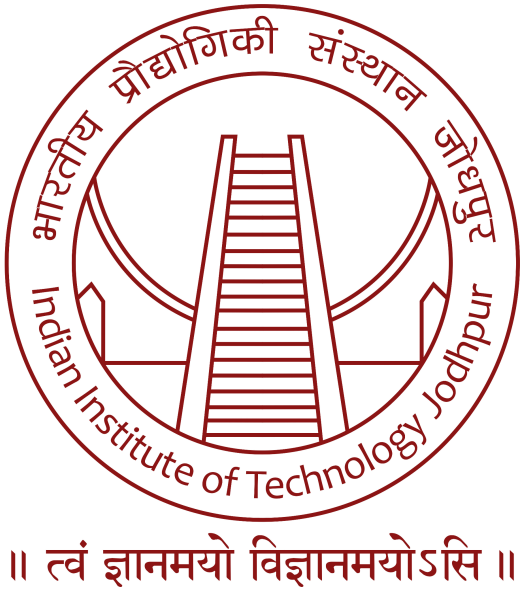
Software and Data Engineering: CSL7090
Code documentation

Serverless Image Processing Application



Instructor: Dr. Sumit Kalra

Team members
Ayush Jain (B21BB006)
Lokesh Kiran Chaudhari (B21CS041)
Shrashti Saraswat (B21CS081)

Github Link - Link

The coding part is divided into 2 parts: Cloud setup and Non-cloud setup.

Cloud Setup

JavaScript (mjs) File: AWS Lambda Function for Image Processing

Overview:

We are using the AWS Lambda function written in Node.js using the AWS SDK and Sharp (an image processing library). It processes images uploaded to an Amazon S3 bucket, applies transformations (resize, greyscale, blur), and stores the processed images back into a destination S3 bucket.

Dependencies:

- **@aws-sdk/client-s3**: AWS SDK module to interact with S3 (get and put objects).
- **sharp**: Image processing library to apply transformations like resizing, greyscale conversion, and blurring.

Code Breakdown:

1. Importing Required Libraries:

```
import { S3Client, GetObjectCommand, PutObjectCommand } from "@aws-sdk/client-s3";
import sharp from "sharp";
```

- **S3Client**: Creates an S3 client to interact with S3 service.
- **GetObjectCommand**: Used to retrieve an object (image) from the source S3 bucket.
- **PutObjectCommand**: Used to upload the processed image to the destination S3 bucket.

- **sharp**: A library used to manipulate images (resize, greyscale, blur).

2. Defining Constants:

```
const S3 = new S3Client();
const DEST_BUCKET = process.env.DEST_BUCKET;
const THUMBNAIL_WIDTH = 200; // px
const SUPPORTED_FORMATS = { jpg: true, jpeg: true, png: true };
```

- **S3**: Initializes the S3 client to interact with the S3 service.
- **DEST_BUCKET**: The destination S3 bucket where processed images will be stored. This value is loaded from the environment variables.
- **THUMBNAIL_WIDTH**: Default width (200px) for resizing images.
- **SUPPORTED_FORMATS**: An object that defines the supported image file formats (jpg, jpeg, png).

3. Lambda Handler Function:

```
export const handler = async (event, context) => {
  const { eventTime, s3 } = event.Records[0];
  const srcBucket = s3.bucket.name;
  const srcKey = decodeURIComponent(s3.object.key.replace(/\+/g, " "));
  const ext = srcKey.replace(/^.*\./, "").toLowerCase();

  console.log(`${eventTime} - ${srcBucket}/${srcKey}`);
```

- **event**: Contains details about the event that triggered the Lambda function (e.g., an S3 file upload).
- **context**: Provides information about the runtime of the function.
- **srcBucket**: The name of the source S3 bucket where the image was uploaded.
- **srcKey**: The key (filename) of the uploaded image, decoded to handle spaces and special characters.
- **ext**: Extracts the file extension to check if the format is supported.

