NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TOMÁS EMANUEL TABORDA MENDES DA SILVA

Bachelor in Computer Science

WEB-BASED APPLICATION FOR ASSISTED ELIMINATION OF DUPLICATE PHOTOGRAPHS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

September, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

WEB-BASED APPLICATION FOR ASSISTED ELIMINATION OF DUPLICATE PHOTOGRAPHS

TOMÁS EMANUEL TABORDA MENDES DA SILVA

Bachelor in Computer Science

Adviser: Fernando P. R. Birra
Assistant Professor, NOVA University Lisbon

Co-adviser: João M. S. Lourenço
Associate Professor, NOVA University Lisbon

Examination Committee:

Chair: Vitor A. Duarte
Assistant Professor, NOVA University Lisbon

Rapporteur: Pedro M. F. Centieiro
Magycal

Member: Fernando P. R. Birra
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2021

Web-based Application for Assisted Elimination of Duplicate Photographs

For my family and friends, who gave me life and made it worth living.

Acknowledgements

First, I would like to give my thanks to what has been my academic home for the vast majority of the past 6 years, the *Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa* and, more specifically, its Department of Computer Science. There, I experienced several new and different facets of life, all of which were instrumental in me becoming the person that I am today, and for that I will forever be grateful.

Secondly, I would like to express my gratitude to my advisors in this work, Professors João Lourenço and Fernando Birra. Their help and advice was invaluable for my success, and I could not have asked for better guidance in this project. My time with them was an exquisite mixture of mentorship, humour, camaraderie, and motivation to keep me going. I will carry with me for the rest of my life many of the lessons that they taught me. For all you've done for me in this past year of work, you have my deepest thanks.

On a more personal note, I want to first thank my family. Throughout my academic journey, they have given me nothing short of unwavering support and guidance whenever I needed it, as well as a bottomless and incredibly warm love which kept me going through many tiring and seemingly unending nights of work. Thank you so, so much.

Not all family is blood however, and thus I must mention my closest group of friends, who have long since crossed that distinct border. I realize that description is somewhat vague and pretentious, but I cannot seem to find a specific word to identify our group that would match the serious tone that this dissertation will (hopefully) have. Y'all know who you are. You have kept me sane for the duration of this work, while I was beset on all sides by pandemics, lockdowns, and just general life situations that repeatedly tried to (and sometimes succeeded in) knocking me down. I have nothing short of the utmost gratitude for all that you have done for me.

Last (but definitely not least) I want to thank my Buddy, Ânia. For the past 4 years, you have been my companion through this degree and life in general. We've worked together, we've lived together, we travelled together to distant lands. We've shared laughter, hardship, and many scholarly and somewhat productive all-nighters. Memories I hope to never lose. I cannot thank you enough.

Thank you all, truly, for everything. I hope not to disappoint you moving forward.

Abstract

The high availability of digital cameras, both on SLR and in cell phones, has caused the proliferation of digital photographs. When it comes to photography, it is a common practice for those who do not have high technical knowledge to take several pictures of the same motif, hoping that at least one of them looks good enough, and later sorting through them to delete those that are not interesting. This sorting process has the potential to be extremely time-consuming and exhausting, leading a lot of photographers to postpone this process and simply leave all the pictures in storage and unsorted.

This leads to a lot of wasted storage space, as several of the pictures that are kept are essentially lower quality or less interesting repetitions of some other image. It also might cause the users to avoid looking through their photo library, due to the repetition of motifs making the albums themselves less engaging and their inspection more tiresome.

In this work, we developed a web-based application for desktop systems that simplifies the process of selecting which pictures in a photo gallery shall be kept and which ones shall be deleted. This is achieved by aggregating photographs of the same subject in a similarity group, and then sorting the images contained in these groups based on their technical quality. Through this process, the user will be able to identify the best pictures on a technical level for each similarity group, and use this knowledge in conjunction with subjective opinion/taste to choose which ones shall persist and which ones shall be removed from storage.

As a result, it is expected that the analysis and removal of similar images will become much less time-consuming and more enjoyable for the stakeholders, this way solving the presented problem while at the same time improving their photographer experience in general.

Keywords: Digital Photography, Photography Library Management, Web-based Application, Similar Image Detection, Picture Technical Quality Assessment

Resumo

A alta disponibilidade de câmaras digitais, tanto no formato SLR como em telemóveis, causou a proliferação da prática da fotografia digital. No mundo fotográfico, é uma técnica comum para quem não tem conhecimentos técnicos tirar várias fotografias do mesmo tema, esperando que pelo menos uma dela esteja boa o suficiente, e mais tarde analisar as fotos tiradas para apagar aquelas que não são interessantes. Este processo de classificação tem o potencial de ser extremamente moroso e cansativo, desta forma causando com que muitos fotógrafos adiem o seu começo, e simplesmente armazenem todas as fotografias sem as classificarem.

Esta situação causa um mau aproveitamento do espaço de armazenamento, visto que muitas das fotografias guardadas são essencialmente repetições menos interessantes ou com menor qualidade de alguma outra imagem. Pode também fazer com que os utilizadores evitem examinar a sua biblioteca de fotografias, devido à repetição de temas fazer os álbuns menos cativantes e a sua análise mais cansativa.

Neste trabalho desenvolvemos uma aplicação *web-based* para sistemas *desktop* que simplifica o processo de selecionar que fotografias numa galeria devem ser mantidas ou apagadas. Isto é realizado através da organização de imagens com o mesmo tema num grupo de semelhança, seguida da ordenação das imagens contidas nestes grupos com base na sua qualidade técnica. Através deste processo, o utilizador será capaz de identificar quais as melhores fotografias em termos técnicos para todos os grupos de semelhança, e tomar este conhecimento em conta em conjunto com a sua opinião subjetiva e gosto para escolher quais imagens devem persistir e quais devem ser removidas do armazenamento.

Como resultado disto, é esperado que a análise e remoção de imagens semelhantes se torne menos demorada e mais agradável para os fotógrafos, resolvendo assim o problema apresentado e, ao mesmo tempo, melhorando no geral a sua experiência de fotógrafo.

Palavras-chave: Fotografia Digital, Gestão de Biblioteca de Fotografias, Aplicação Web-based, Deteção de Imagens Semelhantes, Análise da Qualidade Técnica de Fotografias

Contents

List of Figures

List of Tables

Glossary

Axure RP 9 Axure RP 9 [6] is a prototyping software developed by Axure that can be used to create an application prototype using a simple and fast drag-and-drop interface and without any need for coding.

Base64 Base64 [7] is the name given to a set of binary-to-text encoding schemes that can be used to represent binary data in an ASCII string format.

npm Node package manager or npm is the world's largest online software registry, used to store packages that can be integrated into a node project to increase its functionalities and capabilities.

WebAssembly WebAssembly [91] is a low level assembly-like language that has near-native performance and allows for web and web-based applications to access compiled versions of software developed in programming languages which would normally be unavailable in a web development context, such as C, C++, Rust, among others.

Acronyms

AGAST Adaptive and Generic Corner Detection Based on the Accelerated Segment Test
API Application Programming Interface

BRIEF Binary Robust Independent Elementary Features
BRISK Binary Robust Invariant Scalable Keypoints

CSAT Customer Satisfaction Score

DOM Document Object Model

FAST Features From Accelerated Segment Test
FLANN Fast Library for Approximate Nearest Neighbors

GB Gigabyte

JSX JavaScript XML

KB Kilobyte

MB Megabyte
MP Megapixel
MPA Multi-Page Application
ms millisecond
MVC Model-View-Controller

NPS Net Promoter Score

ORB Oriented FAST and Rotated BRIEF

QA Quality Assurance

SEO Search Engine Optimization
SIFT Scale Invariant Feature Transform
SPA Single Page Application
SUS System Usability Scale

UI User Interface
UX User Experience

XSS Cross-Site-Scripting

1

Introduction

1.1 Context

Photography is the name given to the process of capturing electromagnetic radiation (such as, for instance, light), and reproducing it in a way that can be later seen, analysed and appreciated.

In recent years, through the rise of digital photography and the proliferation of cheap cellphones with access to cameras, this process has become extremely common-place, with an increasingly overwhelming amount of the population attaining the ability to produce a virtually unlimited amount of photographs for a very low cost. This has incentivized the birth of several social networks and platforms that are used to display these pictures in order to show them to friends and acquaintances, or even potentially gain fame through them.

1.2 Underlying Problem

With the increasing interest and availability of digital photography, more and more amateur users are being drawn to this practice. Many times, these amateurs do not have a complete grasp of the several techniques necessary to take high quality photographs, and therefore go more for an approach of quantity over quality, taking several pictures of each motif, and later going through the pictures to select which ones are the best.

For the purposes of this selection, several criteria can be used to determine which of several photographs of the same motif is the best. While some of these criteria are objective, others are subjective. In a subjective sense, it is not particularly hard to decide which picture is the best, since it mostly relies on the photographer's personal taste and emotional attachment to the motif. However, in an objective sense, several technical points can be taken into account, such as motion blur, focus, white balance, among others. It is of course possible to use a mix of subjective and objective criteria in order to choose which picture to keep, but even then it is necessary to consider the aforementioned technical points.

In order to decide on the best picture, photographers usually have to perform a visual comparison of the candidate pictures. To this effect, they usually try to place the pictures side by side (or, in cases where the screen is not big enough to display both pictures, switch back and forth between the pictures several times) and decide which one looks better.

A problem arises when considering the screen that is used to display the pictures that are being compared: in today's world, it is common for the camera resolution to be much higher than the display resolution. For instance, while there exist phones that have cameras of 12 Megapixels (MPs), the displays of these phones are nowhere close to this resolution (an example of this is the iPhone 11, which has dual 12MPs cameras, but has a display with only 1792x828 pixels, which translates to about 1.5MPs [30]). Even when it comes to the display resolution of computers and televisions, these 12MP pictures will probably not be displayed with their full quality (for instance, the display of a 13 inch MacBook has about 4.1MPs [39], and even 4k televisions usually only get up to about 8.5MPs [75]).

This means that the image the photographer sees on screen does not necessarily show the exact quality of the picture, since the image has to be scaled down before being displayed (using some algorithm defined by the device in question), which might have the effect of hiding certain imperfections that could only be seen in full resolution. This way, in order to visually grasp the actual calibre of the image in a device whose display has a resolution lower than the image, the photographer is forced to manually zoom on specific parts of the picture. When comparing two pictures, the photographer has to repeat this process of zooming in and out in both of the pictures, possibly multiple times, and also possibly switching between them if the screen is not big enough to display the pictures side by side.

This is complicated, impractical, and might not even work as a proper comparison technique, since it relies on the capacity of the photographer to accurately collect and memorize visual information from the images, and being able to process this information mentally when comparing them.

1.3 Approach

To address this problem of selecting the best photograph from a set of quasi-identical ones, we propose the development of a web-based (and desktop-focused) version of the photo|uniq application that allows its user to determine which is the best picture on a technical sense from a grouping of several photographs of the same motif.

When using the application, the photographer is able to upload their photos, which are then aggregated into similarity groups (with each group containing the photographs that are considered to be similar among themselves). To help the user do this, the application has an automatic grouping function that is able to identify the images capturing the same motif and aggregate them into similarity groups. The application also provides

the user with the capacity to both edit the outcome of the automatic grouping and to manually create and populate groups with the photos that they consider to be similar.

After the creation of the similarity groups, the next step is for the user to identify which pictures should be kept (if any) from each group. For these purposes, the application provides tools that can help the user with the analysis of the quality of similar pictures. These tools include a sorting function that would automatically sort the photographs contained in a given group by objectively measuring some of their features such as focus and white balance, as well as side-by-side picture comparison with synchronized zoom and pan capabilities that automatically display the same motif on both of the shown images.

Through the use of these tools and functionalities, the user is then able to easily and quickly make an informed decision on which pictures should be kept from the initial similarity group, which consequently helps them remove the remaining uninteresting repeated photographs from their photo gallery.

1.4 Expected Results and Contributions

With this work, we hope to provide the users with a web-based application that provides the means to better and more efficiently identify which pictures among the several ones that they have taken are worth keeping, allowing them to make a more productive use of their storage space and time.

Several steps must be taken in order to design and implement one such application, including:

- Definition of the main functionalities and workflow of the proposed web-based application;
- Design and development of the user interface of the application;
- Definition of the architecture of the application;
- Development and implementation of the web-based application that provides its users with the capabilities needed to solve the presented problem;
- Testing and validation of the developed application, in order to ensure its correctness and stability;
- Evaluation of the developed app with respect to its main objective, i.e., providing its users with the capacity to better and more efficiently identify which pictures among the several ones that they have taken are worth keeping.

1.5 Document Structure

This document is composed of five chapters.

This **first chapter** serves as an introduction to the theme of the dissertation, explaining the context behind its theme, presenting the problem that this work aims to solve, and how it was addressed.

Then, the **second chapter** elaborates upon the technologies and concepts which will be relevant for the coming work. We first start by presenting the photo|uniq application concept as it was defined and developed in the context of previous dissertations [21, 43]. We then discuss other existing solutions that try to tackle the presented problem. Following this, we move on to an explanation of the relevant concepts and technologies related to web-based development which were used in this work. We then end this chapter by identifying the relevant concepts of image analysis that were used for the development of the web-based photo|uniq application.

The **third chapter** provides a summary of the developed web-based application and its development process. For these purposes, we will begin by introducing the application architecture and its data model, followed by a presentation of its User Interface (UI). After this, we will present and discuss the results of the development process of the application itself, with some extra focus being drawn to the application's key functionalities. Then, the chapter finishes off by presenting some external packages and libraries that were used to enhance the application or help it deliver its promised functionalities.

Following this, the **fourth chapter** provides a view into the testing and validation of the developed application, first discussing the validation of the application's functional aspects, namely, if it functions stably and correctly. Then we present the tests and metrics used to verify the correctness of the application's non-functional aspects, which deal with matters such as the application's usability and its customer satisfaction, and how the application's image analysis algorithms measure up to the needs of its users. We then finish up with a presentation and discussion of the tests used to validate the application's performance. These three sections follow a similar structure, in which we first provide a definition of the tests and metrics that were defined to validate the application, followed by the results of the performed tests, which will then be analysed and discussed.

Then, the **fifth chapter** finishes off the document by providing some final considerations, starting with the conclusions that were reached through the development of this work, and then a presentation of the next steps that should be taken for its continuation, including some possible functionalities that could be added to the developed application in order to further increase its usefulness and value.

2

Related Concepts and Technologies

2.1 Photo|uniq

As mentioned previously in this document, the proposed approach to solving the presented problem is based in the implementation of a web-based version of the photo|uniq application.

Photo|uniq is an application concept that has been the focus of two previous works whose goal was to develop an android version of this application [21] and a back-end server that could handle its computation, authentication and storage necessities [43].

The main goal of photo|uniq is to help its users identify similar photos from within their photo library and, following this, help them with the technical analysis of these similar photos, in order to identify the best pictures on a technical level. With this information, the user can then decide which photos should be kept or deleted, and use this information to help reduce their photo library to a purer and less redundant state.

The purpose of this section is to introduce and explain this application concept by elaborating on its basic workflow and functionalities.

2.1.1 Application Workflow

The photo|uniq application workflow can be summarized as:

1. The user uploads a set of photos containing similar images to the application;

2. The photo|uniq application aggregates the photos into similarity groups, where each group only contains similar photos;

3. The best photos on a technical level from within a similarity group are identified by using the functionalities provided by the photo|uniq application;

4. The user then selects which photos from those contained in the similarity groups should be kept and which should be deleted, by using a combination of subjective taste and the knowledge that was obtained regarding the images' technical quality.

2.1.2 Application Functionalities

The photo|uniq application provides its users with several useful functionalities to deliver on each step of the described workflow. Among these, special relevance can be given to those that are referent to the second and third steps of the workflow: the grouping and technical analysis of similar photos.

We will now explain the tools provided by the photo|uniq application in order to handle these two steps.

Grouping Similar Images

The photo|uniq application provides the user with two ways of aggregating the images into similarity groups.

The most basic way of doing this involves the manual creation and population of similarity groups by the user themselves. Here, the user is given tools to create empty similarity groups and move photos into or out of these groups.

The other option provided by the application involves the analysis and comparison of the uploaded images in order to automatically identify which ones are similar among themselves, followed by the creation of groups that are then populated using these images. This is done through the identification of the image keypoints for every image that is being considered for grouping, followed by the correlation of these keypoints with the keypoints that were detected from every other image, to try to see if they match.

The images are then grouped with those other images with which they have the most matching keypoints, if there was any image with enough matching keypoints to be considered potentially similar. If, however, it happens that one image does not contain enough matching keypoints with any other image to be considered similar to it, then that image will be simply put alone in a similarity group.

The processes of identifying image keypoints and descriptors, as well as the correlation of these elements between two different images, is further described in Section B.1.

Technical Quality Analysis

When it comes to the comparison and grading of similar images based on their level of technical quality, the photo|uniq application also provides some automatic means to reach this end, as well as some tools that can be used to assist the user while they perform a manual comparison.

The process of automatically analysing the technical quality of the similar images is provided to the users through a function that they can access while viewing the contents of a similarity group. When the user calls this function, every image contained in the similarity group is analysed using some quality metric defined by the application (e.g., level of focus, white balance, and/or contrast), which returns a numeric value representing the

quality of the image in question. Then, the images in the group are sorted by comparing the calculated numeric values, and ordering the pictures based on their quality.

The application also provides the user with tools that can assist them and simplify the process of manually comparing two similar images. Here, the application allows for a side-by-side comparison of two similar images with synchronized zoom and panning capabilities that provide a synchronization of the motif shown on both images.

This essentially means that while comparing the two images, changes in zoom level are applied to both images, and then while panning on one image, the shown coordinates of the second image are changed so that the motif shown on both images is the same. This is done through the calculation of two homography matrixes that are used to translate the coordinates of a point that is displayed in one image into a point that displays the same motif on the other. Using this tool, the user can better compare two images by zooming in on specific motifs to show how well they are displayed, which in turn compensates for the scaling down that is performed to fit the images in the device's screen.

The user can also manually sort the images contained in the group, and can therefore change the image order to represent their conclusions that were obtained through the images' manual comparison.

The processes associated with using a homography matrix and measuring image quality are further described in Sections B.2 and B.3 respectively.

2.2 Competing Solutions

An important step to take before the implementation of the web-based photo|uniq application was to analyse how other competing solutions try to solve the proposed problem: which functionalities they provide to the user, and how they can be used. Through this process, we can better identify their strengths and weaknesses, and determine what will be the additional value of our solution.

As mentioned in the previous chapter, the main problem that the proposed application seeks to solve is the proliferation of similar pictures of the same motif. For this purpose, the photo|uniq application provides several functionalities that can be used to:

- Automatically identify similar pictures and aggregate them into similarity groups;
- Manually change the contents of these groups (for instance, if the user does not agree with the groupings performed by the application);
- Automatically sort the identified similar pictures based on their technical quality;
- Allow the user to manually compare and sort similar pictures;
- Choose the pictures from among the ones contained in the similarity group that should remain in storage (and the ones that should not).

Then, if possible, the decisions made regarding the provided photographs should be permeated back to the source device they were initially uploaded from.

It would also be desirable to allow the users to continue their work on the application even while they are offline, as long as they do not require any additional data from an external source (for instance, the server that is used to create the similarity groups or judge the quality of the pictures). Another interesting functionality would be to allow the photographs to be uploaded from several different sources, such as the computer running the application, Google Photos, among other options.

In this section, we will summarize the findings of our analysis of how the identified competitors to the proposed application deal with some of the key steps of the photo|uniq application's workflow, in particular image upload, similar image identification, image quality grading, image comparison, and clearing the user's storage space, as well as how they handle offline work. A more detailed and thorough analysis of these competitors and functionalities is presented in Appendix A.

By looking at the analysed competing solutions, it is clear that there are several different approaches to these functionalities, some of which are superior to others. However, one extremely important functionality was not found in any of these competitors: the ability to objectively grade the technical quality of the several different identified similar pictures, in order to help the user choose which pictures should be deleted, and which should be kept in storage.

It can also be said that some of the functionalities that were implemented by these competitors were not as complete as they should be to better help the user fulfill their needs. The best example of this type of limited functionality can be found when it comes to image comparison: while all the competitors have a solution to handle this necessity, most of them do not deal with the need to zoom in on the picture to observe its sections at their true resolution (since, as mentioned before in this document, the display resolution will often not be able to show a single image in its full quality, let alone two).

The one application that handles this in some way (digiKam [16]) tries to help the user through the use of synchronized zoom and pan, making it so that any changes made to the visualization of one of the pictures will also be replicated on the other one. While this is a step in the right direction, it still does not deal with the possibility of similar pictures of the same motif displaying this motif on different image coordinates (for instance, two pictures of the same building, one of which presents the building in the centre of the image, while the other presents it on the left). In this situation, synchronizing the changes made to the visualization of the two pictures will make it so the user cannot compare how the same motifs are represented in the images that are being compared.

Another interesting conclusion that can be reached by the presented analysis is that the competing web applications (which are, in a sense, the most direct competitors to the proposed application, as they share the same platform) are all simple and/or lacking in several of the key functionalities that are essential to our proposed application.

Table A.1 contains a visual summary and representation of the results of this research into the competing solutions.

2.3 Web-based Development

Since the main objective of this work will be accomplished through the design and implementation of a web-based application, it is important to introduce and discuss certain related topics and technologies that were researched and should be understood before diving into the discussion of the actual development process that eventually took place.

2.3.1 Basics of Web-based Development

There are some basic concepts associated with web and web-based development which should be understood in order to describe and develop a web-based application.

Front End

In application development, front end is the name given to the presentation layer that the user will interact with. This layer contains the interface that the user uses to interact with the different functionalities of the application.

While discussing the client-server model that is usually used in a web-based application, the front end is completely contained in the client side (and therefore in the machine with which the user interacts with the application).

In its most basic sense, front-end development is usually done using a combination of HTML, JavaScript and CSS. These three languages each have a role in the creation of the front end:

- HTML files are used to define the structure of the web page, and are later translated by the browser so that they can be shown to the user;
- JavaScript files contain the scripting of the front end, and it is through them that the components defined in the HTML files become interactive;
- CSS files are used to style the components defined in the HTML files, in order to control their visual representation.

Back End

In opposition to front end, back end is the name given to the data access layer, which contains and manages the data that the application needs to operate. The back end may also contain some other operations that are not suited for the client-side (such as those requiring a lot of computational power). Whenever the application requires some information that is not stored or created in the front-end portion of the application, it will be requested from its back-end portion.

In a web-based application, the back end can be contained in the local computer or in a remote location (or even both, if this is more appropriate for the specific application).

Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern tackles application development by dividing the application into 3 different parts [49]:

Model, which corresponds to the data being managed by the application;

View, which corresponds to the UI logic used to generate the pages that the user interfaces with;

Controller, which acts as a bridge between the aforementioned components, responding to the inputs provided by the user through the view and using them to retrieve the data stored using the model.

While using an MVC pattern, the developer has to develop their project while taking into account these three aspects: it is necessary to create the model that will be used to represent and store the data, implement the controllers that will correspond to the possible actions that the user may take while interacting with the application, and create views for every possible interface that the controller may have to generate in order to service these requests.

2.3.2 Architectural Pattern

While developing a web or web-based application, an architectural pattern determines how many different layers the application will have and how these layers will interact.

For the purposes of our work, two types of architectural patterns were considered: a Multi-Page Application (MPA) implemented using a web-oriented MVC framework, and a Single Page Application (SPA) with a web Application Programming Interface (API).

Multi-Page Application Approach

Basic MPAs are the result of the more traditional approach to web development, in which an application's UI is composed of several different pages that have to be loaded or generated by the back end and sent to the front end of the application when the user's interactions require a change in the UI. This way, whenever data is sent back and forth, the new pages have to be loaded or generated on the server and sent to the client to be displayed in the web browser, which is a process that takes some time to complete (possibly negatively affecting the User Experience (UX)).

Over the years, several web development frameworks have been created that are specifically focused in the creation of web and web-based applications following the MVC pattern, due to its several advantages.

When developed using one of these frameworks, a web-based application's workflow can be summarized as: when the user requests access to a web page, the controller receives that request, uses the model to collect the data necessary to generate a web page, generates

this page using a view, and then returns this page to the user. The user can then use the page to interpret the data, and possibly change it by interacting with the generated interface, which will send another request to the controller. The controller can then use the model to update the data and, in conjunction with a view, generate a new page to return to the user.

There are some advantages to this approach, namely:

+ The structure of the MVC pattern makes the application's structure, based in controllers, models and views, very loosely coupled [95, 60];
+ Due to the separation of concerns provided by the pattern, future modifications and development efforts may be less complex [95, 60];
+ Ability to create multiple views for any model [95];
+ High Search Engine Optimization (SEO) (since there are several different pages for the different functionalities).

There are also some disadvantages:

– Separating the functionalities into 3 different artifacts causes scattering, and forces developers to simultaneously maintain consistency in multiple representations [95];
– MPAs are arguably less suited to the development of small applications than SPAs;
– Applications may become very bulky, which may negatively affect performance [60];
– The levels of abstraction introduced by the framework might make it harder or at the very least more time consuming to navigate through the code [95].

Single Page Application Approach

When dealing with SPAs, instead of the application client being made up of several different web pages that the user traverses in order to use its different functionalities, it is instead composed by a single web page, which will be dynamically modified based on the interactions that the user has with its interface [32]. In this pattern, the application front end is composed by this single web page and the back end is a web API, which the client will use to access, submit and alter the data.

Essentially, this pattern makes it so that all the code necessary to generate the interface of the application is loaded all at once (only data will be requested from the server after this loading) [32], as opposed to MPAs, that need to either load or generate other pages when the user travels to them (e.g., to access other functionalities) [76].

This carries some advantages, such as:

+ Decrease of server load while using the application, since the page is loaded all at once, and only data will be loaded later [32, 76, 85, 100];

+ Faster and more responsive interfaces [76, 85, 100];
+ Integrated caching capabilities, which allows for usage while lacking strong connection to the server [76, 85, 100];
+ Providing a UX closer to a native desktop app, by having a faster and more responsive interface, as well as avoiding the need to traverse through several different links [85, 100];
+ Easy to debug, through the use of Chrome developer tools [76, 100];
+ Easier to test due to the separation of the client side and the server side (as they can be tested on an individual level);
+ Allows for an easier re-use of the server for other platforms, or to re-use a previously created server as a web API.

