```java
package com.rtnimageresizer;

import android.content.Context;
import android.content.ContentResolver;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Matrix;
import androidx.exifinterface.media.ExifInterface;
import android.net.Uri;
import android.os.Build;
import android.provider.MediaStore;
import android.util.Base64;
import android.util.Log;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Date;

/**
 * Provide methods to resize and rotate an image file.
 */
public class ImageResizer {
  private final static String IMAGE_JPEG = "image/jpeg";
  private final static String IMAGE_PNG = "image/png";
  private final static String SCHEME_DATA = "data";
  private final static String SCHEME_CONTENT = "content";
  private final static String SCHEME_FILE = "file";
  private final static String SCHEME_HTTP = "http";
  private final static String SCHEME_HTTPS = "https";


  // List of known EXIF tags we will be copying.
  // Orientation, width, height, and some others are ignored
  // TODO: Find any missing tag that might be useful
  private final static String[] EXIF_TO_COPY_ROTATED = new String[]
    {
      ExifInterface.TAG_APERTURE_VALUE,
      ExifInterface.TAG_MAX_APERTURE_VALUE,
ExifInterface.TAG_METERING_MODE,
ExifInterface.TAG_ARTIST,
ExifInterface.TAG_BITS_PER_SAMPLE,
ExifInterface.TAG_COMPRESSION,
ExifInterface.TAG_BODY_SERIAL_NUMBER,
ExifInterface.TAG_BRIGHTNESS_VALUE,
ExifInterface.TAG_CONTRAST,
ExifInterface.TAG_CAMERA_OWNER_NAME,
ExifInterface.TAG_COLOR_SPACE,
ExifInterface.TAG_COPYRIGHT,
ExifInterface.TAG_DATETIME,
ExifInterface.TAG_DATETIME_DIGITIZED,
ExifInterface.TAG_DATETIME_ORIGINAL,
ExifInterface.TAG_DEVICE_SETTING_DESCRIPTION,
ExifInterface.TAG_DIGITAL_ZOOM_RATIO,
ExifInterface.TAG_EXIF_VERSION,
ExifInterface.TAG_EXPOSURE_BIAS_VALUE,
ExifInterface.TAG_EXPOSURE_INDEX,
ExifInterface.TAG_EXPOSURE_MODE,
ExifInterface.TAG_EXPOSURE_TIME,
ExifInterface.TAG_EXPOSURE_PROGRAM,
ExifInterface.TAG_FLASH,
ExifInterface.TAG_FLASH_ENERGY,
ExifInterface.TAG_FOCAL_LENGTH,
ExifInterface.TAG_FOCAL_LENGTH_IN_35MM_FILM,
ExifInterface.TAG_FOCAL_PLANE_RESOLUTION_UNIT,
ExifInterface.TAG_FOCAL_PLANE_X_RESOLUTION,
ExifInterface.TAG_FOCAL_PLANE_Y_RESOLUTION,
ExifInterface.TAG_PHOTOMETRIC_INTERPRETATION,
ExifInterface.TAG_PLANAR_CONFIGURATION,
ExifInterface.TAG_F_NUMBER,
ExifInterface.TAG_GAIN_CONTROL,
ExifInterface.TAG_GAMMA,
ExifInterface.TAG_GPS_ALTITUDE,
ExifInterface.TAG_GPS_ALTITUDE_REF,
ExifInterface.TAG_GPS_AREA_INFORMATION,
ExifInterface.TAG_GPS_DATESTAMP,
ExifInterface.TAG_GPS_DOP,
ExifInterface.TAG_GPS_LATITUDE,
ExifInterface.TAG_GPS_LATITUDE_REF,
ExifInterface.TAG_GPS_LONGITUDE,
ExifInterface.TAG_GPS_LONGITUDE_REF,
ExifInterface.TAG_GPS_STATUS,
ExifInterface.TAG_GPS_DEST_BEARING,
ExifInterface.TAG_GPS_DEST_BEARING_REF,
ExifInterface.TAG_GPS_DEST_DISTANCE,
ExifInterface.TAG_GPS_DEST_DISTANCE_REF,
ExifInterface.TAG_GPS_DEST_LATITUDE,
ExifInterface.TAG_GPS_DEST_LATITUDE_REF,
ExifInterface.TAG_GPS_DEST_LONGITUDE,
