS_KEY',
  secretAccessKey: 'YOUR_SECRET_KEY',
  region: 'ap-south-1'
});
const s3 = new AWS.S3();

// Multer storage using S3
const upload = multer({
  storage: multerS3({
    s3: s3,
    bucket: 'your-bucket-name',
    contentType: multerS3.AUTO_CONTENT_TYPE,
    key: function (req, file, cb) {
      cb(null, Date.now() + '-' + file.originalname); // Unique filename
    }
  })
}, {
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB limit
});

// Route to upload file to S3
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) return res.status(400).send('No file uploaded!');

  // File uploaded successfully
  res.send({
    message: 'File uploaded to cloud successfully!',
    url: req.file.location // URL of uploaded file in S3
  });
});
```

Explanation:

- ✓ **AWS SDK** is used to connect to S3.
- ✓ **multer-s3** stores files directly in S3.
- ✓ A unique filename is generated using `Date.now()`.
- ✓ After upload, S3 returns the **file URL**.
- ✓ The URL is saved in the database for future use.



Amazon S3



Google Cloud Storage



Cloudinary

21.7.4 Use Cases

- * Profile picture uploads.
- * Document management systems.
- * E-commerce product images.
- * Video and media file storage.
- * Backup and archive storage.

Key Points:

- ✓ Cloud storage provides **reliable** and **scalable** storage for files.
- ✓ It **reduces** the load on the application server.
- ✓ Files are accessed using **URLs**.

Key Takeaway:



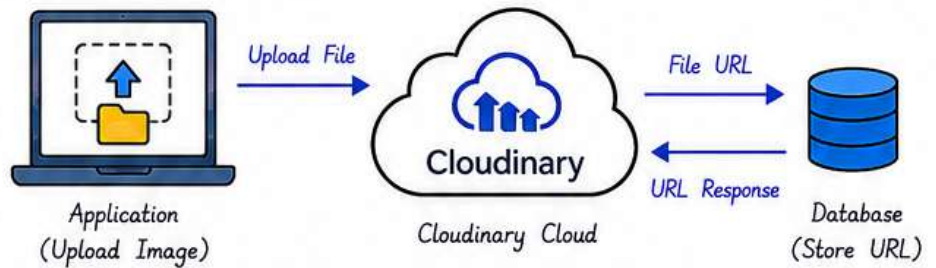
Cloud storage helps in storing files **securely** in the cloud, making them accessible from anywhere while improving **scalability**, **performance** and **reliability** of applications.



21.8 Cloudinary Integration

21.8.1 Introduction

- * Cloudinary is a cloud storage service that specializes in **image and video** management.
- * It provides features like **upload**, **transform**, **optimize** and **deliver** media files.
- * It returns a **secure URL** that can be used to access the media.
- * Cloudinary has a **Node.js SDK** that makes integration easy.

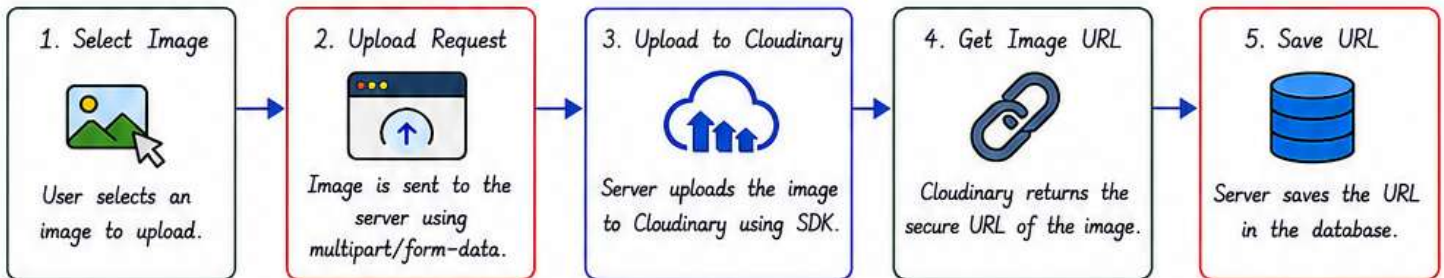


Benefits of Cloudinary

- ✓ Easy media **upload** and management.
- ✓ Automatic **optimization** of images and videos.
- ✓ Supports **transformations** (resize, crop, format, etc.).
- ✓ Delivers media through a **fast CDN**.
- ✓ Reduces **server storage** and bandwidth usage.



21.8.2 Cloudinary Integration Flow



21.8.3 Example (Cloudinary Integration in Express)

