PicPro

CAB432 Assignment 2

Matthias Eder
11378093

Eric Zhang
10319191

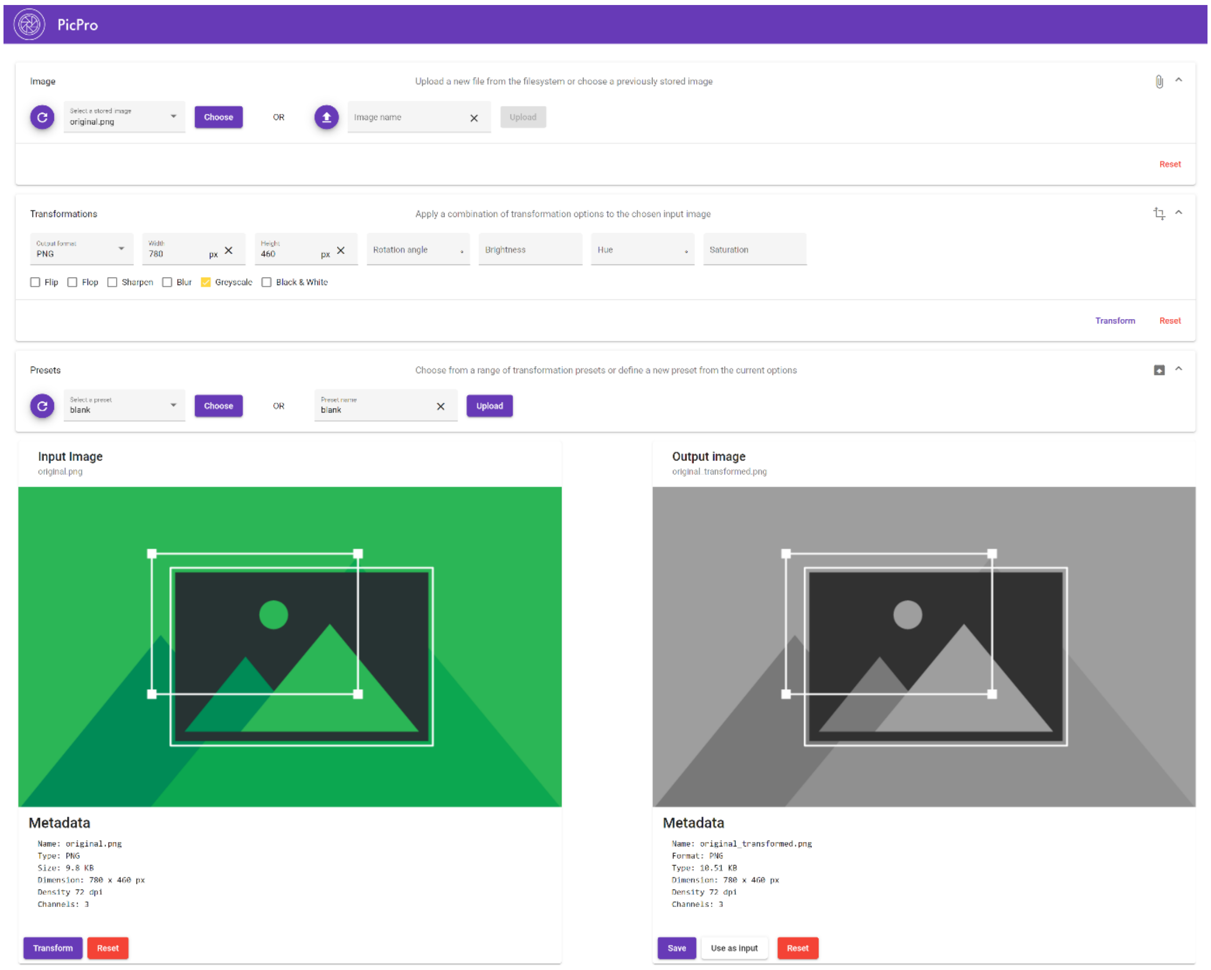
3 November 2022

Contents

Introduction

Purpose & description

The PicPro Image Transformer allows users to alter images using various predefined options, such as changing the filetype format, resizing the image, or adding effects like blur and greyscale. It allows new and transformed images to be stored for users that wish to use the same image multiple times, along with saving presets for transformations so that users don't need to repeatedly fill in the transformation values if they wish to use the same transformation over multiple images.


PicPro
Image
Upload a new file from the filesystem or choose a previously stored image
Choose
OR
Image name
Upload
Reset
Transformations
Apply a combination of transformation options to the chosen input image
Rotation angle
Brightness
Hue
Saturation
Transform
Reset
Presets
Choose from a range of transformation presets or define a new preset from the current options
Choose
OR
Upload
Input Image
Output image
Transform
Reset
Save
Use as input
Reset

Select a stored image: original.png

Output format: PNG
Width: 780 px
Height: 460 px
☐ Flip ☐ Flop ☐ Sharpen ☐ Blur ☑ Greyscale ☐ Black & White

Select a preset: blank
Preset name: blank

Input Image
original.png

Metadata
```
Name: original.png
Type: PNG
Size: 9.8 KB
Dimension: 780 x 460 px
Density 72 dpi
Channels: 3
```

Output image
original_transformed.png

Metadata
```
Name: original_transformed.png
Format: PNG
Type: 10.51 KB
Dimension: 780 x 460 px
Density 72 dpi
Channels: 3
```

Services used

Sharp Node.js module (v0.31.1)

The Sharp module accepts and reads an image file input to allow for various transformations to be done, such as filetype format, resizing the image, or adding effects like blur and greyscale etc.

Endpoint: `N/A as it is not an API, but a Node.js module`

Docs: https://sharp.pixelplumbing.com/

Use cases

US 1

As a	graphic designer
I want	to transform images like changing the format type or adding filters
So that	I can manipulate images to suit my design.

US 2

As a	graphic designer
I want	save previously uploaded images
So that	I don't need to reupload the same image and can access it from the service with ease.

US 3

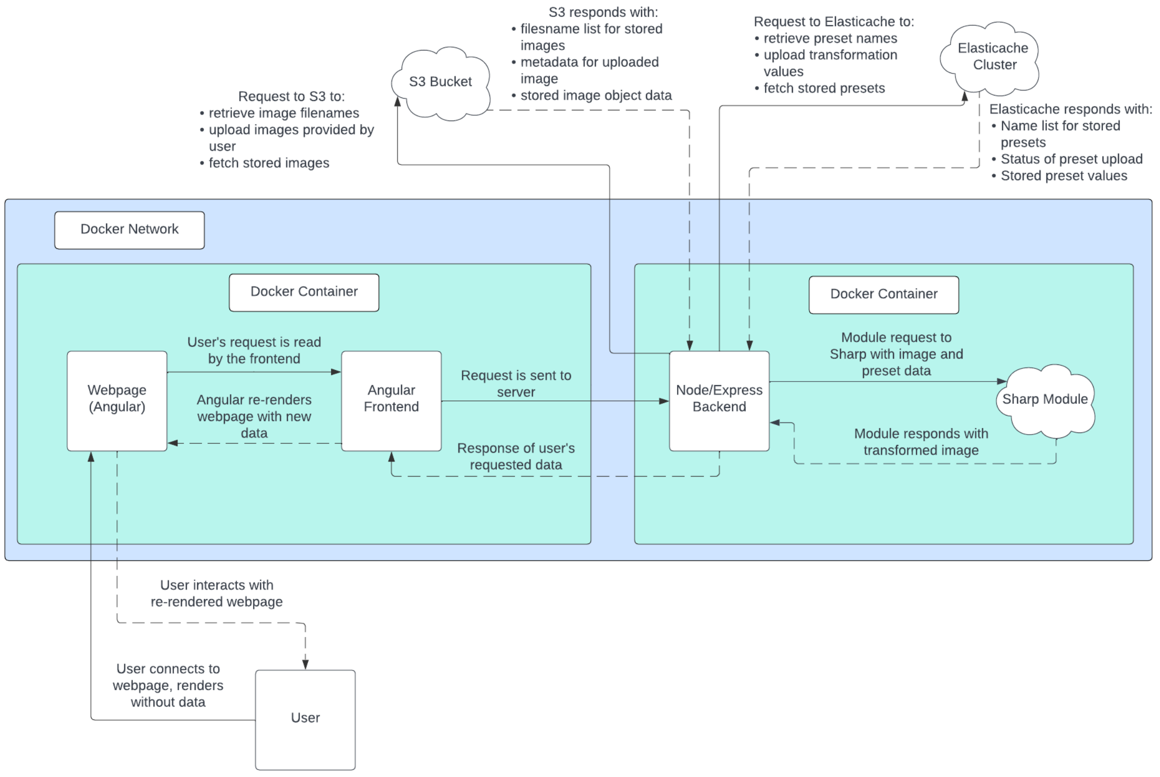
As a	graphic designer
I want	save previous transformation presets/values
So that	I can apply the same transformation to multiple images without having to fill in the values every time

Technical breakdown

Architecture

The flow of data can be seen in the context diagram below. The application consists of two docker containers, one containing the frontend, and one containing the backend. The frontend is a single page web application written with Angular, that processes and sends the user's input data to the backend. The backend is an Express application based on Node.js, receiving the input data and filtering the desired attributes that will be used for either storing/fetching images via S3, storing/fetching presets via ElastiCache, or transforming images via Sharp. Once the response is received, the backend sends the output back to the frontend to be displayed to the user.

Context Diagram


S3 responds with:
• filesname list for stored images
• metadata for uploaded image
• stored image object data
S3 Bucket
Request to Elasticache to:
• retrieve preset names
• upload transformation values
• fetch stored presets
Elasticache Cluster
Request to S3 to:
• retrieve image filenames
• upload images provided by user
• fetch stored images
Elasticache responds with:
• Name list for stored presets
• Status of preset upload
• Stored preset values
Docker Network
Docker Container
Docker Container
User's request is read by the frontend
Webpage (Angular)
Angular re-renders webpage with new data
Angular Frontend
Request is sent to server
Response of user's requested data
Node/Express Backend
Module request to Sharp with image and preset data
Sharp Module
Module responds with transformed image
User interacts with re-rendered webpage
User connects to webpage, renders without data
User

These containers are built separately with *docker compose* with the Docker file content shown in linked Appendix. The backend Docker file sets the work directory to the backend folder, copies the package.json file from the node folder so that it can install the necessary dependencies using npm install. Afterwards, it copies the rest of the code in the container to run the complied *server.js* file to start the backend. This gets exposed to port 3000. The Dockerfile of the client uses a multi-stage build. The first stage builds the Angular application and copies the artifacts. The second stage serves the app on a lightweight NGINX web server, which reduces the size of the Docker image heavily to around 25 MB. The original size, if serving the app on the development server, would have been over 2 GB. This setup follows best-practice guidelines for serving Angular applications in a production environment, and the server also acts as a reverse proxy to redirect requests between client and server in any environment. This gets exposed to port 80. These two Docker files are then used by the docker compose file to be built and tagged together, exposing them to the appropriate ports, 3000:3000 for the backend and 4200:80 for the frontend. Port 80 is needed so that the NGINX reverse proxy can listen on a separate port to redirect requests accordingly.

The way the application is deployed to AWS by pulling the images for frontend and backend from DockerHub, creating a Docker Network so that they can be accessed by container name and then running containers with the *--restart unless-stopped* flag set so that the containers are run on machine startup. This is needed for the dynamic creation of machines during scaling. The pm2 configuration file (ecosystem.config.js) is available in Appendix 4A. but it was not used in the end because pure Docker can achieve the same thing.