There are also, of course, some disadvantages:

– The existence of a single page makes it difficult to share links, leads to poor SEO, and invalidates the use of the browser's "Back" and "Forward" buttons to navigate in the recent page history [32, 76, 85, 100];
– Requires the development of two different projects: the server and the client (as opposed to an MVC framework, that only requires a single project);
– More difficult to keep track of the code, especially when using modules or extensions [32];
– Requires the use of JavaScript by the client's browser [76, 85];
– There might be some security issues due to the application's source code being exposed to the user. Vulnerable to script injection due to Cross-Site-Scripting (XSS) [76, 100];
– The large JavaScript data might take some time to load in a low bandwidth connection [76, 85].

2.3.3 Modern Approaches

With the ever-expanding importance that the internet has gained in our lives, the fields of web and web-based development have been under constant improvement. With this, several technologies were created to facilitate the development of more complex and secure applications.

This section aims to provide an introduction to some of these technologies that were either considered for usage in this work or were in fact used in the development process.

TypeScript

TypeScript [86] is an open-source programming language that was created and is maintained by Microsoft. It is a typed superset of JavaScript or, in other words, it is built on top of JavaScript, which makes it backwards compatible with JavaScript.

The syntax used to code in TypeScript is the same that is used in JavaScript, and code that is produced in TypeScript is compiled into simple JavaScript code that can run in any web browser. However, there are several advantages to programming in TypeScript over JavaScript, with the main one being the fact that it allows the programmer to use static typing, classes, and interfaces, making the code much more secure and easy to read [73].

React

React is an open source JavaScript library that was developed and released in 2013 by Facebook. On GitHub, this library has 6.7k watchers, 163k stars, 32.7k forks and 1531 contributors [23]. There are also 282k questions tagged with reactjs in Stack Overflow [62]. While this library is mainly used in conjunction with JavaScript, it also supports development in TypeScript.

React works through the use of several different technologies and tools, including components, JavaScript XML (JSX), and the manipulation of the state of the application.

Components are defined by the developer in a single JavaScript or TypeScript file, as a way to identify the elements of the application with which the user will interact with, and they contain any variables that are required to keep track of the state of the application.

These components contain within themselves functions that define their behavior and how they should respond when their state has been changed, as well as a specification of how the UI should be rendered in the user's browsers. These specifications are written using JSX, which is an extension to the JavaScript syntax introduced by React that is used to render the component in HTML (so that the browser can later display the UI to the user). This way, every component contains within itself a JSX specification of how its interface should be rendered, so that the user is able to interact with it in order to change the state of the application (these changes might cause the component to be re-rendered, if they have an effect the UI of the component). The JSX specification of a component may contain within it other components, in a tree-like hierarchy.

The state variables stored in the components are used to represent the current state of the application, and these are the variables that will be changed by the user's interactions with the application.

When the state of an application is altered in any way, only the components whose state has been affected will be considered for re-rendering, this way increasing the performance (since the whole page does not have to be re-rendered).

Another important feature comes regarding the way that React handles its data flow. React only supports one-way data flow [65] (as opposed to two-way data binding). With

two-way data binding, any changes to the UI will also change the model state automatically, and vice-versa. On the other hand, with one-way data flow, these changes can only be propagated from one side to the other (in React's case, from the model state to the UI, and not the other way around[1] [68]). One-way data flow is generally considered to be safer and has slightly better performance, as the data can only flow in one way (which reduces complexity and the possibility of errors to occur).

React uses virtual Document Object Model (DOM) rendering [4, 68] (as opposed to direct DOM rendering)[2]. With virtual DOM rendering, before a page is going to be changed in any way, a copy of the DOM is created, and the changes are made in this copy. The copy is then compared to the real DOM, which will then be updated with the differences. This is a more efficient process than direct DOM rendering, since it is much faster to work with JavaScript objects than with DOM.

React has a slight learning curve since, even though the library itself is quite simple to work with, one is required to learn how to write in JSX in order to develop the components (which are an essential part of how this library works).

Another aspect of React comes from its highly active developer community, that creates and publishes several components in the form of third-party libraries, which can be used to get access to more advanced features that the basic library does not contain. This does, of course, have positive and negative sides: while one has access to several more features than one would normally have, there are always some difficulties associated with working with code that was not developed by oneself.

Angular

Angular is a complete rewrite of AngularJS in the TypeScript programming language. It was developed by the Angular programming team at Google and a community of dedicated developers and companies, and released in 2016.

On GitHub, this library has 3.2k watchers, 70.5k stars, 18.5k forks, and 1333 contributors [22]. There are also 245k questions tagged with angular in Stack Overflow [61].

The first big difference between Angular and React is the fact that Angular is a TypeScript-based framework (as opposed to JavaScript-based). However, much like React, Angular uses components (which are referred to as directives) to define the different elements that will be rendered to create the UI that the user interacts with.

While React can be used in conjunction with TypeScript, Angular is completely developed in this programming language. Because of this, developers should be relatively familiar with this language before working with the framework.

[1] In order for UI changes to affect the model state in React, the component's JSX specification must include certain events (e.g., *onChange* and *onClick*) that will be raised in response to the user's actions. The developer must also define what should happen if such events are raised, so that the component can change the model state as intended.

[2] DOM, or Document Object Model, is another name for the web page interface that is shown to the user.

Regarding the components (or directives), Angular separates the UI part of the component and the behavior part of that same component in an HTML file and TypeScript file respectively. This allows for a more clear separation of these two elements, at the expense of making the applications bulkier.

Much like React, when the component state is modified, Angular is able to determine if and how to change the web page automatically by comparing the new version with the previous one and applying the differences to the already rendered page, making sure that only the necessary changes are performed. However, this process uses direct DOM rendering [68], which is less efficient than virtual DOM rendering (due to the reasons that were previously discussed in this section).

Differently from React, Angular only supports two-way data binding [65], which means that any changes to the UI will also change the model state automatically and vice-versa (as described before).

Angular is described as having a very steep learning curve, as it requires developers to learn TypeScript, a more recent and less frequently used programming language, and because it deals with complex syntax and concepts and contains a longer and more verbose documentation than React.

The Angular framework is more suitable for complex, enterprise-scale applications that are expected to have continuous scaling, as its steep learning curve stops being a factor the more you work with it, and its performance score tends to rise with the complexity of the application.

Vue.js

Vue.js is an open-source JavaScript framework created by Evan You and released in 2014, maintained by him and several other core developers. Much like React, it is also possible to use this framework while developing in TypeScript rather than JavaScript.

On GitHub, this library has 6.4k watchers, 179k stars, 28.1k forks and 382 contributors [24]. There are also 72k questions tagged with vue.js in Stack Overflow [63].

When compared to React and Angular, it is clear that the number of contributors behind Vue.js is much smaller, which is explained by the fact that this framework does not have the support of any major corporations, making it completely driven by the open-source community. However, this does not seem to affect the quality or popularity of the framework, as its GitHub stats easily surpass those of Angular and somewhat match those of React.

Vue.js is also based on the use of components. However, the framework allows for the rendering of these components to be defined in two different ways: either through the interpretation of JSX, or through the interpretation of templates. These templates, which are written in an HTML-based syntax, will be compiled into render functions, which will be used to render the components into Vue's memory before updating the browser.

Much like React, Vue also uses virtual DOM rendering [4]. The framework contains within itself a reactivity system so that every component monitors its reactive dependencies, which will allow Vue to know when a component has to be re-rendered based on the changes to the application state. This reactivity system combined with the information stored in Vue's memory allows Vue to calculate the smallest amount of components that it will need to re-render in order to perform the least amount of changes to the interface when the application state is altered. After these calculations are completed, the UI is updated to reflect the new state of the application.

While React only supports one-way data flow and Angular only supports two-way data binding, Vue.js supports both [65] (this way giving the developer the opportunity to use the option that is better suited for their application's needs).

Vue.js is described has having a very gentle learning curve, due to its high customization and extremely developer-friendly documentation. You are also only required to work with HTML, CSS, and JavaScript (which are commonly used programming languages, and therefore more developers have knowledge of how to work with them).

When compared to the other two options, Vue.js is generally considered to be more suitable for smaller and simpler projects, with React and Angular being a better option if the application is expected to reach a certain level of complexity.

2.3.4 UX and UI Design

In any application, there are two elements which must be taken into special consideration before the beginning of the development process: UX and UI. While these two terms are often used interchangeably, they refer to two different (albeit interconnected) concepts [34].

User Experience (UX) refers to the actual experience that a user has with that given product: how they feel about it, the emotions that it triggered, whether or not they are interested in using it again, among other topics.

User Interface (UI) is a concept that is referent to the interface that enables the user to interact with the functionalities and data that a product (or in this particular case, an application) seeks to provide.

As Dain Miller put it [45], in theory "UI is the saddle, the stirrups, and the reigns. UX is the feeling you get being able to ride the horse, and rope your cattle.".

Special care must be taken while designing these elements, in order to make sure that the user understands how to operate the application, finds it enjoyable to do so, and is able to efficiently and effectively complete their objectives through their interactions with this interface. To this effect, it is important to consider the characteristics of the stakeholders and their intended tasks while designing your product, to ensure that this product provides increased value to them, improves the system that is already in place (if there is one), and will in fact be usable by them in an enjoyable way. In certain cases,

it might even be interesting to include these stakeholders in the development process, be it as interviewees, as sketch/prototype testers, or an even more active role.

For the remainder of this section, we will present two concepts which should be taken into account while designing UX and UI: the UX honeycomb, and usability.

UX Honeycomb

The UX honeycomb is a diagram created by Peter Morville used to represent the seven crucial facets of UX [46, 40].

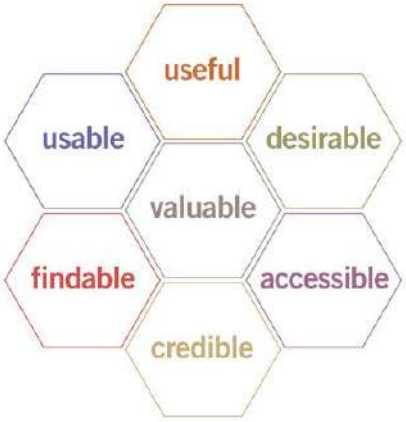


Figure 2.1: UX Honeycomb [46].

According to Morville, to have a satisfying UX every product should be [46, 40]:

- **Useful**, in the sense that it must serve a purpose to its target users;
- **Usable**, meaning that the product must efficiently and effectively allow its users to complete the task they set out to do upon use of the product;
- **Desirable**, or in other words, it is important for the user to be interested in using the product;
- **Valuable**, that is to say, the user must derive some value from their use of the product;
- **Findable**, in that it is extremely important that your product's users are able to easily access the product and navigate through it in order to find the information and functionalities that they desire to use;
- **Accessible**, so that your product is not limited in the user base that can utilize it, and is accessible to as many people as possible (even those with disabilities that would normally hinder this usage);
- **Credible**, in order to make sure that the users have confidence and trust in the product that you are providing to them.

Usability

Usability, when applied to the field of UI design, relates to how well a user can use the system's functionalities. When discussing this concept, Jakob Nielsen defined it as being associated with five different attributes [50]:

- **Learnability** – the interface of the system should be easy to learn by the users, so that they can reach maximal performance with it as fast as possible;
- **Efficiency** – after learning how to use the system's interface, the user should be able to reach a high level of productivity;
- **Memorability** – the interface should be easy to remember, so that the learning process does not have to be repeated whenever a more casual user spends some time without interacting with the system;
- **Errors** – the system should have no catastrophic errors, have a low error rate[3], and should allow the user to easily recover whenever an error does occur;
- **Satisfaction** – the use of the system should be enjoyable, in order to make sure that the users are subjectively satisfied when they interact with it.

Of course, it is not always possible to simultaneously reach optimal scores in every one of these attributes. For instance, in order to reach a low error rate, the system's efficiency might have to be reduced (one example of this can be found when a system requires extra confirmation before permanently deleting a file contained in it). In these cases, it is important to try to find a solution that is as close to a win-win situation as possible, or try to understand which of the attributes is more important for the specific task and system that is being developed.

2.3.5 Client-side Data Storage

One of the requirements for this web-based version of photo|uniq is to remain functional while temporarily disconnected (or, in other words, allow a user to continue their previous work without needing to contact the server). This would require the application to store information in between sessions, so that it is not necessary to remake computations and requests from the server that were already done in the past.

For these purposes, we did some research regarding how to use the application to store data in the local browser storage, and how to later access and modify this data. Two interesting technologies were identified in this research: localStorage and IndexedDB.

[3] The error rate is measured by counting the number of errors (actions that do not accomplish the desired goal) made by the user while performing a certain task.

LocalStorage

Writing to and reading from localStorage allows web and web-based applications to store data locally within the user's browser [64], therefore allowing later access to that same data and avoiding unnecessary requests to the server to get the data again. This is different from session storage, which is only kept until the user leaves the session (or, in other words, until they close the browser tab/window) [98, 64, 29], as well as cookies (which are used to send information to the server to change its behaviour and responses to better suit the user's needs) [64, 29].

In order to access the localStorage on a browser, the application only needs to use the Web Storage API. This API provides four JavaScript methods to the application [98]: *localStorage.setItem*, *localStorage.getItem*, *localStorage.removeItem*, and *localStorage.clear*.

Both the keys and values that are kept in the localStorage are in string format [98]. This means that some of the values that the application wishes to store will have to be stringified/turned into a JSON object, so that they can be stored and later used in a consistent manner.

In order to use the localStorage to, for instance, store and later use the pictures that the user uploads to an application, the application simply has to use the *getItem* method upon the initial load of the application for every picture that was previously uploaded. The application also has to use the *setItem* method when the user uploads any new pictures, and the *removeItem* when the user removes a picture from the application (while, for instance, using the picture name as a key, assuming that the picture name will be unique).

While the approach described in the previous paragraph works, it might not be the best performing solution. This is because every read and write request issued to the localStorage API are synchronous disk operations, which are much slower than accesses to the computer's main memory [89].

A more efficient solution that still could guarantee access to all previously uploaded pictures would be to store all the pictures in a single key-value pair (through serialization of the different pictures). This would allow for a significant reduction of the read operations that would be issued by the application, from N to 1 (with N being the number of pictures in question).

LocalStorage adheres to a same-origin policy, with the origin being defined by the domain, application layer protocol, and port of a URL of the document where the script is being executed [37]. Because of this, an origin cannot access the data that was stored in localStorage by another origin [37].

While most modern browsers are compatible with the Web Storage API [13, 98], every browser has different approaches to localStorage, which might lead to applications not having access to this service in certain situations [13]. Thus, in order to reduce unexpected behaviours, it is important to account for these differences in approach when developing the application.

It is also important to discuss the limits of localStorage. As previously discussed,

every browser has a different approach to localStorage, and therefore some browsers might allow for the application to store more data than others. When looking at a few of the most popular desktop browsers, we found that the current versions of Google Chrome, Mozilla Firefox and the Opera Browser have a limit of 10 Megabytes (MBs) per origin, while the current version of Safari has a limit of 5MBs per origin [89]. These limits were also verified using the tests found in [90], and [82] (note that every character stored occupies 2 bytes of storage space [98]). If these limits are at any time exceeded the Web Storage API triggers a QuotaExceededError [58], which can then be handled according to the developer's desires for the application.

IndexedDB

IndexedDB [28] is a low-level API that allows web and web-based applications to permanently or temporarily store more significant amounts of data client-side in formats available in the JavaScript programming language (such as strings, objects, arrays, among others) using a noSQL database with a key/value architecture [28, 8]. It also uses indexes to enable high-performance searches of this data [28, 8].

When compared to localStorage, the main draws it has are the ability to store a lot more data and in much more complex data types and formats [8]. Another significant difference is the fact that localStorage is limited to a synchronous API, in which the application thread must wait for the operations of the Web Storage API to finish before continuing its functions, while IndexedDB instead uses an asynchronous API [28, 8].

With the enhanced functionalities and capabilities of IndexedDB comes a much more complex and much harder to work with API [28], especially if the developer is unfamiliar with basic database concepts. It also requires more code to be developed, which might lead to a bulkier, more complex and verbose application, and consequently to this application being more prone to bugs.

One of the main differences between IndexedDB and localStorage lies in the complexity of the key-value pairs stored in an IndexedDB database. Since IndexedDB is not a relational database with tables representing collections of rows and columns, and is instead an example of an object-oriented database, an object store for a type of data must be created to store the data in question, and IndexedDB will simply persist JavaScript objects to that store [8]. These object stores can and should have in them a collection of indexes that makes it efficient to query and iterate across, which greatly increases the complexity of the development process [8].

Since the values stored in IndexedDB can be complex structured objects, the keys to which they are assigned (which can be of type string, date, float, binary blob, and array) can also be better suited to those values, and even be properties of the stored objects [8]. Through this, one can also better iterate and search through the stored values, for instance, as one can essentially sort the values by a known property, or retrieve only the objects that fulfill a certain condition [8].

One other difference lies in how one modifies the databases in question. In order to store and access data using IndexedDB, one must create a transaction and define the effects that it will have on a previously created database [8]. These transactions follow a N-reader-1-writer behaviour[4], have a well-defined lifetime (meaning that an attempt to use a transaction after it has completed will throw an exception), and cannot be committed manually (they instead are committed automatically by the API) [8]. Through this transaction model it is possible for a user to open two instances of a given web app (that utilizes IndexedDB) in two different tabs simultaneously without these two instances interfering with each other's modifications [8].

As mentioned previously, IndexedDB is an asynchronous API [28, 8]. This means that interactions with the database do not happen in real-time. Instead, the application will simply issue a transaction to the database, which will contain in it a callback function detailing what to do [8]. Afterwards, when the operation finishes, the application will receive a DOM event, which can be interpreted to verify if the operation succeeded or not, allowing for the application to respond to this success or lack thereof [8].

However, and much like localStorage, IndexedDB adheres to a same-origin policy (with the origin being defined by the domain, application layer protocol, and port of a URL of the document), so that different origins cannot access each other's databases (allowing only for an application to access and modify its own database) [8].

Much like the Web Storage API and localStorage, most modern web browsers are compatible with IndexedDB [14], but these browsers have differences in how they implement this API, which directly affect its limitations. However, there are still some browsers which are not fully compatible with this API [14] and some that are, but not always (such as Firefox in private mode [87, 14]). These particularities must be accounted for while developing the application (the developer should first verify if the application has access to IndexedDB before trying to access it for write or read operations).

Regarding storage space limitations, and looking at the four browsers previously discussed in the localStorage subsection, we can say that:

- **Chrome** allows for the IndexedDB API to use a maximum of 60% of the machine's disk space [35] (no information was found regarding the limits per origin);
- **Firefox** allows for the whole browser to use 50% of the machine's disk space for storage (this is called the global limit), and for an origin to use a maximum of 2 Gigabytes (GBs) or 20% of the global limit (whichever is lower) to store data [11];
- While no specific recent information regarding the storage limits of **Opera** was found, some articles state that its policies when it comes to IndexedDB are similar to those of Chrome (as they are both based on the chromium code) [11, 47] and we will therefore assume the same limits for these two browsers;

[4]This means that several transactions can read from the same scope/data at the same time, but only one can write to it at once. If several write operations are started at the same time, a queue will be created, and the transactions will follow that queue's defined order.

- **Safari** does not seem to have any hard limit to the usage of IndexedDB, as it allows for the API to store data freely until 1GBs is stored, and will then prompt the user in order for them to allow for more of storage to be used, in 200MBs increments [35].

From the numbers shown above, it is clear that IndexedDB has an enormous advantage over localStorage when it comes to storage space. Similarly to localStorage, any attempts made by an application to occupy more than its assigned storage quota will fail, with the corresponding write operation not having any effect on the database and throwing a DOMException with the message QuotaExceededError [11], so that the application can handle the situation in the best manner for the situation that triggered it [35].

2.3.6 Testing and Validation

An extremely important part of developing any piece of product is testing and validation. This is the part of the development process where the product is put through tests to make sure of its utility towards its target audience and correctness when executing its proposed task.

When dealing with a software application, several different types of testing and validation can and should be performed, which can be differentiated into two main categories:

- Functional validation or, in other words, ensuring that the developed application is stable and functions as intended without any serious or critical bugs, and is capable of achieving the end goals of its users;
- Non-functional validation, used to verify the correctness of the application's non-functional aspects that are not covered by functional validation (e.g., usability and customer satisfaction).

The division of what constitutes functional and non-functional validation is dependant on the application's goals and requirements. For instance, if the application needs to be able to perform at a certain level to fulfill it's requirements, then performance validation of this level constitutes an example of functional validation. However, if the application does not explicitly need to reach any rigid performance requirements (or, in other words, if reaching a certain level of performance would simply be desired, but not strictly necessary), then performance validation would be an example of non-functional validation.

This section will be used to further elaborate upon these two concepts, as well as their importance and how one might approach these steps in the context of the development process of an application such as the photo|uniq web-based application.

Functional Validation

According to the 5th Software Fail Watch [83] done by Tricentis, it was found that in 2017, 606 software failures harmed over 3.6 billion people and caused $1.7 trillion in lost

revenue. When added together, the time wasted by these failures caused about 268 years of downtime.

It is common to run into software bugs while developing an application, or really any piece of software. As the complexity of the software increases, it is unreasonable to expect a single developer (or even a team of developers) to consider every possible scenario that their program will be used in, which might lead to vulnerabilities, bugs, and a general decrease in quality of life for the user. This is why one of the most important steps that should be taken when developing any piece of software is to validate its correctness through Quality Assurance (QA) and testing.

The sooner the software errors are found and fixed during the development process the better, as this will lower the time required for fixing them (since there will be less code built upon the already flawed foundations), substantially less monetary cost (since, according to a study made by IBM, the costs required to fix a software error go up as you proceed through the software development life cycle [44]), and lower damage made to the application's reputation (with this damage being possibly completely prevented if the error is found and corrected before the users interact with it).

It is for these reasons that functional validation is so important. Through proper testing and validation, the developer can ensure that the developed application and its functionalities operate as intended and deliver the appropriate and desired results when the user interacts with them.

This validation can be done by continuously integrating the newly developed code into the main project and testing the resulting application, thus minimising the probability of an error reaching the later parts of the application's development. It is also advantageous to make these tests automated and independent from the developer's input (through the development of unit tests or integration tests, for instance), as this allows them to better distribute their time and reduces the need for repetitive and mindless work.

It is also important to make sure that the application is tested by people outside the development team, such as professional independent testers or even potential users that have not participated in the development process.

This will likely make it so the application is put through more intensive tests in ways that the developers might not have previously thought of, since these testers are likely to either have more experience with testing in general, or be capable of better simulating the experience that users will have when first interacting with the application, before they learn how it is meant to work and function (since the application developers will always be influenced by their knowledge of the application's intended workflow).

Non-Functional Validation

As previously explained, non-functional validation is the term used to label any validation that is not directly related to functional aspects of an application. Some interesting and valuable examples of this can be seen in the application's System Usability Scale (SUS),

Customer Satisfaction Score (CSAT) and Net Promoter Score (NPS), which, despite being all essential aspects of the application that should be verified, are not a function of the application, and thus their validation does not constitute functional validation.

These three aspects are calculated by putting the application's users through a survey after they have had contact with the application (either through normal use or through user testing), and interpreting their responses to calculate a metric which will allow us to understand how usable an application is (via calculation of the SUS), how satisfied the user was when using the application (via calculation of the CSAT) and how likely they are to recommend it to their friends and colleagues (via calculation of the NPS).

The **System Usability Scale** calculates an application's usability by analysing the responses that its users gave in a survey regarding their experiences with the application [80, 71]. This survey contains 10 statements based on the following generic template, which the user can respond to with a scale from 1 to 5 (with 1 meaning strongly disagree, and 5 meaning strongly agree) [80, 71]:

1. I think that I would like to use [this system] frequently.
2. I found [the system] unnecessarily complex.
3. I thought [the system] was easy to use.
4. I think that I would need the support of a technical person to be able to use [this system].
5. I found the various functions in [this system] were well integrated.
6. I thought there was too much inconsistency in [the system].
7. I would imagine that most people would learn to use [this system] very quickly.
8. I found [the system] very cumbersome to use.
9. I felt very confident using [the system].
10. I needed to learn a lot of things before I could get going with [the system].

As one can see, the odd-numbered statements are related to positive traits, while the even-numbered statements are related to negative traits.

After obtaining a significant number of answers to these questions, the System Usability Scale can be calculated by first subtracting 1 from the responses that users gave to odd-numbered questions, followed by subtracting the users' responses to the even-numbered questions from 5, which translates every statement to a value between 0 and 4, with 4 corresponding to the most positive response. Then, for every user that responded to the survey, we can add the values related to each of these statements, and multiply the result of this sum by 2.5, which will provide us with a SUS of 0 to 100 for every user. After this, we simply need to calculate the average of all the user responses, and we will have the application's SUS [80, 71].

We can then analyse this value to understand the usability score of the system. If an application's SUS is above 68, it is considered to have an above-average usability score, and a value above 80.3 is considered to be in the top 10% of applications [80, 71].

The **Customer Satisfaction Score** is computed by analysing the responses that the users provide to a question following the template "How would you rate your overall satisfaction with the [goods/service] you received?", to which they can answer with a value ranging from 1 (very unsatisfied) to 5 (very satisfied) [92].

After collecting the user responses, the CSAT can be calculated using the formula:

$$\frac{NSU}{NTU} \times 100$$

where NSU represents the number of users who were satisfied with the application (those who responded to the survey with a value of 4 or 5 or, in other words, "satisfied" or "very satisfied") and NTU represents the total number of users who responded to the survey [92].

This will provide a score that will show the percentage of users that found their interaction with the developed application to be satisfying.

