**Gallery API Documentation**

**Introduction**

The Gallery API is designed to facilitate the management of image files within a web application. It allows users to upload images, store them on a cloud service (like Cloudinary), and manage these images via CRUD (Create, Read, Update, Delete) operations. The API is built using Spring Boot for the backend and Angular for the frontend. Images are stored in Cloudinary, and metadata is stored in a MySQL database.

Main Features

Image Uploading: Users can upload images via the API, which are stored on Cloudinary.

Image Retrieval: The API provides endpoints to fetch a list of uploaded images.

Image Deletion: Users can delete images from both Cloudinary and the local database.

Image Metadata Management: The API manages the image metadata, including the image URL, public ID, and other relevant information.

Backend: Spring Boot

Dependencies

1. Spring Web: For creating RESTful web services.

2. Spring Data JPA: For database operations.

3. MySQL Driver: For connecting to the MySQL database.

4. Spring Boot DevTools: For development and testing.

5. Lombok: To reduce boilerplate code.

6. Cloudinary: For uploading and managing images in the cloud.

Package Structure

1. `com.example.gallery`: Root package.

2. `com.example.gallery.config`: Contains configuration classes, including Cloudinary configuration.

3. `com.example.gallery.controller`: Contains controller classes for handling HTTP requests.

4. `com.example.gallery.service`: Contains service interfaces.

5. `com.example.gallery.service.impl`: Contains the implementation of service interfaces.

6. `com.example.gallery.repository`: Contains repository interfaces for database operations.

7. `com.example.gallery.model`: Contains entity classes mapped to database tables.

Entity Class: Image

```java

@Entity

public class Image {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String publicId;

private String url;

// Other fields like timestamp, uploader, etc.

// Getters and Setters

}

```

Purpose: Represents an image entity that is stored in the MySQL database. It contains fields like `publicId` (the unique identifier from Cloudinary) and `url` (the link to the image).

Service Interface: ImageService

```java

public interface ImageService {

void saveImage(Map<String, Object> uploadResult);

List<Image> getAllImages();

void deleteImage(String publicId);

}

```

Purpose: Defines the operations that the API will support, including saving, retrieving, and deleting images.

Service Implementation: ImageServiceImpl

```java

@Service

public class ImageServiceImpl implements ImageService {

@Autowired

private ImageRepository imageRepository;

@Override

public void saveImage(Map<String, Object> uploadResult) {

Image image = new Image();

image.setPublicId((String) uploadResult.get("public\_id"));

image.setUrl((String) uploadResult.get("url"));

imageRepository.save(image);

}

@Override

public List<Image> getAllImages() {

return imageRepository.findAll();

}

@Override

public void deleteImage(String publicId) {

imageRepository.deleteByPublicId(publicId);

}

}

```

Purpose: Implements the operations defined in the `ImageService` interface.

`saveImage`: Persists image metadata into the database.

`getAllImages`: Retrieves all images stored in the database.

`deleteImage`: Deletes an image by its public ID.

Controller: GalleryController

```java

@RestController

@RequestMapping("/api/gallery")

public class GalleryController {

@Autowired

private ImageService imageService;

@PostMapping("/upload")

public ResponseEntity<?> uploadImage(@RequestParam("file") MultipartFile file) {

Map uploadResult = cloudinary.uploader().upload(file.getBytes(), ObjectUtils.emptyMap());

imageService.saveImage(uploadResult);

return new ResponseEntity<>(uploadResult, HttpStatus.OK);

}

@GetMapping("/images")

public List<Image> getImages() {

return imageService.getAllImages();

}

@DeleteMapping("/delete/{publicId}")

public ResponseEntity<?> deleteImage(@PathVariable String publicId) {

cloudinary.uploader().destroy(publicId, ObjectUtils.emptyMap());

imageService.deleteImage(publicId);

return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

}

```

Purpose: Acts as a bridge between the frontend and the backend. It handles HTTP requests and uses the `ImageService` to perform the necessary operations.

Endpoints

1. `POST /api/gallery/upload`:

Purpose: Upload an image to Cloudinary and save its metadata to the database.

Parameters:

`file` (MultipartFile): The image file to be uploaded.

Response: The Cloudinary response containing details like `public\_id`, `url`, etc.

2. `GET /api/gallery/images`:

Purpose: Retrieve a list of all uploaded images.

Response: A list of `Image` objects.

3. `DELETE /api/gallery/delete/{publicId}`:

Purpose: Delete an image by its `publicId`.

Parameters:

`publicId` (String): The unique identifier of the image in Cloudinary.

Response: HTTP 204 No Content.

Cloudinary Configuration

```java

@Configuration

public class CloudinaryConfig {

@Bean

public Cloudinary cloudinary() {

return new Cloudinary(ObjectUtils.asMap(

"cloud\_name", "your\_cloud\_name",

"api\_key", "your\_api\_key",

"api\_secret", "your\_api\_secret"));

}

}

```

Purpose: Configures the Cloudinary SDK with your account credentials.