Client / server demarcation of responsibilities

The best way to describe the demarcation of responsibilities for the application is to again separate it into **frontend and backend**.

The **frontend** oversees the GUI (Graphical User Interface). What the users see on the webpage, from the header, to the three sections representing the Image, Transformation, and Preset features, along with the input/output images, are rendered from the frontend via Angular. The frontend also reads any input from the user so that it can be formatted inside a JSON request that gets sent back to the frontend, unpacking and re-rendering new output when the backend responds.

The **backend** handles the data processing. When the frontend makes a request, it sends it to the appropriate route, which can be categorised as the image and preset routes. The Appendix code has been linked below.

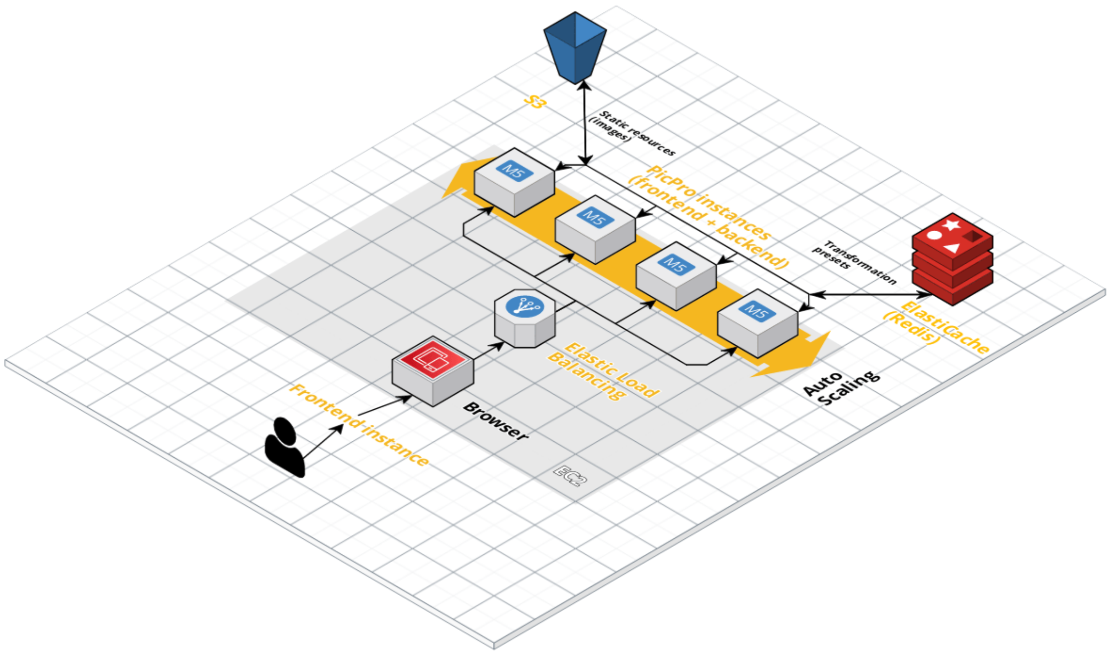
The **image routes** handle the requests from the Image and Transformation sections of the webpage, and there are 4 of them. The base route "images/" retrieves the filenames of stored S3 images so users can choose one to fetch. The "iamges/fetch" and "images/upload" routes handle storing and retrieving images from the S3 bucket, fetch responding back the chosen image information that will be displayed on the webpage while upload only responds with the metadata. The "iamges/transform" value again retrieves the desired image, using the transformation values sent from the frontend to apply the transformations via Sharp, before creating and sending the response containing the information for the newly edited image.

The **preset routes** handle the requests from the Preset section of the webpage, and there are 3 of them. These are similar to the image routes, and all of them interact with Redis via an ElastiCache Cluster: "presets/" responds with the preset names that users can choose from, "presets/fetch" retrieves the chosen preset that the frontend can use to fill in the Transformation section, and “presets/upload” stores the current Transformation values along with a preset name provided by the user.

Looking at the architecture diagram below to explain the demarcation of responsibilities done via AWS, most of the application is contained within EC2. As can be seen, the user interacts with the frontend instance using the webpage, and this interacts with the backend via an Elastic Load Balancer. This scales the instances up and down depending on the computational load the application is receiving.

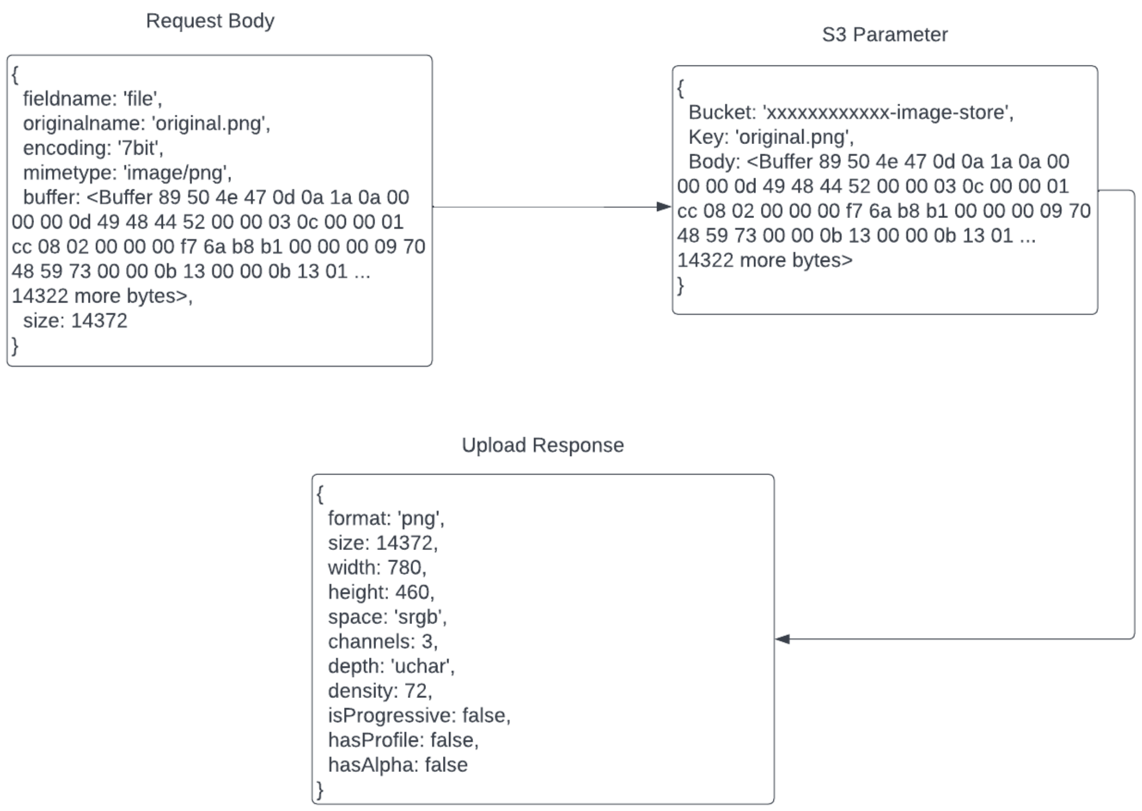
As shown in the diagram, the backend instances interact with the S3 bucket and the Elasticache Cluster when needed. S3 is a blob store, often referred to as object storage. It was chosen to store the images uploaded from the application due to how large and flexible it is, as users may upload several original and transformed images in quick succession. S3 can scale alongside the data stored in it, making it very suitable for large amounts of images.

ElastiCache for Redis is a memory-based cache and was chosen to store the presets that contain the transformation values due to its speed. As these presets are stored as JSON objects, they do not require the large flexibility in storage scaling that S3 boasts, and need to be accessed quickly, which ElastiCache provides very fast access time with very little latency, which is suitable for quickly retrieving the preset values that users will use to make their transformations.



Response filtering / data object correlation

The below diagram shows the data manipulation of the image upload route. The original request contains the image data, like the original name and the image buffer.



```
{
  fieldname: 'file',
  originalname: 'original.png',
  encoding: '7bit',
  mimetype: 'image/png',
  buffer: <Buffer 89 50 4e 47 0d 0a 1a 0a 00
00 00 0d 49 48 44 52 00 00 03 0c 00 00 01
cc 08 02 00 00 00 f7 6a b8 b1 00 00 00 09 70
48 59 73 00 00 0b 13 00 00 0b 13 01 ...
14322 more bytes>,
  size: 14372
}
```

```
{
  Bucket: 'xxxxxxxxxxxx-image-store',
  Key: 'original.png',
  Body: <Buffer 89 50 4e 47 0d 0a 1a 0a 00
00 00 0d 49 48 44 52 00 00 03 0c 00 00 01
cc 08 02 00 00 00 f7 6a b8 b1 00 00 00 09 70
48 59 73 00 00 0b 13 00 00 0b 13 01 ...
14322 more bytes>
}
```

```
{
  format: 'png',
  size: 14372,
  width: 780,
  height: 460,
  space: 'srgb',
  channels: 3,
  depth: 'uchar',
  density: 72,
  isProgressive: false,
  hasProfile: false,
  hasAlpha: false
}
```

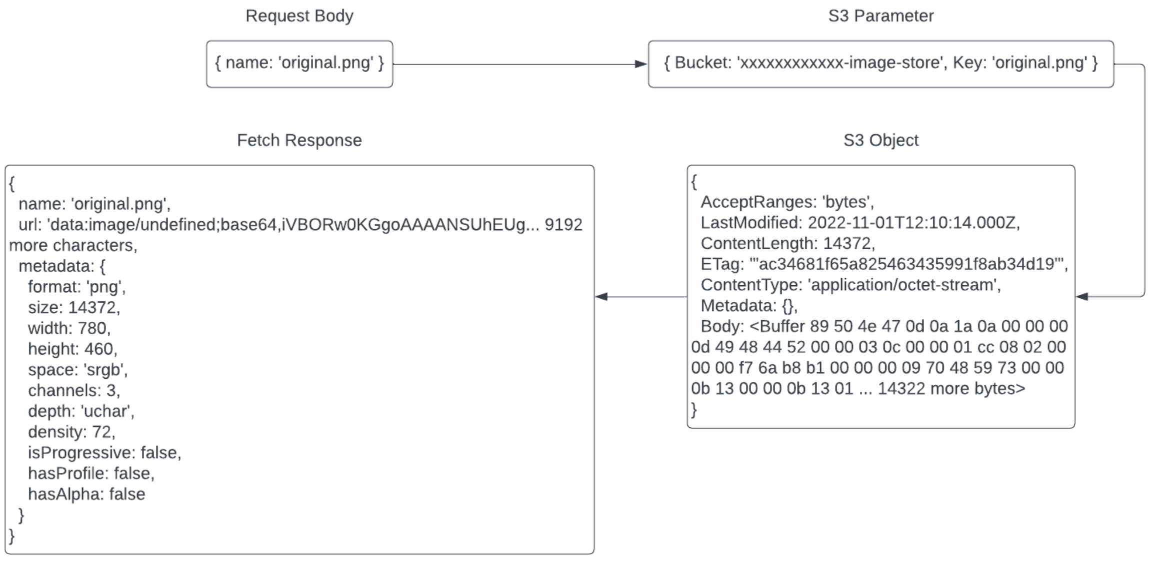
Buffer is used due to many file readers like the Sharp module often manipulating files as Buffer data, as they are suitable for fine-grained data manipulation, according to Digital Ocean **(2020, para. 3)**, along with its ease of conversion to other data types like a base64 URL, used to re-render the images to the webpage so they can be displayed back to the user.