```
const express = require('express');
const multer = require('multer');
const cloudinary = require('cloudinary').v2;
const app = express();

// Configure Cloudinary
cloudinary.config({
  cloud_name: 'YOUR_CLOUD_NAME',
  api_key: 'YOUR_API_KEY',
  api_secret: 'YOUR_API_SECRET'
});

// Multer setup (store in memory before uploading to Cloudinary)
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

// Route for image upload
app.post('/upload-image', upload.single('image'), async (req, res) => {
  try {
    if (!req.file) return res.status(400).send('No file uploaded!');

    // Upload to Cloudinary
    const result = await cloudinary.uploader.upload_stream(
      { folder: 'uploads' },
      (error, result) => {
        if (error) return res.status(500).send('Upload failed!');

        // Save result.secure_url in database (example)
        // UserModel.create({ imageUrl: result.secure_url });

        res.send({
          message: 'Image uploaded successfully!',
          imageUrl: result.secure_url,
          public_id: result.public_id
        });
      }
    );

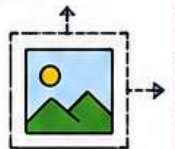
    // Pipe the file buffer to Cloudinary
    result.end(req.file.buffer);
  } catch (err) {
    res.status(500).send('Server error: ' + err.message);
  }
});
```

Explanation:

- ✓ We configure Cloudinary with **cloud_name**, **api_key** and **api_secret**.
- ✓ We use multer with **memoryStorage** to store the file in buffer.
- ✓ **cloudinary.uploader.upload_stream()** uploads the image to Cloudinary.
- ✓ Cloudinary returns a secure URL (**secure_url**) and **public_id**.
- ✓ We save the **secure_url** in the database and return it in the response.

Common Transformations (Examples)

- * w - Width
- * h - Height
- * c - Crop mode (fill, limit, scale, crop)
- * q - Quality (auto, 1 to 100)
- * f - Format (jpg, png, webp)
- * e - Effect (sharpen, blur, etc.)



Example URL:

https://res.cloudinary.com/demo/image/upload/w_300,h_300,c_fill,q_auto,f_auto/sample.jpg

21.8.4 Popular Use Cases

- * User profile picture upload.
- * Product image upload in e-commerce.
- * Blog images and media management.
- * Video upload and streaming.
- * Image optimization and resizing.



Key Takeaway:

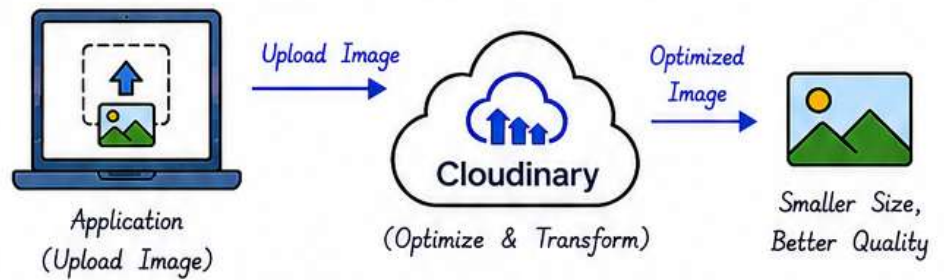
Cloudinary integration **simplifies** media management by providing easy upload, **optimization** and delivery of images and videos through **secure URLs**.



21.9 Image Optimization

21.9.1 Introduction

- * Image optimization improves performance by **reducing file size** without losing quality.
- * Optimized images load **faster**, use less bandwidth and improve **user experience**.
- * Cloud services like **Cloudinary** provide automatic optimization and transformation.
- * We can **resize**, **compress** and convert images to the **best format** for web.

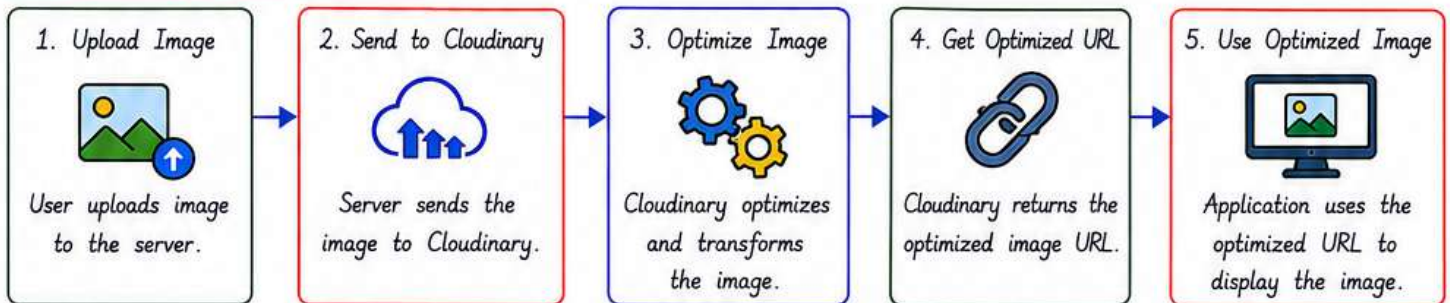


Benefits of Image Optimization

- ✓ Faster page load and better **performance**.
- ✓ Saves **bandwidth** and **storage**.
- ✓ Improves **SEO** and **user experience**.
- ✓ Automatic optimization using **Cloudinary**.



21.9.2 Optimization Flow



21.9.3 Example (Image Optimization using Cloudinary in Express)

