

PicPro

PicPro

CAB432 Assignment 2

Matthias Eder

11378093

Eric Zhang

10319191

23 October 2022

Contents

[Introduction 3](#_Toc118238784)

[Purpose & description 3](#_Toc118238785)

[Services used 3](#_Toc118238786)

[Sharp Node.js module (v0.31.1) 3](#_Toc118238787)

[Use cases 3](#_Toc118238788)

[US 1 3](#_Toc118238789)

[US 2 4](#_Toc118238790)

[US 3 4](#_Toc118238791)

[Technical breakdown 4](#_Toc118238792)

[Architecture 4](#_Toc118238793)

[Context Diagram 4](#_Toc118238794)

[Client / server demarcation of responsibilities 5](#_Toc118238795)

[Architecture Diagram 6](#_Toc118238796)

[Response filtering / data object correlation 7](#_Toc118238797)

[Features 9](#_Toc118238798)

[Scaling and Performance 11](#_Toc118238799)

[Test plan 12](#_Toc118238800)

[Difficulties / Exclusions / unresolved & persistent errors 13](#_Toc118238801)

[Extensions (Optional) 13](#_Toc118238802)

[User guide 13](#_Toc118238803)

[References 13](#_Toc118238804)

[Appendices 14](#_Toc118238805)

[Appendix 1 – Dockerfile Frontend 14](#_Toc118238806)

[Appendix 2 – Dockerfile Backend 15](#_Toc118238807)

[Appendix 3 – docker-compose.yml 15](#_Toc118238808)

[Appendix 4 – pm2 ecosystem.config.js 16](#_Toc118238809)

[Appendix 5 – Image “/” route 16](#_Toc118238810)

[Appendix 6 – Image “/fetch” route 17](#_Toc118238811)

[Appendix 7 – Image “/upload” route 18](#_Toc118238812)

[Appendix 8 – Image “/transform” route 19](#_Toc118238813)

[Appendix 9 – Preset “/” route 20](#_Toc118238814)

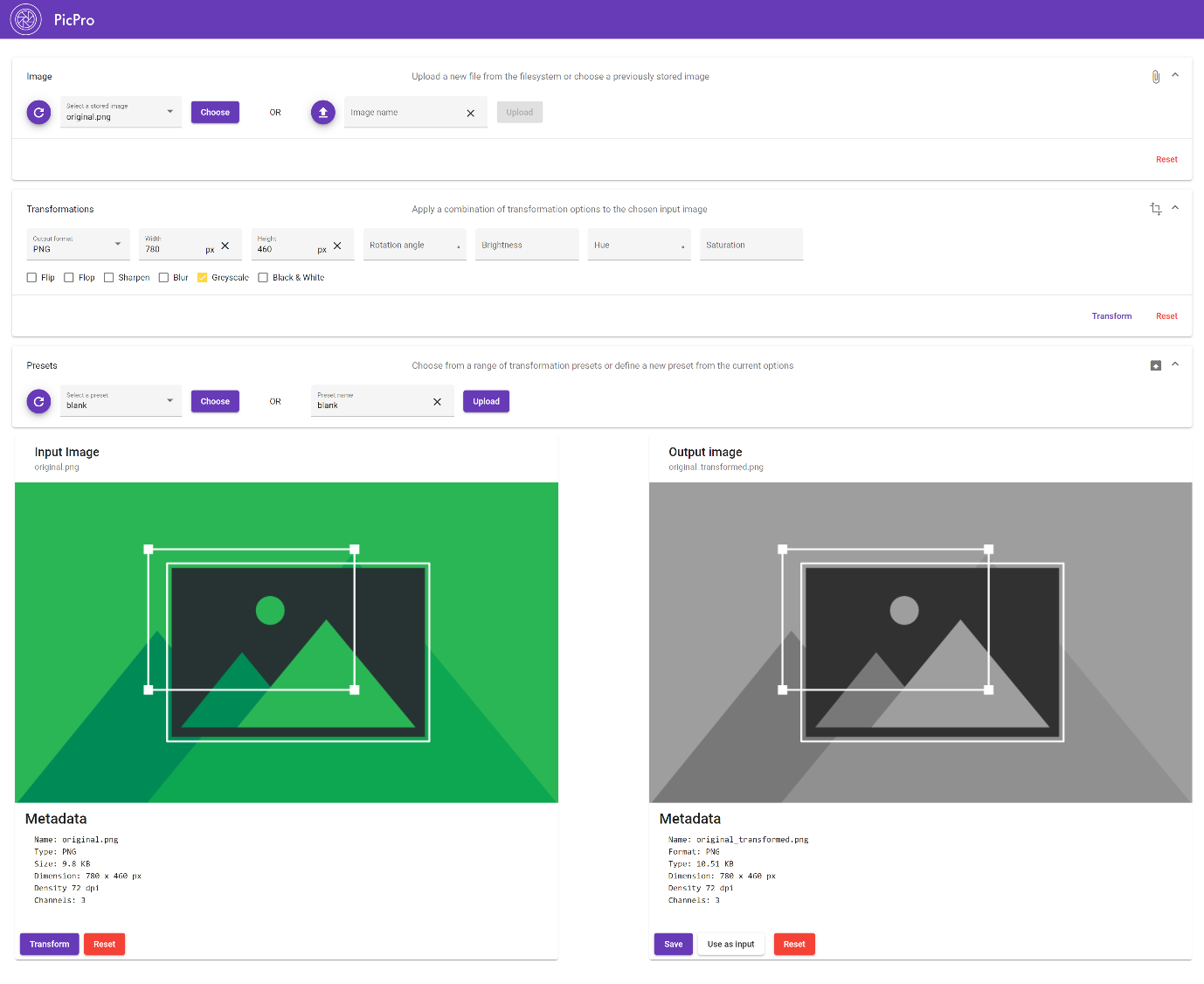
[Appendix 10 – Preset “/fetch” route 20](#_Toc118238815)