The upload parameters contain the bucket name along with the filename from the request body and the image buffer from the image data. This gets uploaded to S3, while the response contains the metadata that is obtained using Sharp, as shown in the code segment below.

```
sharp(file.buffer)
        .metadata()
        .then(metadataResult => {
```

Fetch can be seen as going backwards from upload path. The original request is the filename of the image the user chose to retrieve. This filename is placed within a parameter JSON for S3, again containing the bucket name, which is used to retrieve the data. The object that is returned holds various information, like the last record of it being modified, and the body which contains the image buffer.

As metadata is not stored in S3 and is therefore blank, the route again uses Sharp to obtain it, which gets placed in the fetch response, also containing the filename and the image buffer which has been converted to a base64 URL.



```
{ name: 'original.png' }
```

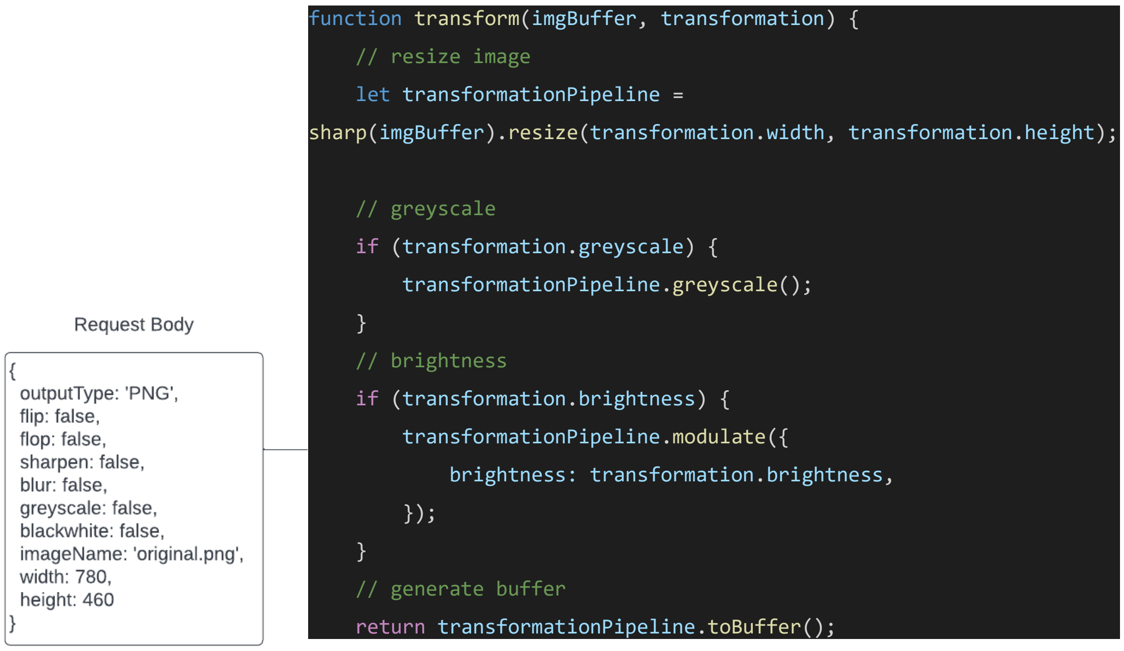
```
{ Bucket: 'xxxxxxxxxxxx-image-store', Key: 'original.png' }
```

```
{
  AcceptRanges: 'bytes',
  LastModified: 2022-11-01T12:10:14.000Z,
  ContentLength: 14372,
  ETag: '"ac34681f65a825463435991f8ab34d19"',
  ContentType: 'application/octet-stream',
  Metadata: {},
  Body: <Buffer 89 50 4e 47 0d 0a 1a 0a 00 00 00
0d 49 48 44 52 00 00 03 0c 00 00 01 cc 08 02 00
00 00 f7 6a b8 b1 00 00 00 09 70 48 59 73 00 00
0b 13 00 00 0b 13 01 ... 14322 more bytes>
}
```

```
{
  name: 'original.png',
  url: 'data:image/undefined;base64,iVBORw0KGgoAAAANSUhEUg... 9192
more characters,
  metadata: {
    format: 'png',
    size: 14372,
    width: 780,
    height: 460,
    space: 'srgb',
    channels: 3,
    depth: 'uchar',
    density: 72,
    isProgressive: false,
    hasProfile: false,
    hasAlpha: false
  }
}
```

Transform has a very similar path to fetch. Using the image name to obtain the desired image from S3, the request body containing the transformation values are sent to the transform function, that calls Sharp to apply the transformations using several if-statements to check through the request and apply the necessary transformations when values are supplied or marked as true.

The function first creates the Sharp pipeline by reading the image buffer and resizing based on the supplied width and height. If they haven't been changed from the original image, then it just stays the same size. Every if statement afterwards stacks a prebuilt sharp function to the pipeline to apply the transformation, below is only a short example of the full function. The full example is shown in Appendix 12A and 13A. At the end, it returns the pipeline containing all the transformations converted back into an image buffer.



```
{
  outputType: 'PNG',
  flip: false,
  flop: false,
  sharpen: false,
  blur: false,
  greyscale: false,
  blackwhite: false,
  imageName: 'original.png',
  width: 780,
  height: 460
}
```

```
function transform(imgBuffer, transformation) {
    // resize image
    let transformationPipeline =
sharp(imgBuffer).resize(transformation.width, transformation.height);

    // greyscale
    if (transformation.greyscale) {
        transformationPipeline.greyscale();
    }
    // brightness
    if (transformation.brightness) {
        transformationPipeline.modulate({
            brightness: transformation.brightness,
        });
    }
    // generate buffer
    return transformationPipeline.toBuffer();
```

The transform function returns the new image buffer for the transformed image. As seen in the code below, the original filename is split at the full stop so that '_transformed' can be added to differentiate the image from the original. This is only a default naming and when the user decides to download or save the image, they will be queried to supply a custom file name. The response is the same as previously shown in the fetch diagram, returning the new name, converted image URL and metadata obtained from Sharp.

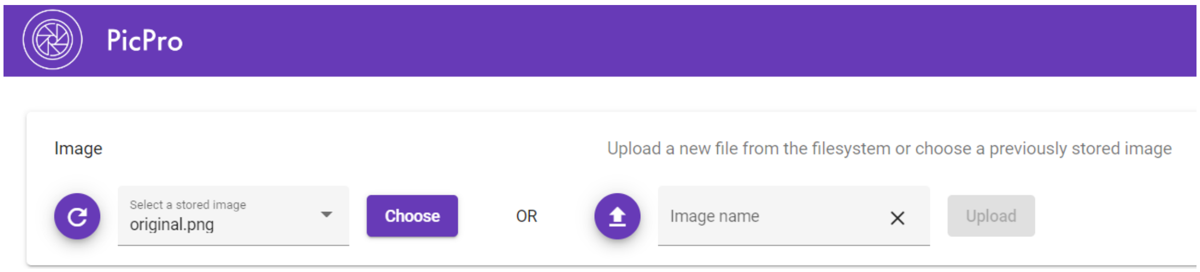
```
const fileNameParts = inputImageName.split('.');
const outputImageName = fileNameParts[0] + '_transformed.';
```

Presets stay largely the same, a JSON object containing the transformation values, alongside a unique name or key. The values are displayed on the frontend, uploading retrieving currently filled in values, and fetching filling in values from the preset. Examples of the preset JSON are explained in the Presets section of Features.

Features

The flow of the data in our project, as well as its architecture, can be best described by the three main features of PicPro: Image, Transformations, and Presets

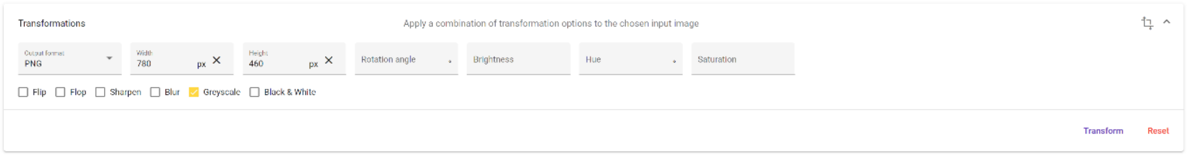
Image


PicPro
Image
Upload a new file from the filesystem or choose a previously stored image
Choose
OR
Image name
Upload

Select a stored image: original.png

The **Image** section allows users to either upload an image or choose from a set of stored images. Uploading an image creates a POST request to store them as a Buffer object in an S3 bucket. When accessing stored images, the keys are retrieved from S3 with a GET request so users can select which one to fetch (which uses the user-defined file names as keys, like "original.png"). Only fetching the file names to show in the selection element instead of the whole image data is intentional to reduce network load if the user decides to upload a new file instead. Thus, the actual data – which can be quite large depending on the picture – is only loaded once the user makes the intentional decision to load an image by clicking the "Choose" button. Once an image is chosen, a POST request is made to fetch the actual image data from S3, which is converted to a base64-encoded data URL and sent as a JSON response along with the filename key and metadata from the Sharp module. This gets re-rendered and displayed by the frontend to display the output to the user.

Transformations

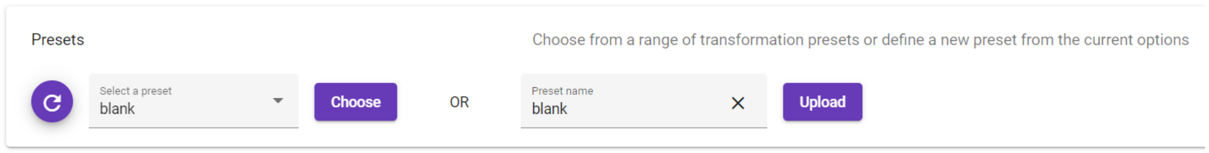

Transformations
Apply a combination of transformation options to the chosen input image
Rotation angle
Brightness
Hue
Saturation
Transform
Reset

Output format: PNG
Width: 780 px
Height: 460 px
☐ Flip ☐ Flop ☐ Sharpen ☐ Blur ☑ Greyscale ☐ Black & White

```
{
   outputType: 'PNG',
   flip: false,
   flop: false,
   sharpen: true,
   blur: true,
   greyscale: false,
   blackwhite: false,
   imageName: 'original.png'
   width: 5,
   height: 5,
   rotationAngle: 5,
   brightness: 5
}
```

The **Transformations** section allows users to select how they want to alter the chosen image. The values adjusted here are stored in a JSON request that gets sent alongside the data of your chosen image. Referring to the code example to the left, the transformation JSON contains the "imageName" which just stores a unique key for the requested image, and "transformations" that stores the values. The checkboxes are stored as Boolean values to identify whether they are to be applied to the image, and the text values get stored as string and number values as they control transformations that can be adjusted or have multiple options. Numerical values are optional so if the user does not define values there, they are also not sent to the server. Users can use the buttons on the bottom right to either apply the transformations or reset the values.