ExifInterface.TAG_GPS_DEST_LONGITUDE_REF,
ExifInterface.TAG_GPS_DIFFERENTIAL,
ExifInterface.TAG_GPS_IMG_DIRECTION,
ExifInterface.TAG_GPS_IMG_DIRECTION_REF,
ExifInterface.TAG_GPS_MAP_DATUM,
ExifInterface.TAG_GPS_MEASURE_MODE,
ExifInterface.TAG_GPS_PROCESSING_METHOD,
ExifInterface.TAG_GPS_SATELLITES,
ExifInterface.TAG_GPS_SPEED,
ExifInterface.TAG_GPS_SPEED_REF,
ExifInterface.TAG_GPS_STATUS,
ExifInterface.TAG_GPS_TIMESTAMP,
ExifInterface.TAG_GPS_TRACK,
ExifInterface.TAG_GPS_TRACK_REF,
ExifInterface.TAG_GPS_VERSION_ID,
ExifInterface.TAG_IMAGE_DESCRIPTION,
ExifInterface.TAG_IMAGE_UNIQUE_ID,
ExifInterface.TAG_ISO_SPEED,
ExifInterface.TAG_PHOTOGRAPHIC_SENSITIVITY,
ExifInterface.TAG_JPEG_INTERCHANGE_FORMAT,
ExifInterface.TAG_JPEG_INTERCHANGE_FORMAT_LENGTH,
ExifInterface.TAG_LENS_MAKE,
ExifInterface.TAG_LENS_MODEL,
ExifInterface.TAG_LENS_SERIAL_NUMBER,
ExifInterface.TAG_LENS_SPECIFICATION,
ExifInterface.TAG_LIGHT_SOURCE,
ExifInterface.TAG_MAKE,
ExifInterface.TAG_MAKER_NOTE,
ExifInterface.TAG_MODEL,
// ExifInterface.TAG_ORIENTATION, // removed
ExifInterface.TAG_SATURATION,
ExifInterface.TAG_SHARPNESS,
ExifInterface.TAG_SHUTTER_SPEED_VALUE,
ExifInterface.TAG_SOFTWARE,
ExifInterface.TAG_SUBJECT_DISTANCE,
ExifInterface.TAG_SUBJECT_DISTANCE_RANGE,
ExifInterface.TAG_SUBJECT_LOCATION,
ExifInterface.TAG_USER_COMMENT,
        ExifInterface.TAG_WHITE_BALANCE
      };



  /**
   * Resize the specified bitmap.
   */
  private static Bitmap resizeImage(Bitmap image, int newWidth, int
newHeight,
                                      String mode, boolean
onlyScaleDown) {
      Bitmap newImage = null;
      if (image == null) {
        return null; // Can't load the image from the given path.
      }

      int width = image.getWidth();
      int height = image.getHeight();

      if (newHeight > 0 && newWidth > 0) {
        int finalWidth;
        int finalHeight;

        if (mode.equals("stretch")) {
          // Distort aspect ratio
          finalWidth = newWidth;
          finalHeight = newHeight;

          if (onlyScaleDown) {
            finalWidth = Math.min(width, finalWidth);
            finalHeight = Math.min(height, finalHeight);
          }
        } else {
          // "contain" (default) or "cover": keep its aspect ratio
          float widthRatio = (float) newWidth / width;
          float heightRatio = (float) newHeight / height;

          float ratio = mode.equals("cover") ?
            Math.max(widthRatio, heightRatio) :
            Math.min(widthRatio, heightRatio);

          if (onlyScaleDown) ratio = Math.min(ratio, 1);
```

```
            finalWidth = (int) Math.round(width * ratio);
            finalHeight = (int) Math.round(height * ratio);
        }

        try {
            newImage = Bitmap.createScaledBitmap(image, finalWidth,
finalHeight, true);
        } catch (OutOfMemoryError e) {
            return null;
        }
    }

    return newImage;
  }

  /**
   * Rotate the specified bitmap with the given angle, in degrees.