[Appendix 11 – Preset “/upload” route 21](#_Toc118238816)

## 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.



### 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

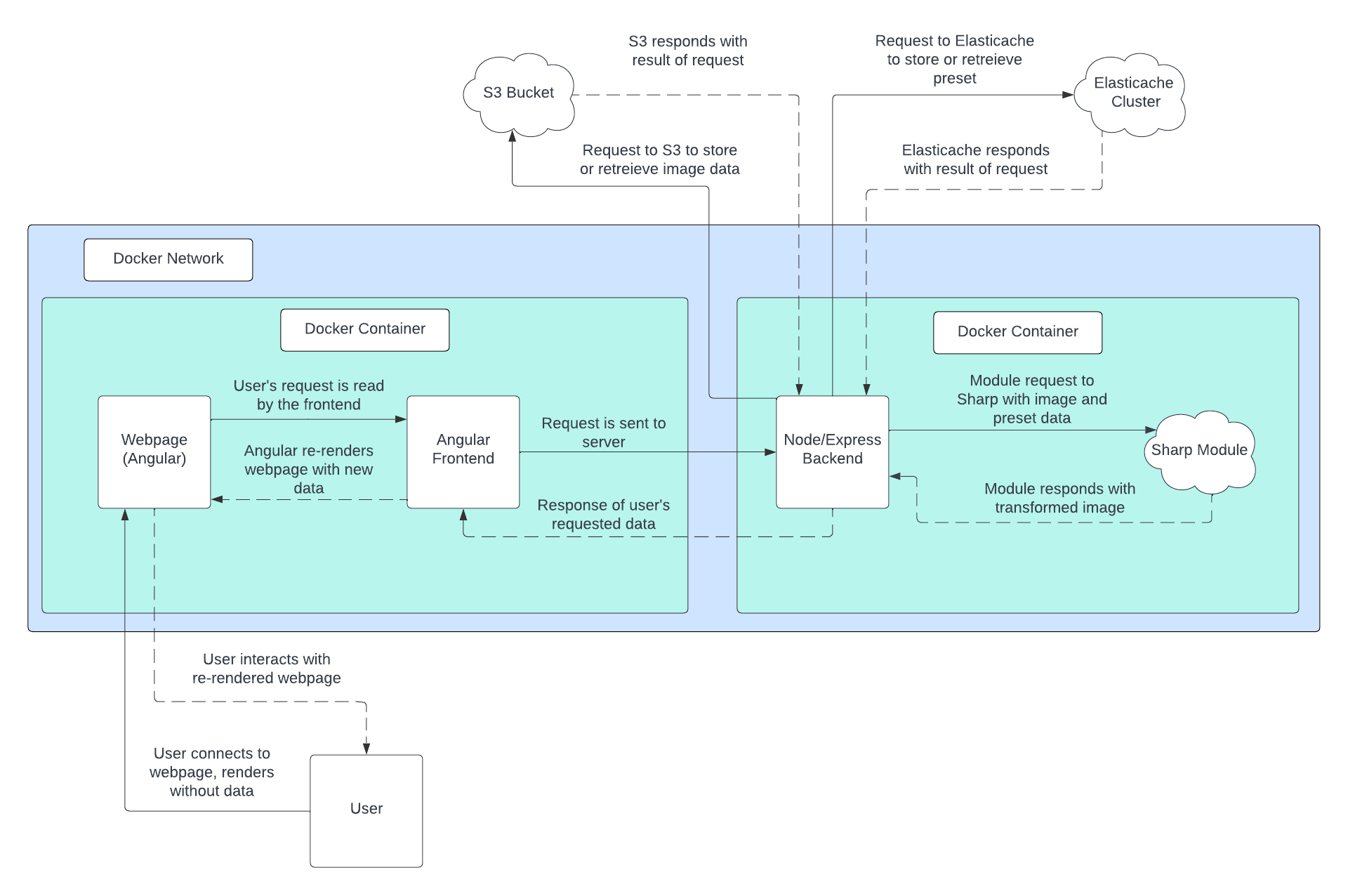
|  |  |
| --- | --- |
| As a | graphic designer |
| I want | Save previous transformation presets/values |
| So that | I can apply the same transformation to multiple images without having to fill in the values every time |