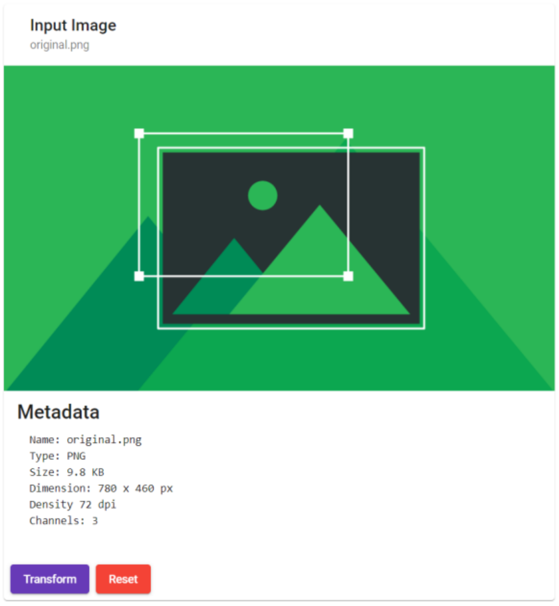
Presets


Presets
Choose from a range of transformation presets or define a new preset from the current options
Select a preset
blank
Choose
OR
Preset name
blank
Upload

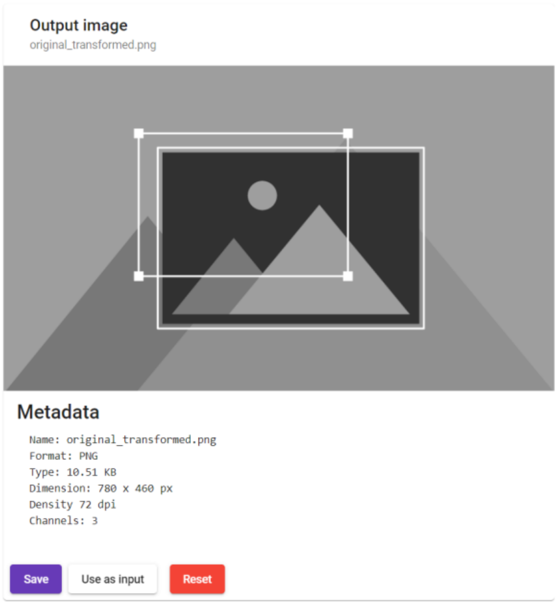
```
{
  name: 'blank',
  transformation: {
    outputType: 'JPEG',
    flip: false,
    flop: false,
    sharpen: true,
    blur: true,
    greyscale: false,
    blackwhite: false,
    width: 5,
    height: 5,
    rotationAngle: 5,
    brightness: 5
  }
}
```

The **Presets** section allows users to store previously used transformations. Uploading a preset takes the values from the Transformation section, as can be seen in the code example to the left. The preset JSON is slightly different from the transformation JSON shown before, being separated into 2 parts: “name” which just stores a unique key for the transformation that can be used later, and “transformation” that stores the values. This is sent as a POST request to the backend that stores the preset JSON in an ElastiCache cluster using Redis. Any previously stored presets have their keys retrieved with a GET request, (like how filenames are displayed in the Image section) displaying the names of all the presets the user can choose from. The same reasoning can be applied here for the choice to only fetch names instead of the whole data as we wanted to focus on computational load while reducing network load. Once chosen, a POST fetch request is made to retrieve the desired preset, which gets sent to the frontend to replace the values in the Transformations section so that they can be used on an image.

Input/Output



```
Name: original.png
Type: PNG
Size: 9.8 KB
Dimension: 780 x 460 px
Density 72 dpi
Channels: 3
```



```
Name: original_transformed.png
Format: PNG
Type: 10.51 KB
Dimension: 780 x 460 px
Density 72 dpi
Channels: 3
```

The images are displayed below these 3 sections, the original input shown on the left and the transformed output on the right. When a transformation is made, users have the option to save the new image, which brings up a popup dialog (shown below) that lets you choose a filename to either download to the local machine or upload to the S3 bucket. Users can also choose to use the output image again as the input image if further transformations need to be applied to the new image, or reset to use a new image or transformation.

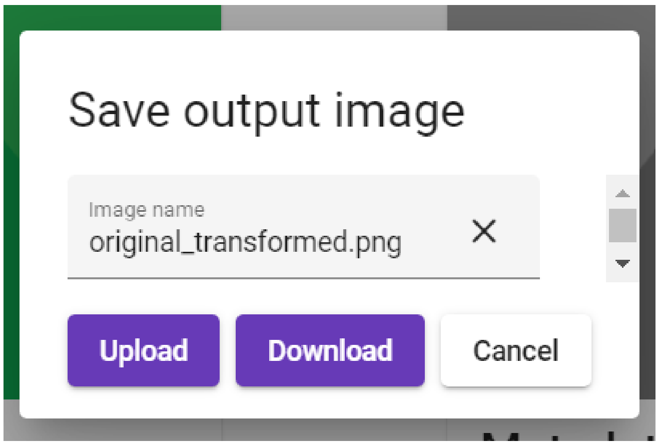

Save output image

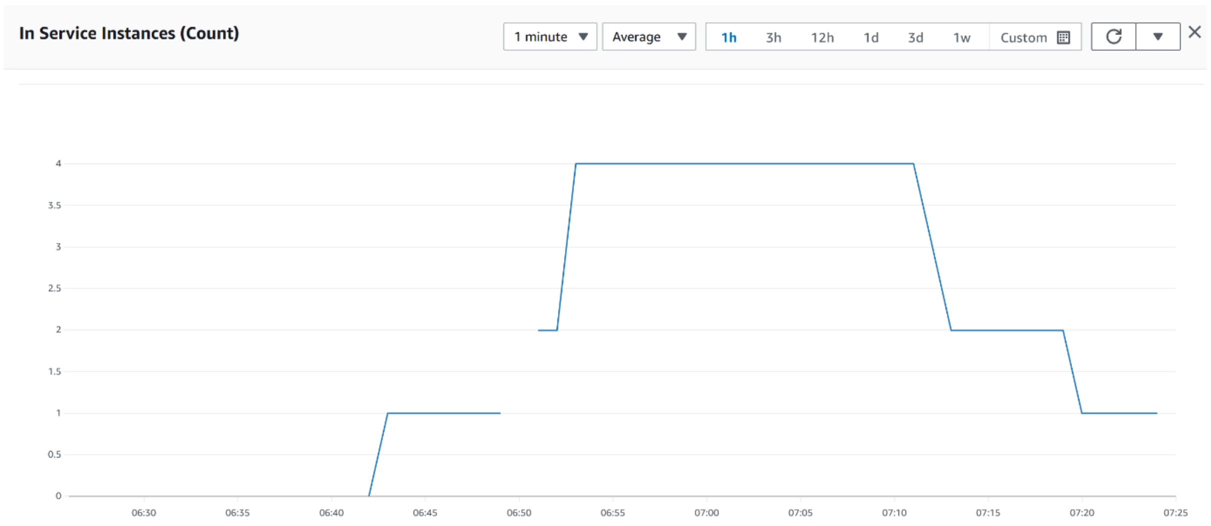
Image name
original_transformed.png

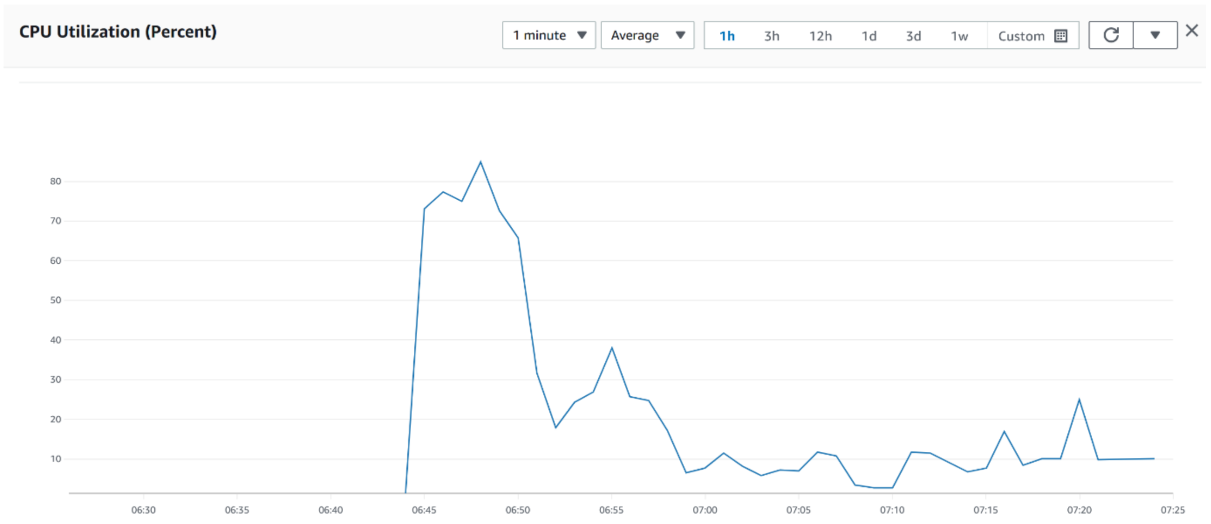
Scaling and Performance

The application receives load from the computationally expensive image transformations and the amount of requests received. A few successive requests with relatively intense transformations like transcoding a large JPEG to a GIF and resizing to e.g. 8000px x 8000px will also max out the CPU capabilities of a single machine. While Sharp is very well optimised and usually performs a transformation pipeline in a few hundred miliseconds, using large input images with high resolution will easily create enough computational load from processing to allow easy scaling.

The scaling setup on AWS is very similar to the respective prac. Machines running frontend and backend are hosted on an LoadBalancer. The scaling policy in the AutoScalingGroup is set to use the avergae CPU utilization as the scaling metric and will scale out at a threshold value of 50%. We experimented with this value a while and found it easier to set it to a lower value for the sake of presentation and testing. A more flexible real-world setting with the potential of more resources would allow increasing the target value for CPU utlization, but we encountered some problems with higher values. If the ASG scaled out to the maximum number of machines quickly and further requests were sent, then the resources were also quickly maxed out resulting in Gateway problems. Investigating these issues were deemed to be out-of-scope for this assignment.

The screnshots shown below provide evidence of a proper deployment of our application in an elastic cloud setting with the ASG configured according to the above mentioned metrics. The first image shows the amount of in-use instances of the LoadBalancer over time. The second image shows the respective CPU utilization over the same time period. It can be observed that the CPU load increased rapidly at first before being cushioned by successful scaling-out. In this experiment, a Postman runner was setup to send 50 high-intensity transformation requests.





The cloud setup was configured according to our architecture diagram. A sidenote would be that in the diagram, the LoadBalancer is placed "behind" the browser block while technically, a single machine will run both the frontend and backend containers. In case of scaling-out, the LoadBalancer will thus also create new instances of the frontend that could be accessed via their instance address and respective port. However, this was pictured intentionally in this way to hide complexity as we will only interact with a single browser and focus very much on data processing in the backend. Thus, the architecture diagram is shown from a user perspective.

