Image Processor API

Justin Thoreson

Contents

I. Architecture Overview

The image processor API allows for transformations to be made to an image via a RESTful API utilizing HTTP requests and responses. Thus, the API is designed off the **client-server** architectural style. The API itself functions in the back-end and connects to front-end clients. The image processor API is designed such that the server code is as decoupled as possible from the image transformation code. Hence, the image processor almost takes the appearance of a library, but given the server component, which acts as the driver for incoming image transformation requests, the API is more than just a library; it is designed to be cloud-hosted.

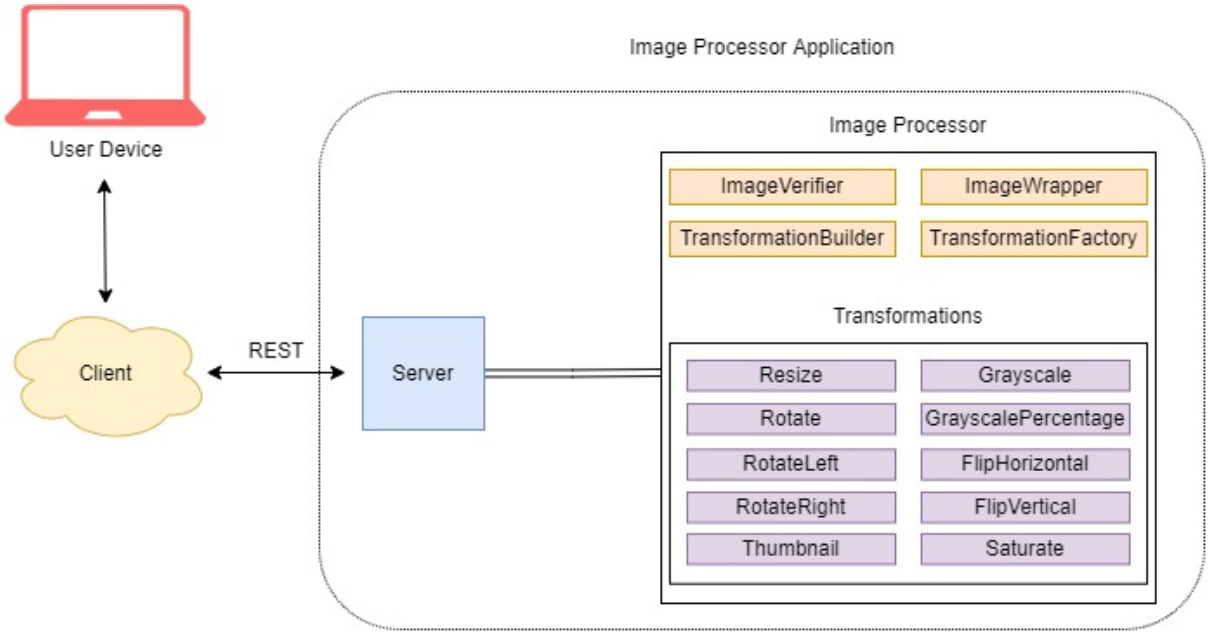


Figure 1: High-level physical view: The image processor API interacting with a client.

Note: Though a user interface was not the focus of this project, a front-end component was developed purely out of interest, the desire to build something more, and the venture for further experience. Furthermore, the front-end client is meant as a means of conducting a visual demonstration. Plus, it was fun to make.

II. Transformations

Ten Transformations

The image processor API allows for ten central transformations: left rotation, right rotation, rotation of ***n*** degrees, flipping horizontally, flipping vertically, grayscale conversion, grayscale conversion with a provided percentage, resizing, saturation/desaturation, and thumbnail generation.

Command (233)

Each transformation acts as a command. Hence, the **Command (233)** design pattern is applied to the transformations. Each transformation command must perform a single transformation on the image. Therefore, each command must implement a transform() method. The underlying methodology of the transformation is implemented differently depending on the command. Each transformation must take in the underlying image object in order to transform the image and return the result. Resize, rotate, grayscale with percentage, and saturate require additional arguments in order to transform the image. Hence, the constructors of these respective commands must take in this additional data. The other commands essentially implement no-op constructors (**Figure 2**). The use of Command (233) is further explained in the "Processing Transformations" Section.