4. File Type Validation:

```
if (!SUPPORTED_FORMATS[ext]) {
  console.log(`ERROR: Unsupported file type (${ext})`);
  return;
}
```

- This section ensures that the uploaded file is of a supported format (jpg, jpeg, png). If not, the function logs an error and returns early.

5. Processing Action Determination:

```
const action = event.Records[0].eventName || 'resize';  // Default action is 'resize'
```

- This line checks if an action (resize, greyscale, or blur) is provided in the event metadata. If no action is specified, it defaults to resizing the image.

6. Retrieving the Image from S3:

```
try {
  const { Body, ContentType } = await S3.send(new GetObjectCommand({ Bucket: srcBucket,
  const image = await Body.transformToByteArray();
```

- **GetObjectCommand**: Retrieves the image from the source S3 bucket.
- **Body**: The image data in binary format.
- **transformToByteArray()**: Converts the image into a byte array for further processing by the `sharp` library.

7. Image Processing (Resize, Greyscale, Blur):

```
let outputBuffer;
if (action === 'resize') {
  outputBuffer = await sharp(image).resize(THUMBNAIL_WIDTH).toBuffer();
} else if (action === 'greyscale') {
  outputBuffer = await sharp(image).greyscale().toBuffer();
} else if (action === 'blur') {
  outputBuffer = await sharp(image).blur(10).toBuffer();
} else {
  console.log(`ERROR: Unsupported action (${action})`);
  return;
}
```

- Depending on the **action** (resize, greyscale, blur), the `sharp` library applies the appropriate transformation to the image:
    - **resize**: Resizes the image to 200px width.
    - **greyscale**: Converts the image to greyscale.
    - **blur**: Applies a Gaussian blur with strength 10.

8. Uploading Processed Image to S3:

```
await S3.send(new PutObjectCommand({ Bucket: DEST_BUCKET, Key: srcKey, Body: outputBuffer,
```

- **PutObjectCommand**: Uploads the transformed image back into the destination S3 bucket with the same filename (`srcKey`) and content type (`ContentType`).

9. Final Response:

```
const message = `Successfully processed ${srcBucket}/${srcKey} with action ${action} and
console.log(message);
return { statusCode: 200, body: message };
```

- The function logs and returns a success message after the image has been successfully processed and uploaded.

HTML: Front-End Image Upload and Processing Form

Overview:

This HTML file provides the front-end interface for users to upload images, choose processing options (resize, greyscale, blur), and view both the original and processed images..

1. HTML Structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Image Processing Website</title>
  <style>
    /* CSS styles for layout and design */
  </style>
</head>
<body>
  <div class="container">
    <h1>Image Processing</h1>
    <form id="imageForm">
      <label for="image">Select Image:</label>
      <input type="file" id="image" name="image" accept="image/*" required />
```

- **Select Image:** Users can choose an image file for upload.

```
      <label for="action">Select Action (For UI Only, Always Resizes):</label>
      <select id="action" name="action">
        <option value="resize">Resize</option>
        <option value="greyscale">Greyscale</option>
        <option value="blur">Gaussian Blur</option>
      </select>
```

- **Select Action**: Dropdown menu for users to choose the processing action (resize, greyscale, blur). Note that the UI reflects these options, but by default, it may always resize depending on the Lambda logic.

2. JavaScript Logic for Form Submission:

```
<script>
  const form = document.getElementById('imageForm');
  const messageDiv = document.getElementById('message');
  const beforeImage = document.getElementById('beforeImage');
  const afterImage = document.getElementById('afterImage');
  const imagesDiv = document.getElementById('images');

  form.addEventListener('submit', async (event) => {
    event.preventDefault();

    const fileInput = document.getElementById('image');
    const file = fileInput.files[0];

    if (!file) {
      alert('Please select a file!');
      return;
    }

    // Show original image before processing
    const reader = new FileReader();
    reader.onload = function (e) {
      beforeImage.src = e.target.result;
      imagesDiv.style.display = 'flex';
    };
    reader.readAsDataURL(file);