Test plan

Task	Expected Outcome	Result	Screenshot (Appendix B)
Image Select	Drop down displayed currently stored images	PASS	1
Image Upload	Loading bar shows during request, image file displays below	PASS	2
Image Fetch	Loading bar shows during request, stored image displays below	PASS	3
Image Section Reset	Input/Output images are empty again	PASS	4

Transform Values	User input is displayed in transformation fields/checkboxes	PASS	5
Transform Image	Transformed image appears as output below	PASS	6
Transform Reset	Transformation values are reset to the input's original values	PASS	7
Preset Select	Drop down displayed currently stored presets	PASS	8
Preset Upload	Transformation values are stored, now available to select	PASS	9
Preset Fetch	Preset values fill in the above transformation values	PASS	10
Input Transform	Transformed image appears as output below	PASS	6
Input Reset	Input/Output images are empty again	PASS	4
Output Upload	Popup appears, image is saved with new name and is now available to select	PASS	11, 12
Output Download	Popup appears, image is downloaded with new name	PASS	11, 13
Output to Input	Input image is replaced with output image	PASS	14
Output Reset	Output image is empty again	PASS	15
No Image Input Disables Transform	Transform button is not highlighted and can't be clicked/triggered	PASS	16
No Image Chosen Disables Fetch	Choose button is not highlighted and can't be clicked/triggered	PASS	17
No File Chosen Disables Upload	Upload button is not highlighted and can't be clicked/triggered	PASS	18
No Preset Chosen Disables Fetch	Choose button is not highlighted and can't be clicked/triggered	PASS	19

Please refer to the images provided in Appendix B to confirm the passing of the test cases. Some are hard to prove with a simple screenshot. We will be happy to show that these work in the demo.

Difficulties and Reflexion

The setup of the application was fairly straight-forward. We re-used most of the template Matthias implemented for the first assignment with regard to the frontend and backend projects, the dockerization especially for the NGINX webserver and reverse-proxying and the basic setup of the routing in the backend were re-used. The key points in implementation were to figure out in what way the image data can be stored efficiently in S3 so that few steps are necessary to upload and retrieve it. The connection and upload/download from and to S3 was also largely re-used. For Redis and ElastiCache it was mostly the connection and understanding how the cluster works that posed a slight challenge but was quickly resolved.

There aren't any notable differences when comparing the proposal to the final product. The features and use cases mentioned in the proposal were successfully implemented.

When images or presets are uploaded, if the same name is used as a previously stored object, the stored object will be overridden by the new image or preset. There currently isn't a warning in place to let users know they are about to overwrite a stored object, and this could be something to add in a future version.

As no external APIs were used, there weren't any constraints there. The sharp module used to apply the transformations and retrieve metadata was relatively simple to learn and use. The application is however limited to what Sharp can do, so something like applying a mosaic blur would most likely require another module or API to add additional features for future versions.

When uploading presets, users can save them with a blank name, the current application does not require users to fill in the upload text field. This is something that should be fixed in a future version, detecting if there is text in the name field before allowing users to upload a preset

Extensions

These are a few extensions that could possibly be added onto future versions of the application

- Allow users to have a private storage, uploading images/presets that can't be accessed by others
- Add additional image transformation features, like mosaic blur, background removal, etc
- Allow the input and output image to be downloaded as a photoshop psd file, separating some of the transformations as separate layers for further editing
- Presets provide a preview of what transformations values them contain, so users don't need to fetch them to see what they are
- Transformations provide a preview of what the image looks like after it's transformed, so users can make more fine-tuned edits

User guide

The "Features" section explains the components of our application in a visual way. In general, a user should either choose a previously uploaded image or upload a new one in the "Image" panel. Then the "Transformation" panel can be opened and settings can be made. The "Transform" button will trigger the processing and display the output image. The transformation settings can be saved in the "Presets" panel. Generally, the UI explains where the functionalities are contained and how it should be used.

For a guide on how to run the Docker containers, please refer to the README file(s) in the repository.

References

Stack Abuse. (2020). Using Buffers in Node.js (Introduction). Digital Ocean. Retrieved October 28, 2022, from https://www.digitalocean.com/commun1.ity/tutorials/using-buffers-in-node-js

Appendices

Appendix A

Appendix 1A – Dockerfile Frontend

```
# Stage 1: Build
FROM node:16-alpine as build
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build --prod

# Stage 2: Run
FROM nginx:stable-alpine
# Remove default nginx website
RUN rm -rf /usr/share/nginx/html/*
# Copy dist folder from build stage to nginx public folder
COPY --from=build /usr/src/app/dist/frontend /usr/share/nginx/html
# Copy nginx config file
COPY nginx.conf /etc/nginx/conf.d/default.conf
# Expose port
EXPOSE 80
```

Appendix 2A – Dockerfile Backend

```
FROM node:16-alpine

# Set the work directory to app folder.
# We will be copying our code here
WORKDIR /backend

# Copy package.json file in the node folder inside container
COPY package.json .

# Install the dependencies in the container
RUN npm install

# Copy the rest of the code in the container
COPY . .

# Run the node server with compiled server file
CMD ["node", "server.js"]

# Expose port
EXPOSE 3000
```

Appendix 3A – docker-compose.yml

```yaml
version: '2.2'
services:
  backend:
    build: ./backend
    container_name: backend
    image: meder96/picpro:backend
    ports:
      - "3000:3000"
  frontend:
    build: ./frontend
    container_name: frontend
    image: meder96/picpro:frontend
    ports:
      - "4200:80"
```

Appendix 4A – pm2 ecosystem.config.js

```javascript
module.exports = {
    apps: [{
        name: "docker network",
        script: "sudo docker network create net"
    },
    {
        name: "backend",
        script: "sudo docker run --network net --name backend -p 3000:3000 -t 
meder96/picpro:backend"
    },
    {
        name: "frontend",
        script: "sudo docker run --network net --name frontend -p 4200:80 -t 
meder96/picpro:frontend"
    }]
}
```

Appendix 5A – Image "/" route

```
router.get('/', (_, res) => {
    console.log("⚡ Received request to /images");

    const params = { Bucket: s3BucketName };
    const fileNames = [];
    s3.listObjectsV2(params).promise()
        .then(listRes => {
            for (var i = 0; i < listRes.Contents.length; i++) {
                fileNames.push(listRes.Contents[i].Key);
            }
            console.log(`ℹ Retrieved ${listRes.KeyCount} images from S3`);
            res.send(fileNames);
        })
        .catch(error => {
            console.error(`❌ Error during image listing from S3: ${error}`);
        });
});
```

Appendix 6A – Image "/fetch" route

```
router.post('/fetch', (req, res) => {
    console.log("⚡ Received request to /images/fetch");
    const fileName = req.body.name;
    const params = {
        Bucket: s3BucketName,
        Key: fileName
    }
    s3.getObject(params).promise()
        .then(data => {
            const imgBuffer = data.Body;
            sharp(imgBuffer).metadata()
                .then(metadataResult => {
                    console.log(`✅ Successfully fetched image
\'${fileName}\' from \'${s3BucketName}\'`);
                    res.send({
                        name: fileName,
                        url: getDataUrlFromBuffer(imgBuffer,
metadataResult.mimeType),
                        metadata: metadataResult
                    });
                })
                .catch(error => {
                    console.error(`❌ Error during image analysis for
metadata: ${error}`);
                });
        })
        .catch(error => {
            console.error(`❌ Error during image retrieval of file
\'${fileName}\' from S3: ${error}`);
        });
});
```

Appendix 7A – Image "/upload" route

```
router.post('/upload', upload, (req, res) => {
    console.log("⚡ Received request to /images/upload");
    const file = req.file;
    const fileName = req.body.name;
    sharp(file.buffer)
        .metadata()
        .then(metadataResult => {
            // use multipart upload for potentially large files
            var upload = new AWS.S3.ManagedUpload({
                params: {
                    Bucket: s3BucketName,
                    Key: fileName,
                    Body: file.buffer
                }
            });
            upload.promise()
                .then(() => {
                    console.log(`✅ Uploaded image \'${fileName}\' to
\'${s3BucketName}\'`);
                    res.send(metadataResult);
                })
                .catch(error => {
                    console.error(`❌ Error during image upload to S3:
${error}`);
                });
        })
        .catch(error => {
            console.error(`❌ Error during image analysis for metadata:
${error}`);
        });
});
```

Appendix 8A – Image "/transform" route

```
router.post('/transform', (req, res) => {
    // main endpoint for image transformation, use sharp here
    console.log("⚡ Received request to /images/transform");
    const inputImageName = req.body.imageName;
    const params = { Bucket: s3BucketName, Key: inputImageName };
    s3.getObject(params).promise()
        .then(data => {
            const imgBuffer = data.Body;
            transform(imgBuffer, req.body)
                .then(outputImgBuffer => {
                    const url = getDataUrlFromBuffer(outputImgBuffer);
                    const fileNameParts = inputImageName.split('.');
                    const outputImageName = fileNameParts[0] +
'_transformed.';
                    sharp(outputImgBuffer).metadata()
                        .then(outputMetadata => {
                            res.send({
                                name: outputImageName +
outputMetadata.format.toLowerCase(),
                                url: url,
                                metadata: outputMetadata
                            });
                        })
                        .catch(metadataError => {
                            console.error(`❌ Error during image analysis for
metadata: ${metadataError}`);
                            res.status(500).send(metadataError);
                        });
                })
                .catch(transformationError => {
                    console.error(`❌ Error during image transformation:
${transformationError}`);
                    res.status(500).send(transformationError);
                });
        })
        .catch(retrievalError => {
            console.error(`❌ Error during image retrieval of file
\'${inputImageName}\' from S3: ${retrievalError}`);
            res.status(500).send(retrievalError);
        });

});
```

Appendix 9A – Preset "/" route

```
router.get('/', (_, res) => {
    console.log("⚡ Received request to /presets/");

    // return the "name" attributes of all stored preset objects in redis

    const keyNames = [];
    redisClient.keys('*').then((result) => {
        for (var i = 0; i < result.length; i++) {
            keyNames.push(result[i]);
        }
        console.log(`ℹ Retrieved ${result.length} presets from redis`);
        res.send(keyNames);
    })
        .catch((err) => {
            console.error(`❌ Error during preset listing from redis:
${err}`);
        });
})
```

Appendix 10A – Preset "/fetch" route

```
router.post('/fetch', (req, res) => {
    console.log("⚡ Received request to /presets/fetch");

    // return a specific preset object by its "name" attribute that is passed
in the request body
    // retrieve from redis
    const redisKey = req.body.name;

    redisClient.get(redisKey)
        .then((result) => {
            if (result) {
                console.log(`✅ Successfully fetched preset \'${redisKey}\'
from Redis`);
                res.send(JSON.parse(result));
            }
        })
        .catch((err) => {
            console.error(`❌ Error when fetching preset \'${redisKey}\' from
Redis: ${err}`);
        });
})
```

Appendix 11A – Preset "/upload" route

```
router.post('/upload', (req, res) => {
    console.log("⚡ Received request to /presets/upload");

    // upload a a preset object that is passed in the request body to redis and return a boolean if the operation was successful
    // we might change this later to a PUT method depending if we really need the success boolean, but it's fine for now
    const redisKey = req.body.name;
    const redisJSON = req.body;

    redisClient.set(redisKey, JSON.stringify({ ...redisJSON }))
        .then((result) => {
            console.log(`✅ Successfully uploaded preset \'${redisKey}\' to Redis: ${result}`);
            res.send(true);
        })
        .catch((err) => {
            console.error(`❌ Error during upload of preset \'${redisKey}\' to Redis: ${err}`);
            res.send(false);
        });
})
```