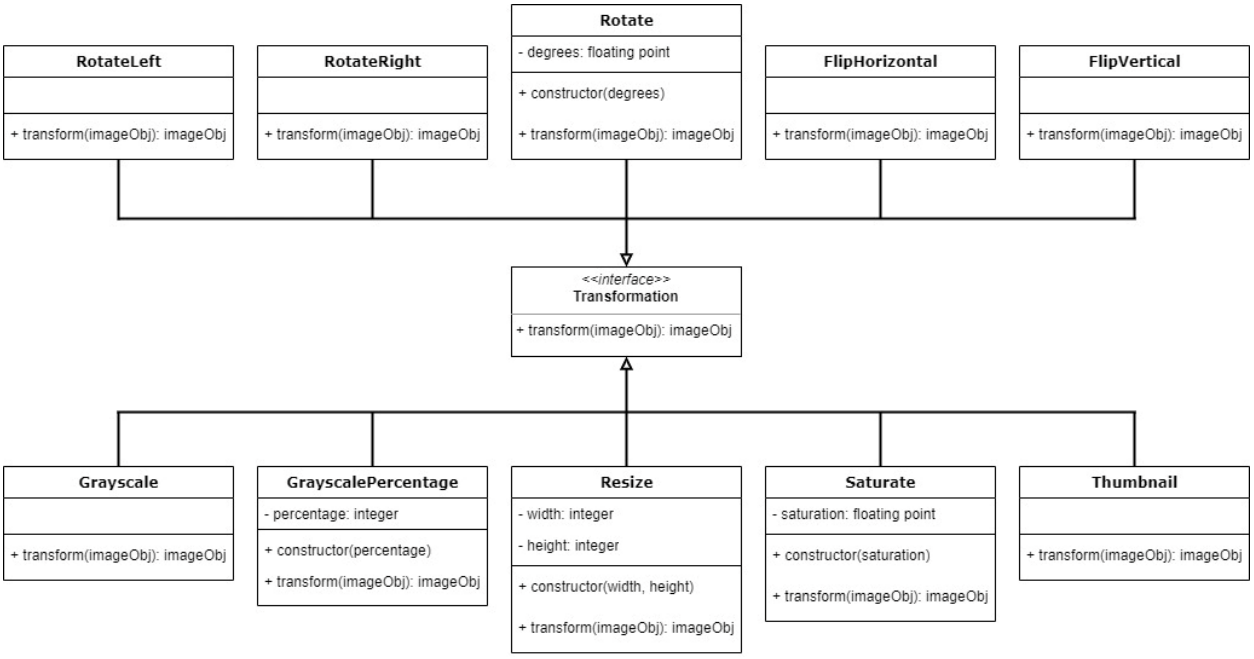


Figure 2: The Transformation interface utilizing the Command (233) design pattern.

Rotation

Rotation requires three separate transformations because rotating left/right and rotating ***n*** degrees have different meanings. Rotating left or right by 90 degrees is a transposition. For example, the dimensions of an image may be 800w x 1200h, which can be considered portrait orientation. When a 90-degree rotation is performed on the image, the dimensions become 1200w x 800h, which is landscape orientation. However, rotating by ***n*** degrees does not alter the dimensions of the image, since an image can only be represented and displayed as a rectangle such that the image's edges are aligned in parallel with the respective edges of the screen. Therefore, it follows that rotating an image by ***n*** degrees keeps the image's dimensions, but instead skews the pixels without altering the orientation (**Figure 3**).