Frontend: Angular

Service: GalleryService

```typescript

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { Image } from './image.model';

@Injectable({

providedIn: 'root'

})

export class GalleryService {

private baseUrl = 'http://localhost:8081/api/gallery';

constructor(private http: HttpClient) {}

uploadImage(file: File): Observable<any> {

const formData = new FormData();

formData.append('file', file);

return this.http.post(`${this.baseUrl}/upload`, formData);

}

getImages(): Observable<Image[]> {

return this.http.get<Image[]>(`${this.baseUrl}/images`);

}

deleteImage(publicId: string): Observable<any> {

return this.http.delete(`${this.baseUrl}/delete/${publicId}`);

}

}

```

Purpose: Interacts with the backend API to perform operations like uploading, retrieving, and deleting images.

Model: Image

```typescript

export class Image {

publicId: string;

url: string;

// Additional fields as needed

}

```

Purpose: Represents the structure of an image object as expected from the backend.

Component: GalleryComponent

```typescript

import { Component } from '@angular/core';

import { GalleryService } from './gallery.service';

import { Image } from './image.model';

@Component({

selector: 'appgallery',

templateUrl: './gallery.component.html',

styleUrls: ['./gallery.component.css']

})

export class GalleryComponent {

images: Image[] = [];

selectedFile: File = null;

constructor(private galleryService: GalleryService) {}

onFileSelected(event: any) {

this.selectedFile = <File>event.target.files[0];

}

uploadImage() {

this.galleryService.uploadImage(this.selectedFile).subscribe(() => {

this.getImages();

});

}

getImages() {

this.galleryService.getImages().subscribe((data: Image[]) => {

this.images = data;

});

}

deleteImage(publicId: string) {

this.galleryService.deleteImage(publicId).subscribe(() => {

this.getImages();

});

}

}

```

Purpose: Handles user interactions in the gallery UI, like selecting an image, uploading it, and displaying the list of images.

HTML Template: gallery.component.html

```html

<div class="gallerywrapper">

<h1>Gallery</h1>

<div class="uploadsection">

<input type="file" (change)="onFileSelected($event)">

<button (click)="uploadImage()">Upload</button>

</div>

<div class="imagessection">

<div ngFor="let image of images" class="imagecard">

<img [src]="image.url" alt="Image">

<button (click)="deleteImage(image.publicId)">Delete</button>

</div>

</div>

</div>

```

Purpose: Defines the structure of the gallery UI, including the image upload and display sections.

CSS: gallery.component.css

```css

.gallerywrapper {

maxwidth: 800px;

margin:

0 auto;

}

.uploadsection {

marginbottom: 20px;

}

.imagessection {

display: flex;

flexwrap: wrap;

gap: 15px;

}

.imagecard {

border: 1px solid ddd;

padding: 10px;

boxshadow: 0 2px 4px rgba(0, 0, 0, 0.1);

}

.imagecard img {

maxwidth: 100%;

height: auto;

}

.imagecard button {

margintop: 10px;

backgroundcolor: f44336;

color: white;

border: none;

padding: 5px 10px;

cursor: pointer;

}

.imagecard button:hover {

backgroundcolor: d32f2f;

}

```

Purpose: Styles the gallery UI, making it responsive and visually appealing.

Explanation of Key Concepts

1. Cloudinary Integration

Purpose: Cloudinary is used to store images in the cloud. This allows for scalable storage and easy management of image assets.

Why Cloudinary?: It provides features like automatic resizing, format conversion, and onthefly image transformations, making it ideal for managing images in web applications.

2. Storing Metadata

Purpose: The image metadata (such as `publicId` and `url`) is stored in a MySQL database to allow easy retrieval and management of images.

Why Store Metadata?: Storing metadata separately from the images allows for fast access to image information without needing to interact directly with Cloudinary for every request.

3. Use of RESTful APIs

Purpose: The API is designed as a RESTful service, which provides a standardized way to create, read, update, and delete resources.

Why REST?: RESTful APIs are stateless and can be easily consumed by frontend applications, mobile apps, or other services.

4. Error Handling and Status Codes

Purpose: The API returns appropriate HTTP status codes and responses, helping clients to understand the outcome of their requests.

Why?: This standardization ensures consistent and predictable behavior across the application, improving the user experience.

5. Multipart File Upload

Purpose: Allows for uploading images via the `POST /upload` endpoint.

Why Multipart?: Multipart forms are the standard way to upload files in HTTP, allowing the server to handle files and form data in a single request.

6. Dependency Injection

Purpose: Spring Boot uses dependency injection to manage components like `ImageService`, `ImageRepository`, and others.

Why?: Dependency injection promotes loose coupling and makes the code easier to test and maintain.

7. Angular Service and Component Interaction

Purpose: The Angular service interacts with the backend API, while the component handles user interaction and displays the images.

Why Separate?: Separating these concerns improves modularity, making the application easier to manage, maintain, and scale.

Conclusion

The Gallery API provides a robust and scalable solution for managing images within a web application. By leveraging Cloudinary for storage, Spring Boot for the backend, and Angular for the frontend, the API ensures that images are handled efficiently, with all relevant metadata stored for easy access and management. This architecture promotes separation of concerns, scalability, and ease of use, making it an ideal solution for modern web applications.