   */
  public static Bitmap rotateImage(Bitmap source, float angle)
  {
    Bitmap retVal;

    Matrix matrix = new Matrix();
    matrix.postRotate(angle);
    try {
        retVal = Bitmap.createBitmap(source, 0, 0, source.getWidth(),
source.getHeight(), matrix, true);
    } catch (OutOfMemoryError e) {
        return null;
    }
    return retVal;
  }

  /**
   * Save the given bitmap in a directory. Extension is automatically
generated using the bitmap format.
   */
  public static File saveImage(Bitmap bitmap, File saveDirectory, String
fileName,
                                          Bitmap.CompressFormat
compressFormat, int quality)
    throws IOException {
    if (bitmap == null) {
        throw new IOException("The bitmap couldn't be resized");
    }

    File newFile = new File(saveDirectory, fileName + "." +
compressFormat.name());
    if(!newFile.createNewFile()) {
        throw new IOException("The file already exists");
    }

    ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
    bitmap.compress(compressFormat, quality, outputStream);
    byte[] bitmapData = outputStream.toByteArray();

    outputStream.flush();
    outputStream.close();

    FileOutputStream fos = new FileOutputStream(newFile);
    fos.write(bitmapData);
    fos.flush();
    fos.close();

    return newFile;
  }

  /**
   * Get {@link File} object for the given Android URI.<br>
   * Use content resolver to get real path if direct path doesn't return
valid file.
   */
  private static File getFileFromUri(Context context, Uri uri) {

    // first try by direct path
    File file = new File(uri.getPath());
    if (file.exists()) {
        return file;
    }

    // try reading real path from content resolver (gallery images)
    Cursor cursor = null;
    try {
        String[] proj = {MediaStore.Images.Media.DATA};
        cursor = context.getContentResolver().query(uri, proj, null, null,
null);
        int column_index =
cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
        cursor.moveToFirst();
        String realPath = cursor.getString(column_index);
        file = new File(realPath);
      } catch (Exception ignored) {
      } finally {
        if (cursor != null) {
          cursor.close();
        }
      }

      return file;
    }

    /**
     * Attempts to copy exif info from one file to another. Note: orientation,
width, and height
     exif attributes are not copied since those are lost after image rotation.

     * imageUri: original image URI as provided from JS
     * dstPath: final image output path
     * Returns true if copy was successful, false otherwise.
     */
    public static boolean copyExif(Context context, Uri imageUri, String
dstPath){
      ExifInterface src = null;
      ExifInterface dst = null;

      try {

        File file = getFileFromUri(context, imageUri);
        if (!file.exists()) {
          return false;
        }

        src = new ExifInterface(file.getAbsolutePath());
        dst = new ExifInterface(dstPath);

      } catch (Exception ignored) {
        Log.e("ImageResizer::copyExif", "EXIF read failed", ignored);
      }

      if(src == null || dst == null){
        return false;
        }

        try{

            for (String attr : EXIF_TO_COPY_ROTATED)
            {
                String value = src.getAttribute(attr);
                if (value != null){
                    dst.setAttribute(attr, value);
                }
            }
            dst.saveAttributes();

        } catch (Exception ignored) {
            Log.e("ImageResizer::copyExif", "EXIF copy failed", ignored);
            return false;
        }

        return true;
    }

    /**
     * Get orientation by reading Image metadata
     */
    public static int getOrientation(Context context, Uri uri) {
        try {
            // ExifInterface(InputStream) only exists since Android N (r24)
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
                InputStream input =
context.getContentResolver().openInputStream(uri);
                ExifInterface ei = new ExifInterface(input);
                return getOrientation(ei);
            }
            File file = getFileFromUri(context, uri);
            if (file.exists()) {
                ExifInterface ei = new ExifInterface(file.getAbsolutePath());
                return getOrientation(ei);
            }
        } catch (Exception ignored) { }

        return 0;
    }

    /**
```

```
     * Convert metadata to degrees
     */
    public static int getOrientation(ExifInterface exif) {
        int orientation =
exif.getAttributeInt(ExifInterface.TAG_ORIENTATION,
ExifInterface.ORIENTATION_NORMAL);
        switch (orientation) {
            case ExifInterface.ORIENTATION_ROTATE_90:
                return 90;
            case ExifInterface.ORIENTATION_ROTATE_180:
                return 180;
            case ExifInterface.ORIENTATION_ROTATE_270:
                return 270;
            default:
                return 0;
        }
    }

    /**
     * Compute the inSampleSize value to use to load a bitmap.