The **Net Promoter Score** can also be derived by analysing survey answers provided by an application's users. In this case, the users are asked a question following the template of "How likely is it that you would recommend [Organisation X/Product Y/Service Z] to a friend or colleague?", to which they can answer on a scale of 0 to 10, with 0 meaning "Not at all likely" and 10 meaning "Extremely likely" [93, 94].

Depending on their answers, the users will then be classified as [93, 94]:

Detractors, if they respond with a score between 0 and 6 (these users are considered to be unhappy with the application, unlikely to use it again, and may even discourage others from using it);

Passives, if they respond with either 7 or 8 (these users are considered to be satisfied with the application, but vulnerable to competing options);

Promoters, if they respond with a score of 9 or 10 (these are users that are typically extremely satisfied with the application, and will more likely than not develop some loyalty to it and suggest it to other potential users).

After obtaining the user scores and sorting them into the three previously explained categories, we can then calculate the NPS by subtracting the percentage of users who were found to be detractors from the percentage of users who were promoters [93, 94]. For instance, if the distribution of a set of surveyed users turned out to be 20% detractors, 10% passives, and 70% promoters, then the tested application's NPS would be 50. This means that the Net Promoter Score can range from -100 to 100, with -100 being the worst score possible, and 100 being the best [93, 94].

Through the use of these three metrics, we can better understand how the user base really feels about the application and its interface, and try to use this information to

further improve the application. Of course, these three metrics also gain value if the surveys are performed again periodically, as this way we can better track how the user base feels about the application over time, and thus see if the changes that have been implemented were positive or negative additions to the application's UX.

2.3.7 Section Summary

There are a few basic decisions that should be made in the planning phase of the development of an application. Of course, one of them is the identification of the several parts that will make up the application, and how they will interact among themselves.

As explained previously in Section 2.3.2, we looked at two possible architectural patterns: MPA implemented using a web-oriented MVC framework and SPA with a web API. While analysing both options, we came to the conclusion that an SPA was a better fit for this work. An SPA architecture allows for a more responsive interface, and a UX much closer to a native application. It also seemed like a more natural fit for the complexity of the proposed application since, while it is not exactly simple, it can still be boiled down to some functionalities that can be easily integrated into a single page.

By deciding to follow this architectural pattern, there will be a more clear and distinct separation between the front end and the back end of the application, since these two parts will not need to be developed as a single project (as opposed to the MPA using an MVC framework, which integrates the development of the several layers of the application into a single project).

It is worth mentioning that while the proposed SPA pattern is not directly integrated with MVC, it is compatible with this pattern and in fact we developed the photo|uniq web-based application following an MVC or at least MVC-like pattern, with a clear distinction being made between the Models, Views, and Controllers of the application.

Researching the most popular and interesting technologies regarding front-end development was one of the most important steps when it came to planning how the proposed application was actually going to be programmed and implemented.

One of the first decisions that was made was regarding the use of TypeScript or JavaScript as a scripting language. Here, it was decided that TypeScript was a much more interesting solution, due to the code correctness and security that it provided when compared with JavaScript.

One other major decision that had to be made came regarding which framework or library to use for the development of the application front end. Since all three considered options are compatible with TypeScript, there were no limiting constraints that would automatically exclude one of them based on the scripting language of choice. Nevertheless, there were positives and negatives that could be used to decide among these alternatives.

The first option to be excluded was Vue.js, because of the framework being described as more suitable for smaller and simpler projects. This created some issues with choosing

this option, since there was a risk that even if Vue.js was suitable for developing the application in its current complexity, these constraints limited the possibilities for expansion and adding of further features in the future.

After this, a choice had to be made between React and Angular. Initially, React seemed to be a more interesting option, as it has a more active community and is generally considered to be a more enjoyable and developer friendly option when compared to Angular[5]. React also has a gentler learning curve when compared to Angular, which makes it a more attractive solution for a project with an estimated smaller development period (when the development team is not already proficient with either option).

However, in order to make sure that the best option was chosen, a very simple web-based application was implemented in TypeScript using React and again using Angular, so as to better compare the development process of these two alternatives. Through this more practical comparison, React also seemed to be the most interesting option since, while the two applications had similar performance and were capable of providing the same functionalities to their users (they were virtually identical), it was much simpler, more intuitive and more enjoyable to develop the application with React.

Through this process of elimination and comparison, we came to the conclusion that React was the best fit for the proposed project, and would therefore be the library to be used to develop the front end of the application.

Regarding client-side data storing, and looking at the details of the two presented APIs, it can be said that if it is not expected that the application will need to save a lot of data, and the data that is saved is not particularly complex (or can be saved in string format), then the application is better suited for the usage of localStorage. However, if the available space of localStorage is not enough to deal with the data storage requirements of the application, or if it is important for the data loading and/or saving to be asynchronous (for instance, to allow the application to load its UI while the data is being loaded/stored), then the application is better suited using the IndexedDB API.

It is, of course, also possible to use both of these APIs in conjunction if, among the several data types or functionalities of the application, some are better suited for localStorage, while others are better suited for IndexedDB.

Considering that the application will be dealing several photographs, each occupying a significant amount of space, IndexedDB was identified as a better option for this particular situation (with localStorage being used to store certain client-specific types of data that do not occupy as much storage space, such as setting-related flags).

As mentioned previously in Section 2.3.6, it is also extremely important to continuously test every piece of software as it is being developed, in order to make sure of its correctness during the whole development process. For this reason, there will be an emphasis on testing every implemented component and functionality of the application

[5] According to the Stack Overflow Developer Survey of 2020, 68.9% of the developers who previously worked with React are interested in doing so again. This percentage is reduced to 54% when talking about Angular [79].

during and after their development (both on an individual level and when integrated with the other components that have been previously implemented).

While the majority of the functional tests performed on the application will be done by the developer, we consider user testing to also be an essential tool that will be used to evaluate how suitable the application is when operated by potential users that have not been exposed to the development process. After these users interact with the application, they will be interviewed, and we will ask them for any suggestions and opinions that they might have and want to share, so that we might use them to further improve the application. These same users will also be asked to fill out a survey which will allow us to calculate the photo|uniq web-based application's SUS, CSAT, and NPS.

Additionally, some performance validation and testing will also be done on some of the application's key functionalities, in order to analyse their time complexity and see how they scale while handling increasingly higher numbers of images (i.e., potentially hundreds of different uploaded photos, which will be analysed and compared among themselves, aggregated into similarity groups, among other operations), as is expected from this work's use case.

2.4 Image Analysis

As explained in Section 2.1, in order to deliver some of its planned key functionalities, the photo|uniq application needs to analyse the images that are uploaded to it, in order to detect similar images (through the use of image feature detection and correlation), measure their technical quality, and calculate the homography matrices that can be used to translate point coordinates from one image to another.

In this section, we summarize our decisions when it comes to how the photo|uniq web-based application will handle these image analysis requirements. A more detailed discussion of these topics and the reasoning behind these decisions is presented in Appendix B.

When it comes to image feature detection and correlation, we chose to use the Oriented FAST and Rotated BRIEF (ORB) feature detection algorithm combined with a brute-force based feature matching algorithm, due to their ability to provide sufficiently accurate and precise results while simultaneously having relatively low computational, temporal, storage, and energy costs.

The precision of these algorithms will also be improved through the use of Lowe's ratio test as it was explained in Section B.1, which will help us to reduce the number of false matches obtained during the process of feature correlation.

After narrowing down the matching keypoints into good (or correct matches), we can then use these matches to verify image similarity between two images (which is essential for the photo|uniq application's automatic generation of similarity groups, further explained in Section 3.4.2), as well as estimate homography matrices capable of mapping points between two similar images through the process of matrix multiplication

presented in Section B.2 (which allows the application to achieve its motif-focused side-by-side image comparison, further explained in Section 3.4.4).

Regarding the objective analysis of the image quality, and as discussed in Section B.3, we chose to implement a simplified version of the workflow proposed by Alves [3]. In this simplified workflow, we will simply use a laplacian-based operator to calculate the variance of the laplacian of an image in order to identify that image's focus level, and thus its objective quality.

While this method for determining an image's objective quality has much room for improvement, especially since it doesn't take into account the other several aspects which can be considered when analysing an image's objective quality (such as motion blur, noise, among others), this method was considered to be sufficient for the main purpose of this work (i.e., the development of the application's front end and UI).

Since this front end was designed to be detachable from the data management and image processing components of the photo|uniq application, the implementation of a more robust and comprehensive image quality analysis algorithm can also be done in a future work if this is desirable, so as to replace or improve the simpler option that was implemented in this development effort.

3

Application Design and Development

3.1 Application Architecture

A simplified perspective on the architecture of the photo|uniq web-based application can be seen in Figure 3.1.

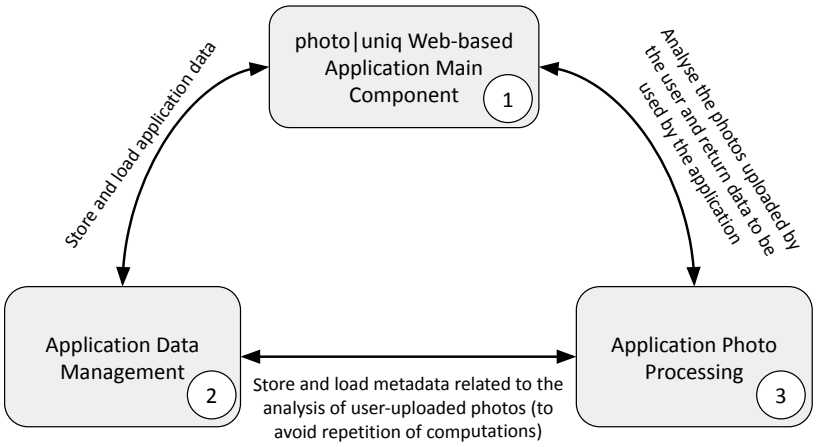


Figure 3.1: Architecture of the photo|uniq web-based application's.

As shown in Figure 3.1, the application is divided into three interconnected parts:

1. The main component of the photo|uniq web-based application, responsible for the application functionalities and operations, as well as the generation of the views and UI that the user will interact with;
2. The application data management, responsible for storing the application's data between sessions;
3. The application photo processing, to which the photo analysis operations are delegated.

These three parts all interact with each other to allow the application to function and maintain its consistency:

- The application data management part is responsible for storing objects that are used in both of the other two components, so that the application state is kept between sessions and to avoid unnecessary repetition of computations related to the analysis of photographs;

- The application photo processing part is used by the main photo|uniq application to perform the computations associated with the analysis of photographs, so that the main application only has to deal with the resulting metadata.

The main photo|uniq application is made up of several React components that change the UI based on the functionalities and views of the application that are accessed by the user. This main application was the main focus of the presented work, and most of the development effort was put into its implementation and design.

In order to access the application data management, an interface (*IDataAPI*) was created, which contains a definition of the various different methods that can be used to store and retrieve values from the chosen method of persistent data storage. In the currently implemented version of the application, this interface is implemented by a class (*IndexedDBDataAPI*) which uses IndexedDB to store and load data between sessions.

Similar to this, access to the application photo processing is done through an interface (*IImageProcessingAPI*) that contains a definition of the upper level methods that the application needs to call in order to process the user-uploaded images and retrieve the resulting metadata. In the currently implemented version of the application, this interface is implemented by a class (*OpenCVImageProcessingAPI*) that processes these images using OpenCV.js, a JavaScript binding of OpenCV that is capable of running in the user's browser to call certain functions and methods of the original C++ OpenCV library[1].

This effectively means that, in its current state, the complete photo|uniq web-based application is entirely contained in the front end, and runs in the user's browser. There are some advantages and limitations associated with this fact.

The main advantage is that, since the application is now completely contained in the user's computer, it is therefore bereft of any need to access a remote server, be it for storage or computation purposes. This means that the application is completely functional even while the computer is disconnected from the internet.

The flip side of this is that, in its current state, the application is also completely cut off from any outside influence and platform, even to the point where accessing the application using multiple browsers will mean that the user will not be able to access any persistent data that was stored using IndexedDB in a different browser (since the implementation of IndexedDB and access to its data is reliant on the used browser).

[1] Opencv.js is further elaborated upon and explained in Section D.4.

This also means that, in its current state, the application does not support cross-platform work (since there is no central server that is accessed by this or any other versions of the photo|uniq application).

However, the existence of the IDataAPI interface means that the application can be compatible with cross-platform work if the additional required infrastructure is eventually created (in other words, if this centralized server is implemented in the future). To do this, we would simply need to create and develop another class that implements IDataAPI to store and load persistent data using this centralized server infrastructure. This class could then replace or complement IndexedDBDataAPI as the *de facto* data management component of the application.

The existence of this centralized server infrastructure could also potentially mean that the computations associated with the application photo processing component could be moved to this remote back end, which would substantially reduce the computations that the user's browser would need to execute in order to provide the application's functionalities, or even potentially allow the use of more complex or refined image analysis algorithms to further improve the application's capabilities.

The final main limitation caused by the application being completely contained in the user's browser is that it means that the application is not able to directly write any data to the user's file system, since web applications running in most modern browsers are not allowed to directly write to the user's file system due to security concerns. This means that the user is not able to use the application to directly delete the photos that they deemed unworthy of keeping.

Nevertheless, the user is still able to propagate their photo selection into their file system, by downloading a zip file generated by the application which contains the photos that the user selected for keeping following the same path structure as they were kept in in the photo|uniq application. The process through which this is done is presented and explained in Section 3.4.6.

3.2 Data Model

The data that is stored and used by the photo|uniq web-based application is structured following a very specific system, in which two main "branches" can be differentiated: the elements related to the workspaces, and the elements related to the photo|uniq gallery.

The bulk of the application's functionalities can be found in the workspaces and the elements contained within them. This is where the user will upload images, analyse them, and then decide which images should be kept or deleted.

The workspace-related branch has four different types of elements that have an associated hierarchy between themselves: workspaces, workspace folders, similarity groups and workspace photos.

The photo|uniq application can contain several workspaces. These workspaces can contain within themselves workspace photos and workspace folders. The workspace

folders may contain both ungrouped workspace photos and similarity groups. Finally, the similarity groups contain only workspace photos within themselves.

Every element found in this branch can also be deleted and undeleted (or, in other words, put back into their original workspace) by the user. If an element is deleted, the elements that are contained within it are also deleted, and a deleted version of every element that is above it in level will also be created if necessary (meaning that if, for instance, the similarity group found in "workspace1/folder2/group3" is deleted, then every photo contained in "group3" will also be deleted, and a deleted version of "workspace1" and "folder2" will also be created for display and organization purposes, if one has not already been created previously). The deleted workspace elements follow the same hierarchy and display logic that is used by their non-deleted forms.

After the user has analysed the workspace photos contained in a similarity group and decided if they should be kept or deleted, they are also able to mark this group as processed. The user is then able to promote the processed groups to the gallery. Through this operation, every photo that was marked as desirable will be promoted to the photo|uniq gallery, while the unmarked photos will be instead deleted. This promotion process is further elaborated upon in Section 3.4.5.

This way, the photo|uniq gallery will serve as a repository where the user will be able to see the images that have been selected for keeping after the analysis process.

The gallery-related branch has three different types of elements that also have an associated hierarchy between themselves: albums, album folders, and gallery photos.

The photo|uniq gallery can contain several albums. These albums can contain within themselves gallery photos and album folders. The album folders contain only gallery photos.

With that in mind and in its current state, the photo|uniq web-based application's data model can be separated into four different parts:

- The data that is related to the workspaces and the elements contained within them, shown in Figure 3.2 and discussed in Section 3.2.1;
- The data that is related to the deleted elements, shown in Figure 3.3 and discussed in Section 3.2.2;
- The data that is related to the elements contained within the photo|uniq gallery, shown in Figure 3.4 and discussed in Section 3.2.3;
- The data that is related to the objects used to support the image analysis functionalities provided by the application, shown in Figure 3.5 and discussed in Section 3.2.4.

We will now elaborate upon each of these different parts in order to explain how they were implemented and why they were designed thusly.

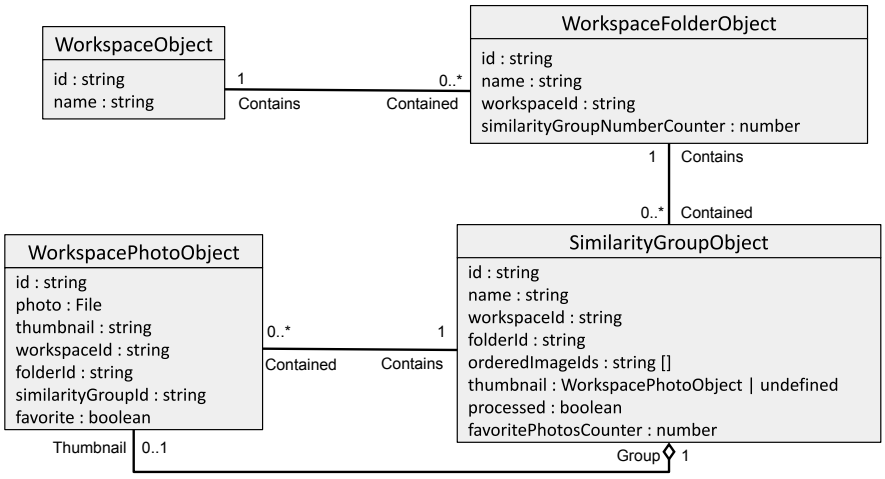


Figure 3.2: Data model of the workspace-related elements.

3.2.1 Workspace-related Elements

As shown in Figure 3.2, the data model associated with the workspace-related elements is divided in four different classes (each one representing an element).

WorkspaceObject is a class used to store the information regarding a workspace, which amounts to its id and name.

WorkspaceFolderObject is used to store the information regarding a workspace folder. For identification purposes, it contains that workspace folder's id and name, as well as the id of its parent workspace. It has a counter that contains information regarding the number of similarity groups that have been created inside of this workspace folder, used for the purposes of naming the similarity groups, whose names are always a variation of "Similarity Group x" (with x being the current counter number, which is incremented after the creation of a group).

SimilarityGroupObject stores the information regarding a similarity group. In terms of identification, this essentially amounts to the group's id and name, as well as the id of the parent workspace folder and the id of the workspace in which that folder is contained. This class also contains an array of workspace photo ids that is used to keep track of the order of the workspace photos that are contained in the group, and a *processed* flag that is used to mark the similarity group as being ready to be promoted to the gallery. For display and representation purposes, the *SimilarityGroupObject* class also contains information regarding its thumbnail (which is either its top rated workspace photo or undefined if the similarity group is empty), and the number of favorite photos that are contained within the group (which is kept in the *favoritePhotosCounter* variable).

Finally, the *WorkspacePhotoObject* class is used to store the information related to the application's photos that have not been deleted or promoted to the gallery. Here, and for identification purposes, it contains the photo's id, as well as the id of the workspace,

workspace folder and similarity group it is contained in. It also contains the photo file itself, and a *favorite* flag that is used to represent whether or not the photo is meant to be kept or deleted when the similarity group containing the workspace photo is promoted to the gallery. The *thumbnail* variable is used for display purposes when listing photos, and contains a Base64 string value version of the image stored in the *photo* variable, resized to a smaller resolution.

It is worth noting that while the photos can be contained directly in a workspace or workspace folder (and not just in a similarity group), for the purposes of simplifying the data model of the application it was decided that when it came to the implementation of the application itself, the photos were to always be contained in a similarity group.

Here, a decision was made that the photos that were directly contained in a workspace folder were placed in a default similarity group (marked with a *similarityGroupId* with a value of "_default"), and the photos that were directly contained in a workspace were placed in a default workspace folder and a default similarity group (marked with a *folderId* and *similarityGroupId* with a value of "_default"). Through this implementation decision, we were able to simplify the data so that, model-wise, the workspaces only contain workspace folders, which contain similarity groups, which contain workspace photos.

Nevertheless, when it comes to the application's views and UI, these photos are displayed as being directly contained in their non-default parent elements (or, in other words, the "_default" workspace folders and similarity groups are transparent to the user).

3.2.2 Deleted Elements

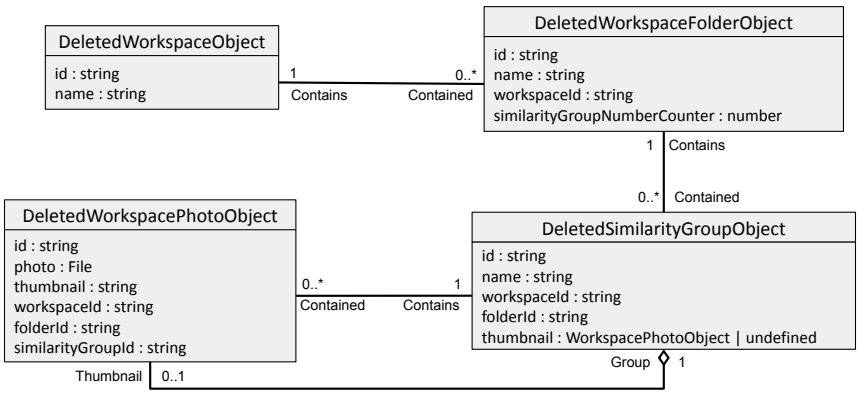


Figure 3.3: Data model of the deleted elements.

As we can see while comparing Figure 3.2 and Figure 3.3, the data models associated with the workspace-related elements and deleted elements are extremely similar.

This is because currently, the only elements that are able to be deleted in the application are those that are contained in the workspace-related data model. When these

elements are deleted, an entry is created containing the necessary information to represent their deleted forms, which are most of the same values as their non-deleted forms.

The reason why we decided to represent the deleted elements using separate classes rather than other methods (such as a *deleted* flag) was due to the separation of concerns that this solution provides, while simultaneously allowing for higher code security since we can, for instance, set all the values of the classes to be read-only (which makes it so trying to change these values after they are initially set causes a compile-time error).

Another point that made this solution seem like a better option was because of the possibility that the same element had to be simultaneously represented as being deleted and non-deleted (for instance, if a workspace was not deleted itself, but previously contained elements that had since been deleted). While dealing with situations such as these, it was simpler and made more sense to create a different object to represent the deleted and non-deleted forms of the same element.

3.2.3 Gallery-related Elements

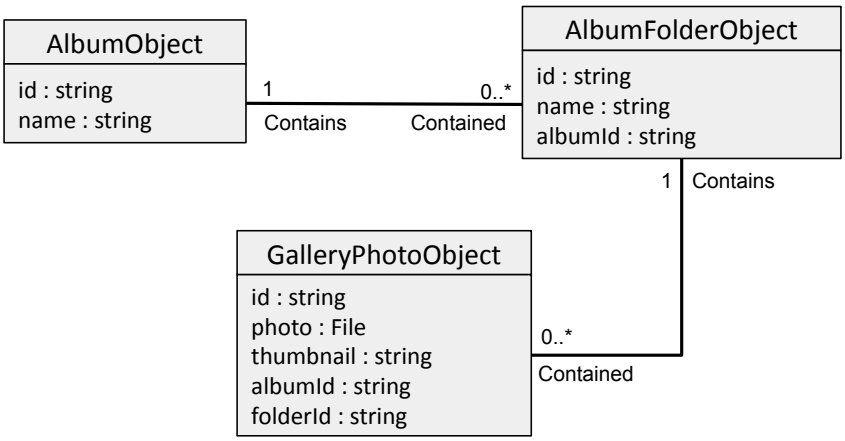


Figure 3.4: Data model of the gallery-related elements.

The data model associated with the gallery-related elements (visually displayed in Figure 3.4) is divided in three different classes (each one representing an element type).

Album is a class used to store the information regarding an album (i.e., its id and name).

AlbumFolderObject contains the information used to identify an album folder: its id and name, as well as the id of its parent album.

Finally, the *GalleryPhotoObject* class is used to store the information related to the application's photos that have already been promoted to the gallery: the photo's own id, the ids of the album and album folder it is contained in, the photo file itself, and a thumbnail version of this photo file (used for listing purposes).

Similarly to the case of the workspace-related elements, while gallery photos can be contained directly in an album, for the purposes of simplifying the data model of the application, it was decided that the objects representing these gallery photos would always be contained in an album folder. This means that the photos that are directly found in the root of an album will be stored in a transparent default album folder (marked with a *folderId* with a value of "_default") for the purposes of the application's source code.

3.2.4 Image Analysis Support Objects

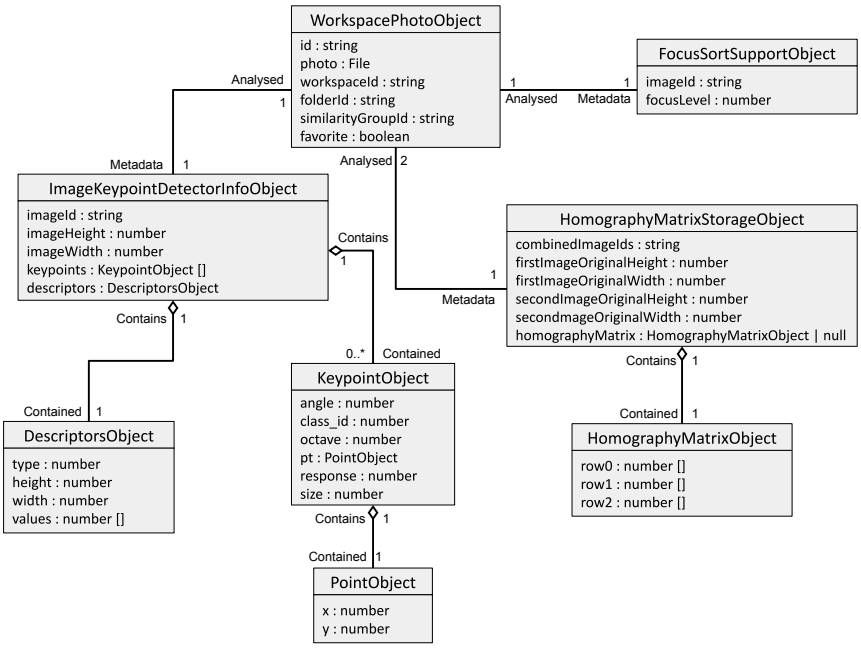


Figure 3.5: Data model of the image analysis support objects.

The data model related to the image analysis support objects, shown in Figure 3.5, contains several different types of classes that are created through the analysis of the *photo* file found in the *WorkspacePhotoObject* class that was introduced in Section 3.2.1.