```
const express = require('express');
const multer = require('multer');
const cloudinary = require('cloudinary').v2;
const app = express();

// Configure Cloudinary
cloudinary.config({
  cloud_name: 'YOUR_CLOUD_NAME',
  api_key: 'YOUR_API_KEY',
  api_secret: 'YOUR_API_SECRET'
});

// Multer setup (store in memory)
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

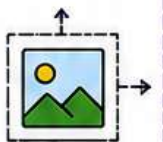
// Route for upload and optimize image
app.post('/upload', upload.single('image'), async (req, res) => {
  try {
    const result = await cloudinary.uploader.upload(req.file.buffer, {
      folder: 'uploads',
      transformation: [
        { width: 1000, height: 1000, crop: 'limit' }, // resize
        { quality: 'auto:good', fetch_format: 'auto' } // optimize
      ]
    });
    // result.secure_url is the optimized image URL
    res.send({
      message: 'Image uploaded and optimized successfully!',
      url: result.secure_url,
      public_id: result.public_id
    });
  } catch (error) {
    res.status(500).send({ message: 'Upload failed', error: error.message });
  }
});
```

Explanation:

- ✓ We upload the image using **multer**.
- ✓ Cloudinary automatically optimizes the image using **transformation** options.
- ✓ **width**, **height** and **crop** resize the image.
- ✓ **quality: 'auto:good'** adjusts the best quality.
- ✓ **fetch_format: 'auto'** delivers the best format (WebP, JPEG, etc.).
- ✓ Cloudinary returns the optimized image URL (**secure_url**).

Common Optimization Options

- * **w** - Width
- * **h** - Height