Appendix 12A – Transform function part 1

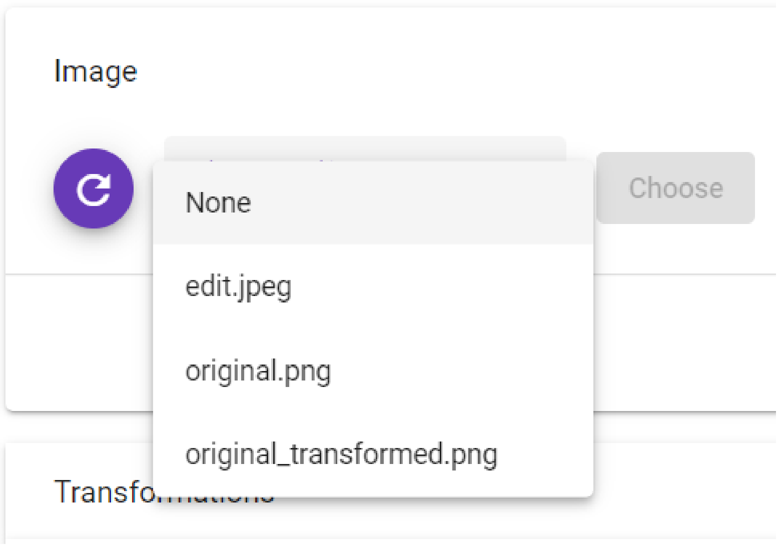
```
function transform(imgBuffer, transformation) {
    // resize image
    let transformationPipeline = sharp(imgBuffer).resize(transformation.width, transformation.height);

    // greyscale
    if (transformation.greyscale) {
        transformationPipeline.greyscale();
    }

    // black and white
    if (transformation.blackwhite) {
        transformationPipeline.threshold(100);
    }

    // brightness
    if (transformation.brightness) {
        transformationPipeline.modulate({
            brightness: transformation.brightness,
        });
    }

    // saturation
    if (transformation.saturation) {
        transformationPipeline.modulate({
            saturation: transformation.saturation,
        });
    }

    // hue
    if (transformation.hue) {
        transformationPipeline.modulate({
            hue: transformation.hue,
        });
    }

    // blur
    if (transformation.blur) {
        transformationPipeline.blur(transformation.blur);
    }

    // rotate
    if (transformation.rotationAngle) {
        transformationPipeline.rotate(transformation.rotationAngle);
    }
```

Appendix 13A – Transform function part 2

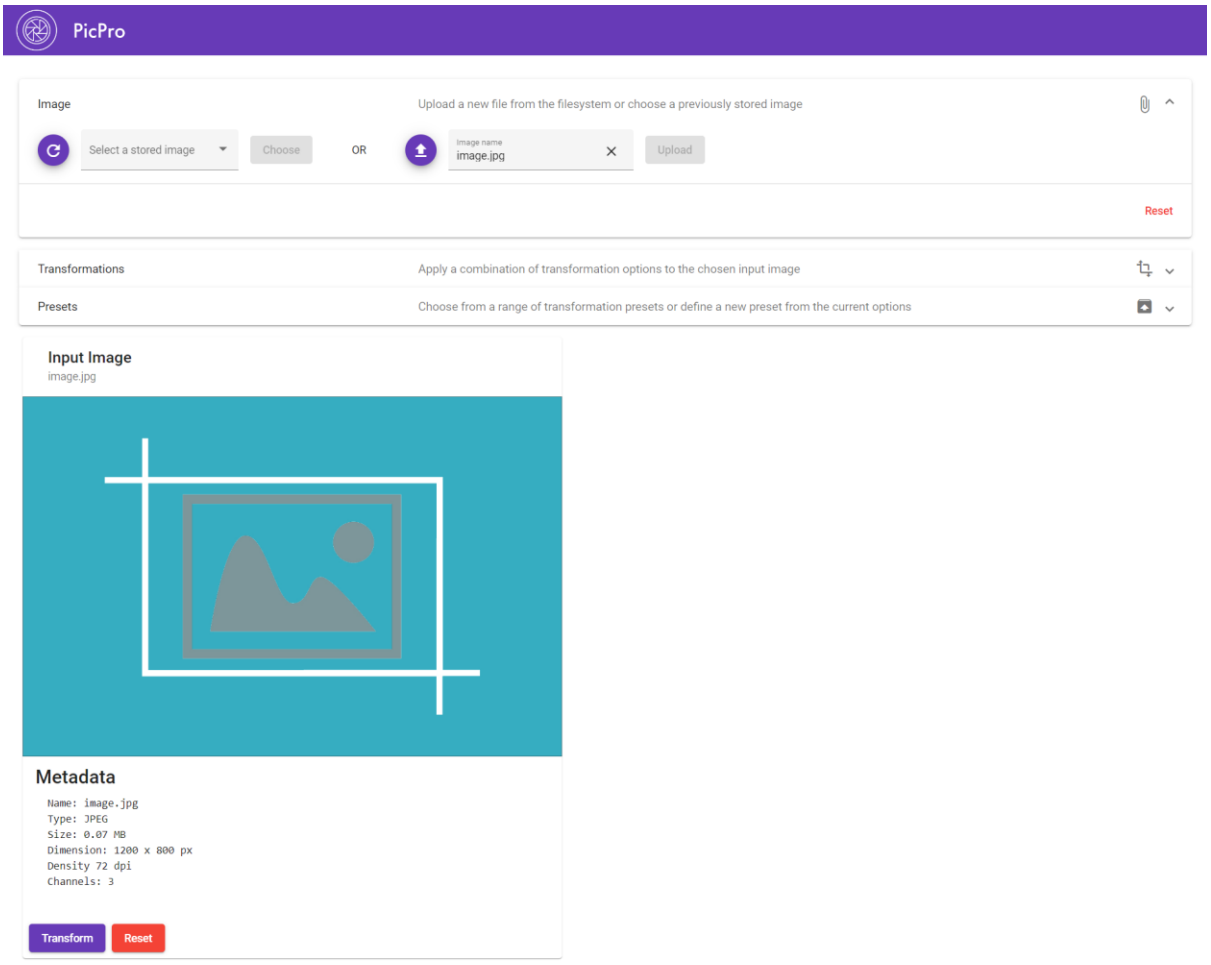
```
    // flip
    if (transformation.flip) {
        transformationPipeline.flip();
    }

    // flop
    if (transformation.flop) {
        transformationPipeline.flop();
    }

    // sharpen
    if (transformation.sharpen) {
        transformationPipeline.sharpen();
    }

    // transcode
    if (transformation.outputType) {
        transformationPipeline.toFormat(transformation.outputType);
    }

    // generate buffer
    return transformationPipeline.toBuffer();
}
```

Appendix B

Appendix 1B



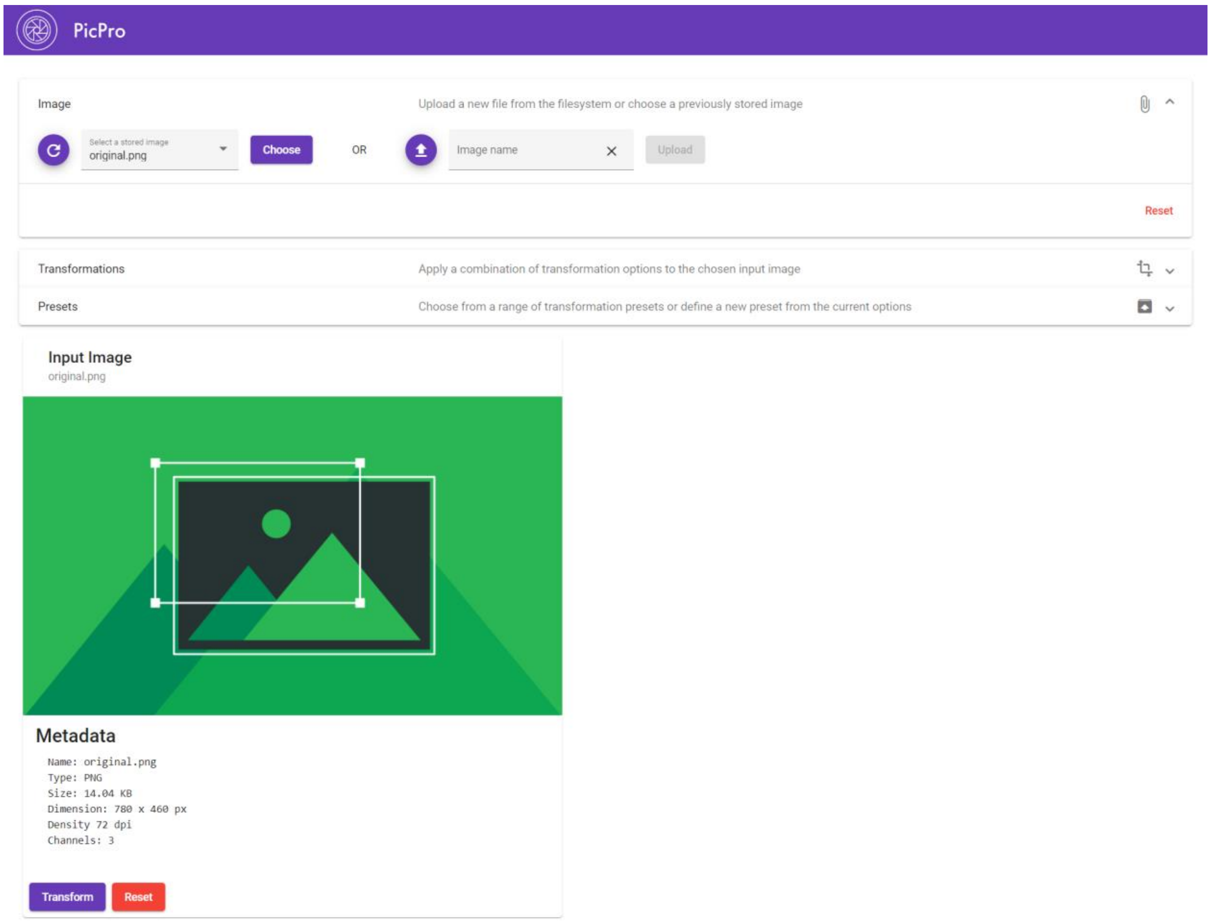
Appendix 2B



PicPro
Image
Upload a new file from the filesystem or choose a previously stored image
Select a stored image
Choose
OR
Image name
image.jpg
Upload
Reset
Transformations
Apply a combination of transformation options to the chosen input image
Presets
Choose from a range of transformation presets or define a new preset from the current options
Input Image
image.jpg
Metadata
Transform
Reset

```
Name: image.jpg
Type: JPEG
Size: 0.07 MB
Dimension: 1200 x 800 px
Density 72 dpi
Channels: 3
```

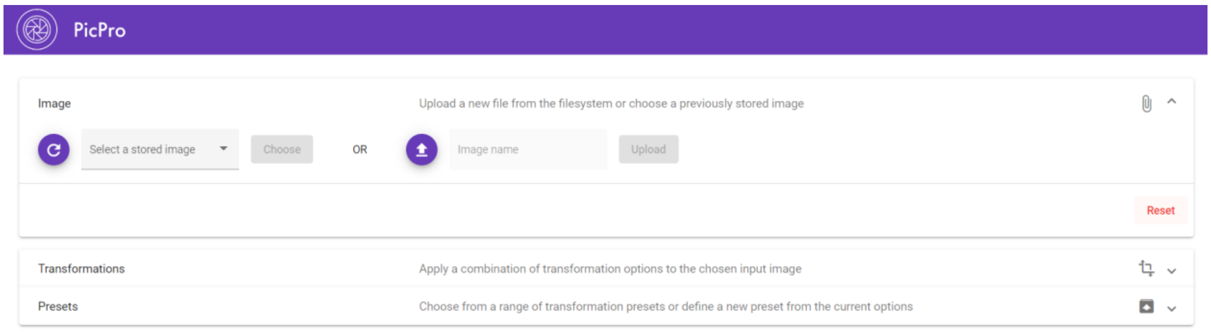
Appendix 3B


PicPro
Image
Upload a new file from the filesystem or choose a previously stored image
Select a stored image
original.png
Choose
OR
Image name
Upload
Reset
Transformations
Apply a combination of transformation options to the chosen input image
Presets
Choose from a range of transformation presets or define a new preset from the current options
Input Image
original.png

