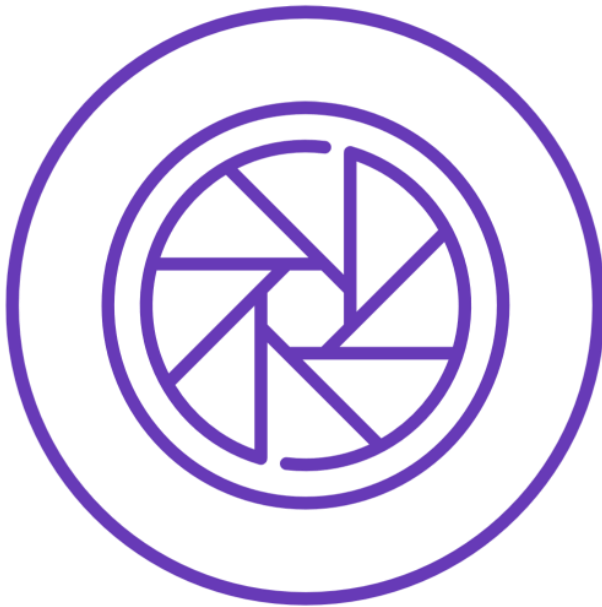


# PicPro



CAB432 Assignment 2

Matthias Eder  
11378093

Eric Zhang  
10319191

3 November 2022

## Contents

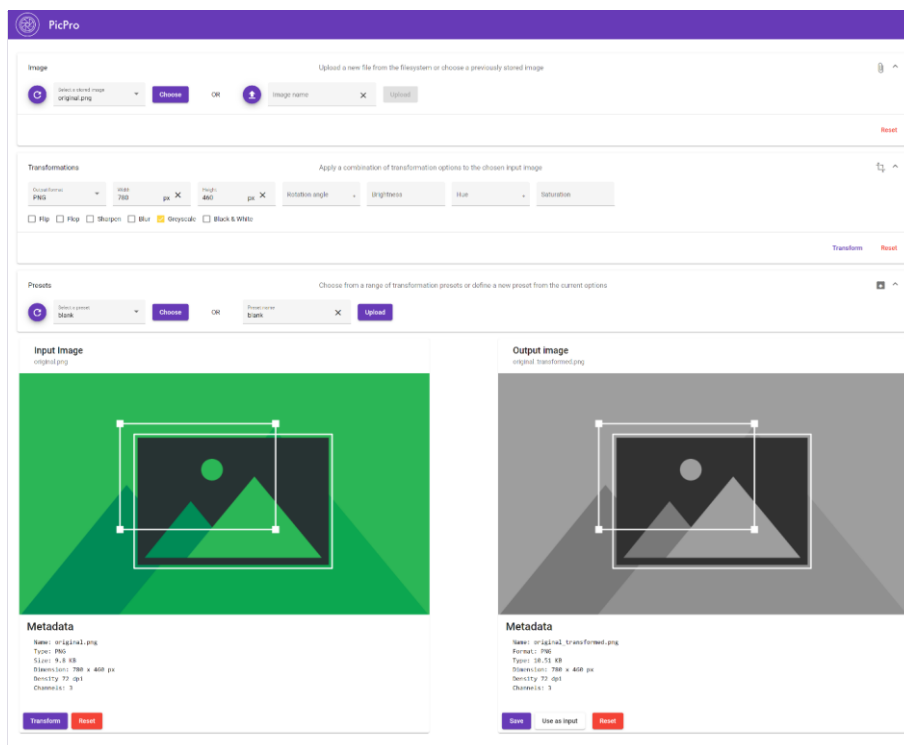
Introduction .....	3
Purpose & description.....	3
Services used.....	3
Sharp Node.js module (v0.31.1).....	3
Use cases.....	3
US 1 .....	3
US 2 .....	4
US 3 .....	4
Technical breakdown .....	4
Architecture .....	4
Context Diagram .....	4
Client / server demarcation of responsibilities.....	5
Response filtering / data object correlation .....	7
Features .....	10
Image.....	10
Transformations.....	10
Presets.....	11
Input/Output.....	11
Scaling and Performance .....	12
Test plan .....	13
Difficulties and Reflexion .....	14
Extensions .....	15
User guide .....	15
References .....	15
Appendices.....	16
Appendix A .....	16
Appendix 1A – Dockerfile Frontend .....	16
Appendix 2A – Dockerfile Backend .....	16
Appendix 3A – docker-compose.yml .....	17
Appendix 4A – pm2 ecosystem.config.js .....	17
Appendix 5A – Image “/” route.....	18
Appendix 6A – Image “/fetch” route .....	19
Appendix 7A – Image “/upload” route .....	20
Appendix 8A – Image “/transform” route .....	21
Appendix 9A – Preset “/” route .....	22