     * Adapted from
https://developer.android.com/training/displaying-bitmaps/load-bitmap.ht
ml
     */
    private static int calculateInSampleSize(BitmapFactory.Options
options, int reqWidth, int reqHeight) {
        final int height = options.outHeight;
        final int width = options.outWidth;

        int inSampleSize = 1;

        if (height > reqHeight || width > reqWidth) {
            final int halfHeight = height / 2;
            final int halfWidth = width / 2;

            // Calculate the largest inSampleSize value that is a power of 2
and keeps both
            // height and width larger than the requested height and width.
            while ((halfHeight / inSampleSize) >= reqHeight && (halfWidth /
inSampleSize) >= reqWidth) {
                inSampleSize *= 2;
            }
        }
```

```
      return inSampleSize;
   }

   /**
    * Load a bitmap either from a real file or using the {@link
ContentResolver} of the current
    * {@link Context} (to read gallery images for example).
    *
    * Note that, when options.inJustDecodeBounds = true, we actually
expect sourceImage to remain
    * as null (see
https://developer.android.com/training/displaying-bitmaps/load-bitmap.ht
ml), so
    * getting null sourceImage at the completion of this method is not
always worthy of an error.
    */
   private static Bitmap loadBitmap(Context context, Uri imageUri,
BitmapFactory.Options options) throws IOException {
      Bitmap sourceImage = null;
      String imageUriScheme = imageUri.getScheme();
      if (imageUriScheme == null ||
!imageUriScheme.equalsIgnoreCase(SCHEME_CONTENT)) {
         try {
            sourceImage = BitmapFactory.decodeFile(imageUri.getPath(),
options);
         } catch (Exception e) {
            e.printStackTrace();
            throw new IOException("Error decoding image file");
         }
      } else {
         ContentResolver cr = context.getContentResolver();
         InputStream input = cr.openInputStream(imageUri);
         if (input != null) {
            sourceImage = BitmapFactory.decodeStream(input, null,
options);
            input.close();
         }
      }
      return sourceImage;
   }

   /**
    * Loads the bitmap resource from the file specified in imagePath.
    */
```

```
  private static Bitmap loadBitmapFromFile(Context context, Uri
imageUri, int newWidth,
                                           int newHeight) throws
IOException  {
     // Decode the image bounds to find the size of the source image.
     BitmapFactory.Options options = new BitmapFactory.Options();
     options.inJustDecodeBounds = true;
     loadBitmap(context, imageUri, options);

     // Set a sample size according to the image size to lower memory
usage.
     options.inSampleSize = calculateInSampleSize(options, newWidth,
newHeight);
     options.inJustDecodeBounds = false;
     //System.out.println(options.inSampleSize);
     return loadBitmap(context, imageUri, options);

  }

  /**
   * Loads the bitmap resource from an URL
   */
  private static Bitmap loadBitmapFromURL(Uri imageUri, int newWidth,
                                          int newHeight) throws
IOException  {

     InputStream input = null;
     Bitmap sourceImage = null;

     try{
        URL url = new URL(imageUri.toString());
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.connect();
        input = connection.getInputStream();

        if (input != null) {

           // need to load into memory since inputstream is not seekable
           // we still won't load the whole bitmap into memory
           // Also need this ugly code since we are on Java8...
           ByteArrayOutputStream buffer = new ByteArrayOutputStream();
           int nRead;
           byte[] data = new byte[1024];
```

```
            byte[] imageData = null;

            try{
               while ((nRead = input.read(data, 0, data.length)) != -1) {
                  buffer.write(data, 0, nRead);
               }
               buffer.flush();
               imageData = buffer.toByteArray();
            }
            finally{
               buffer.close();
            }


            // Decode the image bounds to find the size of the source image.
