**CPAN 212 – Modern Web Technologies**

**Lab 3**

**Task:**

1. **Use Express.js application and React client built in week 5.**
2. **Create route to get multiple images from Express.**
3. **Call get multiple images api from Express and display images in React.**
4. **Add button in React to get random dog image from:** [**https://dog.ceo/api/breeds/image/random**](https://dog.ceo/api/breeds/image/random)
5. **Allow uploading random dog image to Express using submit button.**

**Submission:**

* **Screenshots of results**
  + **Uploading multiple images**
  + **Getting the dog image**
  + **Submitting the dog image into the server**
* **Zip folder with code (please delete the node\_modules)**

**Weight: 4% of final grade**

|  |  |
| --- | --- |
| Rubric | Weightage |
| Express server   * Organized files * Modular structure | 30 |
| React/ Front End application   * This can be in react or another language of your choosing, I want something here * Add some styling, it will be work 10/30. Nothing serious, center the content, make sure it isn’t just slapped together on the left of the screen | 30 |
| Display multiple images in your front end | 20 |
| Get random dog image in React and upload to Express   * The process: fetch for a random dog image, press the upload button to send it to the server -> screenshots | 20 |
| Total | 100 |

CPAN 212 – Modern Web Technologies – Harman Mann

Note:

I will help you with 60% of the work. As we move forward with the labs, I will provide you more freedom to work and break things. +20% per lab. Be sure to google your problems and get help online wherever necessary.

**How to get started**

We are going to build 2 applications. A react frontend and an express backend. The goal will be to separate the frontend and the backend so you can continue working on your frontend skills at home while we build a server in class to handle requests.

You used to use CRA (create react app) and that has been deprecated – it is no longer receiving updates and even they recommend you switch over to something like Next.js, Remix, or in our case Vite. I will make a quick showcase in this document to help you get started with the Vite project, and I will showcase how to get both the server and client running. I will also showcase a sample request on sending a file with Multer to the server, saving it, and calling back to it.

This document might be content heavy as I am trying to push this as an example for your project structure.

Let’s first look at the server. You will be using these modules:

* Express
* Nodemon
* Lodash – this will handle our array operations
* Multer
  + So far, you have sent string from a form
  + What about files? Multer will help you send file data to your form and
* Cors
  + Because we are sending files/data across applications, we must enable sharing files/data between different apps – Cors: Cross origin resource sharing

**What about our App(s) structure(s)?**

We must make a few routes to handle specific tasks, **I will highlight the tasks that I will hand off to you to work on by yourselves or in groups if you are working with them on your project.**

**Server:**

* We will save files into our server -> **we would receive data that needs to be stored -> POST requests**
  + Save one file -> /save/single
  + Save multiple files -> /save/multiple
* We will want to fetch the files on the server -> **we would send data -> GET requests**
  + Fetch random single file -> /fetch/single
  + Fetch multiple random files -> /fetch/multiple
  + Fetch all files -> /fetch/all

Now, I have its setup like this because I want you to recognize that there is a common theme between the routes, so we can set up our server with routers to help organize our routes. Something you did in lab 2, when you made the lab2 router.

**Client:**

We need to handle requesting data from the server and processing the response we get from it. What I want from you is to setup webpages than handle sending files to the server and fetching files from the server. So, using the knowledge from your previous course, take some time to build a frontend. Style it however you need. I will mark the frontend a little bit.

With the corresponding commands from the server, we can setup commands to the front end that sends files and fetch the information from the server. Let’s make something quick for the lab, a 1-page application that handles sending files and fetching files.

**Project Setup**

1. Create a Folder called: Lab3
2. Create 2 sub folders called client, server
3. We will first setup our server
   1. Open an integrated terminal to our server
   2. Setup package.json: npm init -y
   3. Install modules: npm install express nodemon multer cors
4. Create a file: app.js
   1. Import the code from our base server file into this file
   2. Create a folder called “uploads” inside the server directory
   3. Go onto this link and integrate multer into our server: [multer - npm (npmjs.com)](https://www.npmjs.com/package/multer)

We plan to store our information on a folder called uploads. When we receive, we will be using the **disk Storage** feature inside the documentation.