Appendix 10A – Preset “/fetch” route.....	22
Appendix 11A – Preset “/upload” route.....	23
Appendix 12A – Transform function part 1 .....	24
Appendix 13A – Transform function part 2 .....	25
Appendix B .....	25
Appendix 1B.....	25
Appendix 2B.....	26
Appendix 3B.....	27
Appendix 4B.....	27
Appendix 5B.....	27
Appendix 6B.....	28
Appendix 7B.....	28
Appendix 8B.....	29
Appendix 9B.....	29
Appendix 10B.....	29
Appendix 11B.....	29
Appendix 12B.....	30
Appendix 13B.....	30
Appendix 14B.....	31
Appendix 15B.....	31
Appendix 16B.....	31
Appendix 17B.....	32
Appendix 18B.....	32
Appendix 19B.....	32

## 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.



Kommentiert [ME1]: there should be a feature in chrome where you can actually take a screenshot of the whole page, please include that here and also use the updated frontend with the reset buttons and such

Kommentiert [EZ2R1]: New screenshot added

### 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

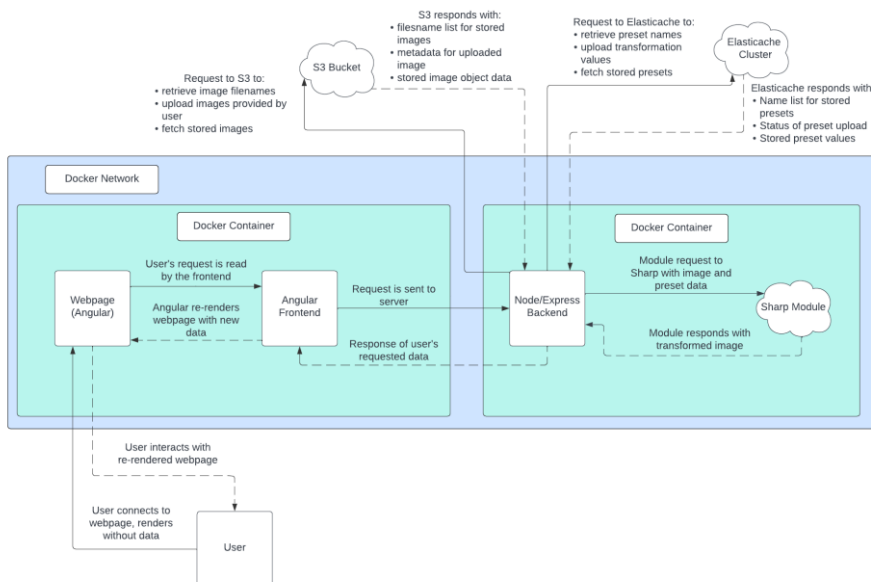
As a	graphic designer
I want	save previous transformation presets/values
So that	I can apply the same transformation to multiple images without having to fill in the values every time

## 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



**Kommentiert [ME3]:** start with a more high-level description. aka the application consists of a frontend and backend, frontend is a single page web application written with Angular, backend an Express app based on Node.js and so on and so forth. they are dockerized SEPARATELY with docker compose, that should really have been clear by now... they are not "contained in docker alongside". and then go deeper into the architecture with persistence services, Sharp, etc.

