Implementing File Upload on the Server Application

Objective:

Implement a server-side file upload system using Node.js and Express that can properly receive, validate, store, and serve files uploaded from the React/Next.js frontend. You will learn how to handle multipart form data on the server, implement file type validation, size limitations, and proper error handling while creating a secure and efficient file storage system.

By connecting the frontend drag-and-drop interface with the backend implementation, you will gain a comprehensive understanding of the complete file upload flow from client to server in modern web applications.

Implementation Flow

1. **Setup the Express server environment** - Install necessary dependencies and create the basic server structure to handle HTTP requests.

2. **Configure middleware for file handling** - Implement Multer middleware to process multipart form data and manage file storage on the server.

3. **Create upload endpoints** - Develop API routes that accept file uploads, validate content types and sizes, and store files securely.

4. **Implement validation and security** - Add checks for file types, size limits, and proper error handling to prevent security vulnerabilities.

5. **Connect the frontend to the backend** - Modify the frontend code to communicate with the Express server instead of the Next.js API routes.

6. **Test the complete system** - Verify that files can be uploaded from the frontend, properly processed by the backend, and accessed when needed.

Refer here for implementation: https://github.com/syangche/Backend_Practicals.git

Step 1: Set Up Your Express Server

First, create a new Node.js project for your backend:

```
mkdir file-upload-server
cd file-upload-server

npm init -y

npm install express cors multer morgan dotenv
```

These packages serve the following purposes:

- `express`: Web server framework
- `cors`: Middleware to enable Cross-Origin Resource Sharing
- `multer`: Middleware for handling multipart/form-data (file uploads)
- `morgan`: HTTP request logger
- `dotenv`: For environment variable management

Step 2: Create the Basic Server Structure

1. Create a basic server structure in `server.js`:

```
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');
const path = require('path');
const fs = require('fs');
require('dotenv').config();

// Initialize express app
const app = express();
const PORT = process.env.PORT || 8000;

// Middleware
app.use(cors());
app.use(express.json());
app.use(morgan('dev')); // HTTP request logging

// Create uploads directory if it doesn't exist
const uploadDir = path.join(__dirname, 'uploads');
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir, { recursive: true });
}

// Serve uploaded files statically
app.use('/uploads', express.static(uploadDir));

// Basic route for testing
app.get('/', (req, res) => {
  res.send('File Upload Server is running');
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

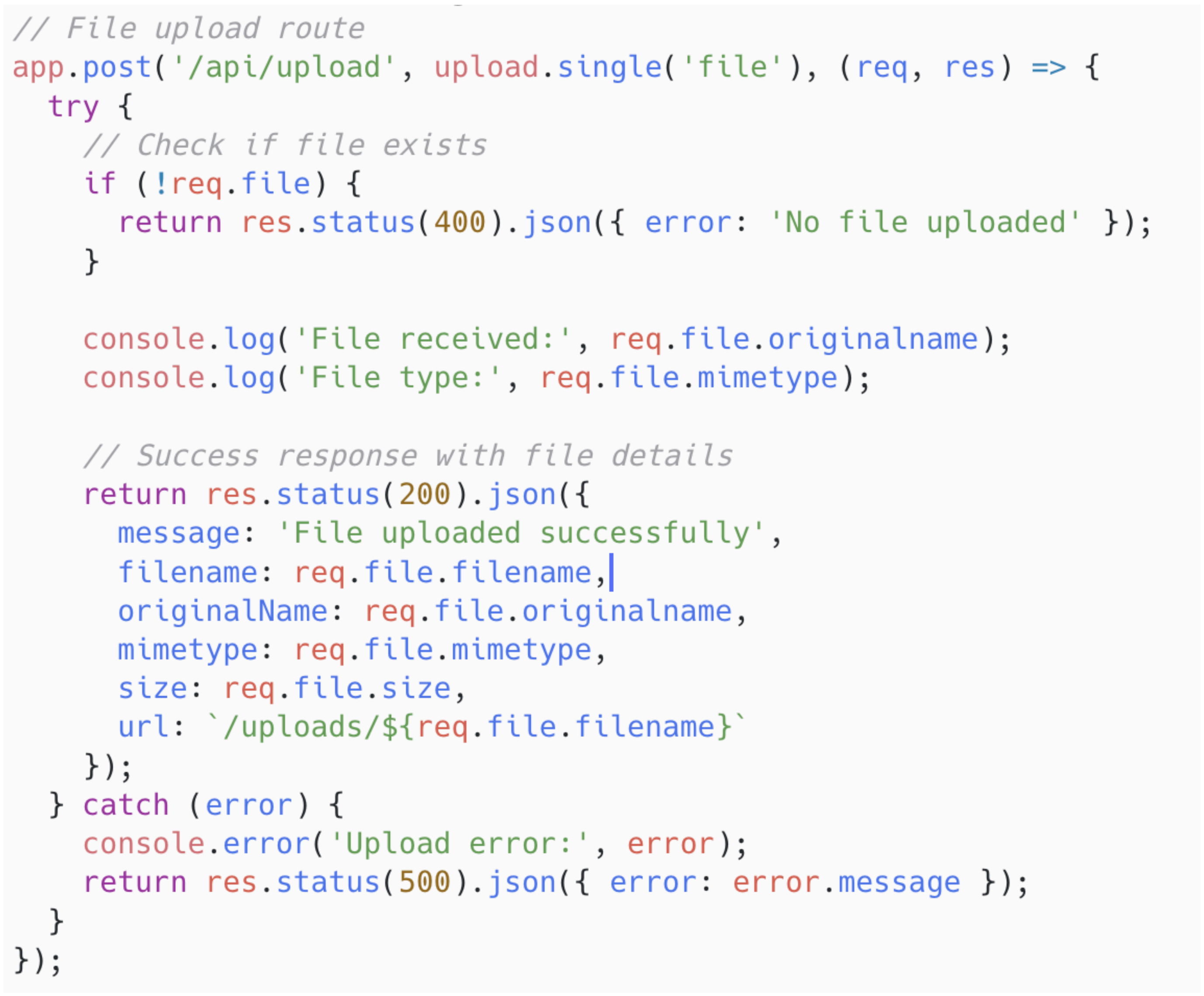
Part 2: Configuring Multer for File Uploads

1. Add Multer configuration to handle file uploads in `server.js` after the middleware setup:

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

// Configure storage
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, uploadDir);
  },
  filename: function (req, file, cb) {
    // Create unique filename
    const timestamp = Date.now();
    const originalName = file.originalname;
    cb(null, `${originalName}`);
  }
});

// Configure file filter
const fileFilter = (req, file, cb) => {
  // Accept only specific mime types
  const allowedMimeTypes = ['image/jpeg', 'image/png', 'application/pdf'];

  if (allowedMimeTypes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(new Error('Invalid file type. Only JPEG, PNG and PDF files are allowed.'), false);
  }
};

// Create the multer instance
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 5 * 1024 * 1024 // 5MB in bytes
  },
  fileFilter: fileFilter
});
```

