Full-Stack Image upload

Workshop

OVERVIEW

In this exercise, you will have the opportunity to practice in building an image upload feature using react, multer, cloudinary, and multer-storage-cloudinary. This workshop will guide you through the entire process of uploading images from the client-side to the server-side, storing them securely, and leveraging the power of Cloudinary for image management and delivery

GOALS

1. Learn how to handle image uploads using React and React Hook Form
2. Using Multer for parsing the multipart/form-data on the server-side
3. Using Cloudinary to save the image on the cloud
4. Using multer-storage-cloudinary to integrate multer and cloudinary easily

SPECIFICATIONS

Create a Full-Stack Image Upload App that will provide a solution for users to upload and manage images

Exercises

Setup

- Create a github repository and clone it
    - make sure it was initialized with a README.md file and a .gitignore file for node
- Create a cluster on mongoDB Atlas if you haven't got one running
- Create a basic express server with the following directories in the root directory `**routes**`, `**controllers**`, and `**models**`
    - Include `**express.json**`, and `**cors**` middelware
    - Setup the env variables using `**dotenv**`
    - Create a connection to the remote db on atlas inside `**db.js**`
- Setup a new **react** application with routing

Exercise 01

- Create a `**models/user.js**` file that contains the schema and model for the **users** collection
    - **Create** the **userSchema** with the following fields
        - **email** => String, Unique, Required
        - **name** => String, Required
    - **Create** and **export** the user model
- Create a `**models/products.js**` file that contains the schema and model for the **products** collection
    - **Create** the **productSchema** with the following fields
        - **name**=> String, Required
        - **price**=> Number, Required
        - **image=>** String, Required
        - **owner=>** ObjectId, ref: UserModel (use the same name for the user model)
    - **Create** and **export** the product model

Exercise 02

- Create **POST** request route inside of `**routes/user.js**` that should create a new **user**
- Create **POST** request route inside of `**routes/products.js**` that should create a new **product** (test using an http client, insomnia, postman, thunder client, etc...)
    - Create a new file called **uplaodImage.js** inside of a directory called **middlewares** `**/middlewares/uploadImage.js**`
    - use `**multer**` middleware to parse the incoming form data and save the image on our server
    - export the multer instance then import and use it in the products router
    - express has a limit size for the body that can be modified using `**express.json()**`, if the uploaded image exceeds the size then you must increase the default size for the request body
    - Check the documentation to get some help
- Create a **GET** request route to retrieve all the **products** with the owner information

Key takeaways from Exercise 02:

- Saving the filename as a temporary solution is just the initial step. Our goal is to enhance the image upload process by uploading it to an online service specifically designed for hosting images, such as **Cloudinary**. Once the image is successfully uploaded to **Cloudinary**, we can obtain the corresponding **URL**.

Exercise 03

- In the frontend create a **component** that displays a **form** for creating a **product**, the **image** input should be of type **file**
    - Use `**react-hook-form**` to manage form state, validation, and submission
    - in the POST request **body** send the data as FormData, to be able to send the **file** as content type of `**multipart/form-data**`
    - when **appending** to the **FormData** the provided `**name**` will be used by the backend when **parsing** the **file** or **body** data

Key takeaways from Exercise 03:

- In the upcoming lessons, you will discover how to incorporate authentication tokens to retrieve the authenticated user's ID directly from the request. This approach eliminates the necessity of including an input field for the user's ID
- **FormData** is useful when dealing with file uploads because it supports the `**multipart/form-data**` content type and provides a built-in way to send files as part of an HTTP request. This is beneficial because file data **cannot** be easily **serialized** into a **JSON** object. By using **FormData**, we can properly **format** and **transmit** file data within the request, ensuring compatibility and effective handling of file uploads

Exercise 04

- Create a new account on Cloudinary
- Set up Cloudinary (in the getting started section in your console)
    - Install the cloudinary npm package and multer-storage-cloudinary
    - Update the **uploadImage.js** to use both cloudinary and multer-storage-cloudinary, to upload the image to cloudinary
    - Use `**cloudinary.config**` to add the configuration for your own account. Make sure to use **env variables** to securely store and access your cloudinary credentials (cloud name, api key, api secret)

```
const cloudinary = require("cloudinary").v2;

cloudinary.config({
 cloud_name: process.env.CLOUD_NAME,
 api_key: process.env.API_KEY,
 api_secret: process.env.API_SECRET,
});
```

- use **CloudinaryStorage** from multer-storage-cloudinary as your multer storage like it shows in the documentation, when

```
const storage = new CloudinaryStorage({
 cloudinary: cloudinary,
 params: {
   // configuration
 },
});
const upload = multer({ storage });
module.exports = upload;
```

- The params for **CloudinaryStorage** can be found here, for this exercise you only need the `**folder**` and ` **allowedFormats**`

```
params: {
   folder: "folder_name",
   allowedFormats: ["png", "jpg", "jpeg"]
 },
```

- Update the **POST** request endpoint at `**/product**s` to save the url for the uploaded image, you will find the url for the uploaded image in `req.file.path`

Key takeaways from Exercise 04:

- **Cloudinary** is a cloud-based media management platform that offers a comprehensive suite of tools and services for **managing**, **optimizing**, and **delivering images**, **videos**, and **other media assets** in web and mobile applications

Exercise 05

- Create a component to display all the **products** in the frontend, you must show the product **image** alongside the **name**, **price**, and the owner's **name**
- Create **DELETE** request route inside of `**routes/products.js**` that should **delete** a **product** based on the **id**

- When deleting a product it is wise to delete the associated image from Cloudinary as well to achieve that do the following:
    - Create a new schema called `**imageSchema**` in `**models/products.j**s`, it will be used as a sub schema in side the `**productSchema**` and that is why it is in the same file
        - `**url**` => String, Required
        - `**publicId**` => String, Required
    - Modify your **POST** request to `**/products**` to reflect the changes on the `**productSchema**`, you can get the `**publicId**` from `req.file.filename`
    - Modify your **DELETE** request to use the `**destroy**` method from **Cloudinary** to delete the file with the matching `**publicId**` from **Cloudinary**

Key takeaways from Exercise 05:

- In Mongoose, subschemas are a way to define reusable schemas that can be embedded within other schemas. A subschema is essentially a schema definition that can be nested within another schema as a field. This allows for creating complex and structured data models by combining multiple schemas together
- In Cloudinary, the `**publicId**` is a unique identifier that represents a specific resource (image, video, etc.) stored in your Cloudinary account. It is used to reference and manipulate resources within the Cloudinary system