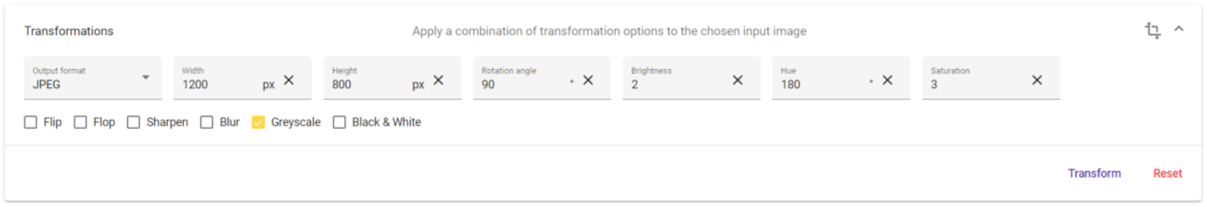
Metadata

```
Name: original.png
Type: PNG
Size: 14.04 KB
Dimension: 780 x 460 px
Density 72 dpi
Channels: 3
```

Appendix 4B


PicPro
Image
Upload a new file from the filesystem or choose a previously stored image
Select a stored image
Choose
OR
Image name
Upload
Reset
Transformations
Apply a combination of transformation options to the chosen input image
Presets
Choose from a range of transformation presets or define a new preset from the current options

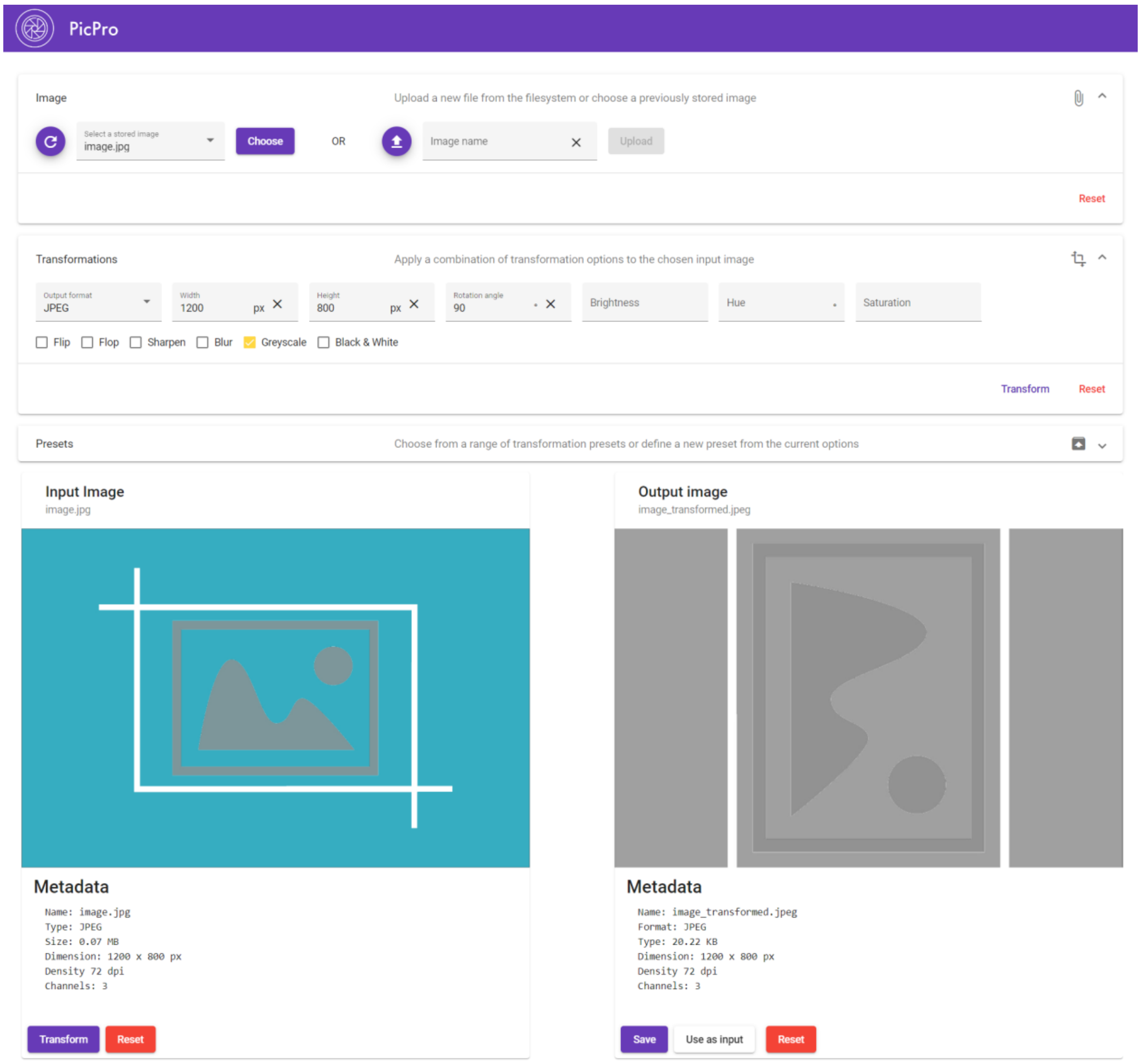
Appendix 5B


Transformations
Apply a combination of transformation options to the chosen input image

Output format JPEG
Width 1200 px
Height 800 px
Rotation angle 90
Brightness 2
Hue 180
Saturation 3

☐ Flip ☐ Flop ☐ Sharpen ☐ Blur ☑ Greyscale ☐ Black & White

Appendix 6B



Image

Upload a new file from the filesystem or choose a previously stored image

Select a stored image: image.jpg

Transformations

Apply a combination of transformation options to the chosen input image

Output format: JPEG

Width: 1200 px

Height: 800 px

Rotation angle: 90

Flip [ ] Flop [ ] Sharpen [ ] Blur [ ] Greyscale [✓] Black & White [ ]

Presets

Choose from a range of transformation presets or define a new preset from the current options

Input Image

image.jpg

Metadata

```
Name: image.jpg
Type: JPEG
Size: 0.07 MB
Dimension: 1200 x 800 px
Density 72 dpi
Channels: 3
```

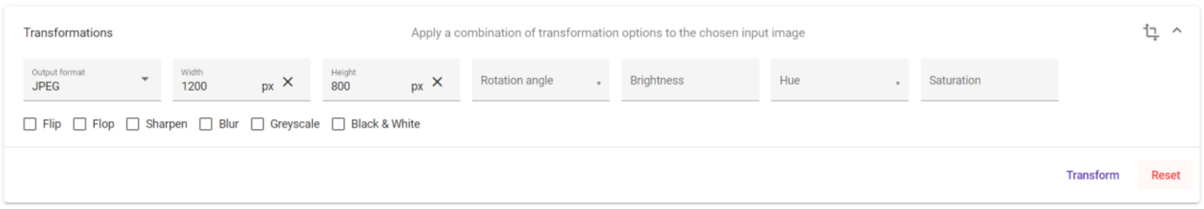
Output image

image_transformed.jpeg

Metadata

```
Name: image_transformed.jpeg
Format: JPEG
Type: 20.22 KB
Dimension: 1200 x 800 px
Density 72 dpi
Channels: 3
```

Appendix 7B



Transformations

Apply a combination of transformation options to the chosen input image

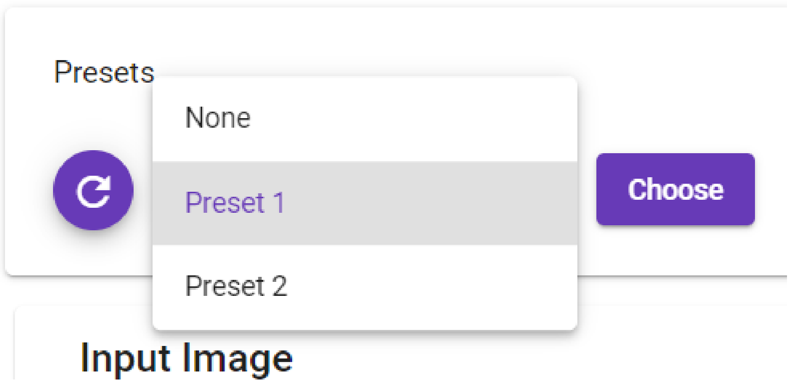
Output format: JPEG

Width: 1200 px

Height: 800 px

Flip [ ] Flop [ ] Sharpen [ ] Blur [ ] Greyscale [ ] Black & White [ ]

Appendix 8B


Presets

None

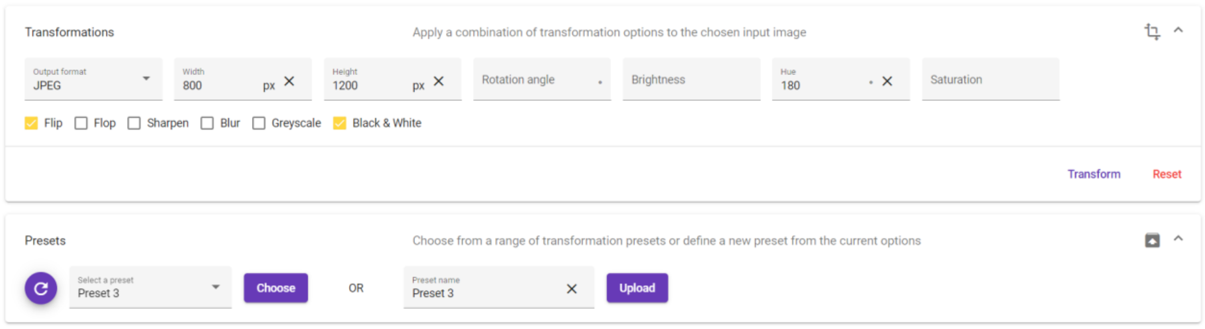
Preset 1

Preset 2

Choose

Input Image

Appendix 9B


Transformations

Apply a combination of transformation options to the chosen input image

Output format: JPEG

Width: 800 px

Height: 1200 px

Rotation angle

Brightness

Hue: 180

Saturation

[✓ Flip] Flop Sharpen Blur Greyscale [✓ Black & White]

Transform Reset

Presets

Choose from a range of transformation presets or define a new preset from the current options