- * **c** - Crop mode (fill, limit, scale, crop)
- * **q** - Quality (auto or 1 to 100)
- * **f** - Format (auto, jpg, png, webp)
- * **e** - Effect (sharpen, blur, grayscale, etc.)



21.9.4 Popular Use Cases

- * Speed up website images.
- * E-commerce product images.
- * User profile picture optimization.
- * Gallery and portfolio websites.
- * Mobile applications for faster loading.



Key Takeaway:



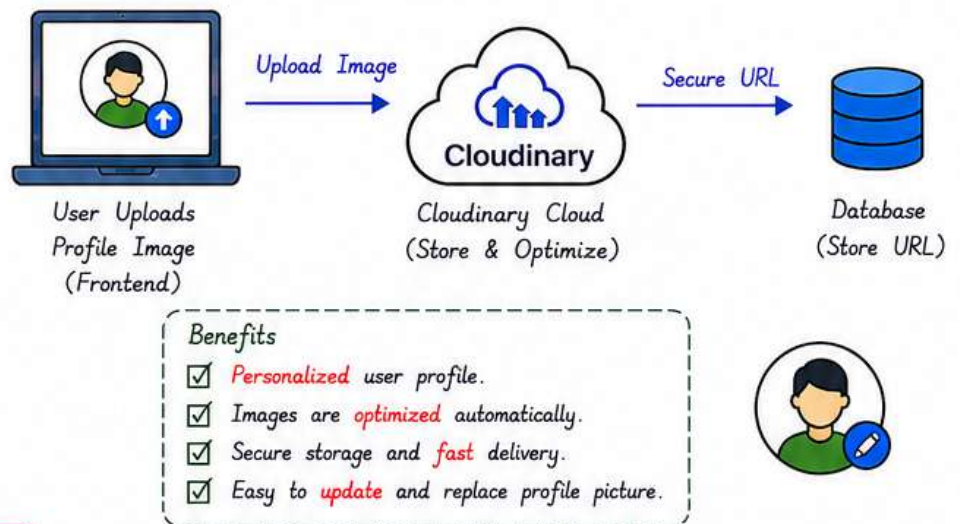
Image optimization **reduces the file size** and improves loading speed without compromising quality. Cloudinary makes it easy with **automatic optimization** and powerful **transformation** options.



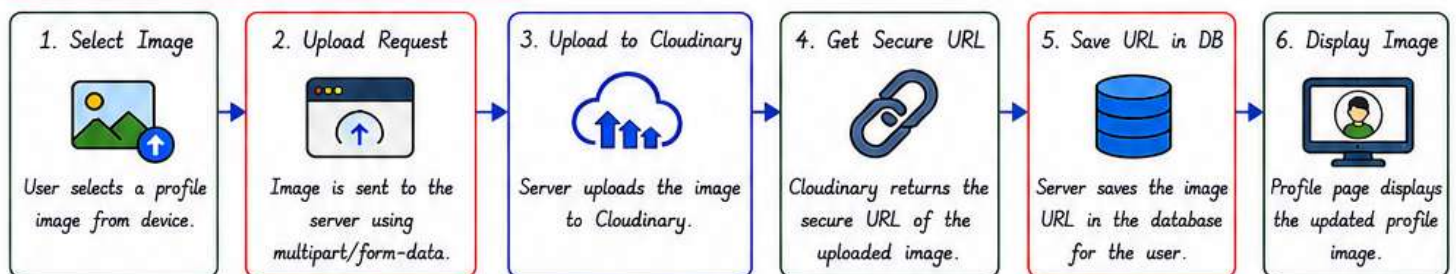
21.10 Profile Image Upload

21.10.1 Introduction

- * Profile image upload allows users to **upload** and **update** their profile picture.
- * The image is stored in the cloud (Cloudinary) and the **secure URL** is saved in the database.
- * The image is displayed on the user profile page.
- * When a new image is uploaded, the old image is **replaced**.



21.10.2 Profile Image Upload Flow



21.10.3 Example (Profile Image Upload using Express + Cloudinary)

```
const express = require('express');
const multer = require('multer');
const cloudinary = require('cloudinary').v2;
const User = require('../models/User');
const authMiddleware = require('../middleware/auth');
const router = express.Router();

// Configure Cloudinary
cloudinary.config({
  cloud_name: 'YOUR_CLOUD_NAME',
  api_key: 'YOUR_API_KEY',
  api_secret: 'YOUR_API_SECRET'
});