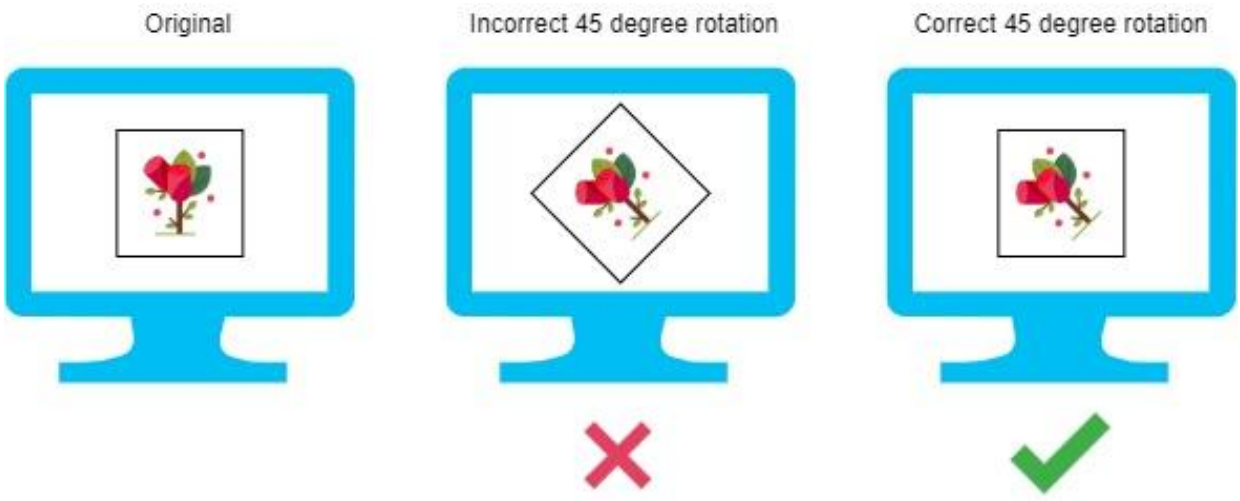


Figure 3: Rotating an image 45 degrees rotates the pixels, but not the image orientation.

Resize vs. Thumbnail

The resize and thumbnail commands are similar, but they are defined differently. Resizing allows for the client to input any positive dimension for width and height. Thus, the dimensions of the image will be converted to the specified width and height, and the resolution will be scaled to match the new dimensions. Thumbnail generation is defined by a maximum width and height of 100 pixels. An image of equal width and height dimensions will be converted to a 100 x 100 image, thus reducing the resolution. If the image's dimensions are not equal, the maximum of the two dimensions will be converted to a size of 100 pixels. The other dimension will be converted to a smaller size such that the original aspect ratio of the image is maintained. For example, a 700 x 1400 image will be converted to 50 x 100 when a thumbnail is generated.

III. Processing Transformations

There is a sequence of steps that come with transforming an image. These steps are described below. They are also illustrated in **Figure 4**.

1) The client performing a transformation request is the first step. The ImageProcessor resource will receive the POST request through the server. The request must come in the form of a JSON object. The JSON object must contain two keys: “image” and “transformations”. The value of the “image” key is an encoded base 64 string representation of the image itself, and the value of the “transformations” key is an array of desired transformations to be performed on the image. The transformations in the array will be performed in order of first to last specified.

2) Second, when a post request is received, the payload is validated to ensure that an image was provided. If an image was not provided, the API responds with an error message with status code 400 BAD REQUEST. However, if the payload contains data within the image key-value pair, the data is checked for the proper MIME type. Since a base 64 string representation of the image is passed via JSON, the program must check if the string is a supported image. If the image is not valid, then the server returns status code 415 UNSUPPORTED MEDIA TYPE. Otherwise, if the image is supported by the image processor, the process continues.

3) Third, the incoming image is converted to an image file object, and the image file is stored in an Image wrapper object. This Image wrapper allows the image file to be transformed. When the Image object is created, the requested transformations can be performed on the image via the transformImage() mutator native to the Image wrapper. This method will take in the list of transformations provided by the client.

4) Once the transformImage() mutator is invoked, the fourth step is to convert the list of transformation requests into a list of Transformation commands. This is done by implementing a TransformationBuilder. Hence, the **Builder (97)** design pattern is applied. The TransformationBuilder loops through each transformation request in the list. **Factory Method (107)** is applied, as each request is sent to a TransformationFactory, where the requests are converted to their respective commands. If a transformation request is not recognized as valid, it is disregarded, and nothing is returned from the factory; thus, it follows that no transformation is performed. Each recognized command is added to the list of transformations in the builder.

5) After the TransformationBuilder has built a list of Transformation objects, the fifth step is for the transformations to be applied to the image. Therefore, each command in the list is executed via a loop. Since the transformations are applied in the same order they were specified via the client’s request, the order and number of transformations differ. Hence, the transformImage() mutator acts as a sort of **Template Method (325)**. Once the transformations are complete, the transformImage() method finished execution.

6) Finally, an accessor is used to extract the encoded base 64 string representation of the transformed image from the Image wrapper, which is then sent back to the client via a JSON response. The JSON response consists of one key-value pair, where the key is “image”, and the value is the base 64 string representation of the image.