## Technical breakdown

### Architecture

The flow of data can be seen in the context diagram below. The application consists of a 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 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 get sent to, making the required request of 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.

#### Context Diagram



These containers are built separately with docker compose, the docker file layouts shown in [Appendix 1-3](#_Appendix_1_–). The backend Docker file sets the work directory in 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 frontend Docker file follows similar steps to build the container, and then removes the default nginx website, which is the web service used for this application. The docker copies the dist folder from the build stage to the public nginx folder, while also copying the necessary ngnix config file that will be used. This gets exposed to port 80. These two Docker files are then by the docker compose file to be tagged for the pushing to the repository and exposed to the appropriate ports, 3000:3000 for the backend and 4200:80 for the frontend.

The way the application is deployed to AWS is via pm2, similarly to the example from the scaling prac. The pm2 configuration file (ecosystem.config.js) shown in [Appendix 4](#_Appendix_4_–) runs the scripts for the 3 sections of the application: The docker network, the frontend container, and the backend container. This is used to run the docker application within the EC2 instance, allowing it to be started when the instance boots up for ease of scaling.

#### 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 that are currently being displayed, 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 categories as the image and preset routes. The Appendix code has been linked below.

The image routes handle the requests from Image and Transformation sections of the webpage, and there are 4 of them. The base route [“/”](#_Appendix_5_–) is a GET request that retrieves the filenames of stored S3 images so users can choose an image to fetch. The [“/fetch”](#_Appendix_6_–) and [“/upload”](#_Appendix_7_–) 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 [“/transform”](#_Appendix_8_–) 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.