**Kommentiert [EZ4R3]:** I've tried to rewrite this to be more high-level

**Kommentiert [ME5]:** technically, Sharp is not an API but a Node.js module, the diagram indicates that Sharp is a web API hosted somewhere else while it is actually a module that is executed locally

**Kommentiert [EZ6R5]:** I'll adjust my wording to better reflect this

**Kommentiert [ME7]:** not bad, but could be improved. why is there an Angular instance outside of the "Docker Container" block? also, backend and frontend are in separate Docker containers inside a single Docker network, so this block is technically wrong. also, Sharp should actually be inside the backend container as it is executed there and not an external API (see above comment). where are the persistence services in the data flow/context diagram?

**Kommentiert [EZ8R7]:** Updated context diagram

**Kommentiert [ME9R7]:** as this diagram should show the data flow in our application, I would rather have an obvious distinction of the data sent to S3/ElastiCache e.g. S3 gets requests with image data to store files, retrieve all file names, retrieve a file for a specific image, aka reflect the endpoints in the server that are relevant for S3. and for ElastiCache the same but with Presets. so that the diagram clearly reflects what kind of data flows where. currently, this diagram is nice but very generic

**Kommentiert [EZ10R7]:** The context diagram has been updated

[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 compiled *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 [“images/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 [“images/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

**Kommentiert [ME11]:** at the end of the architecture, describe the dockerization as well and the deployment on AWS. we are running two docker containers there in a docker network and all of it is started up with pm2 like in the prac

**Kommentiert [EZ12R11]:** I have included the docker/aws deployment here

**Kommentiert [ME13R11]:** nice. I actually changed the deployment slightly so that we don't have to rely on pm2. I changed that now

**Kommentiert [ME14]:** check the last paragraph of the frontend README, there I explain what the NGINX webserver actually does. you can largely copy/paste this from there and explain it a bit better

**Kommentiert [EZ15R14]:** Text replaced with README paragraph

**Kommentiert [ME16]:** "demarcation of responsibilities" you are describing what happens to the instances in case of increased load. please do what is asked in this section and describe what the RESPONSIBILITIES of frontend/backend are as this is kinda fuzzy atm. (frontend=GUI, backend=data processing)

**Kommentiert [EZ17R16]:** I might come back to this one, I'm not sure if I've written this correctly, and I don't know if this falls under "short and focused"

**Kommentiert [EZ18R16]:** I'll leave this for the moment, I don't want to trim it down before you review it again

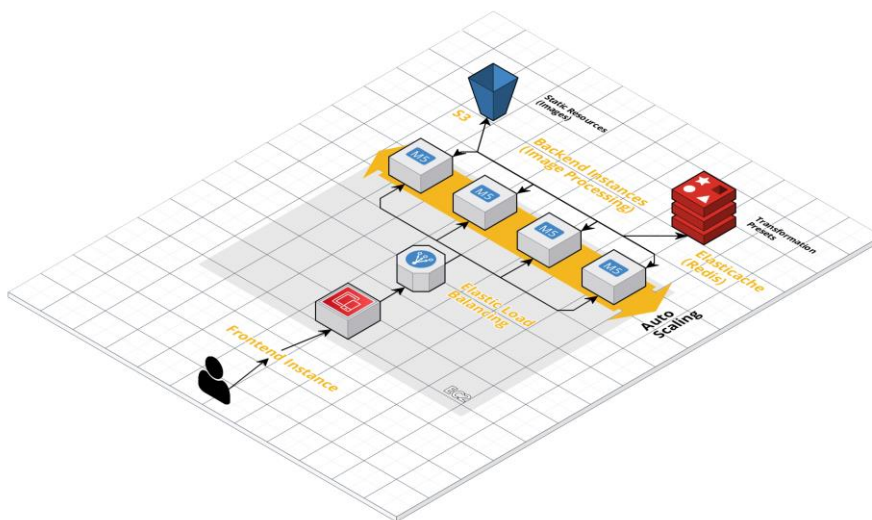
**Kommentiert [ME19R16]:** well done

[“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.