Through these analyses, three types of object can be created: *ImageKeypointDetectorInfoObject*, *HomographyMatrixStorageObject* and *FocusSortSupportObject*, which are instantiated and used during or before the execution of the functionalities that are further elaborated upon in Section 3.4.

ImageKeypointDetectorInfoObject is created after the detection of the keypoints of an image, a computation that may take place either during the automatic grouping of images or before the calculation of the homography matrix used for side-by-side image comparison. The design of this class is highly influenced by the values that are returned when using the algorithms provided by OpenCV.js to detect an image's keypoints, particularly in regards to the values that are stored in the fields *keypoints* and *descriptors*.

Keypoints is an array of *KeypointObject* (which is a class that stores every value that is used to define a keypoint in OpenCV, including a point with x and y coordinates), and *descriptors* is an instance of *DescriptorsObject* which contains the height, width and type of an instance of *Opencv.Mat* (an object type which represents a matrix), as well as the values contained in the *Mat* itself, so that it may be reconstructed again when correlating the keypoints of two different images.

Finally, *ImageKeypointDetectorInfoObject* also contains information regarding the dimensions of the image that was analysed, and the id of the image itself.

HomographyMatrixStorageObject is used to store the information regarding the Homography Matrix used to translate point coordinates from one image to another during the process of side-by-side image comparison. Here, a combination of both of the images' ids is used to identify each instance of *HomographyMatrixStorageObject*. It also contains the height and width of both of the images, as well as the homography matrix itself, stored in the form of an instance of *HomographyMatrixObject* or *null*, if no homography matrix was able to be calculated. *HomographyMatrixObject* is a class that represents a homography matrix using three number arrays with a length of three to represent the 3x3 matrix.

The last class type, *FocusSortSupportObject*, is created when trying to ascertain the objective quality of the images contained in a similarity group. In the currently implemented version of the application, this is done through an analysis of the overall focus level of the provided image, which returns a numeric value. This way, the only information that is necessary to be stored by the *FocusSortSupportObject* is the id of the analysed image (for identification purposes) and the calculated overall focus level.

3.3 UX and UI Development

A lot of consideration was put into the development of the application's UX and UI.

We initiated the UI development process by using Axure RP 9 [6] to create several non-functional prototypes of the application. This allowed us to establish an initial plan for how the application's UI would be implemented, while maintaining a very fast design process in which the prototypes could be created extremely quickly and without any need for programming. These prototypes would then be shown to the application's stakeholders in order to receive suggestions and criticisms which could then be implemented to improve the application's UI.

After this initial prototyping stage was completed, the application's development process began. During this development process, the plan that was created during the prototyping stage was followed as much as possible, while at the same time leaving room for improvement of the UI if and when opportunities for this improvement were found.

In this section, we will display the current status of the application's UI by showing screenshots of the several views that the user can interact with, followed by an interpretation of these screenshots and an explanation of relevant any design decisions.

3.3.1 General UX and UI Approach

Two of the main goals and objectives that were set when it came to the creation of the UX and UI of the application was for it to be simple and clean.

For these purposes, the application's UI mainly follows a colour pallet of black, white, and some shades of grey. Other than that, a light blue tone (hex code #8AE5E1) is also used in certain circumstances, to denote the active/hovered forms of buttons, as well to occasionally add some colour contrast. With a few very specific exceptions[2], these colours make up the entirety of the photo|uniq web-based application UI's colour pallet.

We also decided that, when possible, we would avoid the use of text to establish functionalities or button effects, and would instead favor the use of icons to represent the operations that would be triggered when the user pressed a button or navigated through the application. However, since there are certain operations that cannot be easily represented through a single icon, or even if they are, it is always possible that the user might not understand the meaning behind a button's icon, the application's UI was developed so that a short explanation of the functionalities are provided to the user if their mouse pointer hovers the button that is pressed in order to execute this functionality.

Another important aspect was to make it so it was clear to the user that the buttons were interactive. For these purposes and for every button, two different set of icons were developed, one of which was used to represent these buttons when they were inactive/not hovered, and another which was used to represent the buttons in their active format, be it when they were hovered, or when they were passively active. For all the icons that were developed for the photo|uniq web-based application, the inactive button icons follow a colour scheme of black and white, and the active button icons follow the same exact icon design as their inactive versions, but are instead coloured (to differentiate them from their inactive versions and also to draw the attention of the user). An example of these inactive and active icon versions can be seen in Figure 3.6.

Another necessary aspect of UI design is to understand which actions should be barred behind a confirmation dialog box and which should not. A confirmation dialog box is an element of an application's UI that may be shown to a user before they execute an operation in said application. It typically contains some form of a question inquiring the user if they are sure that they wish to execute the operation in question, and also contains two buttons: one of which can be used to confirm their interest in the operation being executed, and another which can be pressed to cancel the operation.

It is important to strike a balance when it comes to the usage of confirmation dialog boxes: they can make a user's actions more secure and avoid potential mistakes from the user's part, but they should also not be used for every functionality request or button press, since this has the consequence of breaking the flow of the application's actions.

In the photo|uniq web-based application's UI, we attempted to achieve this balance by only using confirmation boxes before the user tries to commit an action with potentially

[2]For instance, the use of a bright red colour (hex code #ED2F2F) to represent negative actions.

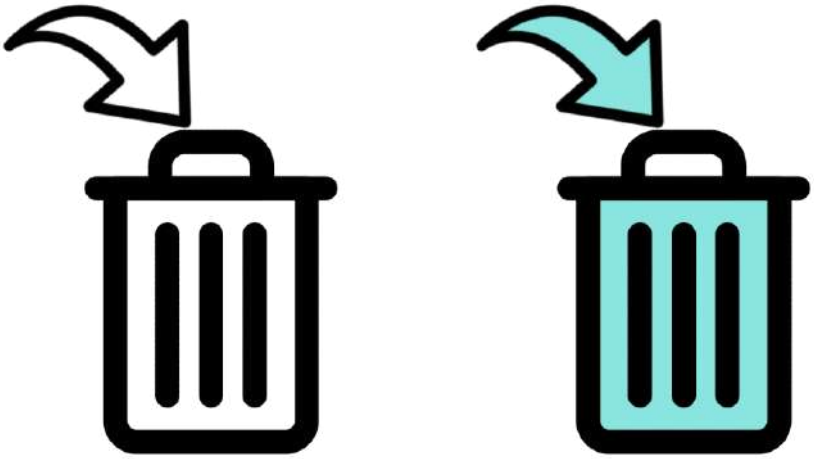


Figure 3.6: Inactive (leftmost icon) and active (rightmost icon) versions of the button used to delete the currently viewed photo|uniq element.

negative or permanent long term effects, before they try to execute lengthy operations that cannot easily be taken back after being completed, or when we considered that a button press might not make sense in the context of the current application state.

3.3.2 Application Header

After the user has logged in to the application, they are immediately redirected to the photo|uniq application's home page. At the top of the application, the user can always find the application header, shown in Figure 3.7.

Figure 3.7: The photo|uniq web-based application's header as it is shown when the user is in the application's home page.

This header was one of the most important UI pieces of the whole application to get right, since the user will be exposed to it for most of their time with the application. Our goal during the design process was for it to be both aesthetically pleasing and informative, while simultaneously being as functionally and spatially efficient as possible.

The header floats at the top of the browser window. This means that it will always be visible to the user, even if they scroll down while viewing the contents of a page.

As shown in Figure 3.7, the photo|uniq web-based application's header can be separated into three main parts:

- The navigation portion of the header, identified in Figure 3.7 with the letter **A**, which allows the user to navigate between different pages of the application that

they have either recently accessed or that are related to the image analysis and selection components of the photo|uniq application;

- The functionality portion of the header, identified in Figure 3.7 with the letter **B**, which contains a variable number of buttons that allow the user to access some functionalities associated with the currently visited page;
- The user-related portion of the header, identified in Figure 3.7 with the letter **C**, where the user can access information and pages related to their user account and experience.

We will now separately present and discuss each one of these three parts.

Navigation Portion

The leftmost part of the application header is dedicated to the navigation between the functional pages of application (those related to workspace-related elements, gallery-related elements, hidden elements, or deleted elements), as well as to the navigation between pages that have previously been accessed by the user.

As explained previously in Section 2.3.2, when choosing to implement a single page application, the browser's back and forward buttons will not be able to fulfill their usual purpose, since the application is composed of a single page (and thus, the user cannot traverse through their most recently visited pages by changing the visited url). It was for these purposes that the two leftmost buttons of the application header were implemented.

These are the page history navigation buttons which, as the name implies, can be used to navigate through the user's page history. They functionally replicate/substitute the back and forward buttons that are typically found on the top left of a web browser, and thus were also placed in the leftmost part of the application header. If, in the currently visited page, the user's page navigation history does not have any previous or next page, then the corresponding button that would allow them to visit these pages will be disabled (this is the case in Figure 3.7, where both buttons are disabled, whereas in Figure 3.8 only the forward button is disabled).

The next four buttons found in the navigation portion of the application header can be used to navigate to the four main functional hubs of the photo|uniq application. From left to right, these buttons allow the user to navigate to:

- The application home page, where the user can access the workspace-related elements of the application;
- The photo|uniq gallery, where the user can access the gallery-related elements of the application;
- The hidden elements hub, which is not implemented in the current version of the application, but is meant to house the uploaded photos that the user has chosen to be excluded from the selection process of the workspaces;

- The deleted elements hub, which contains the user's deleted elements.

The navigation portion of the application header can also be used to quickly identify the path that the user took to access the currently viewed page, if it is accessible through one of the four main hubs of the photo|uniq application.

In the case of Figure 3.7, we can see that the shown screenshot was taken from the application home page, as the icon of the button that is pressed to access this page is displayed in its coloured version. If the user is currently viewing a page associated with a lower level element (e.g., a workspace folder or similarity group), this is also shown in a similar fashion, as shown in Figure 3.8.

Figure 3.8: The photo|uniq web-based application's header as it is shown while the user visualizes a workspace folder.

In this figure, the shown application header was displayed in a page associated with a workspace folder, and thus one slightly smaller[3] button with a coloured icon is shown after the photo|uniq home page icon. This icon is the visual representation of a workspace, and the user can click it to navigate to the associated element (or, in other words, the parent workspace of the currently viewed workspace folder). They can also hover this button to see the name of the corresponding element (for instance, if they hover the workspace icon, they might be shown "Workspace Foo").

Functionality Portion

The functionality portion of the application header is used to house certain functionalities that are available in the page that is currently viewed by the user. The decision to place these "functionality buttons" in the application header was done in order to both provide a consistent place where the user could know to look for the functionalities of any given page, as well as to take advantage of the existing unoccupied space that already existed in the application header.

Depending on the purpose of the page that the user is currently on, this part of the application header will contain different buttons to provide functionalities that are relevant to the current page. This can be seen while comparing Figures 3.7 and 3.8.

The functionality portion of the application header shown in Figure 3.7 contains two buttons, which allow the user to add a workspace to the photo|uniq application, and select all existing workspaces (these buttons are relevant to the Home page of the application, as this is where the existing workspaces are shown to the user).

Conversely, the functionality portion of the application header shown in Figure 3.8 is associated with a workspace folder, and thus has different buttons: one which allows

[3]The difference in the size of the buttons is meant to visually demonstrate that they are associated to specific elements, as opposed to the main hubs themselves.

the user to upload photos to the currently displayed workspace folder (which, when pressed, will take the user to the upload page presented in Section 3.3.4), one which will automatically generate similarity groups using the existing groups and ungrouped images contained in this folder, one which allows the user to create a new empty similarity group inside the displayed workspace folder, and two which allow the user to select all of the similarity groups and ungrouped images contained in the shown workspace folder.

The buttons shown in the functionality portion of the header can also change depending on the state of the application. For instance, if the user selects any workspace photos that are contained in the currently shown workspace folder, the application header will be changed to the one that is shown in Figure 3.9(b).

(a) Before selecting photos.

(b) After selecting photos.

Figure 3.9: Comparison of the photo|uniq web-based application's header as it is shown before and after the user has selected one or more workspace photos while viewing the contents of a workspace folder.

By comparing Figures 3.9(a) and 3.9(b), one can see that the functionality portion of the application header shown in Figure 3.9(b) contains a different set of buttons.

The button identified with the number **1** was removed, and the button identified with the number **2** has been replaced, changing its original functionality of creating an empty similarity group to another one which also creates a similarity group, but will then move the selected photos to this newly created similarity group.

Two other functionality buttons were also added to the application header. The button identified with **3** moves the selected photos to the root of the workspace that contains the currently viewed folder, and the one marked with **4** deletes the selected photos.

User Portion

The user portion of the photo|uniq application header contains the buttons and UI components that can be used by the user to access their user-related information and pages, as well as to log out of the application.

It was placed in the rightmost part of the header in order to follow the UI practices of some of the most currently popular web pages (i.e., Google [25], Facebook [20] and YouTube [99]), which also keep the buttons that allow the user to access their notification and user-related pages and information in the top right of their application UI.

In its base state, the user portion of the application header is made up of two buttons: the notification bell and the user button (found to the right of the notification bell). These

buttons differ from most of the remaining buttons that can be found in the application header in the sense that, if pressed by the user, they will not immediately redirect them to a different page or activate a functionality, but will instead toggle a portion of the application's UI which is hidden by default, as seen in Figures 3.10 and 3.11.

If the user presses the notifications bell, then the notification box will be hidden or shown, depending on its current status. When the notification box is shown, the application header's appearance will change to the one found in Figure 3.10.

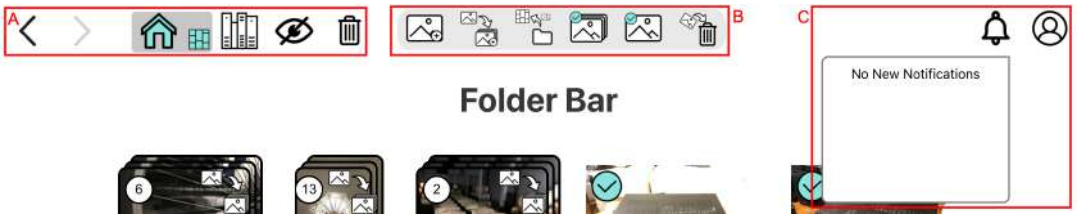


Figure 3.10: The photo|uniq web-based application's header when the notifications box is shown.

The notifications box is a very simple piece of UI that is simply used to show all the relevant and recent notifications to the user. Since the use of notifications is not implemented in the current version of the application, the UI and design of the notifications that will be shown in the notifications box has not yet been developed, and thus will not be shown nor discussed in this dissertation.

If, on the other hand, the user presses the user button (found to the right of the notifications bell), they will toggle the user menu. If the user menu is shown, the application header is changed to that which is displayed in Figure 3.11.

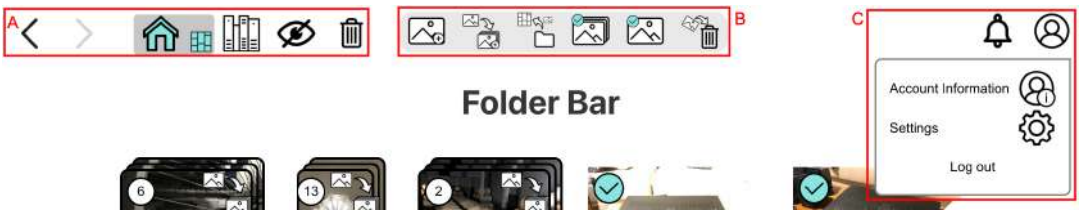


Figure 3.11: The photo|uniq web-based application's header when the user menu is shown.

The user menu follows much of the same design as the notifications box, but rather than having its content be changed depending on the current state of the application, it will always contain the same buttons in it: the first button will take the user to the account information page, the second one will take the user to the settings page, and the third one will log them out of the application, and redirect them to the log in page.

The UI of the account information and settings pages will not be discussed or presented in this dissertation, as they were not a part of the main focus of the developed work, and thus are not as polished as the remainder of the application and are currently not a proper representation of their desired completed status.

Similarly to the buttons found in the navigation portion of the application header, if the user is currently viewing the account information or settings pages, then both the icon of the button used to toggle the user menu and the icon of the button used to access the currently visited page will be changed to their coloured versions, and the background of said button will be coloured gray.

3.3.3 Non-photo Element Visualization

The pages used to view the contents of the several non-photo elements of the photo|uniq application, namely workspaces, workspace folders, similarity groups, their deleted forms, albums, and album folders, are all alike in terms of UI and design, as they mostly aim to fulfill similar purposes and contain elements that can be represented with similar visuals. An example of one such page can be seen in Figure 3.12, which displays the contents of a workspace folder.

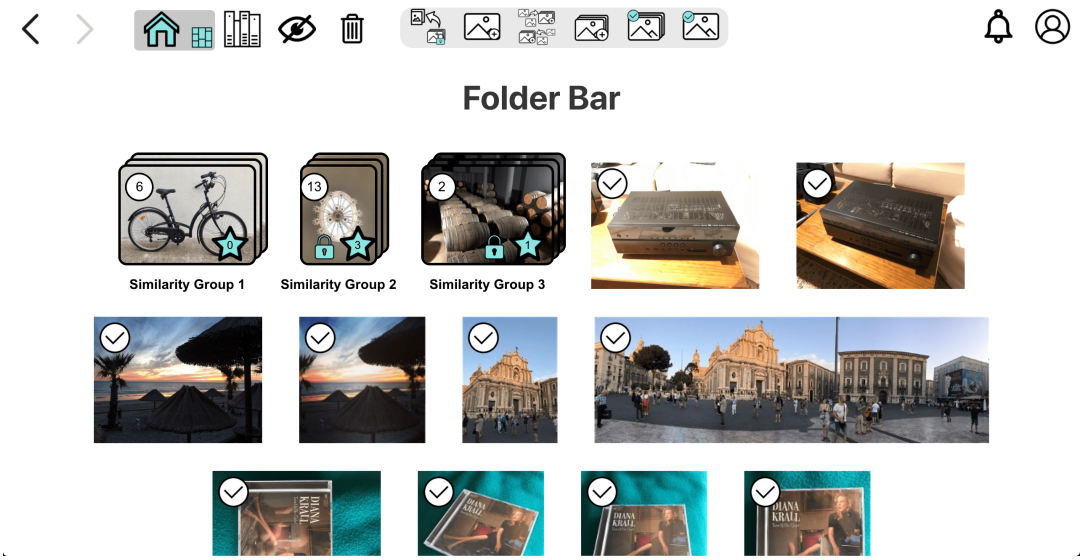


Figure 3.12: View used to display the contents of a workspace folder in the photo|uniq web-based application.

At the top of the page, immediately below the application header, we can find the name of the currently viewed element (in the case of Figure 3.12, “Folder Bar”).

All the space below this title is used to display the contents of this element, starting with the non-photo elements contained within, if there are any, which are then followed by any photos contained in the element.

In the case of Figure 3.12, the element whose contents are being shown is a workspace folder, which can directly contain both similarity groups and workspace photos. Following the explained logic, the first contents that will be shown are similarity groups (if the current element contains any), which are then followed by any workspace photos that may or may not be contained in the shown workspace folder.

It is worth mentioning that if the shown element contains more elements than those that can be shown in the browser window at one time, then the page will continue in a downwards fashion, with the user having to scroll up and down to see all the elements.

As one might notice, different types of elements have different UI representations.

For instance, any non-photo elements have a name associated with them, so that the user might better differentiate between these contained elements. This is invaluable in cases where their icon/thumbnail is not enough to identify them.

Different element types will also be represented by different visual icons. By looking at Figure 3.12, we can see that similarity groups are visually represented by showing their top rated workspace photo, which is also repeated three times to indicate a set of photos (as opposed to a single photo). Conversely, workspace photos are represented by solely showing a thumbnail version of the photo.

Similarity groups also display the number of favorite photos contained inside them in the bottom right of their thumbnail (represented visually by a combination of the icon used to represent favorite photos with the number of favorite photos contained in the group). For instance, of the three similarity groups shown in Figure 3.12, Similarity Group 2 contains three favorite photos, while Similarity Group 3 has one favorite photo, and Similarity Group 1 has none.

If the user clicks on any of the contained elements shown in this UI, the page will be changed accordingly. If they click on a non-photo element, then the page will be changed to show the contents of that element (e.g., if the user clicked on a similarity group, they would be redirected to a page showing the contents of that similarity group, which would have a UI similar to the one shown in Figure 3.14). If, however, they click on a photo, they will be taken to a page that can be used to better see and visually analyse the clicked photo (this page and its UI is presented and discussed in Section 3.3.5).

At the top left of the visual representation of each and every one of the contained elements, the user can also find a select button, which can be used to select these elements.

This select button may also change between different element types. For instance, the select buttons of workspace photos simply show a check mark inside of a circle, while the select buttons of similarity groups instead show the number of photos contained within.

The selection of any one of these elements will affect the UI of the page, as shown in Figure 3.13.

The first thing that might be noticed is that the icon of the select button for the selected elements changes to a coloured one, in order to highlight the change in their status.

Other than that, we can also notice that the selection of elements may cause a change in the functionalities that are available to the user (to reflect the user's ability to specifically interact with the selected elements). Looking at the differences found in Figures 3.12 and 3.13, four completely new functionalities were added and two previously existing ones were removed.

Other than those added to the functionality portion of the application header, another

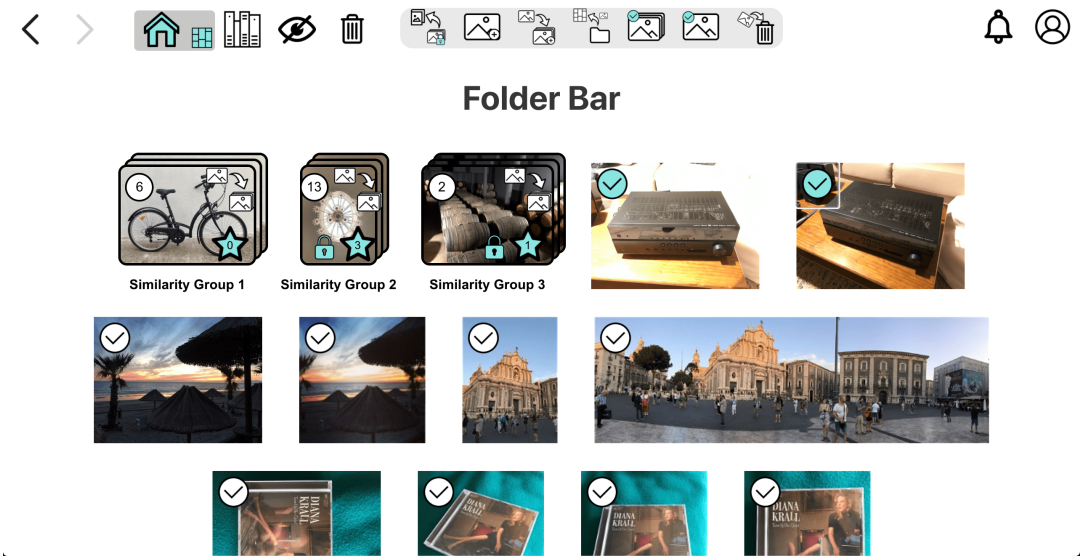


Figure 3.13: View used to display the contents of a workspace folder in the photo|uniq web-based application after the user has selected two workspace photos.

button was added to the page, which can be found on the top right of the visual representation of the similarity groups. This button allows the user to move the selected photos to the similarity group associated with the button (in which case, the photos will no longer be contained in the root of the workspace folder and will instead be moved to the chosen similarity group, causing them to no be longer shown in the current page).

When viewing the contents of a workspace-related element, the UI of the pages such as those shown in Figures 3.12 and 3.14 also contain some drag-and-drop capabilities. For instance, the user can drag a photo contained in a workspace folder into a similarity group contained in that same folder in order to move that photo into the group.

Some particular changes are also performed to this UI when viewing the contents of a similarity group (shown in Figure 3.14).

When the user is viewing the contents of a similarity group, the drag-and-drop capabilities of the page are used to manually change the order of the workspace photos contained in that group. This can be done by clicking on a photo, dragging it to a different place in the photo list, and letting go of the mouse.

Additionally, since it is possible to mark photos contained in similarity groups as favorites, an additional button was added to the thumbnail of the workspace photos contained in similarity groups, which when pressed will toggle the favorite status of these workspace photos.

This button is also different between favorite and non-favorite photos, with the button that is placed on favorite photos showing the coloured version of the favorite photo icon (as is the case with the first, second and fifth workspace photos shown in Figure 3.14), and the button that is placed on non-favorite photos instead showing the uncoloured one by default, and only being changed to the coloured version when hovered (as is the case with the remaining workspace photos).

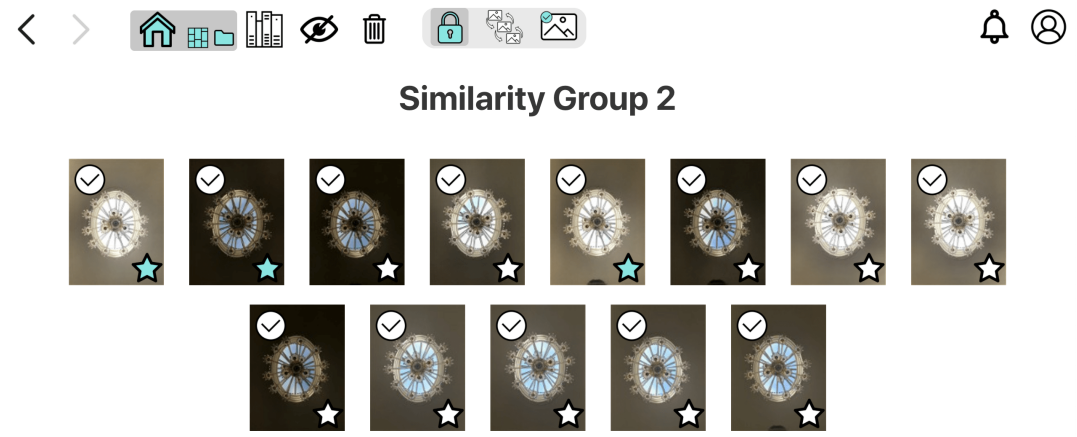


Figure 3.14: View used to display the contents of a similarity group in the photo|uniq web-based application.

