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, and 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

SPECIFICATIONS

Create a Full-Stack Image Upload App that will provide a solution for users to upload and manage images, leveraging React Hook Form, Multer, and Cloudinary for efficient file handling and storage

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 insomnia)
    - use `**multer**` middleware to parse and save the file on the server
    - you can save the filename in the image field (this is temporarily)
    - 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
- 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 ultimate 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**. At that point, we can safely **delete** the temporarily stored file from our server, ensuring efficient storage management and leveraging the benefits of a robust image hosting service

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
  - Connect to Cloudinary using the `` `**cloudinary.config**` `` method. Make sure to use **env variables** to securely store and access the credentials
  - Access the **settings** in the Cloudinary console. **Navigate** to the **Upload** section in the side menu and click on `` `**Add upload preset**` `` in the **Upload presets** section
  - Create a new preset with the `` `**Signing Mode**` `` set to `` `**unsigned**` ``. This ensures that the upload preset does not require a signature, simplifying the upload process
  - **Save** the upload preset to apply the configuration
- update the **POST** request server endpoint at `` `**/product**s` `` to save a new image in Cloudinary (you can view the uploaded image the Media Library section in the console)
  - using `` `**cloudinary.v2.uploader.unsigned_upload**` `` method you can upload an image to Cloudinary
    - - The first parameter is the path for the file on the server, you can access it from `` `**req.file.path** ``
    - the second parameter is the **unsigned** preset **value**, you can find it in the settings section`
  - after the upload is **successful** you can find the **url** in the **response**, specifically the **secure_url** key, use that value to save it in the product's **image** field
  - after the **product** is saved, delete the **file** from the **server** using the `` `**fs**` `` module

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
- The choice between **unsigned** and **signed** uploads depends on the specific requirements of your application. If you prioritize simplicity and speed, and security is not a major concern, unsigned uploads can be a suitable option. However, if you require enhanced security, control, and validation capabilities, signed uploads provide a more robust and secure approach

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 the Cloudinary response after successfully uploading an image
    - 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