    // Show processing message
    messageDiv.textContent = 'Processing image, please wait...';
```

- **Image Preview**: Displays the original image selected by the user before processing.

- **Processing Message**: Shows a "Processing image" message during image upload and processing.

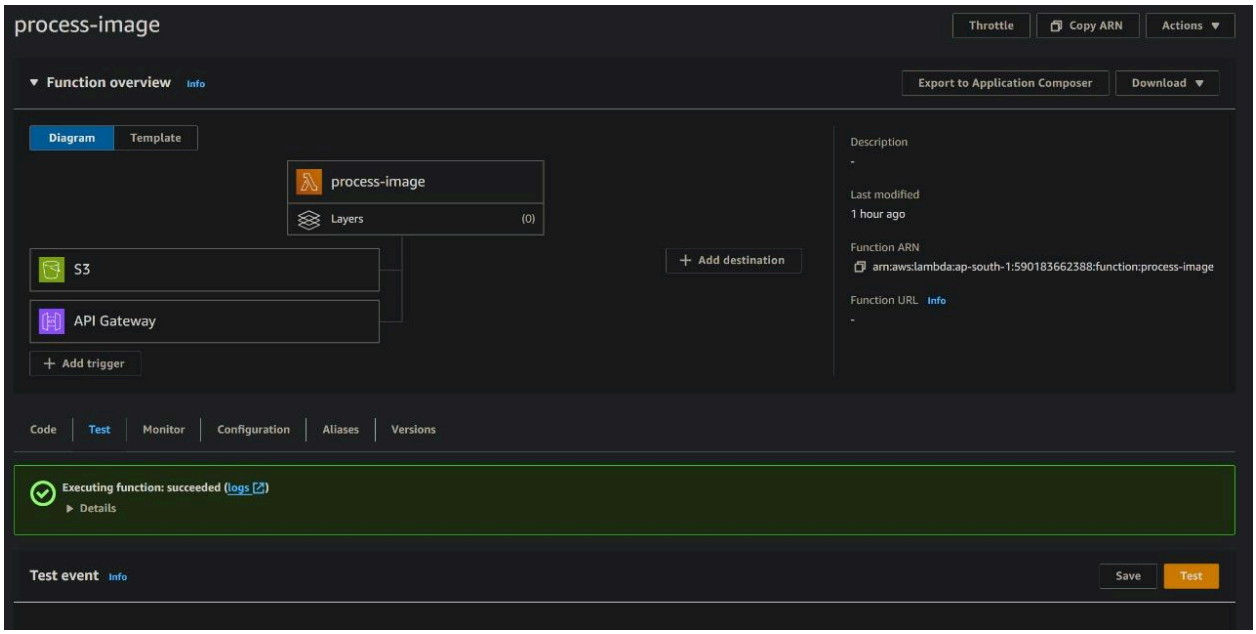
3. API Call to Trigger Lambda Processing:

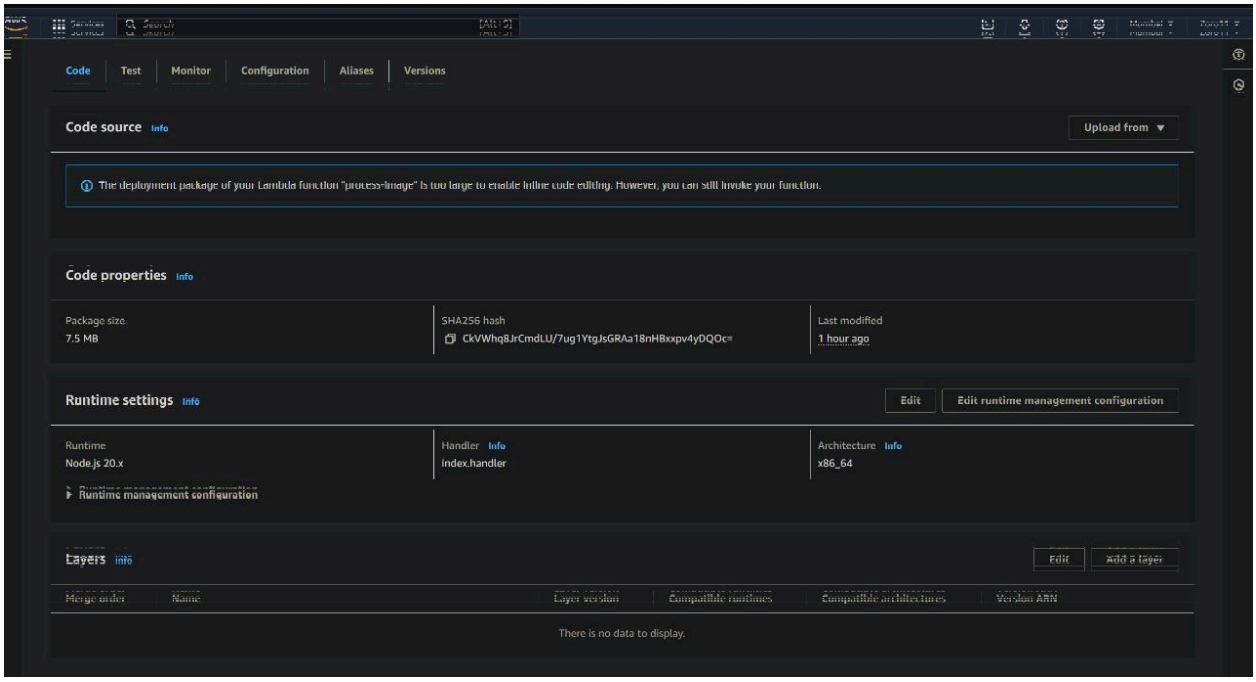
```
    // Upload the image to S3 and trigger Lambda via API Gateway
    const formData = new FormData();
    formData.append('file', file);