Here is a completed version of our app.js at this point

const PORT = process.env.PORT || 8000;

const express = require("express");

const app = express();

const cors = require("cors")

// adding multer

const multer = require("multer");

const storage = multer.diskStorage({

  // the function that saves the file

  destination: function (req, file, cb) {

    // where we are storing the file

    cb(null, "./uploads");

  },

  filename: function (req, file, cb) {

    // what is the filename going to be

    cb(null, `${Date.now()}-${file.originalname}`); // this will save files like: date-filename.extension

  },

});

const upload = multer({ storage: storage }); // the middleware function that handles uploading

// middlelware

app.use(express.urlencoded({ extended: true }));

app.use(express.json());

app.use(cors())

// routes

app.get("/", (req, res) => {

  res.send("Welcome to our server");

});

app.listen(PORT, () => {

  console.log(`http://localhost:${PORT}`);

});

app.use("", (req, res) => {

  res.status(404).send("Page not found");

});

1. Next let’s add a save route, lets follow the example from the document on npm

A screen shot of a computer

Description automatically generated

The idea here, following our middleware class, 1 represents the route we hit, 2 represents the middleware operation (the function) we want to run, and in this case to save the file(s) and finally 3 to send back a response.

Let’s try building the single upload following the example here. This will cover the /save/single and /save/multiple routes we need.

**Note: we need to have a dedicated tag name so that we know what element is being saved from the request. In the example above, it was “avatar”, when you send the information to the server, declare a specific name from the Form for the file.**

Add something like this into your server, put it under the homepage function of “/”

app.post("/save/single", upload.single("file"),  (req, res) => {

  res.send("File uploaded");

});

app.post("/save/multiple", upload.array("files", 100), (req, res) => {

  if (!req.files || req.files.length === 0) {

    return res.status(400).send("No files uploaded.");

  }

  const filePaths = req.files.map((file) => file.path);

  res.send(`Files uploaded successfully: ${filePaths.join(", ")}`);

});

If you are curious on where I found error handling, if you search for “error handling” on the document, it will also provide you an example. What I’m trying to showcase here is the use of documentation and how you can consider making documentation in the future if you are building for developers.

1. Let’s add some fetch commands

Let’s add a fetch command that finds a file in the directory and sends the information to the client. If you consider a few things, we will need to go back and add FS and PATH so we can access and send these files. Now, we are sending back file data, not storing it, SOOOO, we should use GET to send them data to read.

Let’s talk about this for a sec, what do we want to do:

* Find and read the uploads folder (we did this under week 2 – “02-02-fspath-reads.js”)
* Create an operation to read a random file, use whatever operation you want to do this
* Send the file to the front end and have them handle it

Add this into the app.js file, under the /save/multiple route

app.get("/fetch/single", (req, res) => {

  // we read the directory items synchronously to not trip the async speed

  let files\_array = fs.readdirSync(path.join(\_\_dirname, "uploads"));

  // error checking

  if (files\_array.length == 0) {

    // adding return will stop the rest of the operations

    return res.status(503).send({

      message: "No images",

    });

  }

  let filename = \_.sample(files\_array);

  res.sendFile(path.join(\_\_dirname, "uploads", filename));

});

I will leave fetch multiple for your lab work. Here is a tip, look at single and how it works, you can use that to build up a multiple. And lodash has an additional function on top of sample to send multiple items (sampleSize  
).

The completed code for lab 3 starter can be found on the public github.

[CPAN212-F2024-PublicRepository/Labs/Lab03-starter/server at main · HarmanSMann/CPAN212-F2024-PublicRepository (github.com)](https://github.com/HarmanSMann/CPAN212-F2024-PublicRepository/tree/main/Labs/Lab03-starter/server)

**Organizing our Files on the server (good practice – and marks)**

Now we have a server file that’s about 100 lines, lets try organizing it a little.

1. Create a folder called routers
   1. Create 2 files: fetch\_router.js, save\_router.js
   2. Look back at our routes
      1. /save/multiple
      2. /save/single
      3. /fetch/single
      4. /fetch/multiple (you will have to make this later)
   3. There is some commonality with them, lets organize them and make them router files
   4. This is like lab 2

Each file will start like this, and we will move our code into those files:

const express = require("express");

const router = express.Router();

module.exports = router;

1. Lets break apart the routes so we can call them like in lab 2: remember -> app.use(“/lab2”, lab2\_router)
   1. Similarly, like that, using the commonality of our routes -> /save and /fetch

At the end of our journey, we should have something like this:

A screenshot of a computer program

Description automatically generated

**Setting up a client side for React**

1. Open an integrated terminal to the client folder
2. Enter this command: “npm create vite@latest .” (add the period, it means it will make it in the current directory)
   * READ the prompts on your terminal, it will help you setup a react project, pick React and JavaScript when the options come up (use your arrow keys)
3. Next install the packages: npm install
   * This will take a while, there are a lot of packages
4. To run your app: npm run dev
5. Optional
   * If you want to change the default port settings go into vite.config
   * A screen shot of a computer program

     Description automatically generated

Your folder structure will look something like this for the client side

A screenshot of a computer

Description automatically generated

For this lab, lets make a single page application (SPA) that can upload and fetch. This way I can ease you into working with Vite and react again. For now, if you need a form, go and check some of your old code and try adding it in. Oh, is that not enough, fineeeee, here is a sample that I can get started with.