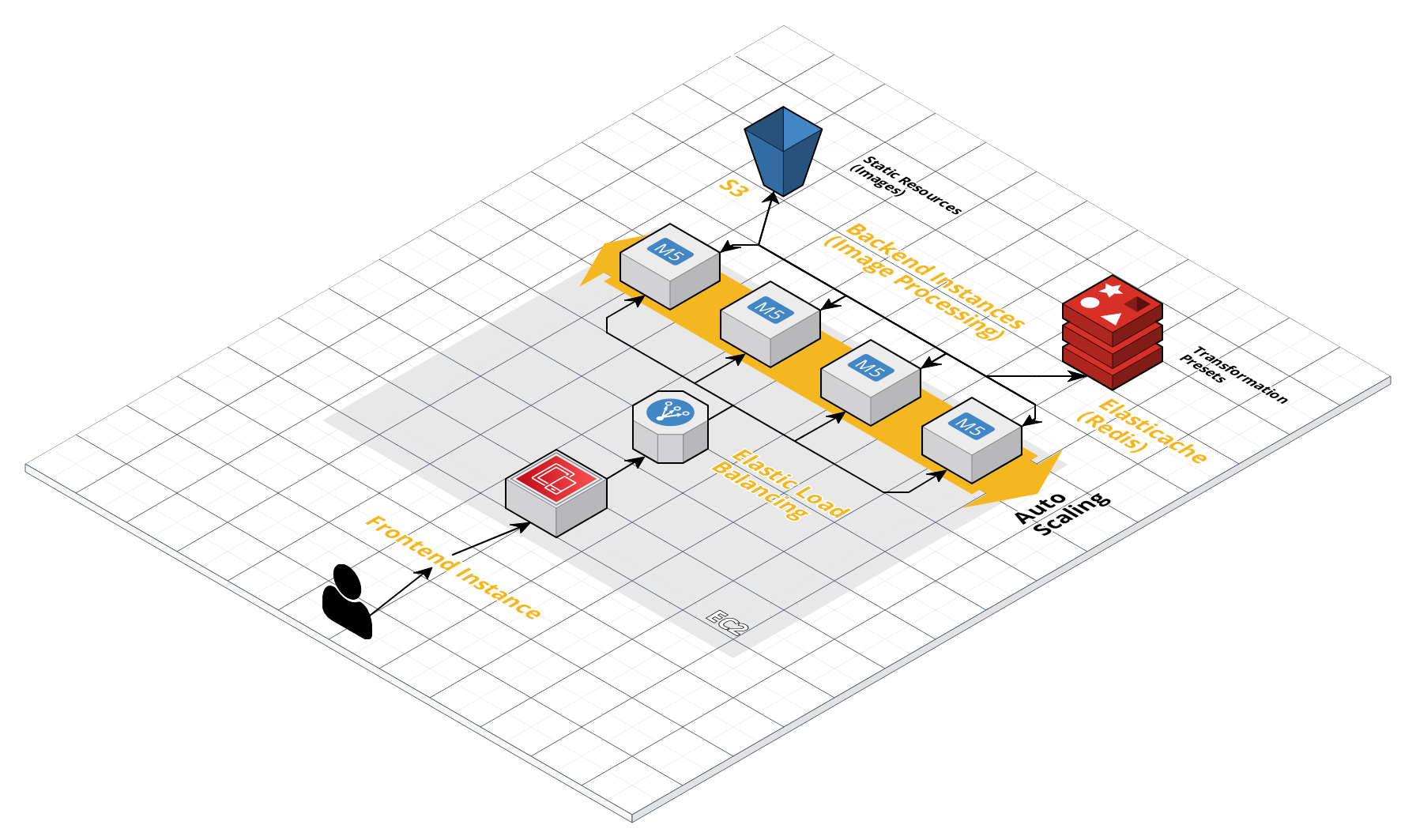
The preset routes handle the requests from the Preset section of the webpage, and there are 3 of them. These do very similar to the image routes, and all of them interact with the Elasticache Cluster via Redis: [“/”](#_Appendix_9_–) responds with the preset names that users can choose from, [“/fetch”](#_Appendix_10_–) retrieves the chosen preset that the frontend can use to fill in the Transformation section, and [“/upload”](#_Appendix_11_–) 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 traffic 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 image in quick succession. S3 can scale alongside the data stored in it, making it very suitable for large amounts of images.