3.3.4 Image Upload

It is through the use of the image upload page that the user is able to upload images into the photo|uniq web-based application. Before the user adds any images to it, the UI of the page will be similar to that which is shown in Figure 3.15.

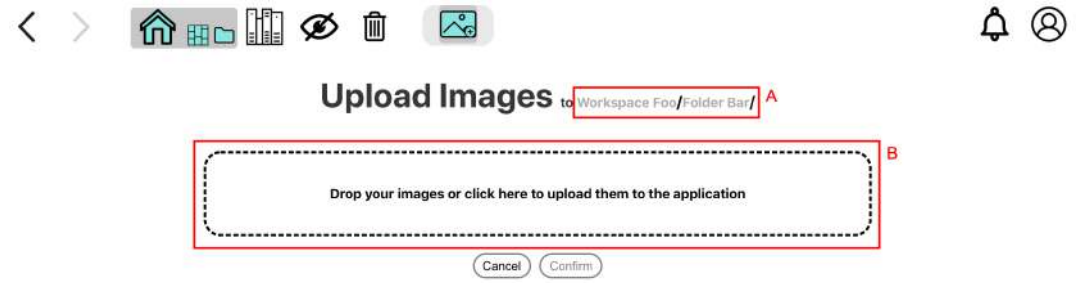


Figure 3.15: The photo|uniq web-based application's Image Upload View prior to the addition of images.

Initially, the view is extremely simple, with only two interactive elements (if one does not count those contained in the application header): the "Cancel" button, which can be pressed to return to the previously viewed location (in the case of Figure 3.15, "Folder Foo", which is contained within "Workspace Bar"), and the image input field (identified with **B** in Figure 3.15).

The view also contains a visual guideline next to the page title, identified with **A** in Figure 3.15. This guideline allows the user to visually confirm the location to which the images will be added to. This visual guideline first identifies the parent workspace to which the images will be added to ("Workspace Foo" in Figure 3.15), and then the parent folder, if there is one[4] ("Folder Bar" in Figure 3.15), separated by forward slashes.

[4]In the current version of the photo|uniq application, images can be uploaded directly to a workspace or

The process of adding images to the application can be started by either dragging and dropping images directly from the user's desktop to the image input field (identified with **B** in Figure 3.15), or by clicking this field and then selecting the images to add from the machine's file system. After any images are added to this view, its UI is changed to that which is shown in Figure 3.16.

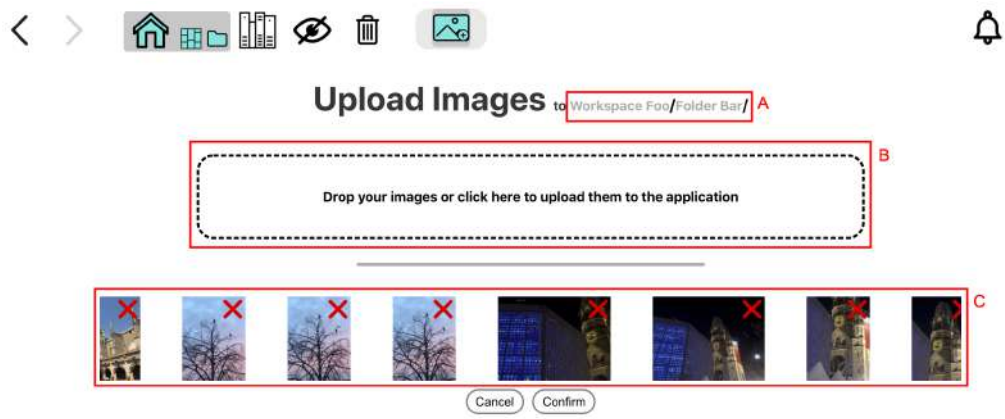


Figure 3.16: The photo|uniq web-based application's Image Upload View after the addition of images.

After adding images to the upload images view, the user can see thumbnails of these added images in the image preview section, identified with **C** in Figure 3.16. These previews/thumbnails are all displayed in a single row, which can be scrolled horizontally by the user to see all the images that are being uploaded to the location shown in **A**. If the user wishes to remove (or, in other words, not upload) one of these images, they simply need to press the red cross that is shown in the top right corner of every thumbnail. They can also add more photos by following the same procedure that was explained previously.

When the user has added all their desired images to this view, they can then hit the "Confirm" button, which is initially disabled, but becomes enabled after the user has added at least one image. After pressing this button, a confirmation dialog box will be shown to the user to ensure that they wish to upload the added images to the photo|uniq application and, if the user responds affirmatively, these images will then be turned into workspace photos and added to the location that is shown in **A**. The user will then be redirected to the page associated with this location (in the case of Figure 3.16, to the page where the contents of "Workspace Foo/Folder Bar" are shown).

3.3.5 Photo Visualization

The users of the photo|uniq web-based application are able to visually analyse all the photos that they have uploaded to the application. The user interface of the view that is generated for these purposes is shown in Figure 3.17.

workspace folder, after which they will be considered to be workspace photos.

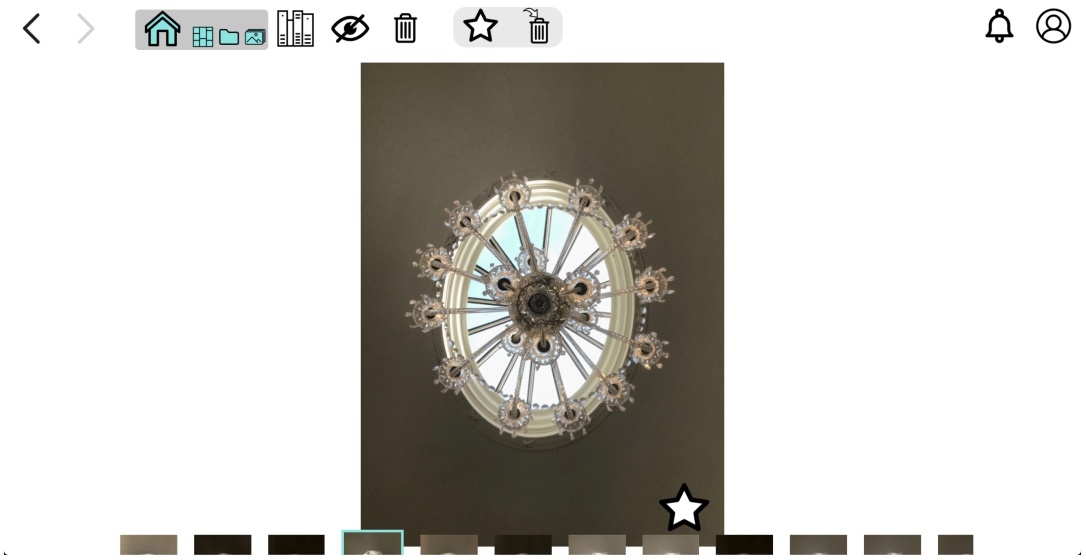


Figure 3.17: Example of the photo|uniq web-based application's photo visualization view.

This user interface is extremely simple, with only two main parts: the main photo which is being analysed and is centered on the screen, and its "sibling" photo list (or, in other words, the list of thumbnails of the photos that are contained on the same level as the main photo), displayed on the bottom of the screen.

When it comes to the main photo, the user can apply basic zoom and panning operations to better visually analyse it. The zoom level of the photo is operated by scrolling while the user's mouse pointer is hovering the photo: they can zoom in on a point by scrolling down while their mouse pointer is hovering that point and, conversely, they are also able to zoom out by scrolling up while the mouse pointer is above the photo. After zooming in on the photo, the user can also pan on it by clicking on the photo and executing a dragging motion in the same direction they want the focus point to go to (for example, the user can move the mouse pointer to the left in order to pan the photo to the left).

The sibling photo list is mostly hidden while in its default state, and can only be fully seen when the user moves their mouse pointer to the bottom of the page. When they do so, the sibling photo list rises in order to show the complete thumbnails of the main photo's sibling photos, as shown in Figure 3.18.

This photo list displays the thumbnails of the sibling photos in a single row that can be scrolled horizontally by the user.

The photos displayed in this list are all contained in the same level and have the same parent (e.g., workspace, workspace folder or similarity group) as the main photo that is being displayed. The main photo is also represented in this list, and can be identified by its light blue border (in the case of Figure 3.18, this can be seen in the fourth photo shown in the list).

The user can click on any of the sibling photos shown on the list (other than the

Figure 3.18: The photo|uniq web-based application's photo visualization view when the mouse pointer is hovers the sibling photo area.

main one that is currently being displayed) in order to switch the focus of the photo visualization view to that sibling photo (thus making the photo whose thumbnail was clicked become the main photo).

The user interface of this view was also designed to have some support for keyboard-based controls. If the user presses the right arrow or left arrow keys on their keyboard, the main photo will be switched to the sibling photo shown in the sibling photo list to the immediate right or left position of the previous main photo. They can also hit the escape key to return to the page which shows the main photo's parent element's contents.

If the photos that are shown are contained in a similarity group, the user can also access three additional functionalities:

- They can use drag-and-drop controls to manually sort the workspace photos contained in this group by pressing and holding on a sibling photo, dragging it to the new position they wish to place it at, and then releasing the photo. This is particularly useful since it allows the user to sort the photos contained in a similarity group while visually analysing them.

- They can change the main photo's favorite status (or, in other words, mark an un-favorite photo as favorite, or mark a favorite photo as un-favorite) by pressing the button found on the bottom right of the main photo (as seen in Figure 3.17), or by hitting the enter key.

- They can also access the Photo Comparison page (presented in Section 3.3.6) by clicking on a sibling photo while also holding the alt key on their keyboard. Doing so will redirect the user to a page which will allow them to visually compare the previous main photo with the sibling photo that they clicked on.

3.3.6 Photo Comparison

If two workspace photos are contained in the same similarity group, the users of the photo|uniq web-based application can access a page which allows them to visually compare them using side-by-side image comparison. The page used to perform this comparison has the user interface shown in Figure 3.19.

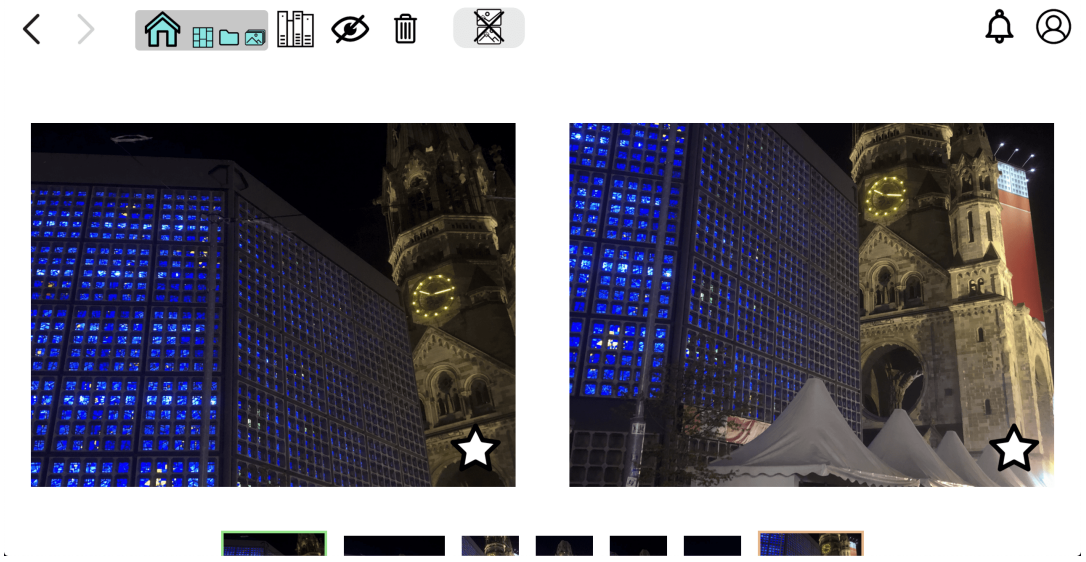


Figure 3.19: Example of the photo|uniq web-based application's photo comparison view.

This user interface is extremely similar to the one that was presented in Section 3.3.5, aside from a few differences that were caused by the process of adapting the UI to the use case of visually comparing two images.

The most relevant difference is, of course, the existence of two main photos, as opposed to a single one. In order to ensure the consistency of the relative positioning of the two main photos, we made it so their positional relation in the sibling photo list would always be maintained in their main photo displays (or in other words, the first/leftmost main photo will also always be displayed on the left of the second/rightmost main photo in the photo sibling list).

Similarly to the interface discussed in Section 3.3.5, the user can press the buttons found in the bottom right of either shown photos to toggle that photo's favorite status.

They can also zoom and pan on either one of the main photos using the same gestures that were explained in Section 3.3.5, with these gestures producing the same effects and results on that photo as those that were previously presented. However, there are also some side effects which occur to the photo that was not interacted with.

If the user zooms in or out of one photo, then the resulting scale changes will be applied on both photos.

If the user pans on one photo, one of two outcomes will occur to the other one:

- If the two photos are similar enough to have homography matrices calculated between them and the user chooses to use these homography matrices, then they will be used so that the center point of both displayed photos will always be focused on the same motif (even if this motif has different image coordinates in the two photographs). An example of this scenario can be seen in Figure 3.20(a).

- If the two photos are not similar enough to have homography matrices calculated between them or the user has chosen to disable the use of homography matrices, then the two images will simply be focused on the point with the same exact image coordinates, without the use of any homography matrix to translate these coordinates (which means that the two images are not guaranteed to be focused on the same motif). An example of this scenario can be seen in Figure 3.20(b).

In order to toggle the use of the homography matrices on and off, the user can press the button marked with the number 1 in Figure 3.20.

The existence of two main photos also caused some changes to the visual appearance of the sibling photo list, as well as to some of the ways that the user can interact with it.

The borders of the thumbnails of the two main photos have different colours, with the leftmost main photo having a light green border and the rightmost main photo having a light orange border.

The sibling photo list can also be used to change which photos are being compared. This can be done in one of two ways:

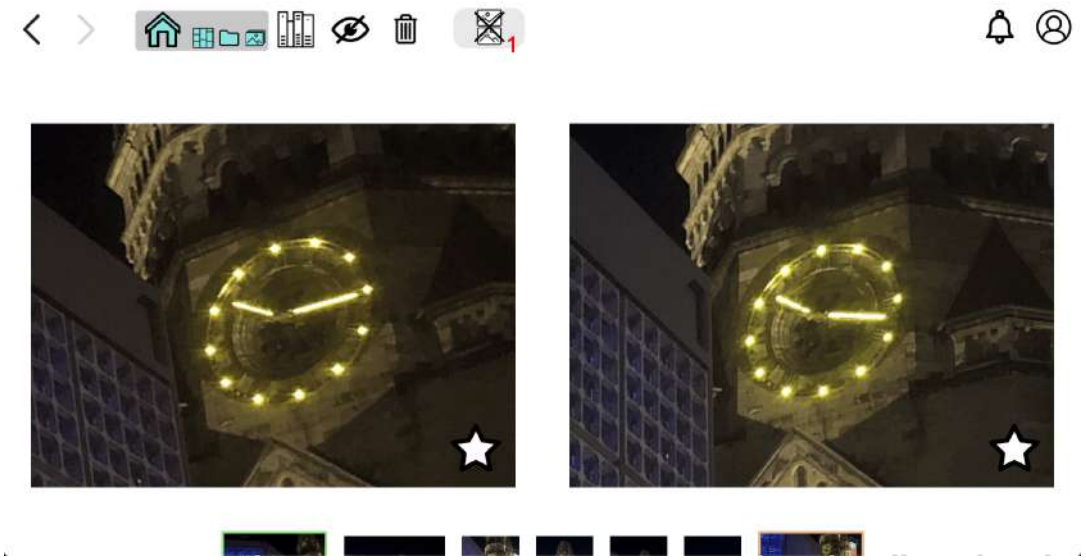
- If the user clicks on the thumbnail of an unselected sibling photo while also holding the alt button on their keyboard, then the leftmost main photo will be replaced with the clicked photo (thus starting a comparison between the current rightmost main photo and the clicked sibling photo);

- If the user clicks on the thumbnail of an unselected sibling photo without pressing the alt button on their keyboard, then that sibling photo will instead replace the rightmost main photo in the current comparison.

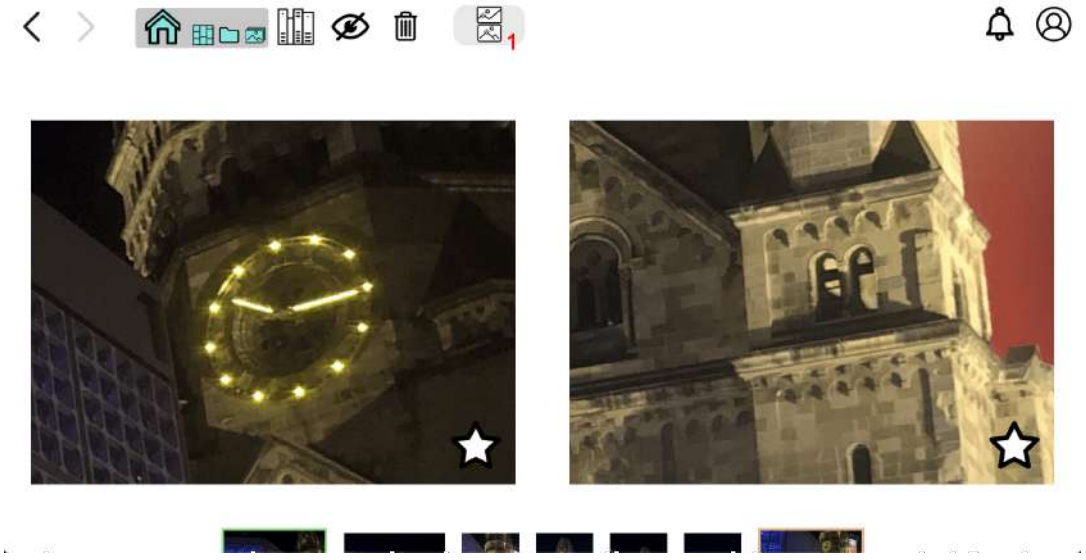
Since only photos contained in a similarity group can be compared using the photo comparison view, the sibling list can always be used to manually sort the workspace photos contained in this group by following the steps that were explained in Section 3.3.5. If the process of changing the position of a photo causes a change in the relative positional relation of the two main photos, then the photos will also change their positional relation on the main display, thus maintaining the aforementioned positional consistency between the main photos and the sibling photo list.

This UI also contains some instances of keyboard support similar to those found in the photo visualization page, with some adaptation being performed to these controls in order to accommodate the existence of two main photos:

- Pressing the enter key by itself will toggle the leftmost main photo's favorite status;



(a) The photo|uniq web-based application's photo comparison view while zoomed on the same motif on both photos (through the use of homography matrices).



(b) The photo|uniq web-based application's photo comparison view while zoomed on the same absolute image coordinates on both photos (without the use of homography matrices).

Figure 3.20: Visual display of the capabilities of both comparison modes provided by the photo|uniq web-based application's photo comparison view.

- Pressing the enter key in conjunction with the alt key will toggle the rightmost main photo's favorite status;
- Pressing the right or left arrows by themselves will switch the rightmost main photo with the first photo to the right or left of that main photo in the sibling photo list that is not currently shown;
- Pressing the right or left arrows while also pressing the alt key will switch the leftmost main photo with the first photo to the right or left of that main photo in the sibling photo list that is not currently shown;
- Pressing the escape key will redirect the user to the page showing the contents of the similarity group which contains the shown photos.

3.4 Application Functionalities

Other than the development of the photo|uniq web-based application's UI, the other main aspect of this work was to implement a functional web-based version of the photo|uniq application that would be able to provide its users with tools that could be used to identify similar images and compare them amongst themselves, in order to help the user decide which should be kept or deleted.

For these purposes, a web-based application was developed, which is capable of providing several different functionalities and operations to its users. In this section, we will introduce and elaborate on the several different functionalities that make up the photo|uniq web-based application in its current state and show how these functionalities work in conjunction with each other to solve the stakeholder's problem, as it was identified and explained in Section 1.2.

To do this, we will discuss those functionalities that are more specific and unique to our application. Additionally, the remaining functionalities that are more generic or whose implementation was less distinct from other applications are presented and discussed in Appendix C.

3.4.1 Workspace-related Element Management

The bulk of the photo|uniq functionalities and operations can be found while accessing the application's workspace-related elements (following the definition set in Section 3.2.1), as these are the elements which allow the user to analyse their uploaded photographs and decide which ones should be kept or deleted. In this section, we will present the several operations that the user can access and execute in order to create, update, and delete these elements.

As previously explained in Section 3.2, there are four different types of workspace-related elements. Looking at them from a bottom-up perspective, these are:

- Workspace photos, which represent the images that the user has uploaded to the application and that have not yet been promoted to the photo|uniq gallery through the process that is explained in Section 3.4.5;
- Similarity groups, which contain similar workspace photos so that the user may compare these photos more easily with each other;
- Workspace folders, which can contain similarity groups and ungrouped workspace photos, and serve as the element in which photos can be stored before they are aggregated into similarity groups;
- Workspaces, which are the top level component and can contain both workspace photos and workspace folders.

These elements can be created by the user by way of a few different actions.

The creation of workspaces can be done while the user is viewing the photo|uniq main page, and only requires that they provide a name that is not currently in use by another workspace.

Workspace folders work in a similar way, with the user being able to create a new workspace folder while viewing the contents of a workspace, and only needing to provide a name that is not used by another workspace folder contained in that workspace.

When it comes to similarity groups, these elements can be created in one of two ways: the user can either manually create a similarity group while viewing the contents of a workspace folder, or they can be automatically created during the process of automatic image grouping (which is presented and explained in Section 3.4.2). In both of these cases, the name given to these similarity groups will be automatically generated by a concatenation of the string "Similarity Group " with the value that is currently stored in the *similarityGroupNumberCounter* variable of the parent workspace folder (which is initialized at 1 and is incremented whenever a similarity group is created).

Finally, the workspace photo elements are created when the user uploads images to the application, via the upload page that is presented in Section 3.3.4. While workspace photos can be contained in all three of the other presented elements, upon their creation they will always be stored directly in the root of a workspace or workspace folder, since the user is not allowed to directly upload images to a similarity group.

After their creation, the user can also interact with these elements. While doing so, they can view their contents (or, in the case of the workspace photos, view the photos themselves) and, if dealing with non-deleted versions of these elements, call upon certain operations that can affect the state of these elements and their contents. An operation that affects the state of any element will update the information that is stored in the object representation of that element in both the application's session and persistent storage.

When it comes to the workspace photos, the user can move them in between elements and, if they are contained in a similarity group, change the order in which they are sorted in that group and mark them as favorites or non-favorites.

When it comes to the process of moving workspace photos between elements, these photos can only be moved between elements which are directly connected. In other words, workspace photos can be moved from the root of a workspace into the root of a workspace folder, from the root of a workspace folder into the root of a workspace, from the root of a workspace folder into a similarity group, and from a similarity group into the root of a workspace folder.

The user can also combine the action of creating a non-top level element (i.e., an element that is not a workspace) with the action of moving workspace photos into a lower level. This essentially means that the application will create a workspace folder or similarity group and, immediately after this, move the workspace photos that they currently have selected into this newly created element. This combination of operations will be executed if the user requests the creation of one of these elements after they have selected at least one workspace photo.

When viewing the workspace photos contained in a similarity group, the user can change the order that those photos are sorted in by clicking and dragging them to another position. Doing so will change the *orderedImageIds* array that is contained in the object that represents the parent similarity group, and also change the order upon which the photos are stored in the application's session storage. If this change in order affects the photo that is stored in the first place of the similarity group, then the group thumbnail will also be changed to become the workspace photo whose id is in the first position of the *orderedImageIds* array after the operation is completed.

If a workspace photo is contained in a similarity group, the user can also mark that photo as "favorite", or if they already are marked as such, they can be marked as "non-favorite". Workspace photos that are marked as favorite will be promoted to the photo|uniq gallery after the user has completed the selection process of the images contained in a similarity group (as opposed to the non-favorite photos, which will instead be deleted).

When the user has completed the selection process of the similar images contained in a similarity group (i.e., marked as favorites the images which should be kept), they can also mark that similarity group as processed. Any group that is marked as processed will be considered for the operation of promoting workspace photos into the photo|uniq gallery (which is further explained in Section 3.4.5).

If a group is marked as processed, the user is also able to manually mark it as non-processed if they wish to do so. Similarity groups will also be marked as non-processed if any change is performed upon their contents (i.e., if a photo is added to or removed from the group, if the order of the photos contained in this group is changed, or if the favorite status of any of the photos contained in the group is changed).

It is also possible to delete and undelete every workspace-related element type.

The process of deleting an element is done by creating an object that represents the deleted version of that element, followed by the removal of the non-deleted version of the element from the application's session and persistent storage.

If any one element is deleted, then all the elements that are contained inside that

element (if any) will also be deleted, and a deleted representation of every higher-level element that contains that element (either directly or indirectly) will also be created, if one does not already exist. For instance, if the similarity group found in "workspace1/folder2/group3" is deleted, then every photo contained in "group3" will also be deleted, and a deleted representation of "workspace1" and "folder2" will also be created for display and organization purposes, if one does not already exist.

This effectively means that all elements other that the workspace photos can simultaneously have deleted and non-deleted representations at any one time.

If a workspace photo contained in a similarity group is deleted, then that photo's id will be removed from the group's *orderedImageIds* array. If this deleted photo's id held the first position in the *orderedImageIds* array prior to its deletion, then this deletion will also change the group's thumbnail so that it becomes the workspace photo whose id previously held the second position in the array (or, if there was no such photo, the group's thumbnail will instead become undefined).

The process of undeleting an element is extremely similar to the one used to delete an equivalent non-deleted element, with the main difference being the reversion of the objects that are added to and removed from the application's session and persistent storage (or, in other words, a non-deleted version of the object is added and the deleted version of the object is removed).