    const apiUrl = 'https://your-api-gateway-endpoint.amazonaws.com/dev'; // Update to y

    try {
      const response = await fetch(apiUrl, {
        method: 'POST',
        body: formData
      });

      const data = await response.json();
      const processedImageUrl = `https://dest-a2.s3.amazonaws.com/${data.processedKey}`;

      afterImage.src = processedImageUrl; // Display the processed image
      messageDiv.textContent = 'Image processed successfully!';
    } catch (error) {
      console.error('Error:', error);
      messageDiv.textContent = 'Failed to process image.';
    }
  });
</script>
```

- **Form Submission**: The selected image is uploaded to the API Gateway, which triggers the Lambda function for processing.
- **API Endpoint**: `apiUrl` should be updated with the correct API Gateway URL.
- **Processed Image Display**: After successful processing, the processed image is displayed next to the original.

Conclusion for the cloud setup:

- **JavaScript (mjs)**: Handles the server-side image processing using AWS Lambda, S3, and Sharp. It processes images based on user-selected actions (resize, greyscale, blur).
- **HTML & JavaScript**: Provides a user-friendly front-end to upload images, select processing options, and display both original and processed images.





Non - Cloud Setup

This code is for the non-cloud setup (local run) of your Serverless Image Processor. This explains the HTML structure, JavaScript functions, and how the front-end handles image processing tasks such as resizing, applying grayscale filters, and blurring images.

HTML: Local Image Processing Application

Overview:

This HTML file provides a local environment for users to upload and process images without relying on a cloud backend. Image processing actions such as resizing, grayscale, and blurring are applied directly in the browser using the HTML5 Canvas API and JavaScript.

1. HTML Structure

This section defines the basic structure of the webpage where users can upload images, select processing actions, and download processed images.

Head Section

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Serverless Image Processor</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/cropper/4.1.0/crop
    <style>
        /* CSS styles for layout and design */
    </style>
</head>
```

- **Meta tags**: Define the character encoding and set the viewport for responsive design.
- **Title**: Sets the title of the web page.