Go into /client/src/App.jsx and let’s modify the file

Let’s start out with this as our file, I erased everything to get us here:

import { useState } from 'react'

function App() {

// variable

// handle event changes

// form submission

  return (

    <>

// form information

    </>

  )

}

export default App

The first thing that we will need to do is make a function that handles our single upload

This needs a variable, a setter function, a form submission, and some html to indicate what to do. (I lay it out almost one for one of how we add it into the file

import { useState } from "react";

function App() {

  const [singleFile, setSingleFile] = useState(null); // variable

  // Handle file input for single upload

  const handleSingleFileChange = (e) => {

    setSingleFile(e.target.files[0]);

  };

  // Upload a single file to the server

  const uploadSingleFile = async () => {

    const formData = new FormData();

    formData.append("file", singleFile);

    try {

      const response = await fetch("http://localhost:8000/save/single", {

        method: "POST",

        body: formData,

      });

      const data = await response.json();

      alert(data.message);

    } catch (error) {

      console.error("Error uploading single file:", error);

    }

  };

  return (

    <>

      <h1>File Upload and Fetch App</h1>

      {/\* Section for uploading single file \*/}

      <div>

        <h2>Upload Single File</h2>

        <input type="file" onChange={handleSingleFileChange} />

        <button onClick={uploadSingleFile}>Upload Single File</button>

      </div>

    </>

  );

}

export default App;

From here, we can do the multiple instances with the same information, with a modification to the form. We will get into this now, but there is an additional attribute keyword “multiple” to allow multiple items to be selected.

Variable setup:

const [multipleFiles, setMultipleFiles] = useState([]);

html:

      {/\* Section for uploading multiple files \*/}

      <div>

        <h2>Upload Multiple Files</h2>



        <input type="file" multiple onChange={handleMultipleFilesChange} />

        <button onClick={uploadMultipleFiles}>Upload Multiple Files</button>

      </div>

And for the function, we get the items and an array, so we need to append them all into the form

  // Upload multiple files to the server

  const uploadMultipleFiles = async () => {

    const formData = new FormData();

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

      formData.append('files', multipleFiles[i]);



    }

    try {

      const response = await fetch('http://localhost:8000/save/multiple', {

        method: 'POST',

        body: formData,

      });

      const data = await response.json();

      alert(data.message);

    } catch (error) {

      console.error('Error uploading multiple files:', error);

    }

  };

The final lap for the lab: Fetching from the server

  const [fetchedSingleFile, setFetchedSingleFile] = useState(null);

fetch function

  // Fetch a random single file from the server

  const fetchSingleFile = async () => {

    try {

      const response = await fetch('http://localhost:8000/fetch/single');

      const blob = await response.blob();

      const url = URL.createObjectURL(blob);

      setFetchedSingleFile(url);

    } catch (error) {

      console.error('Error fetching single file:', error);

    }

  };

Here is the unique thing about this code. We have something called blob(). When we send a file, it must get transformed into something we can send over through HTTP. We convert it from the file into a blob – **b**inary **l**arge **ob**ject. That’s why we call blob on the response, to transform the object back into what it was originally.

Html:

      <div>

        <h2>Fetch Single File</h2>

        <button onClick={fetchSingleFile}>Fetch Single File</button>

        {fetchedSingleFile && (

          <div>

            <h3>Single File</h3>

            <img src={fetchedSingleFile} alt="Fetched Single" style={{ width: '200px', marginTop: '10px' }} />

          </div>

        )}

      </div>

All the code is available on the GitHub to get you moving for your part of the lab. It will be called starter code

**The Lab**

What you need to do:

1. Server Side
   1. Create a route that handles multiple file uploads -> **/fetch/multiple**
   2. Hint: lodash has a function called: sampleSize(path, number) that allows you to pick how many items at random to send
2. Client Side
   1. Create a section on the app that will call to the **/fetch/multiple** from the server and return a collection of images (call to: <http://localhost:8000/fetch/multiple>)
      1. Hint: when you receive the response, use a loop/ for each operator to go through each item and apply blob () to then display the information
   2. Create a section that calls to the API and displays the image
   3. Send the image to the server