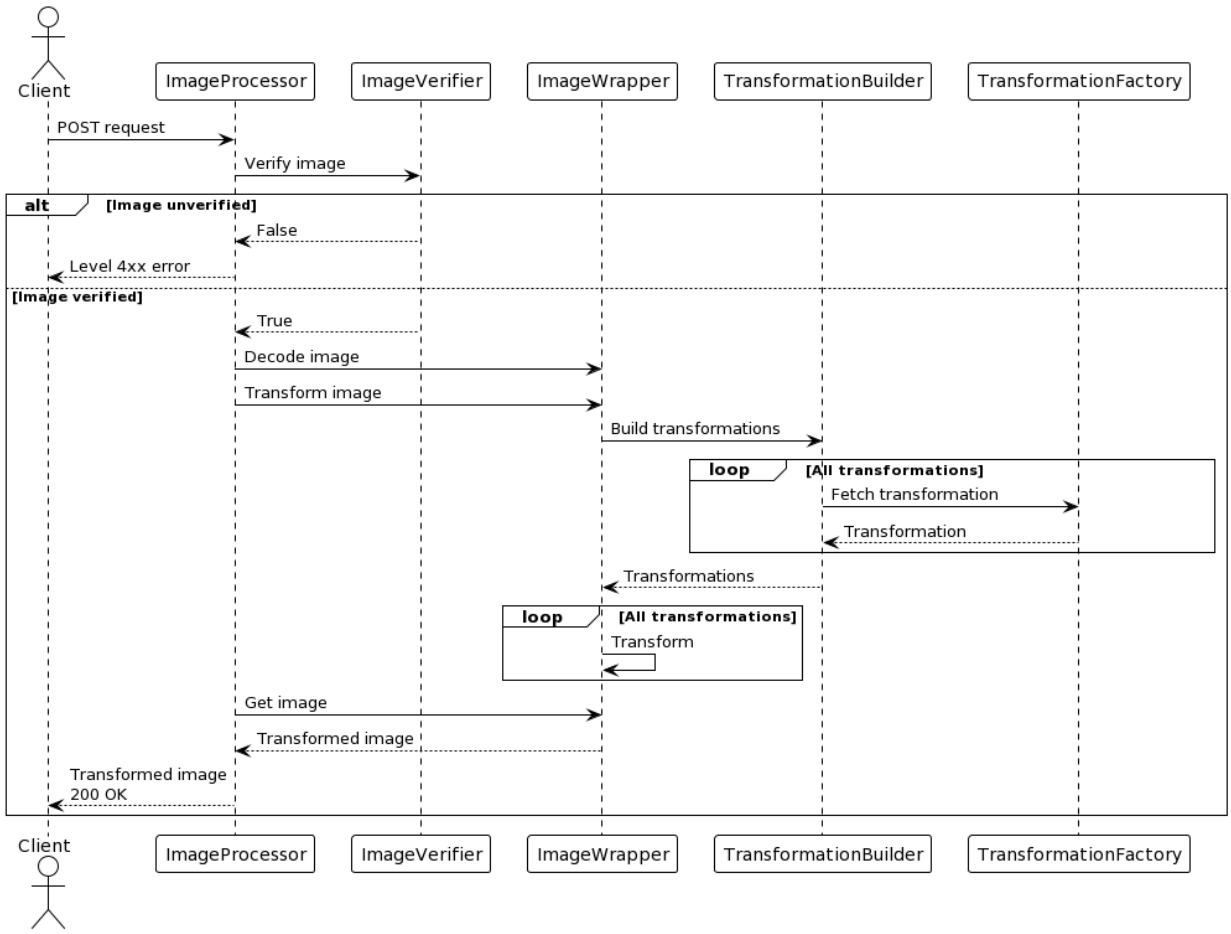


Figure 4: Sequence diagram illustrating the image transformation process.

IV. API Specification

API Endpoint

A client application, such as the one that I have built for fun, will be able to access the API via the following endpoint:

```
http://host-name-goes-here:2022/image-processor/
```

The API is hosted via a host name, which is currently defined at “localhost” when the service runs locally. The port is defined at port 2022, and the resource for processing the image is located at the path “/image-processor/”. Provided below are the crucial lines of server code for hosting the API (for the Python implementation), as well as an example of how a client may access the API through the aforementioned endpoint.

```
api.add_resource(ImageProcessor, '/image-processor/')
app.run(port=2022)
```

Figure 5: The image processor resource as hosted by the Python server.

```
/*** API endpoint ***/

const api = axios.create({
  baseURL: 'http://localhost:2022/image-processor/'
})
```

Figure 6: An example of a client accessing the API via the specified endpoint.

```
api.post('/', req).then(res => {
```

Figure 7: An example of a client issuing a POST request to the image processor.