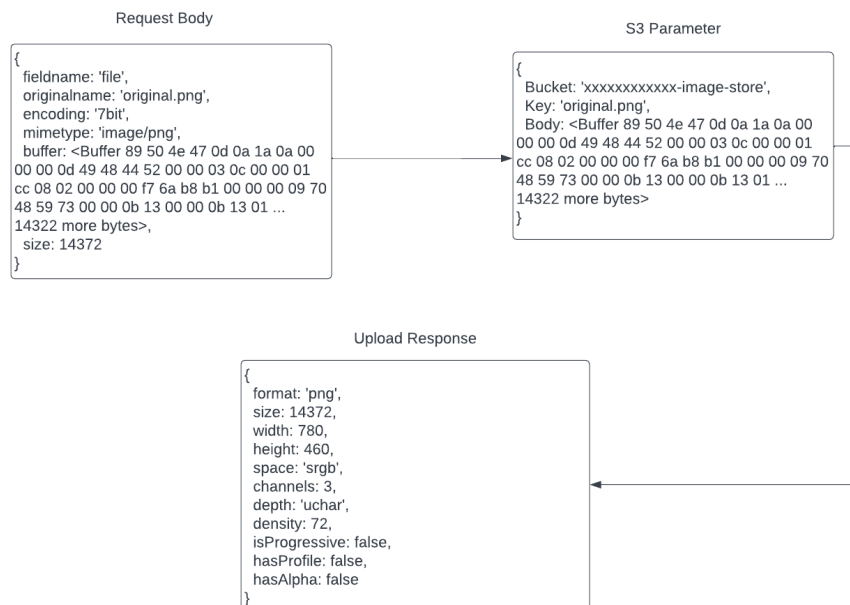


Kommentiert [ME20]: incorrect IMO, there were no differences between our proposed and actual architecture. I removed one of the identical diagrams, please remove the marked section

Kommentiert [EZ21R20]: Section has been removed

### 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.



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 => {
```

**Kommentiert [ME22]:** I feel this could be enhanced by a diagram showing more clearly where what kind of data is being processed and sent/stored. we are dealing with several different types of data, this is not reflected in this paragraph focussing only on the transformation

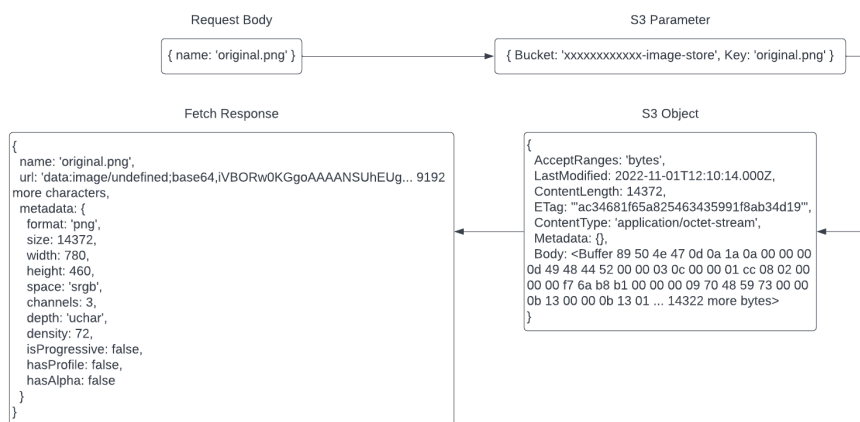
**Kommentiert [EZ23R22]:** I might come back to this, because I'm also not sure the best way to show the data

**Kommentiert [EZ24R22]:** Explained a bit how the frontend handles the response data



[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.



[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.

Request Body

```
{
  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.

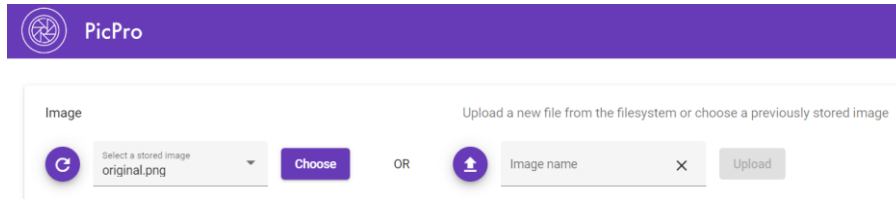
**Kommentiert [ME25]:** I don't think we need to show frontend code in the report, rather just keep the parts about data processing, request data, JSON and stuff from the marked paragraphs and remove frontend stuff, that's not really interesting I think