- **Styles**: Inline CSS is used to design the layout of the page. The webpage has two sections (upload and processing) and uses simple, responsive design elements to ensure the UI is user-friendly.

Body Section

```
<body>
    <h1>Serverless Image Processor</h1>
    <div class="container">
        <div class="upload-section">
            <h2>Upload Image</h2>
            <input type="file" id="imageUpload" accept="image/*">
            <button id="uploadButton">Upload</button>
            <div id="uploadedImageContainer"></div>
        </div>
        <div class="process-section">
            <h2>Process Image</h2>
            <select id="processAction">
                <option value="">Select action</option>
                <option value="resize">Resize</option>
                <option value="grayscale">Grayscale</option>
                <option value="blur">Blur</option>
            </select>
            <button id="processButton" disabled>Process</button>
            <div id="processedImageContainer"></div>
            <button id="downloadButton">Download Processed Image</button>
        </div>
    </div>
    <div id="errorContainer" class="error"></div>
</body>
```

Image Upload Section:

- `input[type="file"]`: Allows users to select an image file from their local system.
- `#uploadButton`: Button to trigger the image upload process.

- `#uploadedImageContainer`: Displays the uploaded image.

Image Processing Section:

- **Dropdown Menu (`#processAction`)**: Allows users to select an image processing action (resize, grayscale, blur).
- **Process Button (`#processButton`)**: Applies the selected processing action.
- **Processed Image Container (`#processedImageContainer`)**: Displays the processed image.
- **Download Button (`#downloadButton`)**: Allows the user to download the processed image.

Error Container: Displays any error messages (e.g., if no image is uploaded or no action is selected).

2. JavaScript Functionality

The JavaScript section handles image uploading, image processing (resize, grayscale, blur), and downloading the processed image.

Event Listeners for Upload and Processing Actions

```
const imageUpload = document.getElementById('imageUpload');
const uploadButton = document.getElementById('uploadButton');
const processAction = document.getElementById('processAction');
const processButton = document.getElementById('processButton');
const downloadButton = document.getElementById('downloadButton');
const uploadedImageContainer = document.getElementById('uploadedImageContainer');
const processedImageContainer = document.getElementById('processedImageContainer');
const errorContainer = document.getElementById('errorContainer');

let uploadedImage = null;
let processedImageData = null;
```

- **Global Variables**:
    - **uploadedImage**: Stores the original image uploaded by the user.

    - **processedImageData**: Stores the processed image data (used for downloading the image).

Upload Button Click Event

- **Image Upload**:
    - When the **upload button** is clicked, the image is read using the **FileReader API** and displayed on the screen.
    - Once the image is loaded, the **process button** is enabled, allowing users to apply image transformations.

Process Button Click Event

- **Processing Image**:
    - This function triggers image processing based on the selected action (resize, grayscale, or blur).
    - If no action or image is selected, an error message is displayed.

Download Button Click Event

- **Download Processed Image**:
    - The processed image can be downloaded by creating an anchor (`<a>`) element, setting the `href` attribute to the processed image's data URL, and triggering a click event to download the image.

Image Processing Function

- **processImage Function**: Handles different image processing actions (resize, grayscale, blur). It draws the uploaded image on a canvas and modifies the image based on the selected action.

Resize Image Function

- **resizeImage Function**: Resizes the image to 50% of its original size for demonstration purposes.

Grayscale Filter Function

- **applyGrayscale Function**: Converts the image to grayscale by averaging the RGB values for each pixel.

Blur Filter Function

- **applyBlur Function**: Applies a 5px Gaussian blur filter to the image.

Error Handling

- **showError Function**: Displays an error message and clears it after 3 seconds.

Conclusion for the non-cloud setup:

This outlines how the **non-cloud local setup** works for image processing, covering both the HTML structure and the JavaScript logic. The local application processes images using the **Canvas API** and provides functionality to resize, grayscale, and blur images directly in the browser without needing a cloud service.