// Multer setup (store in memory)
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

// Upload profile image
router.post('/upload-profile-image', authMiddleware,
  upload.single('image'), async (req, res) => {
    try {
      if (!req.file) return res.status(400).send({ message: 'No image uploaded!' });

      // Upload to Cloudinary
      const result = await cloudinary.uploader.upload(req.file.buffer, {
        folder: 'profile_images',
        width: 500, height: 500, crop: 'fill', quality: 'auto', fetch_format: 'auto'
      });

      // Update user profile image URL in database
      const user = await User.findById(req.user._id);
      if (user.profileImage && user.profileImage.public_id) {
        // Delete old image from Cloudinary
        await cloudinary.uploader.destroy(user.profileImage.public_id);
      }
      user.profileImage = {
        url: result.secure_url,
        public_id: result.public_id
      };
      await user.save();

      res.send({ message: 'Profile image updated successfully', imageUrl: result.secure_url });
    } catch (error) {
      res.status(500).send({ message: 'Upload failed', error: error.message });
    }
  });
```

Explanation:

- ✓ Multer stores the image in **memory**.
- ✓ Image is uploaded to **Cloudinary**.
- ✓ If user already has an old profile image, it is **deleted** from Cloudinary.
- ✓ New image **URL** and **public_id** are saved in the database.
- ✓ The updated image is returned and displayed on the profile page.