**Kommentiert [EZ26R25]:** Frontend explanation has been removed

## Features

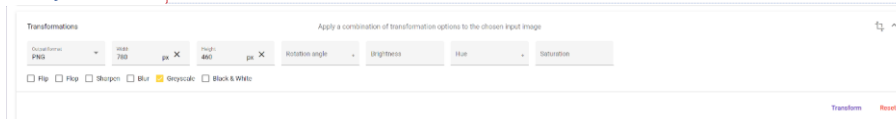
The flow of the data in our project, as well as its architecture, can be best described by the three main features of PicPro: Image, Transformations, and Presets

### Image



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



```
{
  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.

**Kommentiert [ME27]:** maybe just change that sentence to smth like:  
"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, Transformation, and Presets."

**Kommentiert [EZ28R27]:** Thank you, I'll make this change

**Kommentiert [EZ29R27]:** I'm considering shrinking the Features section, because some things talked about here is now mentioned above in the demarcation of responsibilities section

**Kommentiert [ME30R27]:** technically yeah but just leave it as it is now I'd say

**Kommentiert [ME31]:** the architecture of a software project describes its technical design (wrt. backend/frontend, services, hosting, dockerization, etc.), you are describing features here. the architecture is shown in the diagram I made for the proposal

**Kommentiert [EZ32R31]:** This should hopefully be rectified with the above paragraphs

**Kommentiert [ME33]:** please update the screenshot with the current version where the "Transform" and "Reset" buttons are included. you can decide where you want to explain what these buttons do/trigger, either here or down where the images are shown. you are explaining this in the section of data manipulation, please decide where to put this and explain it in a single place

**Kommentiert [EZ34R33]:** I have updated the image, although I worry this screenshot is too small to see

**Kommentiert [ME35]:** this is incorrect. the example JSON shown on the left shows the Preset JSON format, not the Transformation. the latter contains the filename in itself, only the Preset contains the "name" property outside of the "transformation" block. please clarify/rework this in this paragraph and the below one

**Kommentiert [EZ36R35]:** This has been updated to correctly differentiate the transformation and preset JSON

## 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.


**Kommentiert [ME37]:** clarify the difference between Transformation and Preset JSON pls

**Kommentiert [EZ38R37]:** This has been clarified

## Input/Output

Input Image

original.png



Metadata

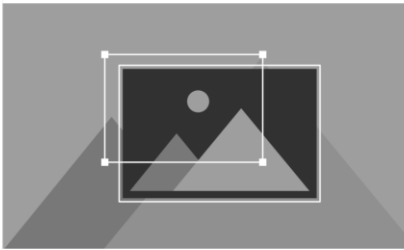
Name: original.png  
Type: PNG  
Size: 9.8 KB  
Dimension: 780 x 460 px  
Density: 72 dpi  
Channels: 3