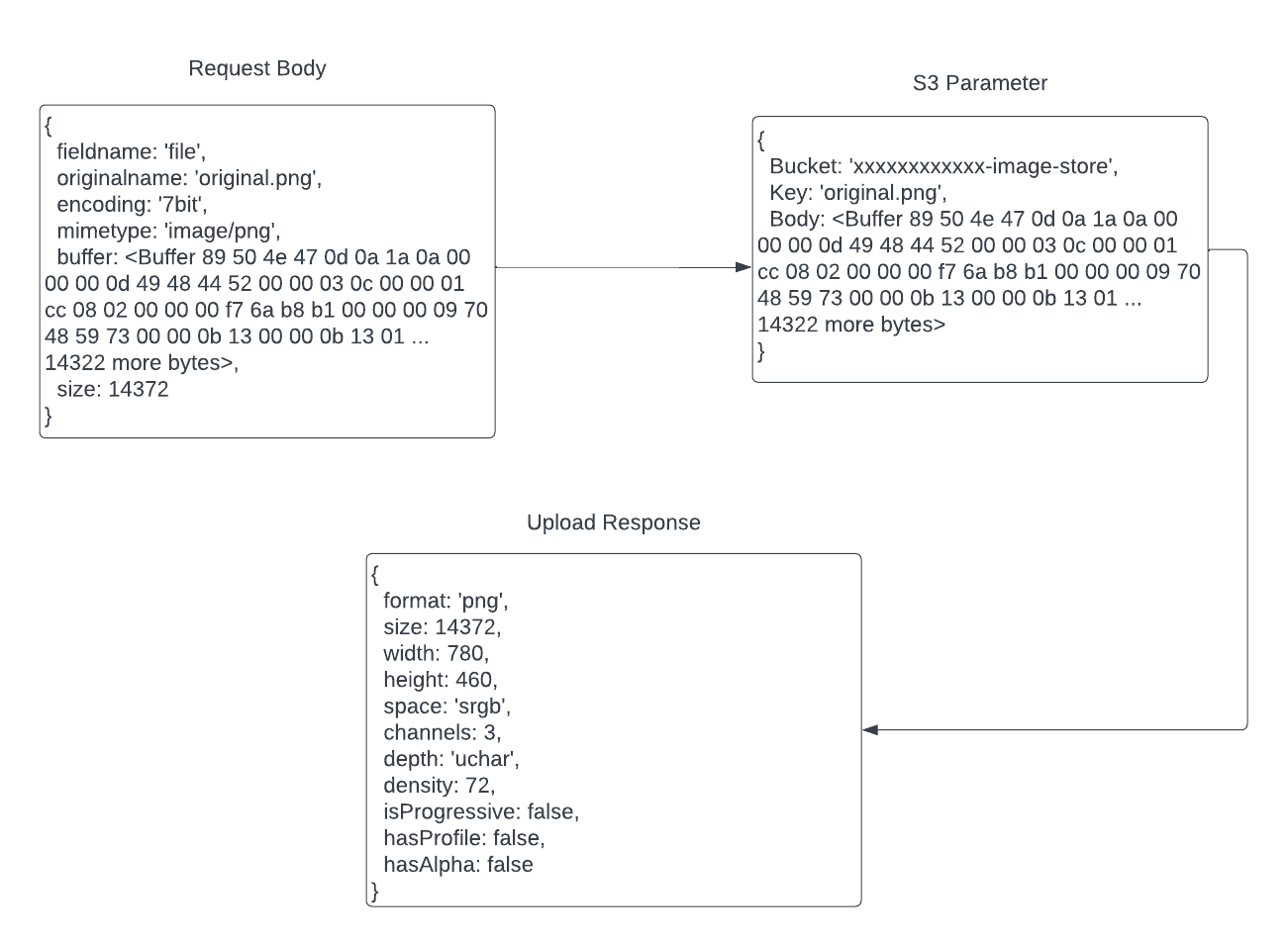
Elasticache is a memory-based cache and was chosen to store the presets that contain the transformation values due to its speed. As these presets are stored as JSON objects, they do not require the large flexibility in storage scaling that S3 boasts, and need to be accessed quickly, which Elasticache 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 transformation.