JSON Structure

Any POST request to the image processor must take the form of JSON. The JSON must contain two key-value pairs: “image” and “transformations”. The “image” key’s value must be an encoded base 64 string representation of an image, whereas the “transformation” key’s value must contain a list of transformation string requests. The transformations are performed on the image from top to bottom, first to last as specified in the transformation list. The list of transformations can contain duplicate transformations if desired, and all transformation types need not be present in the list. An empty list of transformations results in no transformation performed on the image.

To denote which transformations to perform on the provided image, transformations are represented by strings. Each transformation string is the name of the transformation command. The list of transformations will contain these strings verbatim if the specified transformations require no arguments. Otherwise, if arguments are required the transformations strings must be suffixed with open and closed parentheses such that the arguments are contained inside. The specific transformations that require arguments are rotating by ***n*** degrees, resizing, saturating, and grayscaling by a given percentage.

```
{
  "image": "image",
  "transformations": [
    "thumbnail",
    "flip-horizontal",
    "flip-vertical",
    "rotate-left",
    "rotate-right",
    "grayscale",
    "grayscale-percentage(%)",
    "rotate(degrees)",
    "resize(width,height)",
    "saturate(value)"
  ]
}
```

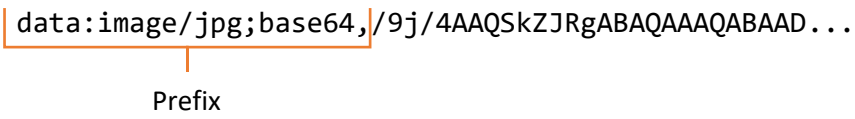
Figure 8: The POST request JSON payload structure.

Encoded Image

The provided image must take the form of a base 64 encoded string. An example of an encoded image may appear as the following:

```
/9j/4AAQSkZJRgABAQAAAQABAAD...
```

Frequently (but not always), an image encoded to base 64 will appear with a detailed media type prefix, such as the following:



The image processor API is not capable of receiving, processing, or sending images with the detailed media type prefix. Therefore, the client is responsible for truncating the prefix. The image processor API will still be able to recognize the media type extension given the first several characters of the encoded image string. If the client wishes for the detailed media type prefix to be reattached to the transformed image, they will be responsible for doing so.

V. Client / User Interface

The central purpose of this project was to design and implement an image processor API that purely operated in the back-end as a server such that clients can interface with it. These clients may already exist in the form of API platforms, such as Postman. However, one may also build their own client to call the image processor. As a bonus to this project, a user interface was created.

Users will upload an image to the image processor client. The image will appear in the "image preview" section of the interface. On the left-hand side, the user will be able to enable the transformations that they wish to process on the image. When the user has selected the transformations to perform and has provided all respective transformation values, the user will invoke the API through a POST request. The image will be sent to the server, transformed, and sent back, where the newly transformed image will be displayed.

Though the image processor API allows for transformations to be performed in any order any number of times, the client that was developed will perform each selected transformation once in the relative order they appear in the left-hand side bar from top to bottom, left to right. To perform transformations more than once, multiple POST requests need to be made.