Part 3: Creating the Upload API Endpoint

1. Add an upload route to `server.js` after multer configuration:



```javascript
// File upload route
app.post('/api/upload', upload.single('file'), (req, res) => {
  try {
    // Check if file exists
    if (!req.file) {
      return res.status(400).json({ error: 'No file uploaded' });
    }

    console.log('File received:', req.file.originalname);
    console.log('File type:', req.file.mimetype);

    // Success response with file details
    return res.status(200).json({
      message: 'File uploaded successfully',
      filename: req.file.filename,
      originalName: req.file.originalname,
      mimetype: req.file.mimetype,
      size: req.file.size,
      url: `/uploads/${req.file.filename}`
    });
  } catch (error) {
    console.error('Upload error:', error);
    return res.status(500).json({ error: error.message });
  }
});
```

2. Add error handling middleware for Multer after the Routes:

```
// Error handling for multer errors
app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    // A Multer error occurred when uploading
    if (err.code === 'LIMIT_FILE_SIZE') {
      return res.status(413).json({ error: 'File too large. Maximum size is 5MB.' });
    }
    return res.status(400).json({ error: err.message });
  }

  // For any other errors
  console.error(err);
  res.status(500).json({ error: 'Server error' });
});
```

Part 4: Configuring CORS for Frontend Connection

1. Since your frontend and backend are running on different ports, you need proper CORS configuration. Update the CORS configuration in server.js:

```
// Replace the simple cors() call with:
app.use(cors({
  origin: process.env.FRONTEND_URL || 'http://localhost:3000',
  methods: ['GET', 'POST'],
  allowedHeaders: ['Content-Type']
}));
```

Create a `.env` file in your backend project:

```
PORT=8000
FRONTEND_URL=http://localhost:3000
```

Part 5: Modify Your Frontend to Connect to the Express Backend

Now, we need to update your React frontend to connect to your Express backend instead of the Next.js API route. Here's how to modify your `page.js` component:

1. Update the onSubmit function to point to your Express backend

```
// Update the onSubmit function to use your Express backend URL
const onSubmit = async (data) => {
  setIsUploading(true);
  setUploadProgress(0);
  setUploadResult(null);

  try {
    const formData = new FormData();
    formData.append('file', data.file[0]);
    formData.append('name', data.name);

    // Log what we're trying to upload for debugging
    console.log('Uploading file:', data.file[0].name, 'Type:', data.file[0].type);

    // Change the URL to point to your Express backend
    const response = await axios.post('http://localhost:8000/api/upload', formData, {
      onUploadProgress: (progressEvent) => {
        const percentage = Math.round(
          (progressEvent.loaded * 100) / progressEvent.total
        );
        setUploadProgress(percentage);
      }
    });

    setUploadResult({
      success: true,
      message: 'File uploaded successfully!',
      data: response.data
    });
  } catch (error) {
    console.error('Upload error:', error);
    setUploadResult({
      success: false,
      message: error.response?.data?.error || 'Upload failed'
    });
  } finally {
    setIsUploading(false);
  }
};
```

The key change here is updating the axios POST URL to point to your Express server's endpoint (`http://localhost:8000/api/upload`)

2. Modify the Dropzone component to handle PDF files better

```
// Inside the onDrop callback:
onDrop: (acceptedFiles) => {
  // Process the dropped files
  onDrop(acceptedFiles);

  // Create preview info for the file (both images and PDFs)
  if (acceptedFiles.length > 0) {
    const file = acceptedFiles[0];

    // For images, create preview URL
    if (file.type.startsWith('image/')) {
      const previewUrl = URL.createObjectURL(file);
      setFilePreview({
        url: previewUrl,
        name: file.name,
        type: file.type
      });
    }
    // For PDFs, just store the name and type (no preview URL needed)
    else if (file.type === 'application/pdf') {
      setFilePreview({
        name: file.name,
        type: file.type
      });
    }
    else {
      // For other file types
      setFilePreview(null);
    }
  }
},
```

3. Update the preview section to display PDF filenames.
   Replace the original preview section with:

```
{filePreview && (
  <div className="mb-4">
    <h3 className="font-medium mb-1">Preview:</h3>
    <div className="border rounded p-2">
      {filePreview.type?.startsWith('image/') ? (
        /* Show actual preview for images */
        <img
          src={filePreview.url}
          alt={filePreview.name}
          className="max-w-full h-auto max-h-40 rounded"
        />
      ) : filePreview.type === 'application/pdf' ? (
        /* Just show file name for PDFs */
        <div className="py-2 px-3 bg-gray-100 rounded flex items-center">
          <svg className="w-6 h-6 text-red-500 mr-2" fill="currentColor" viewBox="0 0 20 20">
            <path d="M9 2a2 2 0 00-2 2v8a2 2 0 002 2h6a2 2 0 002-2V6.414A2 2 0 0016.414 5L14 2.586A2 2 0 0012.586 2H9z"></path>
            <path d="M3 8a2 2 0 012-2h2a2 2 0 012 2v8a2 2 0 01-2 2H5a2 2 0 01-2-2V8z"></path>
          </svg>
          <span>{filePreview.name}</span>
        </div>
      ) : (
        /* Generic file info for other types */
        <div>File selected: {filePreview.name}</div>
      )}
    </div>
  </div>
)}
```

Testing Your Implementation

1. Start your Express backend (`node server.js`)
2. Start your Next.js frontend (`npm run dev`)
3. Navigate to your file upload form in the browser
4. Upload files and check that:
   - Progress is tracked correctly
   - Validation works (file type and size)
   - Successful uploads return proper information
   - Files are saved in the backend's `uploads` directory
   - Error handling works for invalid files

Key Concepts

1. Multipart Form Data

Concept: When a file is uploaded through a form, it's sent as `multipart/form-data`. This format allows both text fields and binary data (files) to be sent in the same HTTP request.

Implementation:

- Frontend: `FormData` object appends the file and other form fields
- Backend: Multer middleware parses this multipart data, extracts the files, and handles saving them to disk

2. File Storage with Multer

Concept: Multer handles the complex process of:

- Parsing multipart form data
- Extracting file objects
- Saving files to disk with proper naming
- Providing metadata about the uploaded files

Key Components:

- `storage`: Defines where and how files are stored
- `fileFilter`: Validates file types
- `limits`: Controls file size and count limits

3. Error Handling

Concept: File uploads can fail for many reasons (invalid type, too large, server errors). Proper error handling ensures users get meaningful feedback.

Implementation:

- Frontend: Catches and displays errors from the backend
- Backend: Custom error middleware catches and formats errors from Multer and other sources

4. CORS (Cross-Origin Resource Sharing)

Concept: Browsers restrict cross-origin HTTP requests as a security measure. CORS is a mechanism that allows servers to specify who can access their resources.

Implementation: The `cors` middleware configures which origins, methods, and headers are allowed when your frontend makes requests to your backend.

5. Progress Tracking

Concept: For large files, tracking upload progress is important for user experience. This is implemented using the `onUploadProgress` callback in Axios.

Implementation:

- Frontend: Uses Axios's `onUploadProgress` to track and display progress
- Backend: Express/Multer processes the upload in chunks, allowing the browser to track progress