Image Transformations (Example URL)

https://res.cloudinary.com/demo/image/upload/w_200,h_200,c_fill,g_face,r_max/v1712312345/profile_images/abc123.jpg

- * **w_200, h_200** → Resize
- * **c_fill, g_face** → Crop & Gravity
- * **r_max** → Round corners (if needed)
- * **q_auto** → Auto quality
- * **f_auto** → Auto format

21.10.4 Use Cases

- * User profile picture upload.
- * Social media applications.
- * User account settings.
- * Employee portal profile update.
- * Messaging apps.

Key Points:

- ✓ Always **validate** file type and size.
- ✓ Use Cloudinary transformations for **optimized** images.
- ✓ **Replace** old image to save storage.
- ✓ Store only the **secure URL** in database.

Key Takeaway:

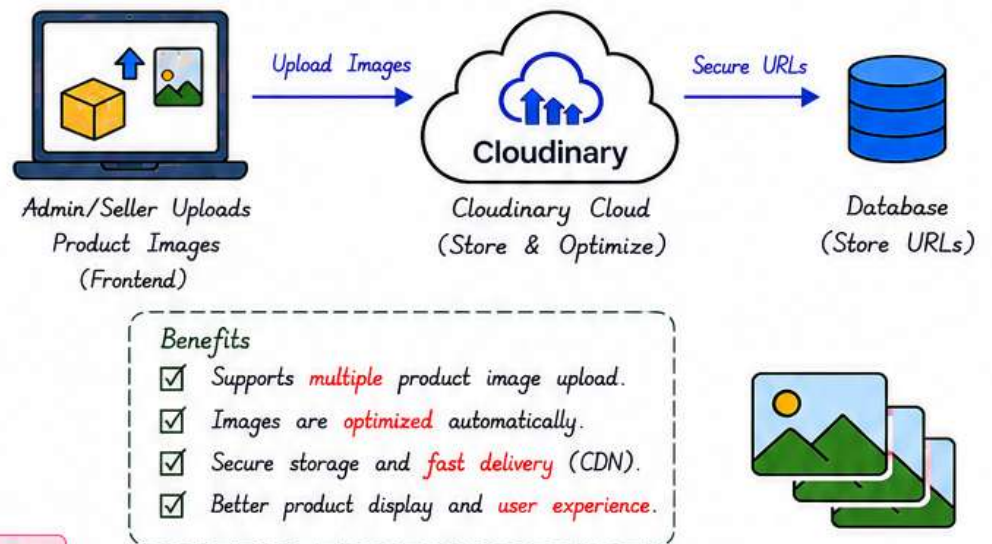
Profile image upload provides a better user experience by allowing users to **personalize** their profile. Using Cloudinary ensures **optimized storage**, **fast delivery** and **secure access**.



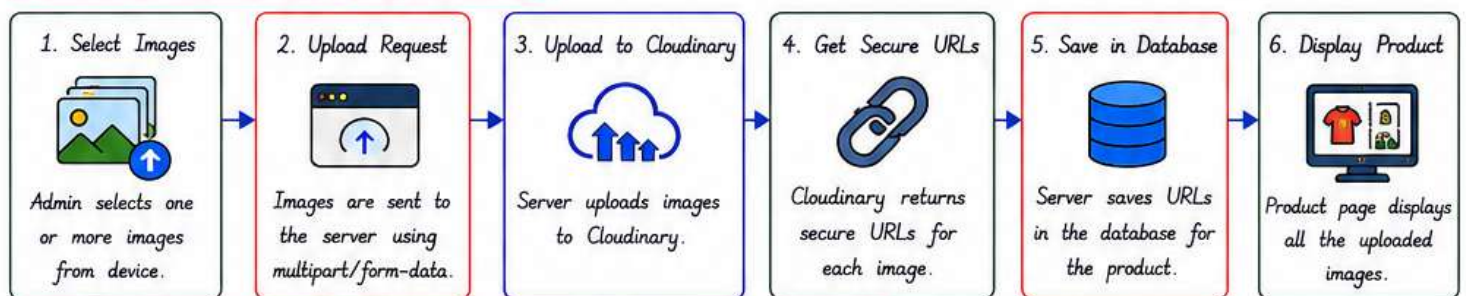
21.11 Product Image Upload

21.11.1 Introduction

- * Product image upload allows admin or sellers to upload **multiple images** for a product.
- * Images are stored in the cloud (Cloudinary) and the **secure URLs** are saved in the database.
- * These images are displayed in the product details page.
- * We can **upload, optimize** and manage multiple product images easily.



21.11.2 Product Image Upload Flow



21.11.3 Example (Product Image Upload using Express + Cloudinary)