Transform

Reset

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

Save

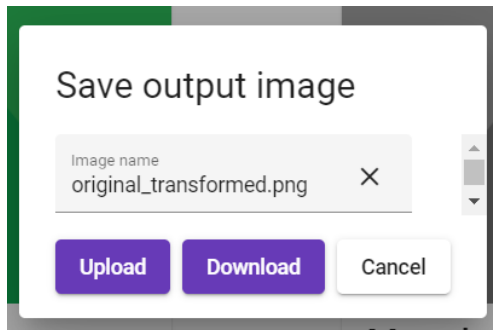
Use as Input

Reset

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.

**Kommentiert [ME39]:** IMO you could add another paragraph below where you more clearly explain what the transformation endpoint in the backend actually does, maybe with code examples

**Kommentiert [EZ40R39]:** I believe I've better explained this above in demarcation and response filtering



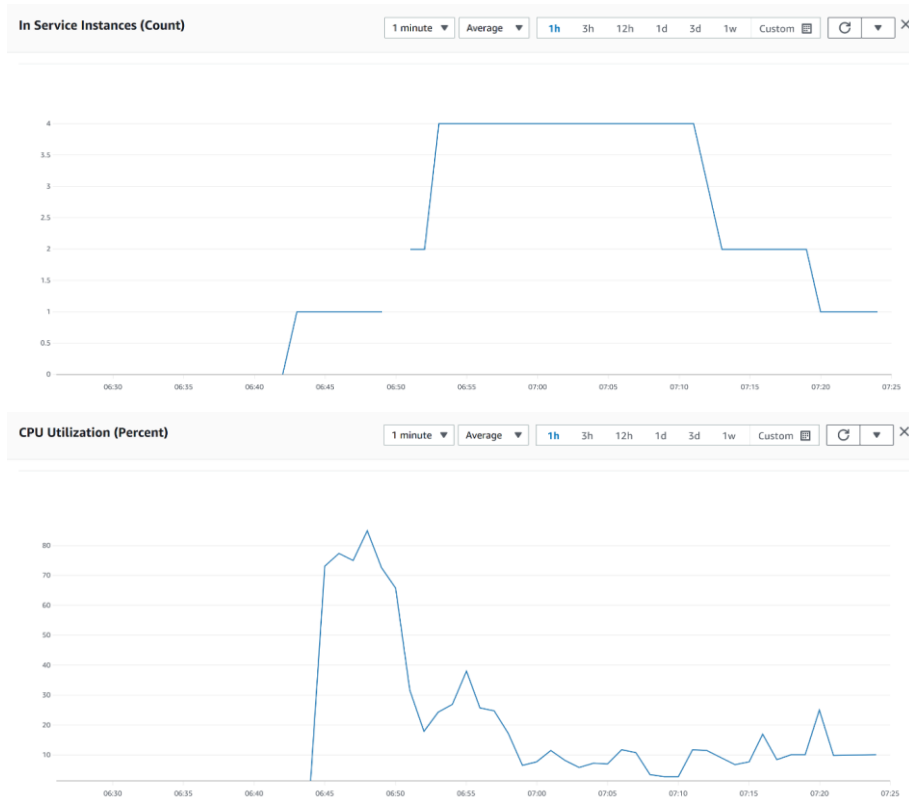
### Scaling and Performance

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 milliseconds, 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 average 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 utilization, 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 screenshots 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.

Kommentiert [ME41]: I'll have to write this part as well...



The cloud setup was configured according to our architecture diagram. A sidenote would be that in the diagram, the LoadBalancer is placed “behind” the frontend 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 to hide complexity as we will only interact with a single frontend instance and focus very much on data processing in the backend.

#### Test plan

Task	Expected Outcome	Result	Screenshot (Appendix B)
Image Select	Drop down displayed currently stored images	PASS	<a href="#">1</a>
Image Upload	Loading bar shows during request, image file displays below	PASS	<a href="#">2</a>
Image Fetch	Loading bar shows during request, stored image displays below	PASS	<a href="#">3</a>
Image Section Reset	Input/Output images are empty again	PASS	<a href="#">4</a>
Transform Values	User input is displayed in transformation fields/checkboxes	PASS	<a href="#">5</a>

Kommentiert [ME42]: you write that part as well pls

Kommentiert [EZ43R42]: Test plan filled out

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

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 they 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/community/tutorials/using-buffers-in-node-js>

Kommentiert [ME44]: I can also write this, will copy/paste the README that I wrote

Kommentiert [ME45]: this is not a proper reference, check the guide and also cite this in the report somewhere

Kommentiert [EZ46R45]: Using below guide to write the reference and make an in-text citation under response filtering section  
<https://www.citewrite.qut.edu.au/cite/qutcite.html#apa-internet-webpage>



## 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
```