#### Architecture Diagram



#### Response filtering / data object correlation

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



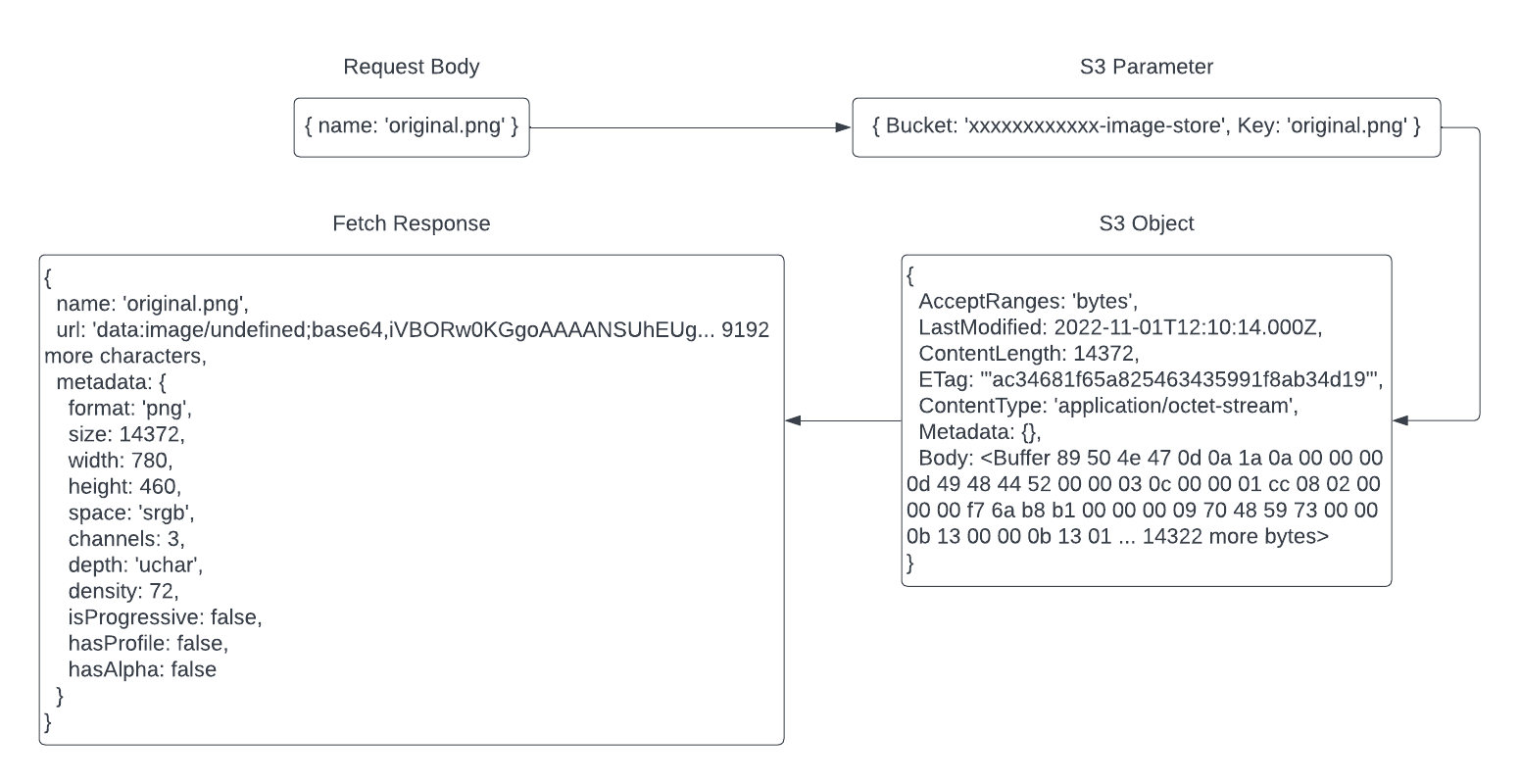
Buffer is used due to many file readers like the Sharp module often manipulating files as Buffer data, as they are suitable for fine-grained data manipulation, along with its ease of conversion to other data types like a base64 URL, used to re-render the images to the webpage so they can be displayed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