```
const express = require('express');
const multer = require('multer');
const cloudinary = require('cloudinary').v2;
const Product = require('./models/Product');
const router = express.Router();

// Configure Cloudinary
cloudinary.config({
  cloud_name: 'YOUR_CLOUD_NAME',
  api_key: 'YOUR_API_KEY',
  api_secret: 'YOUR_API_SECRET'
});

// Multer setup (store in memory)
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

// Route: upload multiple product images
router.post('/upload-product-images', upload.array('images', 5), async (req, res) => {
  try {
    if (!req.files || req.files.length === 0) {
      return res.status(400).send({ message: 'No images uploaded!' });
    }
    const uploadPromises = req.files.map(file => {
      return cloudinary.uploader.upload(file.buffer, {
        folder: 'product_images',
        width: 800, height: 800, crop: 'limit',
        quality: 'auto', fetch_format: 'auto'
      });
    });
    const results = await Promise.all(uploadPromises);
    const imageUrls = results.map(result => result.secure_url);
    // Save product with image URLs in database
    const product = new Product({
      name: req.body.name,
      price: req.body.price,
      images: imageUrls
    });
    await product.save();
    res.send({
      message: 'Product images uploaded successfully!',
      images: imageUrls
    });
  } catch (error) {
    res.status(500).send({ message: 'Upload failed', error: error.message });
  }
});
```

Explanation:

- ✓ We use **multer.array()** to upload multiple images (max 5 in this example).
- ✓ Each image is uploaded to Cloudinary and we get **secure_url**.
- ✓ All URLs are stored in the **images** field of the product.
- ✓ These URLs are used to display product images on the frontend.

Cloudinary Transformation Options (Examples)

- * **w_800** → Resize width
- * **h_800** → Resize height
- * **c_fill** → Crop & fill
- * **q_auto** → Auto quality
- * **f_auto** → Auto format (WebP / JPEG / PNG)
- * **g_auto** → Auto gravity (center / face)

Example URL:

https://res.cloudinary.com/demo/image/upload/w_800,h_800,c_limit,q_auto,f_auto/sample.jpg

21.11.4 Use Cases

- * E-commerce product image upload.
- * Admin product management.
- * User product listing.
- * Gallery style product display.
- * Multiple image preview in product page.



Key Points:

- ✓ Validate file type and size before upload.
- ✓ Use Cloudinary transformations for optimized images.
- ✓ Store only secure URLs in database.
- ✓ Always handle errors and provide proper response.

Key Takeaway:



Product image upload helps to showcase products better. Using Cloudinary ensures **optimized images, secure storage, fast delivery** and **better user experience**.



21.12 Secure File Handling

21.12.1 Introduction

- * Secure file handling ensures that uploaded and downloaded files are safe from **malicious attacks**.
- * It **validates files**, prevents harmful uploads and **protects the server**.
- * Secure handling is essential for web applications.

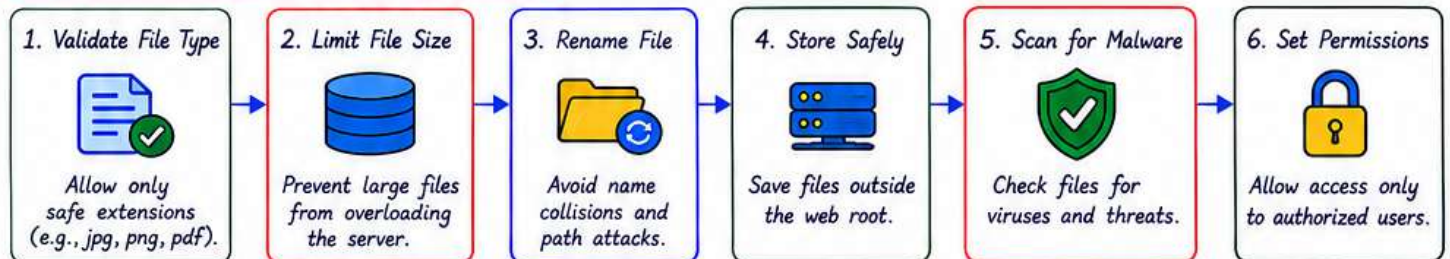


Goal: Keep files safe, valid and controlled.

Best Practices:

- ✓ Validate file type.
- ✓ Limit file size.
- ✓ Rename files.
- ✓ Store outside web root.
- ✓ Scan for malware.
- ✓ Use HTTPS.
- ✓ Set proper permissions.

21.12.2 Secure File Upload Practices



21.12.3 Example (Secure File Handling in Express.js)

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const crypto = require('crypto');

// Configure secure storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, path.join(__dirname, 'uploads'));
  },
  filename: (req, file, cb) => {
    const uniqueName = Date.now() + '-' +
      crypto.randomBytes(6).toString('hex') +
      path.extname(file.originalname);
    cb(null, uniqueName);
  }
});

const upload = multer({
  storage: storage,
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB limit
  fileFilter: (req, file, cb) => {
    const allowed = ['image/jpeg', 'image/png', 'application/pdf'];
    cb(null, allowed.includes(file.mimetype));
  }
});

app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) return res.status(400).send('No file uploaded');
  res.send('File uploaded securely!');
});
```

Explanation:

- ✓ Uses **Multer** with **diskStorage** for secure file saving.
- ✓ Generates **unique filename** to avoid overwriting.
- ✓ Validates file type (**MIME type**).
- ✓ Limits file size to **5MB**.
- ✓ Stores files in a safe **'uploads'** directory.

21.12.4 Secure File Handling Flow



Common Security Risks & Solutions

Risk	Solution
Malicious file upload	Validate file type & scan
File size attack	Set file size limits
Path traversal	Use safe filenames
Unauthorized access	Set proper permissions
Data leakage	Store files outside web root

Key Takeaway:



Always **validate**, **limit**, and **secure files** during upload and storage to keep your application safe.