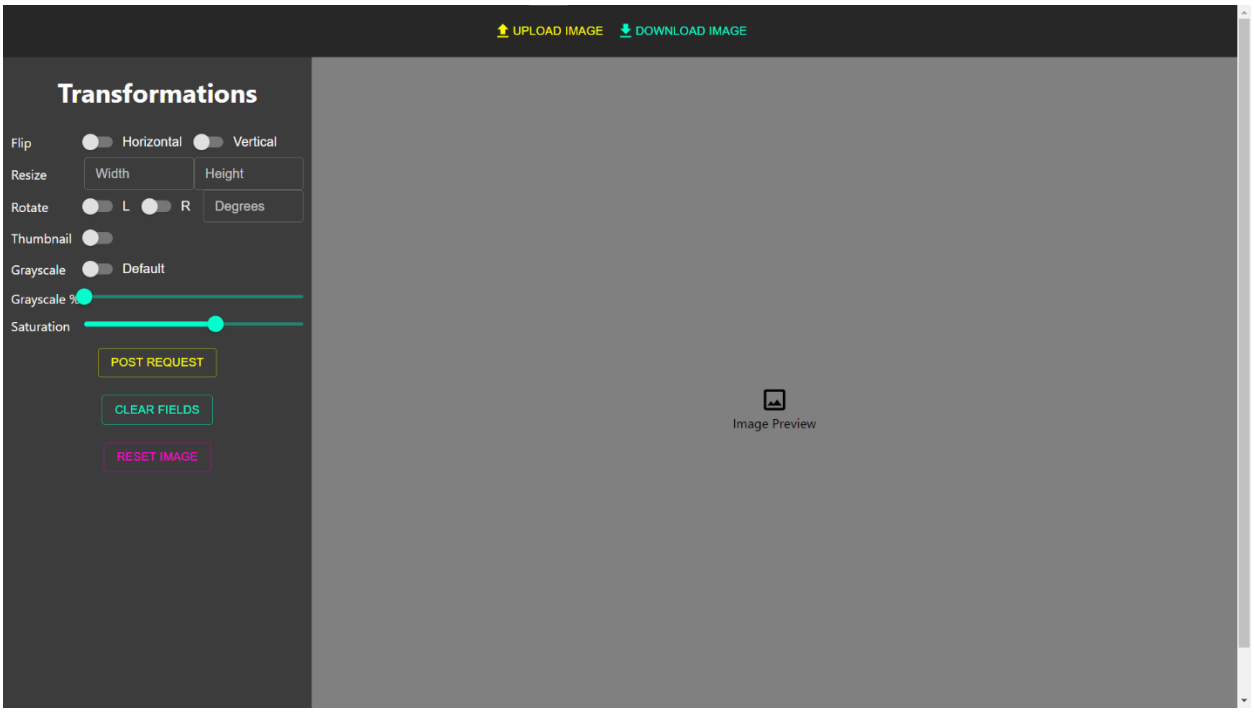


Figure 9: The front-end user interface client.

VI. Implementation

Three Implementations

Initially, the image processor API implementation was built using Python. However, a second implementation was developed later using C#, and a third implementation was developed last using TypeScript. The C# implementation is designated as the main implementation.

Source Code

The source code for the image processor API can be found at the following repository:

https://github.com/thoresonjd/image-processor-api

Tools & Technologies

- Back-end
  - C# API
    - Language: C#, .NET 6
    - HTTP/REST: Web API
    - Imaging: ImageSharp
  - Python API
    - Language: Python
    - HTTP/REST: Flask, Flask Restful, Flask CORS
    - Imaging: Pillow – Python Imaging Library (PIL) fork
  - TypeScript API
    - Language: TypeScript/JavaScript, NodeJS
    - HTTP/REST: ExpressJS
    - Imaging: SharpJS
- Front-end (Bonus)
  - Language: TypeScript
  - Framework: React w/ Material UI
  - HTTP/REST: Axios

Comments & Concerns

The three implementations (C#, Python, and TypeScript) function very similarly, but they do not always have the same results. For example, when provided with a negative saturation, the Python API appears to alter the hues of the image when saturating, whereas the C# API inverts the image before saturating. Furthermore, the TypeScript implementation does not support the Saturate or GrayscalePercentage commands.

- C# API
    - Transformations appear to be slow at times, but it is currently not known why this is the case.
    - ImageSharp does not allow for negative saturation. Therefore, as a workaround, when a negative value is provided, the colors of the image are inverted before saturation is applied.
- Python API
    - Pillow works fine but can have weird graphical side effects that reduce the resolution of the image, specifically when rotating by ***n*** degrees.
    - Rotating by ***n*** degrees cuts the corners off the initial image.
    - Specified grayscale percentage is applied pixel-by-pixel; therefore, larger images may take slightly longer to process.
    - Only JPG/JPEG and PNG extensions are currently supported.
- TypeScript API
    - Finding ways to control image saturation and grayscale by percentage have proven difficult with SharpJS as of now. Thus, the Saturate and GrayscalePercentage commands have not yet been implemented into the TypeScript API.
    - The TypeScript API will return status code 418 I'm a teapot when Saturate and GrayscalePercentage are invoked. This is to acknowledge that the aforementioned commands are valid in theory and within the scope of the design, but the TypeScript implementation is currently not built to handle the commands.
    - If Saturate and GrayscalePercentage are desired, it is best to use either the C# implementation or the Python implementation.