            // Do it here so we only do one request
            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inJustDecodeBounds = true;
            BitmapFactory.decodeByteArray(imageData, 0,
imageData.length, options);

            // Set a sample size according to the image size to lower
memory usage.
            options.inSampleSize = calculateInSampleSize(options,
newWidth, newHeight);
            options.inJustDecodeBounds = false;

            sourceImage = BitmapFactory.decodeByteArray(imageData, 0,
imageData.length, options);
         }
      }
      catch (Exception e) {
         e.printStackTrace();
         throw new IOException("Error fetching remote image file.");
      }
      finally{
         try {
            if(input != null){
               input.close();
            }
         }
         catch (IOException e) {
            e.printStackTrace();
         }
      }

      return sourceImage;

   }

   /**
    * Loads the bitmap resource from a base64 encoded jpg or png.
    * Format is as such:
    * png: 'data:image/png;base64,iVBORw0KGgoAA...'
    * jpg: 'data:image/jpeg;base64,/9j/4AAQSkZJ...'
    */
   private static Bitmap loadBitmapFromBase64(Uri imageUri) {
      Bitmap sourceImage = null;
      String imagePath = imageUri.getSchemeSpecificPart();
      int commaLocation = imagePath.indexOf(',');
      if (commaLocation != -1) {
         final String mimeType = imagePath.substring(0,
commaLocation).replace('\\','/').toLowerCase();
         final boolean isJpeg = mimeType.startsWith(IMAGE_JPEG);
         final boolean isPng = !isJpeg &&
mimeType.startsWith(IMAGE_PNG);

         if (isJpeg || isPng) {
            // base64 image. Convert to a bitmap.
            final String encodedImage =
imagePath.substring(commaLocation + 1);
            final byte[] decodedString = Base64.decode(encodedImage,
Base64.DEFAULT);
            sourceImage = BitmapFactory.decodeByteArray(decodedString,
0, decodedString.length);
         }
      }

      return sourceImage;
   }

   /**
    * Create a resized version of the given image and returns a Bitmap
object
    * ready to be saved or converted. Ensure that the result is cleaned up
after use
    * by using recycle
    */
   public static Bitmap createResizedImage(Context context, Uri
imageUri, int newWidth,
                                           int newHeight, int quality, int
rotation,
                                           String mode, boolean
onlyScaleDown) throws IOException   {
      Bitmap sourceImage = null;
      String imageUriScheme = imageUri.getScheme();

      if (imageUriScheme == null ||
         imageUriScheme.equalsIgnoreCase(SCHEME_FILE) ||
         imageUriScheme.equalsIgnoreCase(SCHEME_CONTENT)
      ) {
         sourceImage = ImageResizer.loadBitmapFromFile(context,
imageUri, newWidth, newHeight);
      } else if (imageUriScheme.equalsIgnoreCase(SCHEME_HTTP) ||
imageUriScheme.equalsIgnoreCase(SCHEME_HTTPS)){
         sourceImage = ImageResizer.loadBitmapFromURL(imageUri,
newWidth, newHeight);
      } else if (imageUriScheme.equalsIgnoreCase(SCHEME_DATA)) {
         sourceImage = ImageResizer.loadBitmapFromBase64(imageUri);
      }

      if (sourceImage == null) {
         throw new IOException("Unable to load source image from path");
      }


      // Rotate if necessary. Rotate first because we will otherwise
      // get wrong dimensions if we want the new dimensions to be after
rotation.
      // NOTE: This will "fix" the image using it's exif info if it is rotated as
well.
      Bitmap rotatedImage = sourceImage;
      int orientation = getOrientation(context, imageUri);
      rotation = orientation + rotation;
      rotatedImage = ImageResizer.rotateImage(sourceImage, rotation);

      if(rotatedImage == null){
         throw new IOException("Unable to rotate image. Most likely due to
not enough memory.");
      }
      if (rotatedImage != rotatedImage) {
         sourceImage.recycle();
      }

      // Scale image
      Bitmap scaledImage = ImageResizer.resizeImage(rotatedImage,
newWidth, newHeight, mode, onlyScaleDown);

      if(scaledImage == null){
         throw new IOException("Unable to resize image. Most likely due to
not enough memory.");
      }

      if (scaledImage != rotatedImage) {
         rotatedImage.recycle();
      }

      return scaledImage;
   }
}
```