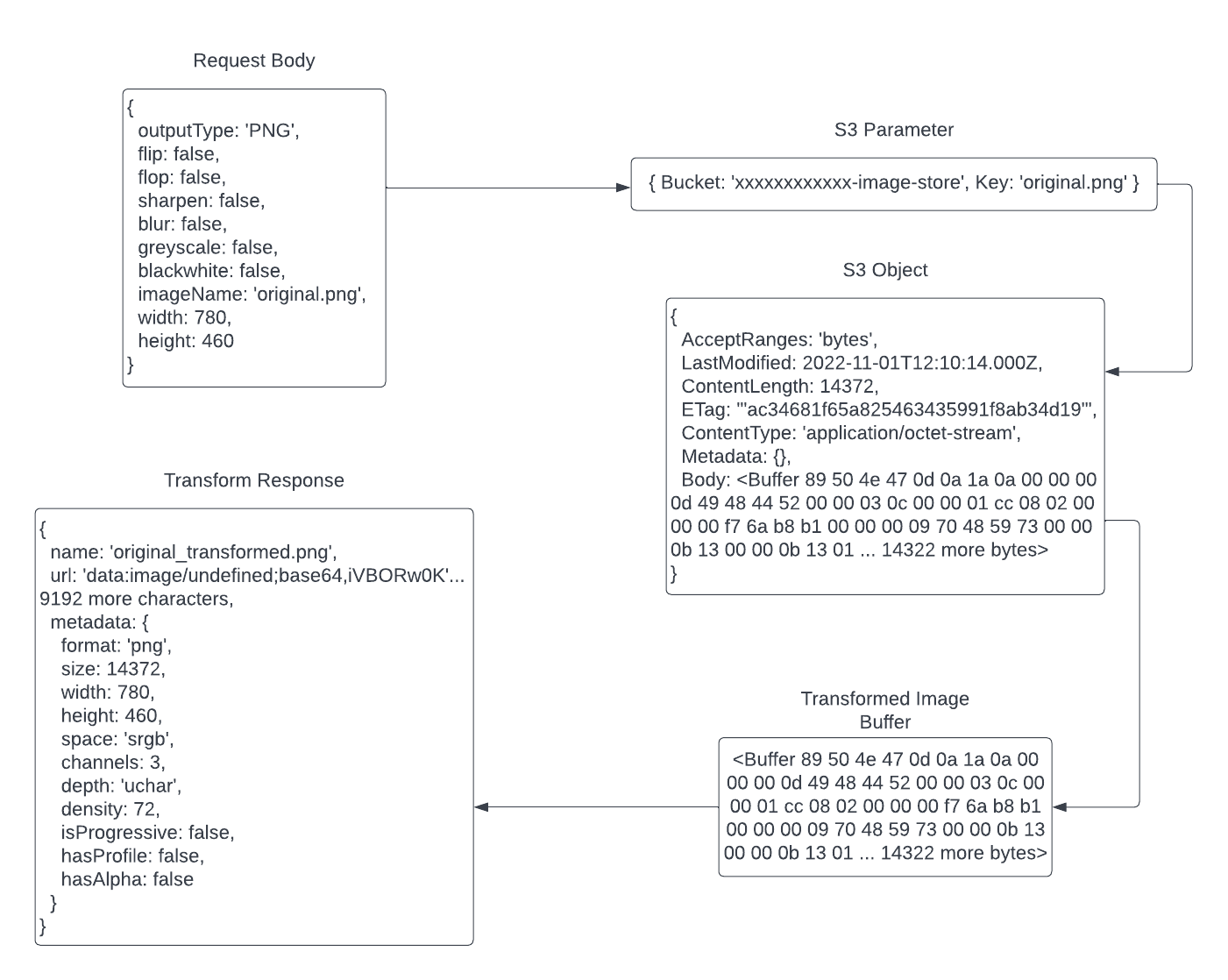


Fetch can be seen as going backwards from upload path. The original request is the filename of the image the user chose to retrieve. This filename is placed within a parameter JSON for S3, again containing the bucket name, which is used to retrieve the data from S3. The object that is returned hold various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

As metadata is blank, the route again uses Sharp to obtain the metadata, which gets placed in the fetch response, also containing the filename and the image buffer which has been converted to a base64 URL.



Transformation is very similar to fetch. Using the image name to obtain the desired image from S3, the request body containing the transformation values are sent to the transformation function, that calls sharp to make the transformations uses several if-statements to check through the request and apply the necessary transformations only if values are supplied.



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. The response is the same as previously shown in the fetch diagram, returning the new name, converted image URL and metadata obtained from Sharp.



Presets stay largely the same, a JSON object containing the transformation value that looks like the request shown in the transformation section. This will be explained more thoroughly in the Features section.

## Features

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

Graphical user interface, application

Description automatically generated

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 in an S3 bucket as a Buffer object. 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 URI 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.

Graphical user interface, application

Description automatically generated

The **Transformation** 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 “transformation” 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.

{

outputType: ‘PNG’,

flip: false,

flop: false,

sharpen: true,

blur: true,

greyscale: false,

blackwhite: false,

imageName: ‘original.png’

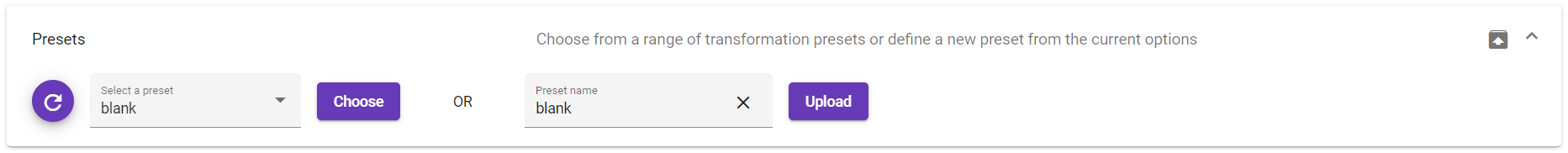
width: 5,

height: 5,

rotationAngle: 5,

brightness: 5

}



The **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.