**Kommentiert [ME47]:** also maybe include code samples from the connections to the persistence services or maybe also the storing/retrieval of images to/from S3. just include code parts of important parts

**Kommentiert [EZ48R47]:** Some code examples have been included

**Kommentiert [ME49]:** include stuff in appendix. when you describe the dockerization somewhere include the docker-compose.yml or the dockerfiles or smth, or include the transformation endpoint for reference and refer to it in the text wherever you explain the transformation in detail

**Kommentiert [EZ50R49]:** Some code examples have been included

*Appendix 3A – docker-compose.yml*

```
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*

```
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();
}

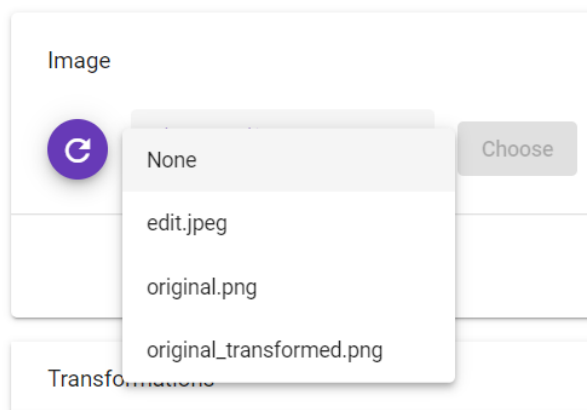
// sharpen
if (transformation.sharpen) {
    transformationPipeline.sharpen();
}

// 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

X

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

Name: image.jpg

Type: JPEG

Size: 0.07 MB

Dimension: 1200 x 800 px


Density 72 dpi

Channels: 3

Transform

Reset

## 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

X

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: 14.4k B  
Dimension: 788 x 468 px  
Density: 72 dpi  
Channels: 3

TransformReset

## 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

600

px

Rotation angle

90

Brightness

2

Hue

180


Saturation

3

☐ Flip ☐ Flip ☐ Sharpen ☐ Blur ☒ Greyscale ☐ Black & White


TransformReset

## Appendix 6B

 PicPro


Image

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

 Select a stored image  
image.jpg

Choose

OR

 Image name

X

Upload

Reset

Transformations

Apply a combination of transformation options to the chosen input image

Output format  
JPEG

Width  
1200  
px X

Height  
800  
px X

Rotation angle  
90  
° X

Brightness

Hue


Saturation

☐ Flip ☐ Flip ☐ Sharpen ☐ Blur ☒ Greyscale ☐ Black & White


Transform Reset

Presets

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

 v

Input image  
image.jpg

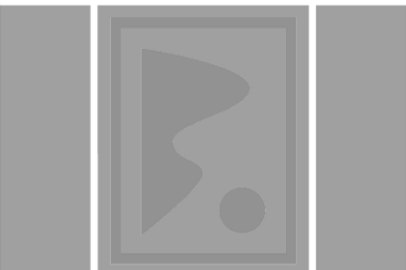


Metadata

Name: image.jpg  
Type: JPEG  
Size: 6.07 KB  
Dimension: 1200 x 800 px  
Density: 72 dpi  
Channels: 3

Transform Reset

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

Save Use as input Reset

## Appendix 7B

Transformations

Apply a combination of transformation options to the chosen input image

Output format  
JPEG

Width  
1200  
px X

Height  
800  
px X

Rotation angle  
°

Brightness

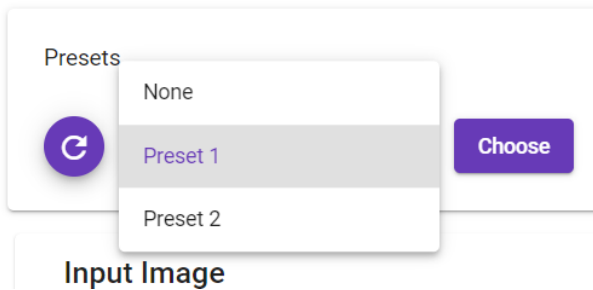
Hue

Saturation

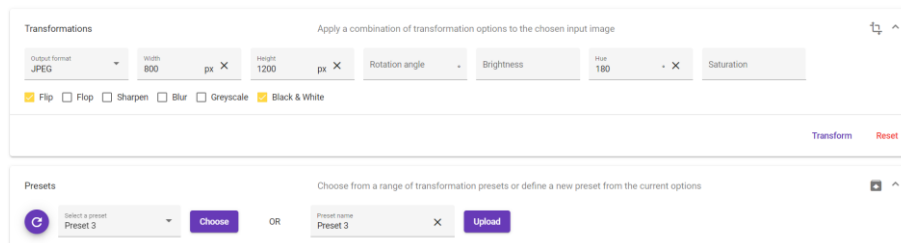
☐ Flip ☐ Flip ☐ Sharpen ☐ Blur ☐ Greyscale ☐ Black & White