If a deleted workspace photo that was previously contained in a similarity group is undeleted, then that photo's id is added to the end of the group's *orderedImageIds* array. If the group did not contain any workspace photos when this was done, then that photo will become the group's new thumbnail.

It is also worth mentioning that the user can empty the deleted elements hub, which removes all deleted elements from the application's session and persistent storage.

3.4.2 Automatic Image Grouping

One of the key selling points of the photo|uniq application is its ability to automatically identify similar images (or, in other words, images that capture the same motif), and aggregate them into similarity groups, so that the user can more easily and quickly compare these similar images between themselves.

This automatic grouping operation can be called by the user when they are viewing the contents of a workspace folder. If they do so, any images that are not contained in a similarity group will be moved into an already existing group, or a newly created one.

Several steps must be taken by the application in order to execute this operation:

1. First, we begin by detecting the keypoints and descriptors for every ungrouped image contained in the folder, using the ORB feature detection algorithm [70].

2. After collecting the keypoints and descriptors for every ungrouped image, we need to do the same for the images that will be used to represent the already existing

groups that contain at least one workspace photo. For these representation purposes, we chose to use the groups' thumbnails (which are also the workspace photos that are currently sorted in the first position of these groups).

3. Thirdly, for every ungrouped image:

   a) We use the Brute-force feature matching algorithm [51] to correlate the keypoints of the ungrouped images with those of every group representative, and see how many matching keypoints are identified.

   b) If there are enough matching keypoints between an ungrouped image and a group representative to consider them similar, we see if this number of matching keypoints is higher than the previous maximum number of matching keypoints. If this is the case, then we will keep information regarding the group to which the representative belonged to, and change the maximum number of matching keypoints to be this new value.

   c) After comparing the ungrouped image with every group representative, we mark the image to be moved to the group whose representative had the highest number of matching keypoints when compared to the ungrouped image. However, if an ungrouped image is found to differ from all of the existing group representatives, then this image will be considered dissimilar to every existing group and will be used instead as the representative of a new group (which means that the following ungrouped images will also be compared to this image to see if they are similar).

4. Finally, we act upon the conclusions that were reached in the previous step. Every previously ungrouped image is either moved into the already created similarity groups that contain images similar to them, or new similarity groups are created that are them populated with the images that were identified as similar.

The first three presented steps are performed in the application's photo processing component[5], with the fourth one being performed in the main application component.

This operation requires a few relatively heavy computations, in particular those related to the detection of image keypoints and descriptors. Because of this, it has the potential to take a lot of time to complete its execution, with the execution time increasing with the number of images that are being analysed. In order to try to reduce this computation time, some measures were taken to improve the temporal efficiency of the keypoint detection process, while still maintaining the effectiveness of the operation.

In the current version of the photo|uniq web-based application, the image keypoints are detected using a version of the image that is about 9% of the image's original size, which substantially reduces the time that is required to detect the image keypoints while still allowing for an accurate process of keypoint detection and correlation.

[5] See Section 3.1 for further information on the three different components of the photo|uniq application.

We also added two extra steps to the presented process which allow the application to store the detected image keypoints and descriptors in an *ImageKeypointDetectorInfoObject*[6] object. After doing so, this object can then be loaded at a later point to avoid the repetition of unnecessary computations related to keypoint detection after the first time that this process is applied on an image.

Taking this change into account, before starting the process of keypoint detection on any one given image, the application first tries to load from its persistent storage an *ImageKeypointDetectorInfoObject* object associated with that image, and only proceeds to keypoint detection if no object is found. Then, after the process of detecting the image's keypoints and descriptors, the application stores these values in the persistent storage so they can be loaded and used at a later time, if necessary.

3.4.3 Automatic Image Quality-based Sorting

Another key functionality of the photo|uniq application is the ability to automatically sort the images contained in a similarity group based on their objective quality, without the need for any user input.

The user can access this functionality by pressing a button found in the functionality portion of the application header[7] while viewing the contents of a similarity group that contains at least two workspace photos.

When it comes to the main photo|uniq web-based application component[8], the process of automatically sorting workspace photos based on their quality can be separated into three different steps:

1. Firstly, the main application component makes a request to the application photo processing component, asking the component to analyse and sort the workspace photos contained in the viewed similarity group based on their objective quality;

2. Following this step, the application photo processing component returns an array containing the ids of the provided workspace photos sorted based on their objective quality, from highest to lowest;

3. Finally, the application's main component changes the order of the workspace photos contained in the viewed similarity group to replicate that which is contained in the array that was returned by the application photo processing component, and changes that group's thumbnail to be the photo whose id is stored in the first position of the array.

The execution of this operation also has the side effect of marking the group as unprocessed, in order to ensure that the user takes the new order of the workspace photos (or, in

[6] See Section 3.2.4 for further information on this class.
[7] See Section 3.3.2
[8] See Section 3.1 for further information on the three different components of the photo|uniq application.

other words, their objective quality relative to each other) into account before promoting the contents of this group to the photo|uniq gallery.

By implementing this operation following the presented steps, we made it so the grading and sorting of the images based on their objective quality is completely contained in the application photo processing component (and thus, separated from the main application). This means that the process through which this grading occurs can be changed at any time to better suit the application's requirements, without any need to change the implementation of the main application component.

As was previously discussed in Section B.3, in the current version of the photo|uniq web-based application's photo processing component, the process of determining an image's objective quality is done using a very simple metric that only takes into account the overall focus level of the images that are being analysed.

This was chosen for our current implementation mostly as a straightforward option that would allow us to have a simple and functioning algorithm that could be developed quickly, and thus would allow us to continue the work on what was the main focus of the application: the development of the main component of the photo|uniq application.

In order to calculate the images' overall focus level, we chose to use the laplacian-based variance operator [57] that is described in further detail in Section B.3.

Following this method, the current implementation of the photo|uniq photo processing component automatically sorts the images contained in a similarity group by:

1. Applying the laplacian operator to of these images and, immediately after this, calculating the variance of the resulting matrices (this variance can then be used as a numeric representation of the overall focus level of an image);

2. Sorting these focus levels from highest to lowest;

3. Creating a string array containing the ids of the analysed images ordered by their focus levels, and returning this array to the main application component.

Similarly to what was previously talked about in Section 3.4.2, this operation also had a risk of taking a lot of time to complete, with its execution time increasing along with the number of workspace photos that were contained in the similarity group. Because of this, and in order to make this process more efficient by avoiding the need to repeat the computations used to calculate the numeric value of the images' objective quality grade, we chose to store this value for later use in the application's persistent and session storage, using a *FocusSortSupportObject*[9] object.

Therefore, in the current version of the application, before the calculation of the image overall focus level, the photo processing component first tries to find a *FocusSortSupportObject* object associated with that image in its session and persistent storage.

[9] See Section 3.2.4 for further information on this class.

If this object is found, then the application will use its *focusLevel* variable to sort the image based on its overall focus level. If, however, no such object exists in either the application's session or persistent storage, the application will proceed with the computation of the image's laplacian matrix and calculation of that matrix's standard deviation, after which a *FocusSortSupportObject* containing this standard deviation will be stored in the application's session and persistent storage to be loaded and used at a later time.

3.4.4 Side-by-side Image Comparison

If two images are contained in the same similarity group, the user can compare them side-by-side using the photo comparison page that is presented in Section 3.3.6.

One of the main particularities of the photo|uniq web-based application's side-by-side image comparison is its use of homography matrices to automatically synchronize the zoom and panning actions, so that both photos are centered in the same object, even when one of the photos suffers from a translation, rotation, scale difference, *etc.*, with respect to the other photo.

Before the application to accesses the photo comparison page, it will first load the homography matrices from its session and persistent storage, so that they can be used to map point coordinates between the two photos being compared. If the necessary homography matrices are not found, the application will compute them.

The process through which a homography matrix is calculated in the photo|uniq web-based application is performed in the photo processing component of the application[10], and can be broken up into a few different steps.

First, we start by obtaining the image features of both images, either by loading them from the application's persistent storage, or by performing keypoint detection using the ORB keypoint detection algorithm [70] (in which case, we will perform this keypoint detection in scaled down versions of the images, and will store the resulting keypoints and descriptors in the application's persistent storage, as explained in Section 3.4.2).

After obtaining the features for both images, we then perform feature correlation between the two images using the Brute-force feature matching algorithm [51].

At this point, one of two scenarios will occur: either there are enough matching keypoints to calculate the homography matrix, or there are not. If there are not enough matching keypoints, then the photo processing component will assume that the two images are not similar, and thus will not attempt to calculate their homography matrix. It will instead return a value of *null* to the main application component, which will signify that the application should not attempt to use a homography matrix for the translation of point coordinates between the two provided photos. If, however, there are enough matching keypoints between the two images, then the photo processing component will estimate the homography matrix using the matched keypoints.

[10] See Section 3.1 for further information on the three different parts that make up the photo|uniq application.

After obtaining this homography matrix, there is still an extra step that must be done before it is ready for use. Since the homography matrix was estimated using keypoints that were detected in scaled down versions of the provided photos, it needs to be re-scaled in order to be applicable to the photos in their original size.

In order to do this, we simply need to execute the following formula [43]:

$$H' = \begin{bmatrix} H_{11} & H_{12} & \frac{H_{13}}{S} \\ H_{21} & H_{22} & \frac{H_{23}}{S} \\ H_{31} \times S & H_{32} \times S & H_{33} \end{bmatrix}$$

where H' represents the re-scaled homography matrix, H_{xy} represents the numeric value found in the x row and y column of the scaled down version of the homography matrix, and S represents the scaling factor which was used to scale down the photos[11].

After the re-scaled homography matrix is calculated and returned to the photo|uniq main application (or *null*, if it was not possible to calculate the homography matrix), this returned value is immediately stored in both the application's session and persistent storage for later use, so that the application will not have to needlessly repeat the computations and calculations that are necessary for the estimation of these matrices.

Having access to the homography matrices that can be used to translate image coordinates between the two photos that will be compared, we can then start the process of synchronizing the zoom and pan operations in both of the displayed photos.

The synchronization of the zoom level on both images was extremely simple to implement: we simply needed to make it so whenever the user zooms in or out on an image, the resulting zoom level is applied on both images. By doing this, we made sure that the two images would always be zoomed using the same scale.

The synchronization of the center point of the image displays is done through the use of a more complex algorithm.

The first thing that should be explained is the two different types of image panning synchronization that the user has access to[12]:

- If the two photos are similar enough to have homography matrices calculated between them and the user chooses to use these homography matrices, then they will be used so that the center point of both displayed photos will always be focused on the same motif (even if this motif has different image coordinates in the two photographs). An example of this scenario can be seen in Figure 3.20(a).

- If the two photos are not similar enough to have homography matrices calculated between them or the user has chosen to disable the use of homography matrices, then the two images will simply be focused on the point with the same exact image

[11] It is worth mentioning that this formula is adapted from the original one presented by Mano [43] due to the photo|uniq web-based application using the same scaling factor for both images.

[12] These explanations are a repetition of the ones that are given in Section 3.3.6.

coordinates, without the use of any homography matrix to translate these coordinates (which means that the two images are not guaranteed to be focused on the same motif). An example of this scenario can be seen in Figure 3.20(b).

As explained in Section 3.3.6, the user can toggle the use of homography matrices on and off (or, in other words, switch between the two presented scenarios) by pressing a button found on the functionalities portion of the application header that is provided by the application while performing side-by-side image comparison.

This option to not use the homography matrices is provided to the users so that they have a functional way to compare the two photos even if the computed homography matrices are invalid (which is always a risk, especially if the user tries to compare images which are not in fact similar, but have enough false-positive matching keypoints for the algorithm that is used to calculate the homography matrices to think that they are).

While the use of homography matrices is turned off, whenever the user executes a panning operation in one image, that same operation is also applied to the other one.

If, however, the user chooses to use homography matrices, then this synchronization process has a few extra steps associated with it. Before we present these steps, it is important to define a few different concepts in order to facilitate the discussion of the performed operations: original image coordinates and display image coordinates.

For the purposes of this section, original image coordinates are the coordinates used to represent a point in the image in its original resolution. Conversely, display image coordinates correspond to the coordinates of a point in regards to the visual display of the image that is being shown to the user (which will more often than not be in a different resolution from the original image resolution).

It is also important to establish that the package[13] that is used to provide the zoom and panning capabilities to the application's UI uses the top left corner point to represent the result of the image translate operations (or, in other words, it returns the display image coordinates of this top left corner point to the application when the user performs a zoom or panning operation).

We should also state that, for the purposes of this section, we will consider the top left point to be the origin point for both the display image and original image coordinates.

Taking these definitions and constraints into account, we will now begin the presentation of the steps that are taken by the application when using homography matrices to translate image coordinates between two photos.

When the user performs a panning operation, the application will have access to the display image coordinates of the top left corner point of the image that was panned.

Since we wish to make the **center** point of the two images contain the same motif, the first step is to translate the display image coordinates from the top left point to the center point of the display in which the user performed the panning operation.

[13] This package is further elaborated upon in Section D.6.

To get the display coordinates of the center point, we simply need to add half of the display's width and height to the x and y coordinates of the top left point (making the coordinates of the point (X+$\frac{Width}{2}$,Y+$\frac{Height}{2}$)).

After this, the next step is to translate the display image coordinates of the center point to the original image coordinates of the point that is shown in that center point. To do this, we need to first calculate the scaling ratio between the original image resolution and the image display resolution. This can be done by following the formula:

$$S = \frac{Height_{\text{Original}}}{Height_{\text{Display}} \times Z}$$

where S refers to the scaling factor, $\text{Height}_{\text{Original}}$ refers to the height of the image in its original resolution, $\text{Height}_{\text{Display}}$ refers to the height of the image in the resolution that is currently displayed to the user, and Z refers to the zoom value that currently applied on the image.

We can then calculate the original image coordinates of the center point of the displayed image by multiplying the display image coordinates by the scaling ratio:

$$(X, Y)_{\text{Original}} = (X_{\text{Display}} \times S, Y_{\text{Display}} \times S)$$

with $(X,Y)_{\text{Original}}$ referring to the original point coordinates, S referring to the previously calculated scaling factor, X_{Display} referring to the X coordinate of the display image coordinates, and Y_{Display} referring to the Y coordinate of the display image coordinates.

After obtaining the original image coordinates of the center point, we can then apply the homography matrix to these coordinates (using the process that was explained in Section B.2), in order to obtain the original image coordinates for the equivalent point on the second image.

After obtaining the original point coordinates of the equivalent point in the second image, we simply need to do the inverse of the calculations that were applied in the first two steps, so as to obtain the display image coordinates of the top left corner point of the second image. Using these point coordinates in conjunction with the display point coordinates that were returned by the package at the beginning of this process, we can then make the display of both images be centered on equivalent points.

By completing this set of steps, the application will have access to the display image coordinates of the top left corner point of both images which allow the application to display equivalent center points for both images. These coordinates can then be used as parameters for the *TransformWrapper* function that is provided by the react-zoom-pan-pinch [67] npm package[14].

3.4.5 Promoting Workspace Photos to Gallery

After the user has aggregated their uploaded workspace photos into similarity groups, flagged the best photos in these groups as favorites (or, in other words, decided which

[14]For further information on this function and package, consult Section D.6.

images should be kept or not), and marked the similarity groups as being processed, they can begin the procedure of promoting the favorite photos contained in these groups to the photo|uniq gallery.

The operation of promoting workspace photos into the photo|uniq gallery is composed by a few different steps, and can be accessed while the user is viewing the contents of a workspace folder that contains at least one processed similarity group.

The first part of this process is the loading or creation of the album and album folders that will contain the gallery photos resulting from the promotion of the workspace photos. These album and album folder elements will have the same names as the workspace and workspace folder elements that contain the similarity group that houses the workspace photos prior to their promotion.

In order to complete this step, the photo|uniq web-based application first tries to find the necessary album and album folder elements from its session storage, and then tries to load them from the persistent storage if they were not found. If they also do not exist in the application's persistent storage, the application will then create these elements and store them in the session and persistent storage, as they are necessary for the creation of the gallery photos that will represent the favorite workspace photos after their promotion.

After the album and album folder elements are created or loaded to the application's session storage, the application looks at every image contained in the processed similarity groups that are found in the viewed workspace folder. For every workspace photo in these groups, one of two things will happen:

- If a photo has been marked as favorite by the user, then it will be "promoted to the photo|uniq gallery", which effectively means that a gallery photo will be created to represent the previously existing workspace photo, and this previously existing workspace photo will be removed from the application's persistent and session storage.

- If a photo has not been marked as favorite by the user, then it will be deleted by following the steps explained in Section 3.4.1.

Finally, the application will then remove the non-deleted version of every processed similarity group (as well as their contents) from its session and persistent storage.

After this process if completed, the user can then view the promoted photographs in the photo|uniq gallery by following the same path that they use to see the workspace folder whose processed similarity groups were promoted. For instance, if the promotion process is executed while viewing the workspace folder found by following the path "Foo/Bar/" after using the photo|uniq application home page as a starting point, then the gallery photos resulting from this promotion can be found in the album folder found by following the path "Foo/Bar/" after accessing the photo|uniq gallery.

3.4.6 Download Gallery Contents

After the promotion of the favorite photos into the photo|uniq gallery, it is natural for the user to wish to store the results of their work into their computer's file system.

For these purposes, the application contains a function which, when called upon, will generate a zip file containing the gallery-related elements of the application, and then trigger a download of this zip file into the user's local computer storage.

In terms of implementation, this function is mainly contained in the application's data management component[15], as the main application component simply issues a call to the function when the user requests the aforementioned download.

In the currently implemented version of the application's data management component, the generation of the zip file is done using the JSZip [31] npm package, which is further explained in Section D.7. The method through which the zip file is constructed and generated follows a simple algorithm:

1. First, an empty instance of JSZip is created, which will serve as a blueprint through which a zip archive containing the gallery-related elements will be generated.

2. Then, a folder called "photo|uniq_gallery" is created in the root of this JSZip.

3. After this, the application's data management component retrieves all albums currently contained in the photo|uniq gallery. Then, for every one of these albums:

    a) The application data management component retrieves the data related to all photos and folders directly contained in the root of this album.

    b) If there is at least one photo or folder contained in this album, then a directory with the album's name is created inside of the "photo|uniq_gallery" folder which was created in the second step of this algorithm. If not, nothing else happens related to this album, and the algorithm proceeds to the next album that was loaded by the application's data management component.

    c) For every gallery photo contained directly in the root of the album in question, an image file is created inside the root of the directory that was created in the previous step.

    d) Then, for every album folder contained in this album:

        i. The application data management component retrieves the data related to all the gallery photos contained in this album folder.

        ii. If there is at least one gallery photo contained in this album folder, then a directory with the album folder's name is created inside of the directory that is associated with its parent album in the JSZip instance. If not, then nothing else happens related to this album folder, and the algorithm

[15] More Information on the several application components can be found in Section 3.1.

simply moves over to the next album folder that was loaded by the application's data management component.

iii. For every photo contained in the folder in question, an image file is created inside the root of the directory that was created in the previous step.

4. Finally, after adding every album, album folder, and gallery photo contained in the photo|uniq gallery to the instance of JSZip which was created in the first step of the algorithm, this instance is then used as a blueprint to generate a zip file following the aforementioned path structure.

After generating the zip file containing all of the user's gallery elements, the application will then trigger a download of this file using an artificial click on a hidden HTML anchor element containing a reference to the generated file.

It should also be mentioned that at any point before or after the download of the contents of the photo|uniq gallery, a button is also provided to the user which, when pressed, will trigger a function that can be used to clear the gallery-related data from the application's persistent and session storage.

3.5 External Packages and Libraries

During the development of the web-based photo|uniq application, some functionalities were required which were not provided by the base TypeScript language, nor the React library. Because of this, and in order to complement these base capabilities, we used seven external packages and libraries:

- UUID [88] was used to generate the unique ids of the elements that were accessed and used by the application (e.g., workspaces and photos);
- IDB [27] allowed the application to more easily write to and read from the browser's IndexedDB, and integrated these accesses with TypeScript Promises (which allow for a higher and much more straightforward control over the asynchronous nature of IndexedDB);
- ts-image-processor [84] was used to generate thumbnail versions of the photos uploaded by the user into the application;
- OpenCV.js [55] allowed the web-based photo|uniq application to achieve the image analysis capabilities which were essential to allow it to deliver some of its key functionalities to its users (such as automatic image grouping and automatic image sorting based on their quality);
- React Sortable HOC [66] was used to improve the UX and UI of the sorting of photos contained in similarity groups, by turning these photos into draggable elements, which could then be rearranged by the users using smooth and responsive controls;

- react-zoom-pan-pinch [67] was used to allow the users to achieve fluid zoom and panning capabilities in the photo visualization and comparison pages (which were discussed in Sections 3.3.5 and 3.3.6 respectively);
- JSZip [31] was used to create zip files and populate them with the contents of the user's photo|uniq gallery, so that this zip file could then be downloaded by the user, thus passing the results of their image selection work into their file system.

A more elaborated discussion of these packages and how they were used for the development of the photo|uniq web-based application can be found in Appendix D.

4

Testing and Validation

As explained previously in Section 2.3.6, it is extremely important to test implemented software in several different ways, be them functional or non-functional.

In this chapter, we will describe the several different tests and types of validation that were performed to the implemented work, in order to ensure that it was capable of delivering its functionalities to its users in a way that allowed them to solve the problem that was identified in Section 1.2 in a stable and efficient way, while simultaneously being an enjoyable and usable product.

For these purposes, the chapter will be divided into three different sections, each one dealing with the validation of a different facet of the photo|uniq web-based application:

- Section 4.1 will deal with the validation of the functional aspects of the application;
- Section 4.2 will deal with the validation of non-functional aspects of the application, with the exception of those related with the application's performance[1];
- Section 4.3 will deal with the validation of aspects related to the performance of the photo|uniq web-based application.

These three sections will all follow the same general structure, in which we will first begin by identifying the metrics and tests which will be used to validate the aspects in question (i.e., functional, non-functional, or performance), followed by a presentation and analysis of the results of said tests, where we will attempt to draw valid conclusions from them and delineate possible ways to improve the work where it is possible to do so.

4.1 Functional Validation

Functional validation is used to verify the correctness and stability of the functional aspects of a developed product.

[1] The performance validation of the application was separated from the rest of its non-functional aspects in order to improve the dissertation document's structure and readability.

In the context of the photo|uniq web-based application, these functional aspects were interpreted to mean the application's ability to solve the problem that was identified in Section 1.2, while running and performing stably and correctly.

4.1.1 Definition of Tests and Metrics

The tests that were performed to measure the functional validity of the photo|uniq web-based application can be differentiated into two main groups: those that were performed solely by the developer of the work (or, in other words, by the author of this dissertation), and those that were performed with the help of a selected set of testers and potential users that were exposed to the implemented application.

The majority of the tests that were performed by the application's developer were done so immediately after the implementation of an application element was completed, and served to ensure that this element worked well and stably by itself, and was also properly integrated and capable of interacting with the remainder of the application's functionalities while producing the correct results.

This meant that, after completing the implementation and integration of a particular concept or functionality into the photo|uniq web-based application, that concept or functionality would be put through some manual testing performed by the developer, to ensure its correct functioning both in normal and abnormal use conditions. This developer testing was also occasionally performed in multiple operating systems (MacOS, Windows, and Ubuntu) and browser clients (Google Chrome, Mozilla Firefox, Opera, Safari, and Microsoft Edge), in order to validate the cross-platform capabilities of the developed work.

The prospective users who were chosen to test the implemented application were subjected to user tests in which they were accompanied by a moderator. This moderator served as a guide and observer to the testing session, in order to help the user if they ever got stuck, and simultaneously observe the user's behaviour when interacting with the application, so as to better understand the mindset of a user in contact with the application and its UI.

The user tests all followed the same general scheme, where the users had four specific tasks that they had to complete without any particular script or guidance (in order to better simulate an unsupervised user experience).

While the users were encouraged to experiment with the application during their tests, the scope of the four planned tasks was mainly focused around the execution of the application's main workflow (i.e., the processes of uploading images, aggregating them into similarity groups, comparing them among themselves to select the favorite images, and promoting the processed groups into the photo|uniq gallery). This was also another way to test the functional validity and the resilience of the application to normal use conditions. Since the users did not know their expected behaviour or how the application is meant to function, they could potentially attempt to use some functionalities in ways

that were not previously considered (and might consequently also not have been tested).

If a bug or unexpected behaviour was detected at any point during the execution of these two aforementioned types of tests, the developer would then try to replicate this bug or behaviour in order to understand its causes, and then improve the implemented solution in order to prevent these situations from arising in the future.

4.1.2 Result Showcase and Analysis

We had previously mentioned that certain tests were performed to verify the correct functioning of the photo|uniq web-based application in a few different browsers and operating systems. In these tests, we attempted to interact with the application's different functionalities and mechanics, in order to see if any incompatibilities or unexpected behaviours were identified while using different browsers.

For this purpose, we tested the application using Google Chrome, Safari[2], Mozilla Firefox, Opera, and Microsoft Edge, running in MacOS 10.14.6, Windows 10 version 20H2, and Ubuntu 18.04.5 LTS. While performing these tests, we encountered the following bugs and unexpected behaviours:

- Safari version 14.1:
  - Incapable of using OpenCV.js [56], due to an error occurring when parsing its source code;
  - Errors occur while attempting to update values of data stored in IndexedDB using IDB [27];
  - Sometimes incapable of loading photo files from IndexedDB using IDB [27].
- Firefox version 91.0.1:
  - Unable to drag thumbnail version of the currently viewed photos while in the photo visualization and photo comparison pages;
  - UX problems while dragging photos for photo sorting purposes in general, i.e. images get "stuck" for a second before being able to be dragged again, making the action of dragging images less fluid than it is in other browsers;
  - Errors occur while using IDB [27] to perform multiple write operations to IndexedDB;
  - **Specific for Firefox for MacOS**: Application crashes while comparing two images using the photo comparison page.
- No bugs were detected for Chrome (version 92.0.4515.159), Opera (version 78.0.4093.147), or Edge (version 92.0.902.78 for Windows 10 and MacOS, version 93.0.961.27 beta for Ubuntu).