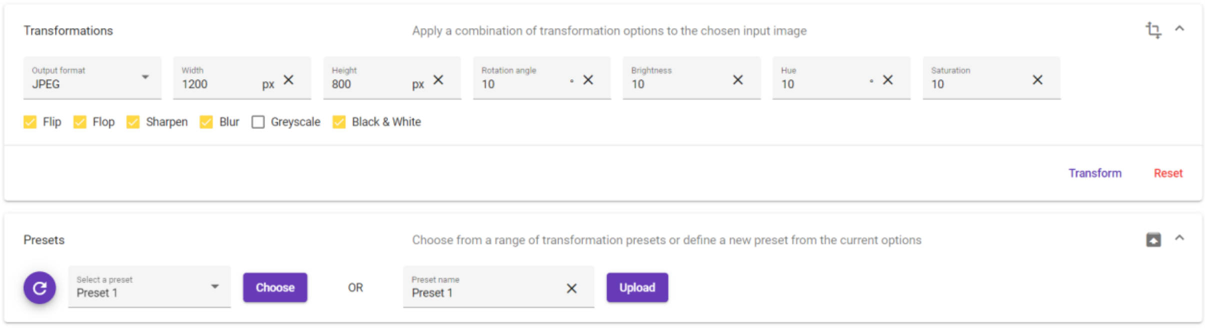
Select a preset: Preset 3

Choose OR

Preset name: Preset 3

Upload

Appendix 10B


Transformations

Apply a combination of transformation options to the chosen input image

Output format: JPEG

Width: 1200 px

Height: 800 px

Rotation angle: 10

Brightness: 10

Hue: 10

Saturation: 10

[✓ Flip] [✓ Flop] [✓ Sharpen] [✓ Blur] Greyscale [✓ Black & White]

Transform Reset

Presets

Choose from a range of transformation presets or define a new preset from the current options

Select a preset: Preset 1

Choose OR

Preset name: Preset 1

Upload

Appendix 11B

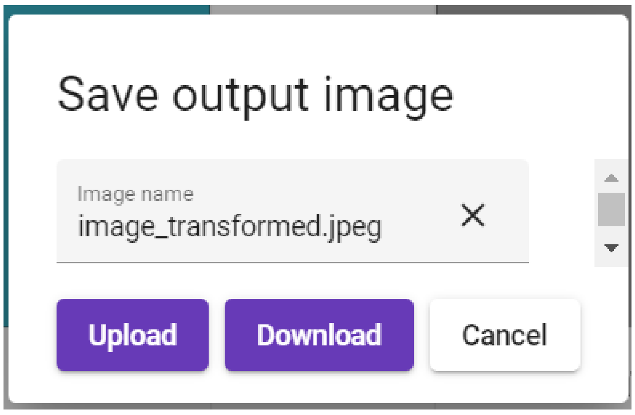
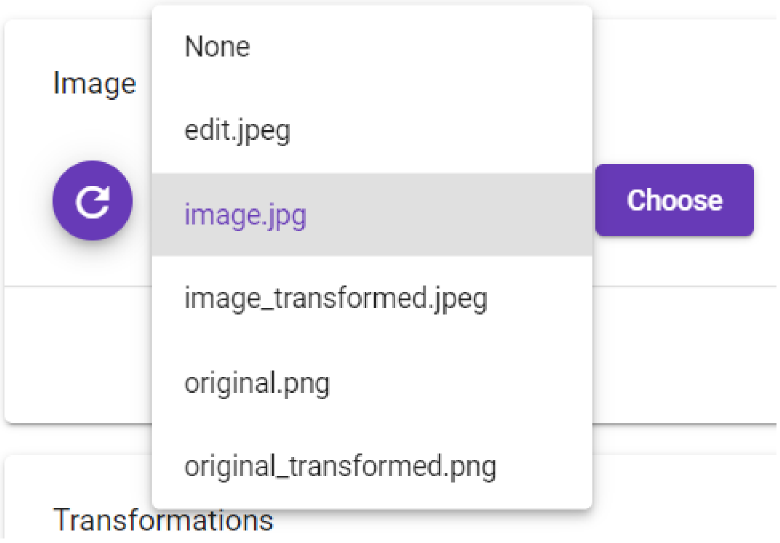

Save output image

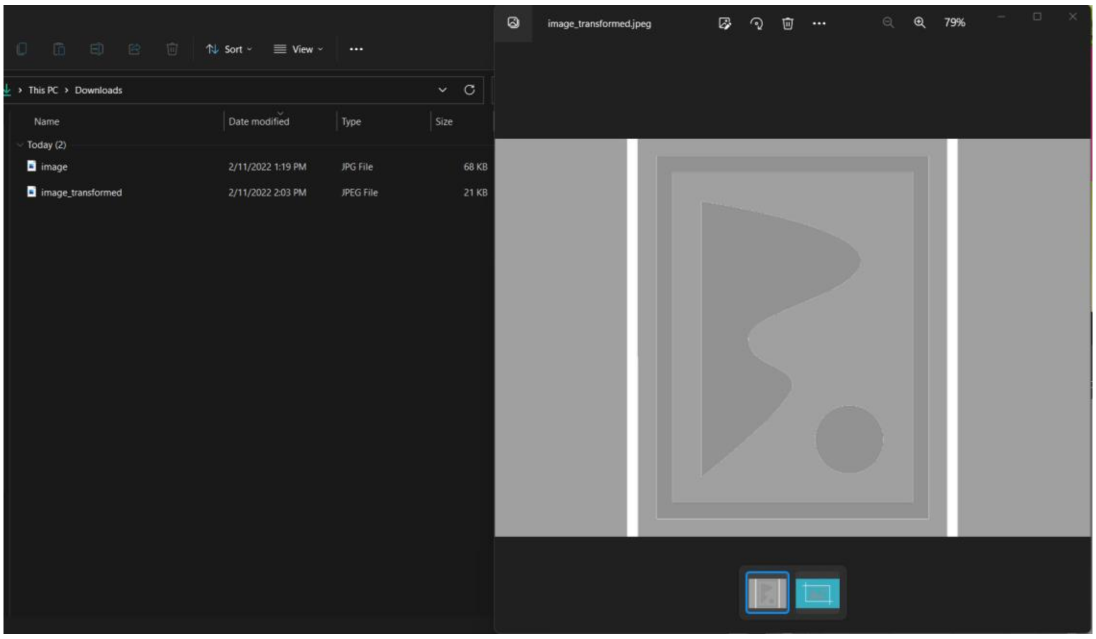
Image name: image_transformed.jpeg

Upload Download Cancel

Appendix 12B

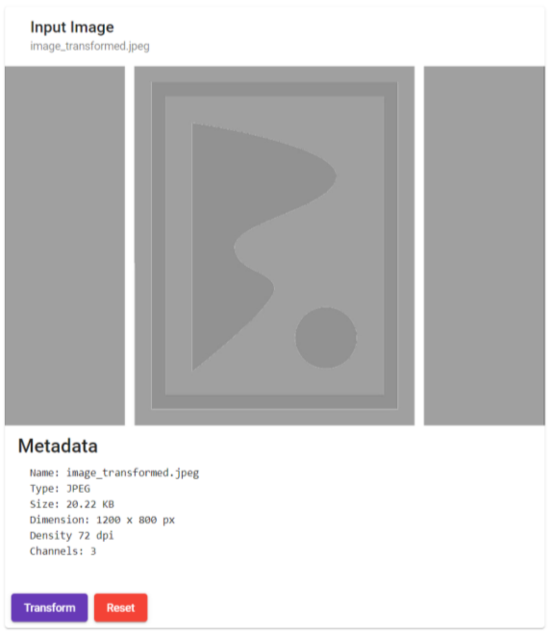


Appendix 13B



```
Name                Date modified        Type        Size
Today (2)
image               2/11/2022 1:19 PM    JPG File    68 KB
image_transformed   2/11/2022 2:03 PM    JPEG File   21 KB
```

Appendix 14B

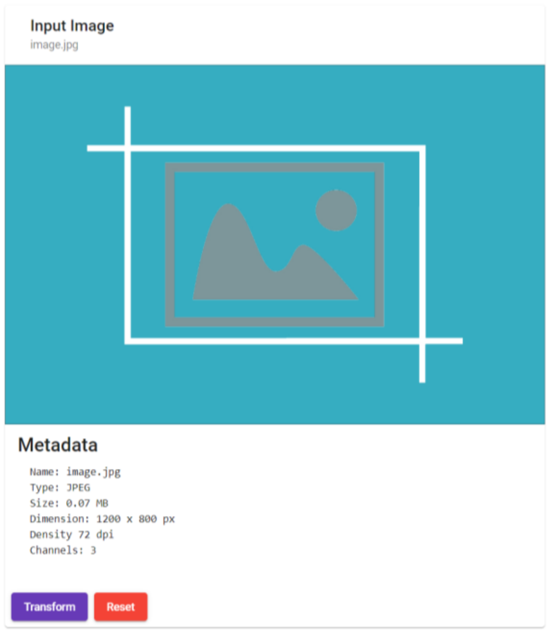


Input Image
image_transformed.jpeg

Metadata

```
Name: image_transformed.jpeg
Type: JPEG
Size: 20.22 KB
Dimension: 1200 x 800 px
Density 72 dpi
Channels: 3
```

Appendix 15B

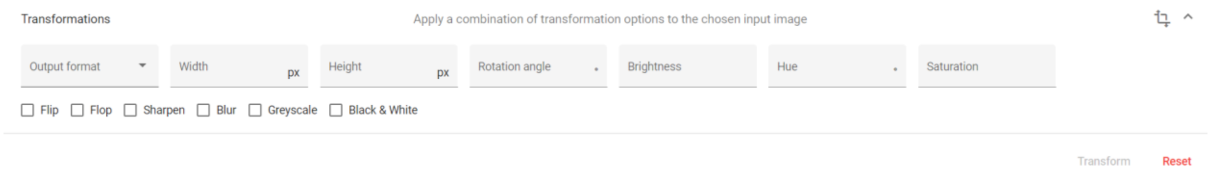


Input Image
image.jpg

Metadata

```
Name: image.jpg
Type: JPEG
Size: 0.07 MB
Dimension: 1200 x 800 px
Density 72 dpi
Channels: 3
```

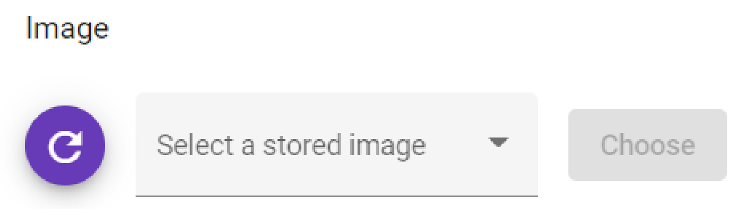
Appendix 16B



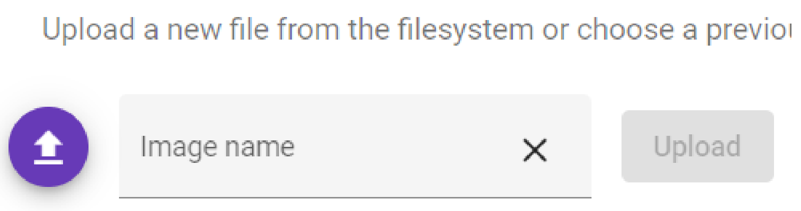
Transformations

Apply a combination of transformation options to the chosen input image

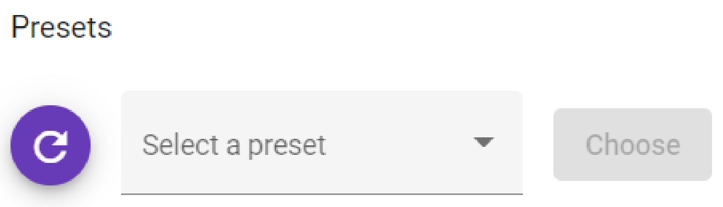
Appendix 17B



Appendix 18B


Upload a new file from the filesystem or choose a previo

Appendix 19B


Presets
Select a preset
Choose