{

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

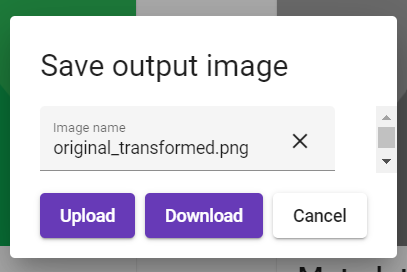
}

}

Graphical user interface, application

Description automatically generated

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.



### Scaling and Performance

*This is a crucial aspect of the report, and you should use this section to document the approach taken to scaling – the nature of the application load, how it was varied and how the scaling infrastructure responded. You should refer to the architectural diagram above or reproduce the relevant aspects here. You should include screenshots of CPU, network or queuing metrics as observed on the cloud services dashboard, together with screenshots of your settings and the scaling pool instance creation and destruction. We expect that your work here will demonstrate successful scale out and scale in as required in the assignment specification. The screen shots that you use here will also very likely be re-purposed as part of the slide deck for the demo.*

*An example scaling image is shown below, and we would normally expect to see this sort of image and some evidence of your group settings. Note the instance count on the left hand axis. As noted, many alternatives are possible.*



### Test plan

*Manual testing is fine and our expectations are in line with the example grid below. You can show the results through a screen shot and point us to these from the table. Your tests should include:*

* *Positive outcome cases*
* *Negative outcome cases (error scenarios)*
* *Edge cases*
* *Non-functional cases*

*Note that the grid below is unrelated to this assignment.*



*As they are common in industry you could define your Acceptance Criteria as GWT statements. This is not compulsory, but see:* [*https://www.agilealliance.org/glossary/gwt/*](https://www.agilealliance.org/glossary/gwt/)*. And here is an example:*



Difficulties / Exclusions / unresolved & persistent errors /

*In this section, you should explain anything that caused you problems and how you overcame those problems. Tell us if there was any issue that prevented you completing the assignment to specification. Tell us about any assumptions or compromises that you have made. Those who worked with an API like Spotify, which presented particular concerns, should discuss the compromises here, and this is also where you can tell us about problems with API keys and responses.*

*More generally, you might consider:*

* *Your major roadblocks and how you resolved them.*
* *Any functionality you didn’t or couldn’t finish*
* *Are there any differences between your brief and what you delivered? If so, explain why.*
* *Are there any outstanding bugs?*

## Extensions (Optional)

*In this section, you can tell us if you wish to how you might extend your app and make it better. This is an opportunity to tell us about good ideas that you had that you didn’t have time to tell us about.*

## User guide

*Tell us how to use your application. You may re-use some of the screenshots from the use case descriptions, but this is more about how to use the app. As long as we can find what we need to do to use your application, this need not be all that long.*

*But either way, screenshots are your friend.*

## References

*Use a standard approach to referencing – see the guidance at* [*https://www.citewrite.qut.edu.au/cite/*](https://www.citewrite.qut.edu.au/cite/)*.*

Stack Abuse. (2020). Using Buffers in Node.js. Digital Ocean.

<https://www.digitalocean.com/community/tutorials/using-buffers-in-node-js>

## Appendices

*Stuff you want to include, but is too long or too complex to include in the main report text. The full Docker file, some longer excerpt from API docs. Whatever helps.*

*[Our thanks to those students who allowed us to use their work in the examples presented above.]*

#### Appendix 1 – Dockerfile Frontend



#### Appendix 2 – Dockerfile Backend



#### Appendix 3 – docker-compose.yml



#### Appendix 4 – pm2 ecosystem.config.js



#### Appendix 5 – Image “/” route



#### Appendix 6 – Image “/fetch” route



#### Appendix 7 – Image “/upload” route



#### Appendix 8 – Image “/transform” route



#### Appendix 9 – Preset “/” route



#### Appendix 10 – Preset “/fetch” route



#### Appendix 11 – Preset “/upload” route