[2]Safari was only tested on MacOS, since since there are no current native versions of Safari for Windows or Linux.

Looking at these results, we can quickly recognize that all the tested chromium based browsers (i.e., Chrome, Opera, and Edge) did not show any unexpected behaviour when running the photo|uniq web-based application. This might be the case due to the fact that Google Chrome was the main browser used during the development of the application, and therefore had the most time put into bug detection and fixing.

We can also see that Safari is not compatible with the current build of OpenCV.js [55], which makes it so the current version of the developed application does not have access to any of its required image analysis capabilities when being run in this browser.

Other than this, there are some issues with the use of IndexedDB (or at very least IDB [27]) while performing multiple accesses to an IndexedDB object store and running the application using Firefox or Safari. This has the consequence of potentially leading to issues in application state consistency between sessions.

Finally, Firefox also is not be perfectly compatible with React Sortable HOC [66], and, in the case of Firefox for MacOS, react-zoom-pan-pinch [67]. These issues bring up problems associated with the application's interface itself, decreasing it's usability and, in the case of Firefox for MacOS, removing the ability to access one of its key functionalities (i.e., its side-by-side image comparing capabilities).

It is therefore clear that additional efforts must be made in order to improve the compatibility of the developed photo|uniq application with browsers which are not based in the chromium browser.

4.2 Non-functional Validation

As previously defined in Section 2.3.6, non-functional validation consists of testing that is performed to ensure that a developed product is correct and well designed in its non-functional aspects.

When looking at this definition through the lens of the photo|uniq web-based application, non-functional validation can be used to test a lot of different things. However, for the purposes of this work, we chose to focus this testing in how much the application is usable and liked by its users, how the application's image analysis algorithms measure up to its users needs, and in the aspects related to the application's performance (which will be addressed and discussed in Section 4.3).

4.2.1 Definition of Tests and Metrics

For the purposes of validating how likeable and usable the photo|uniq web-based application was, we used the three user-based scores that were previously presented and discussed in Section 2.3.6: the System Usability Scale (SUS), Customer Satisfaction Score (CSAT), and Net Promoter Score (NPS).

All these scores were calculated based on the responses that the application's users

provided to questionnaires created by following the templates that are presented in Section 2.3.6. This way, after a potential user had had contact with the application in the context of the user tests that were previously mentioned in Section 4.1.1, they were asked to respond to three sets of statements following the templates that were shown in the non-functional validation portion of Section 2.3.6.

These responses were then used to calculate the SUS, CSAT, and NPS following the steps that are also explained in the non-functional validation portion of Section 2.3.6.

In addition to these three scores, the users were also asked to voice their opinions about the application and its UI while they were interacting with it, and were also interviewed after their tests so that they could give further feedback regarding their experiences with the application.

In these situations, the users were also asked about any further comments or suggestions that they might have for the purposes of improving the application's UX and UI, so that these suggestions could be used to improve the application's usability. This feedback had the potential to be more specific and more informative than the three calculated scores, since it allowed the users to more directly voice their problems and expectations related with the application's UI.

Regarding the validation of the application's image analysis algorithms, this validation was focused in the algorithms used to automatically group images based on their similarity and automatically sort similar images based on their technical quality.

For the purposes of validating these algorithms, three tests were devised:

- In order to test the automatic image grouping function, we asked a group of testers to create similarity groups using the photos found in two provided sets of images (shown in Appendixes E and F). After this, we considered pairs of images which were grouped by the majority of testers to be similar to each other (and thus, obtained the reference pairs through which to compare the results of the image grouping algorithm). We then ran the automatic image grouping function of the photo|-uniq application using the same sets of images, and compared the resulting pairs of similar images with those found in the aforementioned reference pairs. In this comparison, we considered a false negative to occur when two images were grouped by the majority of testers but not by the application, and a false positive to occur if two images were grouped by the application but not the majority of testers.

- In order to test the automatic image sorting function, we artificially blurred 4 different images (shown in Figure G.1) to varying extents, by applying a blur filter on them with a 25, 50, 75 and 100% blur setting. We then used the application to sort 4 different groups of images, each one containing the unaltered and altered versions of one of these four initial images, and analysed the obtained order to see if it matched the expected order in regards to the different blur levels of the images. For the purposes of this analysis, and for every separate group, we considered all possible pairs of images for the 5 images contained in the group, and then verified

the correctness of the relative quality of the images in these pairs. A pair was considered to be correct if the image with the highest relative quality (as identified by the application) had a lower blur value than the second, and incorrect otherwise.

- A second test was also devised to test the automatic image sorting function, which relied on input obtained from testers. Here, we asked the same testers as before to sort four groups of images based on their relative technical quality when compared to each other. After obtaining all sets of results (one for each tester), and for every pair of images contained in these groups, we verified which image in the pair was considered to have better quality according to the majority of testers, which allowed us to create the reference pairs to which the results of the application's algorithm could be compared to. We then ran the automatic image sorting function of the photo|uniq application using the same groups of images, and compared the order of the pairs of images with those found in the aforementioned reference groups.

4.2.2 Result Showcase and Analysis

User Tests

Nine potential users had contact with the developed work during the course of the user testing sessions of the photo|uniq web-based application.

Their average responses to the aforementioned set of statements that were used to calculate the application's SUS can be seen in Table 4.1.

As previously discussed in Section 2.3.6, an application's SUS can be calculated by following a few steps:

1. First, we subtract 1 from the responses that users gave to odd-numbered questions, and subtract from 5 the users' responses to the even-numbered questions. This transforms the responses to every statement into a value between 0 and 4, with 4 corresponding to the most positive response.

2. Then, for every user that responded to the survey, we add the values related to each of these statements, and multiply the result of this sum by 2.5, which will provide us with a SUS of 0 to 100 for every user.

3. After this, we calculate the average of all the user responses, and we will have the application's average SUS.

Following these steps, we reach an average System Usability Score of 85, which places the usability of the photo|uniq web-based application in the top 10% of applications in regards to their usability [80, 71].

The users were also asked to answer to the statement "How would you rate your overall satisfaction with the photo|uniq application?" with a value ranging from 1 to 5,

Table 4.1: System Usability Scale average responses.

	Statement	Average Response	Standard Deviation
1.	I think that I would like to use the photo|uniq application frequently.	3.44	0.53
2.	I found the photo|uniq application unnecessarily complex.	1.56	1.01
3.	I thought the photo|uniq application was easy to use.	4.22	0.67
4.	I think that I would need the support of a technical person to be able to use the photo|uniq application.	1.22	0.44
5.	I found the various functions in the photo|uniq application were well integrated.	4.33	0.71
6.	I thought there was too much inconsistency in the photo|uniq application.	1.22	0.44
7.	I would imagine that most people would learn to use the photo|uniq application very quickly.	4.33	0.71
8.	I found the photo|uniq application very cumbersome to use.	1.33	0.50
9.	I felt very confident using the photo|uniq application.	4.00	0.71
10.	I needed to learn a lot of things before I could get going with the photo|uniq application.	1.11	0.33

with 1 meaning "Very unsatisfied", and 5 meaning "Very satisfied". The responses given by the users to this statement are shown in Figure 4.1.

We can calculate the photo|uniq web-based application's CSAT using these responses by following the formula which was previously presented in Section 2.3.6:

$$\frac{NSU}{NTU} \times 100$$

where *NSU* represents the number of users who were satisfied with the application (those who responded to the survey with a value of 4 or 5 or, in other words, "satisfied" or "very satisfied") and *NTU* represents the total number of users who responded to the survey [92].

This way, and according to the responses provided by the users who tested the developed work, the photo|uniq web-based application's CSAT is 77.78%, which means that the majority of the users that had contact with the developed work were satisfied with the photo|uniq application.

The users were also asked to respond to another statement, in which they were asked about the likelihood of them recommending the photo|uniq web-based application to a

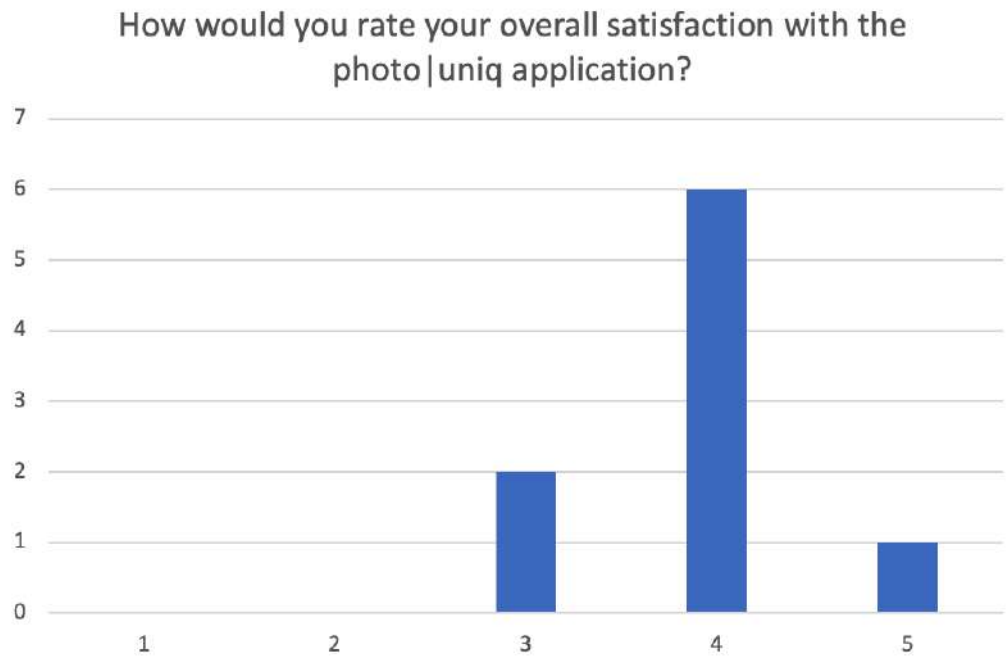


Figure 4.1: Responses given by the users to the statement used to measure the photo|uniq web-based application's CSAT.

friend or colleague. They were able to respond to this statement using a number from 0 to 10, with 0 meaning "Not at all likely", and 10 meaning "Extremely likely".

We used the responses given by the users to this final statement (which can be seen in Figure 4.2) to calculate the photo|uniq web-based application's NPS.

As explained previously in Section 2.3.6, an application's NPS is calculated by first dividing the user responses into three different types:

Detractors, which are the users who responded with a value from 0 to 6, and are considered to be unhappy with the application, unlikely to use it again, and may even discourage others from using it;

Passives, which are the users who responded with a value of 7 or 8, and are considered to be satisfied with the application, but vulnerable to competing options;

Promoters, which are the users who responded with a value of 9 or 10, and are considered to be extremely satisfied with the application, and will more likely than not develop some loyalty to it and suggest it to other potential users.

After doing this, we can calculate the NPS by subtracting the percentage of detractors from the percentage of promoters.

Looking at the results shown in Figure 4.2, we can see that out of the nine responses, one of the users was identified as a detractor, five of the users were identified as passives, and three of them were identified as promoters. This means that the Net Promoter Score of the photo|uniq web-based application is 22.22 (33.33 – 11.11).

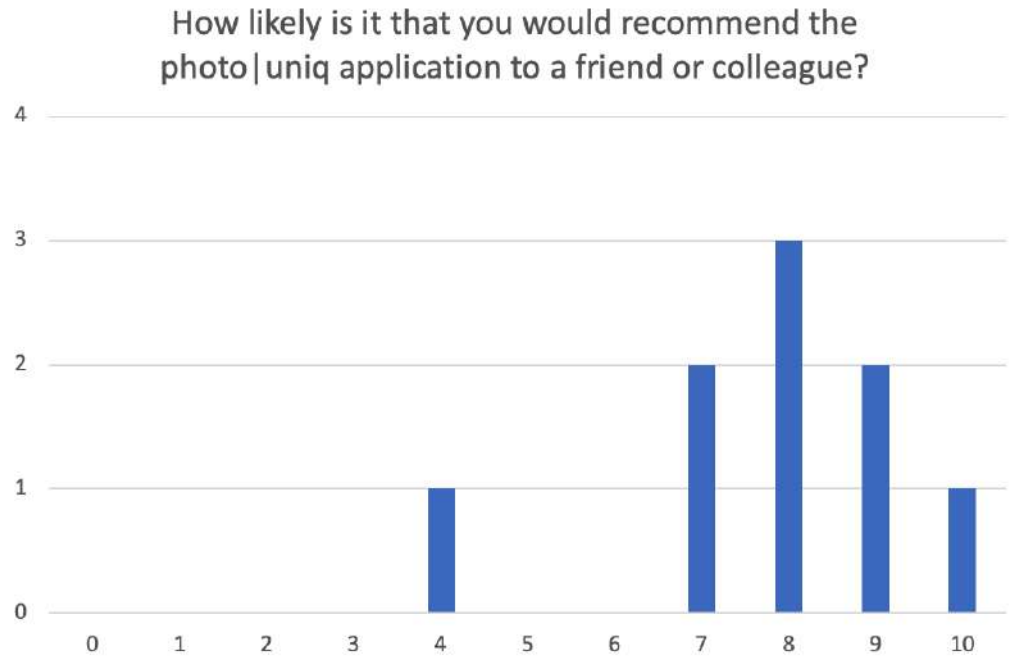


Figure 4.2: Responses given by the users to the statement used to measure the photo|uniq web-based application's NPS.

Considering that the NPS can range from -100 to 100, an NPS of 22.22 demonstrates that some of the user base would be loyal to the developed work to the point where they might feel comfortable recommending it to their acquaintances, but most of this user base would probably not develop any strong feelings towards the photo|uniq application.

These three scores (SUS, CSAT and NPS) are more meaningful and informative if measured multiple times, using more potential users, and after they had had a more complete experience with the application. Because of this, while the obtained results are very much positive, we suggest that these scores should be recalculated after future development efforts, and while using more potential users.

By performing these multiple measures and tests, we could achieve more accurate results concerning the satisfaction and opinion of the application's user base regarding their experiences with the application, as well as a provide us with a way to track of the evolution of these measures, to ensure that the user base's satisfaction does not decrease over time or with any future changes made to the application's functionalities and interface (with the goal being to either maintain the scores or improve them even further).

As mentioned previously, the users were also asked to provide any suggestions that they might have to improve the current state of the application and its UI. These suggestions occurred with varying frequency, with the most frequent and relevant ones being:

- Improve the design of certain button icons, in particular:
  - The similarity group icon, which several users misinterpreted as being related

to the handling of multiple images;

  - The processed group icon, since a padlock can be associated with several different concepts, i.e. security;
  - The group select button icon, since some users did not identify the current icon (which shows the number of photos contained in the group) to mean that it would select a group when pressed. One given suggestion was to make the icon shift to the regular select icon (i.e., a check mark) when hovered.

- Add additional options to assist users with the selection of elements, in particular:
  - Allow users to use *shift+click* to select multiple elements (much like it is used when exploring a computer's file system);
  - Allow users to use *cmd+click* or *ctrl+click* on an element to directly select it, instead of having to press the select button found in the application's UI;
  - Potentially add a button to enter "select mode", in which the user would then be able to select any element just by clicking on it, rather than need to click the aforementioned select button.

- Improve the UI of the photo visualization page by:
  - Adding buttons to the left and right of the main photo which, when pressed, will take the user to the previous and next photos in the sibling photo list;
  - Adding a visual indication that you can scroll on photos to zoom in on them. One suggestion for this was a zoom bar, which could then also be manipulated to change the zoom level of the main image.

- When viewing the contents of a similarity group, it would be interesting to have some visual indication of the order of the photos contained within, other than them being displayed left to right and top to bottom. This might also make it more intuitive for the users to understand that the photos in the groups can be sorted.

- Some users found the concepts of marking images as favorites (to signify that they should be kept) and marking groups as processed (to signify that the group was ready to be promoted to the gallery) to be a bit confusing on their first use of the application. Additional work should be put into these key concepts in order to make them more intuitive to and easily learned by the new users of the application.

- After promoting processed groups into the gallery, a button or dialog box should be provided to the user in order to allow them to go directly to the gallery (or, in other words, follow their promoted images) if they choose to do so.

- Some users suggested that we should move the buttons found in the functionality portion of the application header into some other part of the application's UI, so as

to better represent their functional difference from those found in the navigation portion of the application header, and make them visually stand out more.

- We had previously mentioned in Section 3.3.1 that a short explanation or tooltip of a button's functionality is provided to the user if their mouse pointer hovers that button[3]. When interacting with this tooltip, several users expressed a desire for it to appear faster than it currently does, with a few of them also suggesting that it might be better for it to also have a more appealing and "flashy" visual design.

- Some users also suggested that the application should have a custom design for its dialog boxes, rather than just taking advantage of those provided by the browser client's implementation (as provided when calling the *window.confirm()* function).

- Several users also expressed an interest in the addition of a new functionality to the application which would allow them to simply press a button after adding their images, with this button then calling a function that would automatically group similar images, sort these groups based on quality, and mark the best identified photo as favorite. This would allow the users to skip the majority of the application's workflow if they wish to do so (since they would then only need to confirm the application's choice of best image), while still allowing them to have full control of what would happen to their photos.

Automatic Image Grouping

As mentioned in the previous section, in order to validate the functional correctness of the photo|uniq application's automatic image grouping functionality, we compared the results of this function with the reference pairs obtained by asking a group of testers to manually group the similar images found within two sets of images (shown in Appendixes E and F).

These two sets were designed so that the similarity groups in the first set were very easily differentiated among themselves, in order to allow the testers to get used to the process of creating the groups, and the second set was composed of a more realistic dataset, whose images were taken from a real photo gallery, and therefore had the potentiality of being harder to separate between themselves.

The results of these tests can be seen in Table 4.2.

Analysing these values, we can say that a precision of 85.7% (for the first set) and 84.3% (for the second set) are very good results. We can also remark that, out of the obtained 39 incorrect pairs, 84.6% of these pairs were related to false negatives, as opposed to the 15.4% false positives.

While attempting to explain the higher frequency of false negatives in these results, we hypothesized that these results might have happened due to the algorithm employed

[3]This is currently done through the use of the HTML *title* attribute, which is dependant on the browser client's implementation.

Table 4.2: Functional validation of the automatic image grouping function.

	First Dataset	**Second Dataset**
Number of Images in Dataset	15	18
Total Number of Image Pairs	105	153
Number of Correctly Sorted Image Pairs	90	129
Number of False Positives	6	0
Number of False Negatives	9	24
Accuracy of Pairs	85.7%	84.3%
Number of Groups Created by Users	3	4
Number of Groups Created by Application	4	10

by the application being stricter than the testers' own criteria when it came to the identification of similar images.

In these performed tests, the testers' criteria seemed to take into account a more subjective analysis of the images in order to identify if the same objects or general area were captured in the provided photos. This was different from the application's algorithm, which merely correlates keypoints, and thus takes a more objective approach to the identification of similar images (and more affected by matters of perspective and photo angle, as well as requiring the photos to contain the same key motifs).

We can see an example of the difference in results obtained from the majority of testers *vs.* the photo|uniq application in Figure 4.3.

The three images shown in this figure were all identified as similar by the majority of the aforementioned testers (who were able to identify that the photos were all taken in the same place), while only the second pair (shown in Figure 4.3(b)) was identified as similar by the photo|uniq application.

For the purposes of the chosen metrics for false positives and false negatives, if a reference group is separated into several smaller groups that only contain the same images as that reference group, then no false positives will occur, but several false negatives will happen, as several images which should have been identified as similar are not placed in the same similarity group. Because of this, and in the situation shown in Figure 4.3, since the majority of testers identified a single group with the three images while the application identified two groups, two false negatives and no false positives occurred.

Similar situations occurred with other images, where a single similarity group (as identified by the majority of testers) was divided by the application into multiple smaller groups, causing the high number of false negatives that were previously identified, with 0 accompanying false positives. This is also reflected by the results shown in Table 4.2 regarding the number of created groups in the second dataset (shown in Appendix F),

(a)

(b)

Figure 4.3: Example of a pair of similar images as identified by the majority of testers (4.3(a)), as opposed to one identified by the application (4.3(b)).

where the users created 4 groups *vs.* the application's 10, and even in the first dataset (shown in Appendix E), where the users created 3 groups *vs.* the application's 4.

While there were also some false positives identified with regards to the first set, these can be easily interpreted as being a result of the misidentification of similar images caused by an imprecision in the application's algorithm, which led to a single image (shown in Figure E.3) being placed in the wrong similarity group (otherwise composed by Figures E.2, E.4, E.6, E.8, E.10, and E.11).

From looking at the aforementioned findings, we consider that additional testing should be performed in order to verify if these results remain consistent with different testers and image sets, so as to see if the current direction that is being employed for the identification of similar images is appropriate for the requirements of the application's expected user base, or if different types of algorithms should be studied (and if they should, which are the most interesting for the application's specific use case).

Automatic Image Sorting

As previously explained, two different types of tests were performed for the functional validation of the automatic image sorting algorithm of the photo|uniq application.

The first type of test involved artificially blurring four different images (shown in Figure G.1) using different levels of blurriness (25%, 50%, 75%, and 100%) and using the photo|uniq application to sort these images based on their technical quality. Since we were aware of the different levels of blurriness that each image had, we could then use these values as a reference to see if the application sorted the images according to the ascending level of blurriness.

The second type of test involved four different groups of images (shown in Figures H.1, H.2, H.3, and H.4), which dealt with additional types of image objective quality measures such as changes in contrast, exposure, colour, among others. For this test, we asked five testers to sort the images in these four groups according to their technical quality, while ignoring their subjective quality.

After this, we analysed the five sets of answers and, for every two images contained in the same group, created the reference pairs that represented the relative quality of these two images. This was done by seeing how the testers ranked the images in each pair in relation to each other, and then considering the image with the highest ranking as chosen by the majority of testers to have the best technical quality from the two options.

After obtaining these reference pairs, we could then compare them to the pairs that were returned by the application's function, and through this analyse the accuracy of this function for both types of tests. The results of these tests can be seen in Table 4.3.

Table 4.3: Functional validation of the automatic image sorting function.

	First Type of Test	**Second Type of Test**
Total Number of Image Pairs	40	45
Number of Correctly Sorted Image Pairs	33	28
Accuracy of Results	82.5%	62.2%

An analysis of these two results indicates that, as expected, the application's algorithm is much more accurate when merely dealing with changes in image overall focus level (as was the case with the first type of test, which had a tested accuracy of 82.5%), as opposed to images with technical problems of other varieties (that were sorted in the second type of test, which had a tested accuracy of 62.2%). This was expected since, as previously mentioned in Section 2.4, the current method used by the application to measure an image's objective quality only takes into account its overall focus level.

These findings confirm the application's current method for evaluating an image's objective quality as being incomplete, since it does not take into account several of the metrics that can and should be taken into account for this purpose. Because of this, we

recommend that a future development effort should attempt to add additional steps and complexity to this current method, so that it might take into account additional relevant image technical quality metrics.

We can also say that, when directly analysing the specific results obtained for each one of the groups found in the second test, the application had the worse accuracy when dealing with the groups containing the highest number of artificially altered images (i.e., 40% and 60% accuracy when sorting the images found in Figures H.1 and H.3), which had extremely exaggerated changes performed on them, and therefore should not be considered to be the best examples of a normal dataset.

When it came to the first test, we also noticed that for all groups, while there were slight differences in the expected order between blurry images, the first graded image in all groups was always the original one (or, in other words, the one which had not been artificially blurred for the purposes of the test). This could be interpreted to mean that the algorithm does indeed serve to accurately differentiate blurry images from properly focused ones, but additional tests following this same structure and with different types of images (that might be affected differently by being properly or improperly focused) should be performed to better verify this hypothesis.

4.3 Performance Validation

As the name indicates, performance validation is a process in which the performance of a developed product is put to the test, in order to ensure that it is up to the standards and necessities of its user base.

When it comes to the performance testing and validation of the photo|uniq application, the tests which are defined and presented in this section were performed while running the application in its production build on a computer, local static server and browser with the following technical specifications:

Computer model 13-inch MacBook Pro from mid 2014;

OS macOS Mojave 10.14.6;

CPU 2,8 GHz Intel Core i5;

RAM 8 GB 1600 MHz DDR3;

GPU Intel Iris 1536 MB;

Static server setup software Serve [72] version 11.3.2;

Browser Google Chrome version 91.0.4472.114.

Image Datasets Since the consequences of differences in image resolution and storage cost were not monitored for the purposes of these tests, we used datasets of images

that had an average size of 37 Kilobytes (KBs). The datasets where also composed of different images between themselves (meaning that no image was included in more than one dataset).

4.3.1 Definition of Tests and Metrics

When it came to measuring the performance of the photo|uniq web-based application, we decided to focus our efforts in analysing how some of its most important functions scale with the number of handled images. We considered four different functions:

- The image upload function, which is what allows the user to add photos to the photo|uniq application;
- The image loading function, which loads the images that have been uploaded to the application from the application's session or persistent storage;
- The automatic grouping function, which automatically creates similarity groups using the previously existing groups and ungrouped workspace photos contained in a workspace folder (as was described in Section 3.4.2);
- The automatic image sorting function, which sorts the images contained in a similarity group based on their overall focus level (as was described in Section 3.4.3).

Our method for testing these functions was to first analyse the source code of the developed work in order to estimate their time complexity. Following this, we then ran these functions and measured their execution time, while modifying the number of photos being uploaded, loaded, grouped, or sorted.

We then plotted the obtained results in a graph, and analysed this graph to see if it matched up with the previously estimated time complexity for each function, by comparing it with the expected pattern for its specific type of Big O time complexity, as shown in Figure 4.4.

By performing these types of tests, we achieved a better understanding of how the performance of the developed work scales with the number of handled images, and saw if these results were appropriate or if the developed functions should be further improved.

4.3.2 Result Showcase and Analysis

Image upload time complexity

By analysing the source code of the function used to upload images to the application, we estimated that this function's temporal cost would rise linearly with the number of uploaded images, as well as with the number of elements required to search in order to locate the parent element to which the images would be added to (i.e., the total number of workspaces if uploading to a workspace, and the total number of workspaces and workspace folders contained in the parent workspace if uploading to a workspace folder).