Transform Reset

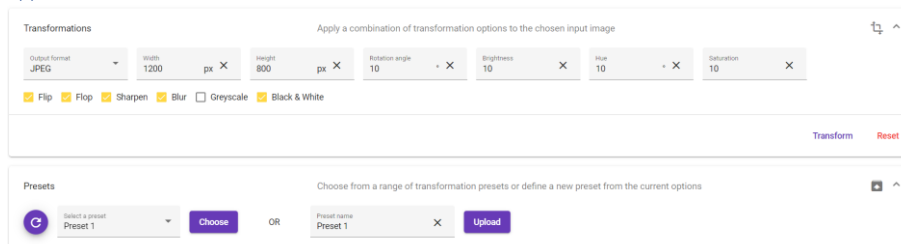
#### Appendix 8B



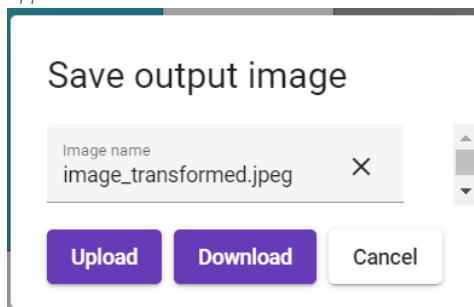
#### Appendix 9B



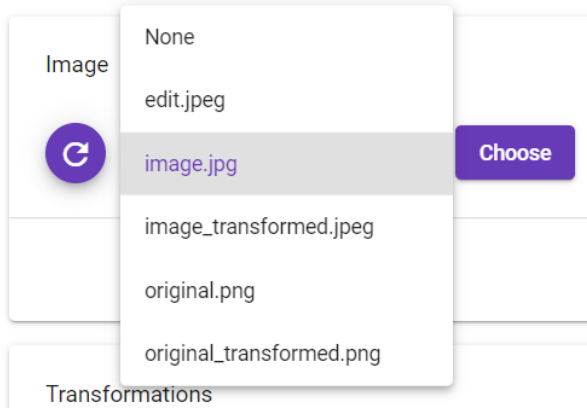
#### Appendix 10B



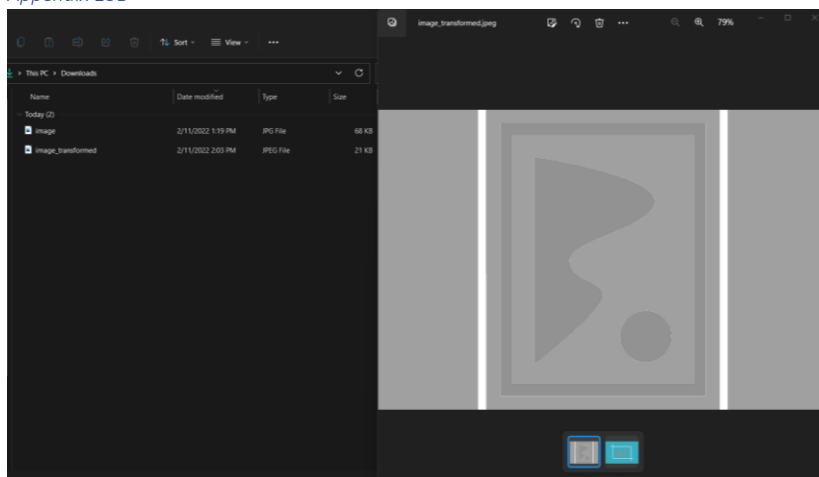
#### Appendix 11B



Appendix 12B



Appendix 13B



### 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

Transform

Reset

### Appendix 15B

Input Image

image.jpg



Metadata

Name: image.jpg

Type: JPEG

Size: 0.07 KB

Dimension: 1200 x 800 px

Density: 72 dpi

Channels: 3

Transform

Reset

### Appendix 16B

Transformations

Apply a combination of transformation options to the chosen input image

Output format

Width

px

Height

px

Rotation angle

Brightness

Hue

Saturation

☐ Flip

☐ Flip

☐ Sharpen

☐ Blur

☐ Greyscale

☐ Black & White


Transform

Reset



Appendix 17B


Image

 Select a stored image ▼

Choose

Appendix 18B


Upload a new file from the filesystem or choose a previous

 Image name ✕

Upload

Appendix 19B

Presets

 Select a preset ▼

Choose