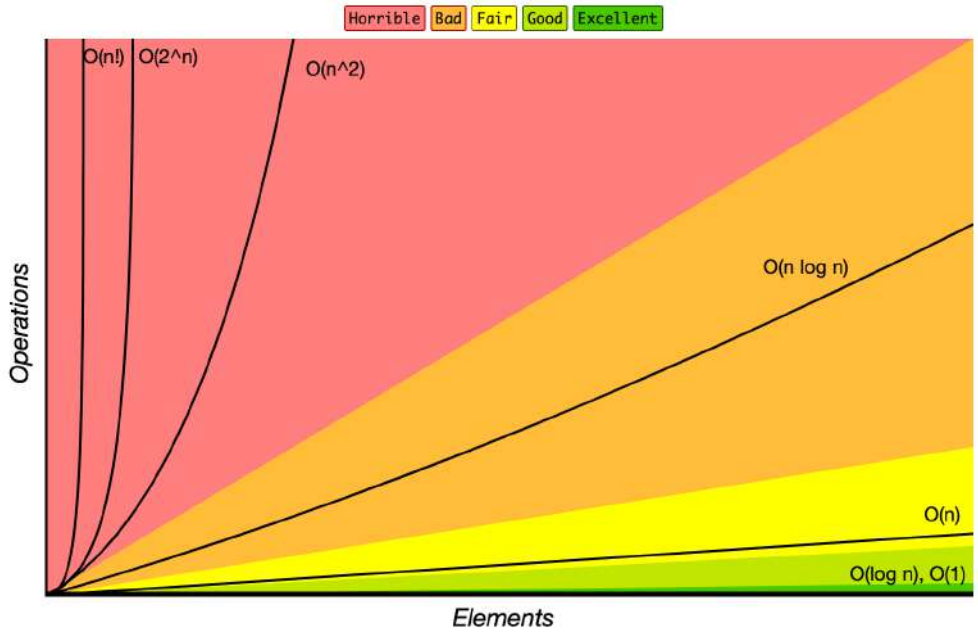


Figure 4.4: The various types of plots obtained from different Big O time complexities [10].

Putting this into Big O notation, the time complexity of the image upload function would be O(w + n) when uploading to a workspace, and O(w + f + n) when uploading to a workspace folder, with **w** representing the number of workspaces existing in the application, **f** representing the number of folders contained in the parent workspace of the folder to which the images are being uploaded to, and **n** being the number of images being uploaded. A more detailed analysis of the time complexity of the several steps that make up this function can be seen in Table I.1.

In order to verify this hypothesis, we measured the time that it would take to upload a varying number of images into the application. This measure of time was obtained by performing 10 sets of tests in which we first observed the time that it took to upload 10, 100, 500, 1000, 5000, and 10000 images.

Due to our previously mentioned choice of focusing our efforts in analysing how the image upload function would scale with the number of uploaded images, we tested this function while there was only one existing workspace in the application, and no workspace folders. While testing the function in these conditions, the previously established **w** and **f** variables would always have a lower value than the previously established **n** variable, which made it so we were able to simplify the expected time complexity to be O(n), since O(w + f + n) = O(n) when n > w and n > f.

With these testing condition in mind, we expected that the time taken to upload images would scale linearly with the number of images being uploaded.

After performing these observations, we then calculated the average results for each number of uploaded images that were tested, and plotted these results in the graph that

is shown in Figure 4.5.

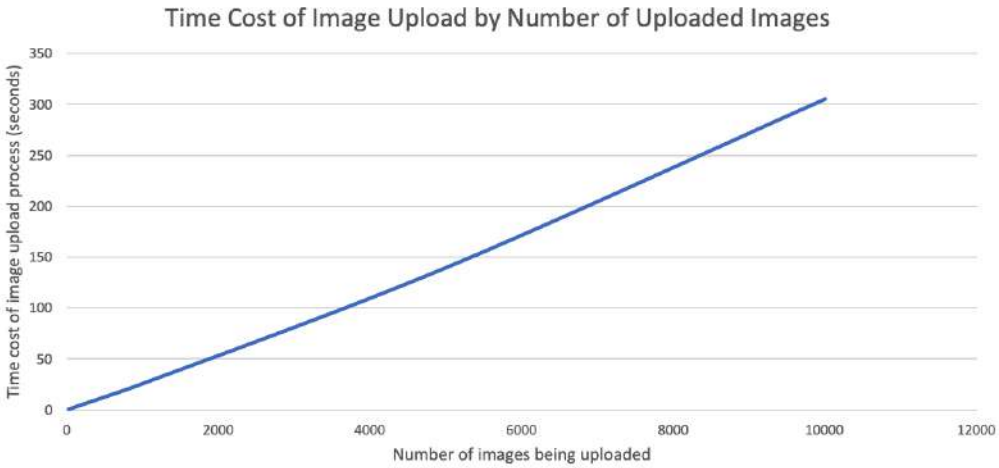


Figure 4.5: Plot representing the results obtained by the practical tests performed to validate the time complexity of the image upload function.

As we can see while analysing the shown graph, the results are consistent with a linear time complexity, since the straightness of the plotted line indicates that the time cost of the image upload process increased linearly with the number of uploaded images.

Image loading time complexity

When loading the photos contained in an element from the application's persistent storage, the current version of the photo|uniq application follows a very simple process, in which it first makes a request to the IndexedDB database to request all of the photos that are associated with that element's id, and then, depending on the type of element whose id was provided, one of three scenarios will happen:

1. If the element whose id was provided is a non-deleted similarity group, the application will first sort the obtained photos according to the photo order defined by the similarity group in question, and then store the sorted list in the application state;

2. If the element whose id was provided is another type of lowest level element whose photo order is not relevant (i.e., a deleted similarity group or an album folder), the application will immediately store the obtained photo list in the application state;

3. If the element whose id was provided is a mid or highest level element (who can contain other non-photo elements), then the photo list obtained from storage will contain every photo that is stored either in the element with the provided id, or in one of its child elements. In this situation, the application will first filter the obtained list so that it only contains the photos that are contained directly in the root of the element with the provided id, and only after that it will store the filtered list in the application state.

The estimated Big O time complexity for the first scenario is O(g + n log(t)), while the other two scenarios have an estimated O(n log(t)) time complexity. In these time complexities, **g** refers to the number of groups contained in the group's parent workspace folder, **n** refers to the number of images which are retrieved from the IndexedDB object store, and **t** refers to the total number of images stored in the accessed object store (which will always be higher or equal than **n**). A more detailed analysis of the time complexity of the several steps that make up these three scenarios is shown in Table I.2.

In order to verify these hypothetical time complexities, we performed three different tests, one for each possible scenario.

All these tests followed the same structure, which was designed to measure the time that is necessary to load a varying number of images from the application's persistent storage. This measure of time was obtained by performing 10 sets of measures in which we first observed the time that it took to load 10, 100, 500, 1000, 5000, and 10000 images.

After performing these observations, and for each one of these tests, we calculated the average results for each number of images that were tested, and plotted these results in a graph, shown in Figure 4.6.

While loading images contained in a non-deleted similarity group, a constant number of groups was contained in the parent workspace folder, which essentially made it so the time complexity of this function could be considered to be O(n log(t)) (since there was no variation in the previously established **g** variable throughout the testing process).

The number of photos contained in the IndexedDB object store was also always the same for all scenarios (i.e., 16610 images). This also meant that the previously established **t** variable was kept constant throughout all of the performed tests.

By setting these two variables to constant numbers throughout all the tests, we made it so the only source of variation was the number of loaded images. By doing so, we expected that the time cost of this function would grow linearly with the number of loaded images or, in other words, follow an O(n) time complexity.

As we can see, all three plots follow the same general structure, which is mostly consistent with a linear progression (as was expected). There is a slight exception to this when reaching the highest number of images, where the plots seem to slightly curve downwards (which is particularly noticeable when it comes to the tests regarding the second scenario). We could not find an explanation for the occurrence this curve.

Automatic image grouping time complexity

The function used to automatically group images based on their similarity[4] is among the most complex processes of the developed work, which caused us to expect that the time complexity of this function would be much more convoluted than the other functions discussed in this section.

[4]Explained in Section 3.4.2.

Figure 4.6: Plot representing the results obtained by the practical tests performed to validate the time complexity of the function used to load images.

After analysing the source code of the function, we estimated a time complexity of $O((f \log(i)) + (n \times g))$ while disregarding variables such as image resolution and number of detected features, since these variables were not measured in the tests that we were performing. Our performed estimation of the time complexity of the several steps that make up this function is shown in Table I.3.

In this presented complexity, **n** refers to the number of ungrouped images which are being grouped through the execution of the function, **f** refers to the number of images whose features are being correlated (or, in other words, the sum of **n** with the number of group representatives which will be analysed during the execution of the function), **i** refers to the total number of *ImageKeypointDetectorInfoObjects*[5] which are stored in the application's persistent storage, and **g** refers to the number of groups which are contained in the parent workspace folder after the end of the function's execution.

In order to verify this estimated time complexity, we executed three sets of tests in which we grouped a varying number of images, starting with 10 images and up to 10000 images (when possible). These tests were performed in an environment in which there were no previously created groups in the application nor *ImageKeypointDetectorInfoObjects* in the IndexedDB object store.

By running the tests in these conditions, we made it so:

- The previously defined **g** variable was equal to the number of groups created during the function's execution, since there were no previously created groups contained in the parent workspace folder;

[5] *ImageKeypointDetectorInfoObject* (previously defined in Section 3.2.4), is a class which is used to store the previously computed features of an image. These features are essential for the purposes of automatically grouping images, since it is through the correlation of image features that the application discerns whether two images are similar or not.

- The previously defined **f** variable was equal to the number of ungrouped images (i.e., **n**), since there were no group representatives to compare these images to;
- The previously defined **i** variable always had a constant value (i.e., 0), since there were no *ImageKeypointDetectorInfoObjects* in the application's persistent storage.

These testing conditions allowed us to simplify the previously estimated time complexity of O((f log(i)) + (n × g)) into O(n log (1) + (n × g)) or, in other words, O(n × g). This allowed us to focus our testing efforts into the analysis of how the number of previously ungrouped images and the number of created groups affected the scalability of the function.

Additionally, the aforementioned three sets of tests were performed in order to test the expected scenario while running this function, the worst case scenario, and the best case scenario:

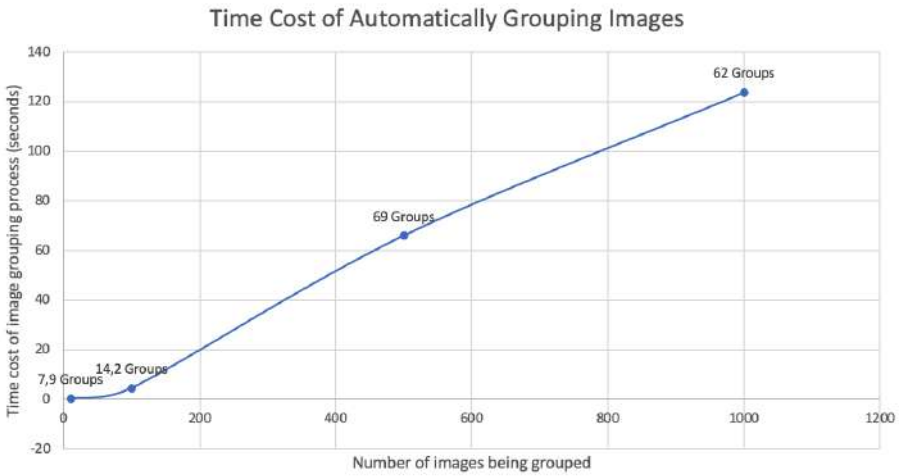
1. To test the expected scenario, we used an unedited version of the automatic grouping function, in which images were placed in groups based on them being identified as similar or not. In this situation, the time complexity remained O(n × g).
2. To test the worst case scenario, the source code was changed so that no image was ever considered similar, causing every image to always be placed in a different group from the others. This meant that the number of groups at the end of the function execution would always be equal to the number of previously ungrouped images, transforming the time complexity into O(n × n), or O(n^2).
3. In the last set, the code was tweaked so that all images would always be placed in the same group, which lead to a single group existing at the end of the function's execution. In this situation, the number of groups became a constant, and therefore the time complexity became O(n × 1), or O(n).

For each number of images and set of tests, we took 10 measures of time, which were then averaged and plotted in Figures 4.7(a), 4.7(b), and 4.7(c), representing the first, second and third sets of tests respectively. The text above each data point in Figure 4.7(a) also indicates the average number of groups that were created during the tests for that specific number of images.

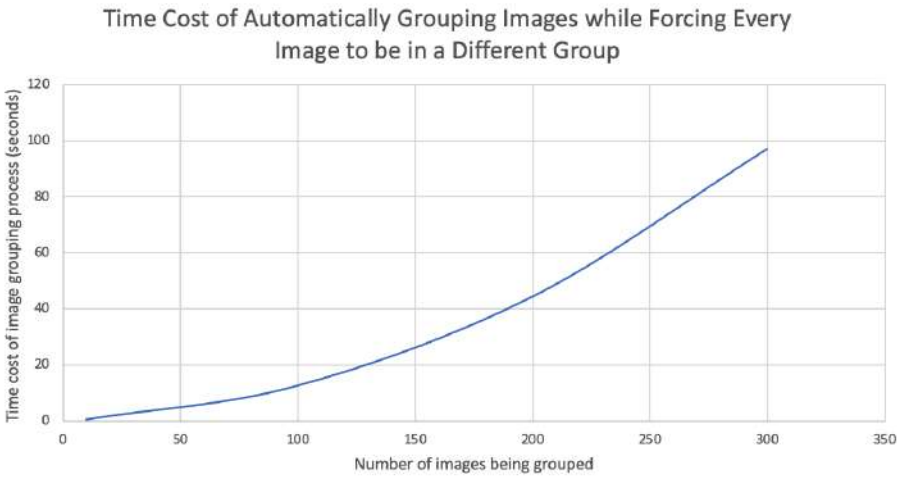
A few things can be gleaned from analysing these three shown graphs.

First of all, we can see that only the set of tests related to the best case scenario managed to reach the attempt to group 10000 images. This occurred due to the expected and worst case scenarios consistently crashing the application at just over 1300 images and just over 300 images respectively.

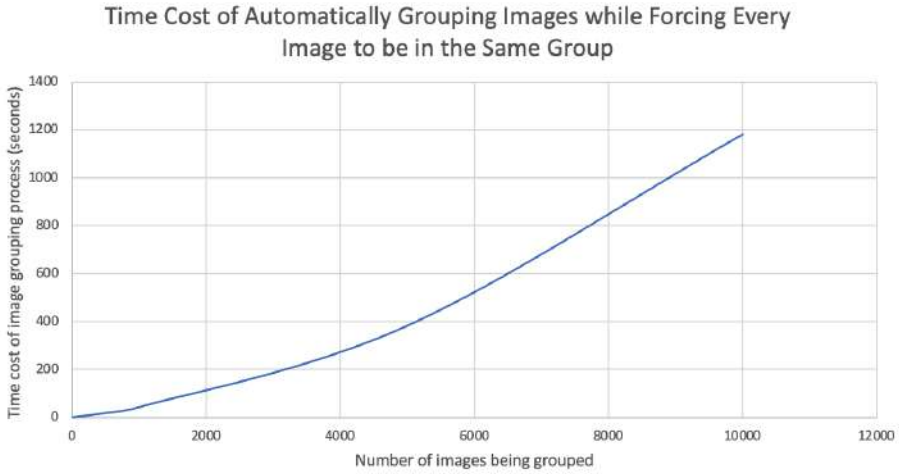
These crashes happened due to an unidentified bug occurring within OpenCV.js, and always occurred at the image feature correlation stage, after all of the image features were identified for all images. No discernible cause could be identified as to why this bug occurred, which led us to think that it might be happening due to an internal error



(a)



(b)



(c)

Figure 4.7: Plots representing the results obtained by the practical tests performed to validate the time complexity of the function used to automatically group images.

occurring within OpenCV.js when performing too many image feature correlation operations. This is substantiated by the crash occurring sooner with the more groups that are created, since every image that the function attempts to group is compared with the representatives from all created groups.

Looking at the specific graphs for each scenario, and starting with the expected scenario, we can see that the while the plot begins with a slight curve up (which might be consistent with an $O(n \times g)$ time complexity), it then seems to follow a more straight line when reaching a higher number of images.

An explanation for this discrepancy might be the relatively low average number of groups which were created when grouping 1000 images (i.e., 62 groups), as opposed to the 69 groups which were created when grouping 500 images. This low number of groups might have caused the execution time of the function to be lower than usually expected (since less image feature correlation operations had to be performed), which would indeed affect the plotted line as is shown in Figure 4.7(a).

Moving on to the worst case scenario, we see that while there is a very clear upwards curve happening on the plotted line, it is not as intense as would normally be expected of an $O(n^2)$ time complexity. We offer two explanations to this discrepancy.

The first is that the line might not seem quadratic due to the fact that $O(n^2)$ is merely a representation of the worst case scenario which never really happens, as the number of groups starts as 0, and keeps increasing as the function is executed. To put it in other terms, during the execution of the tweaked function, the first image is not compared with any other image, the second is compared with the first, the third is compared with the first and the second, and so on and so forth up to the point when the last image will be compared with every other image. Because of this, not all images actually get compared with every other image (since only one image actually does so), and therefore there are not actually $n \times n$ image feature correlations.

Another possible explanation for this is the fact that we could only perform this set of tests with a relatively low number of images, which might not be enough to truly show a proper evolution of the temporal cost of the execution of this scenario. This explanation will be further explored when analysing Figure 4.8.

Finally, when looking at the results of the set of tests related to the best case scenario, we see a line which seems to follow a near linear pattern, except for a very slight curve upwards. We can find no explanation for the existence of this curve, and require further testing to better understand its occurrence.

Due to the sets of tests not being performed with the same number of images, it becomes difficult to compare these results with each other, and simultaneously is harder to analyse the results of the worst case scenario, since we were not able to perform our tests with a high enough number of images to truly reach any worthwhile conclusions.

In order to mitigate these issues as much as possible, a graph which contains the results of running all sets of tests with up to 1000 images can be seen in Figure 4.8.

Two interesting points can be gleaned from this plotted graph:

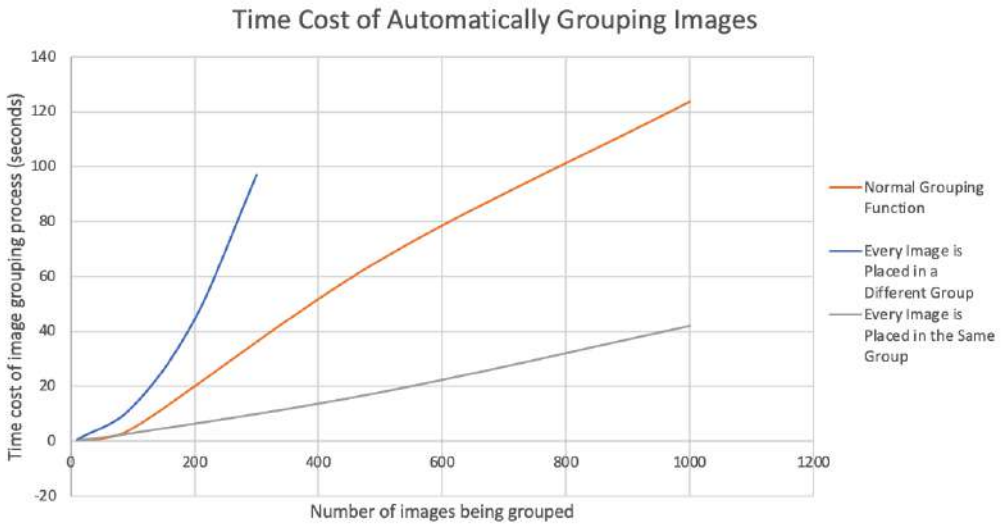


Figure 4.8: Plot showing the results of all three sets of tests obtained by the practical tests performed to validate the time complexity of the function used to automatically group images.

- The relative execution times of the three tested scenarios performed as expected, with a lower execution time being observed when dealing with a lower number of images and/or a lower number of groups;

- When directly compared with the other two plots, the line related to the worst case scenario displays a much more steep curve up, which is more consistent with the expected pattern of an $O(n^2)$ time consistency (as it is shown in Figure 4.4).

While we found the results of these three sets of tests results to be relatively consistent with our expectations (while taking into account the limitations of our testing process), it would still be a worthwhile idea to re-run these tests at a later point with a more stable and powerful computer, in order to verify if the function performs as expected when put through more thorough tests.

Additional efforts should also be put into understanding and, if possible, fixing the identified OpenCV.js [55] bug, in order to make the application more resilient to grouping a higher number of images.

Automatic image sorting time complexity

The function used to automatically sort images based on their technical quality follows a simple process which was previously presented in Section 3.4.3.

After performing an analysis of the source code of the developed function, we estimated that its time complexity would be $O(s \log(f) + n \log(n))$, with **s** being the number of

images whose *FocusSortSupportObject*[6] cannot be found in the application's session storage, **f** being the number of *FocusSortSupportObjects* stored in the application's IndexedDB database, and **n** being the number of images which are being sorted. The support table showing our analysis of the time complexity of the several steps that make up this function can be seen in Table I.4.

We tested this hypothesized time complexity by performing three sets of test, in which we verified the most extreme versions of the possible ways to retrieve/compute the *FocusSortSupportObjects*:

1. The first set of tests verified the time cost of attempting to sort images in a situation where none of the images had a previously computed *FocusSortSupportObject*. In this situation, we expected the time complexity to be O(n log(1) + n log(n)), or O(n log(n)), since the previously defined **s** variable would have the same value as **n**, and there would be no *FocusSortSupportObjects* found in the IndexedDB object store.

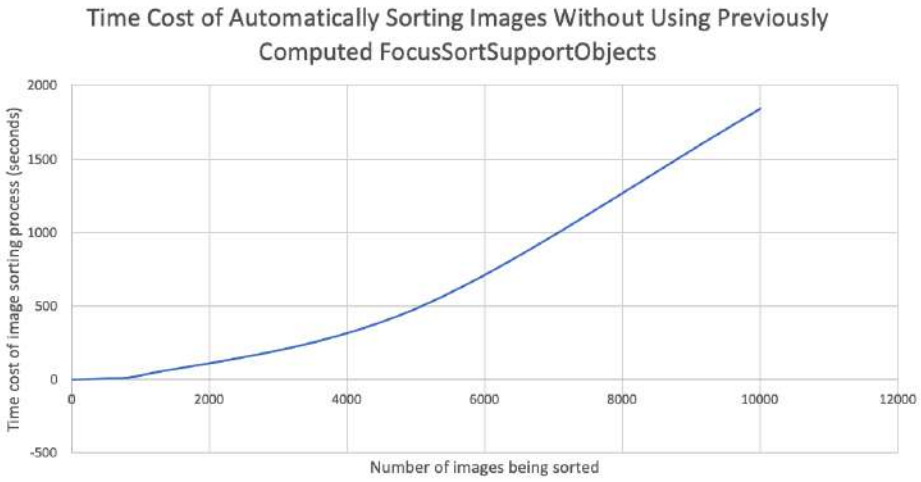
2. The second set of tests verified the time cost of attempting to sort images in a situation where all of the images had a previously computed *FocusSortSupportObject* which had not yet been loaded to the application's session storage (and thus still had to be retrieved from the IndexedDB database). In this situation, we also expected the time complexity to be O(n log(n)), since the previously defined **s** and **f** variables would have the same value as **n**, and O(n log(n) + n log(n)) = O(n log(n)).

3. The final set of tests verified the time cost of attempting to sort images in a situation in which all of the images had a previously computed *FocusSortSupportObject* which had already been loaded to the application's session storage. In this situation, we also expected the time complexity to be O(n log(n)), since the steps related to the loading of *FocusSortSupportObjects* from the IndexedDB object store (which were the cause of the previously defined O(s log(f)) portion of the time complexity) would not occur, therefore simplifying the time complexity into O(n log(n))).

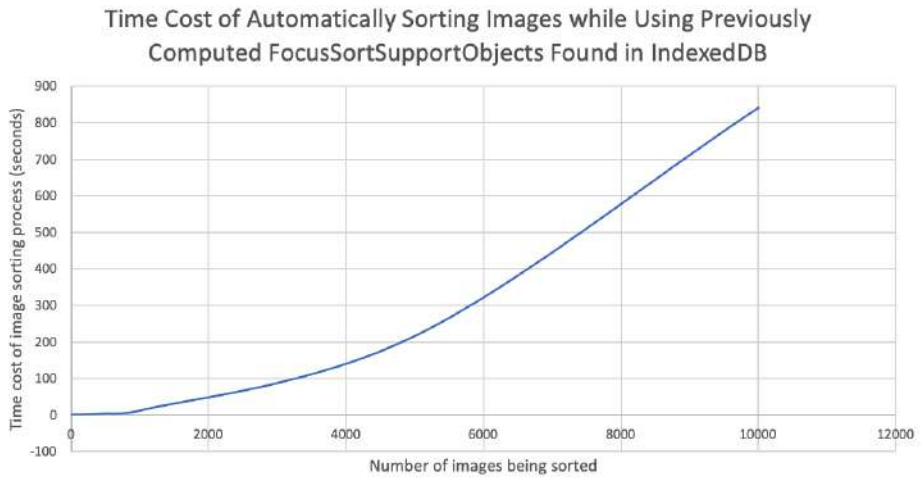
For each one of these test sets, we sorted 10, 100, 500, 1000, 5000, and 10000 images, and measured the time that it took to complete this sorting operation. We performed 10 measures of time (or 5 when dealing with numbers of images which took upwards of 10 minutes to sort), and then averaged and plotted these results, so that they could be easier to analyse. These plots can be seen in Figures 4.9(a), 4.9(b) and 4.9(c) (containing the results of the first, second, and third presented test scenarios respectively).

Looking at the three shown graphs, we can see that all three of them contain the slight upwards curve that is expected from an O(n log(n)) time complexity (as is shown in Figure 4.4), which is consistent with our previously estimated time complexity.

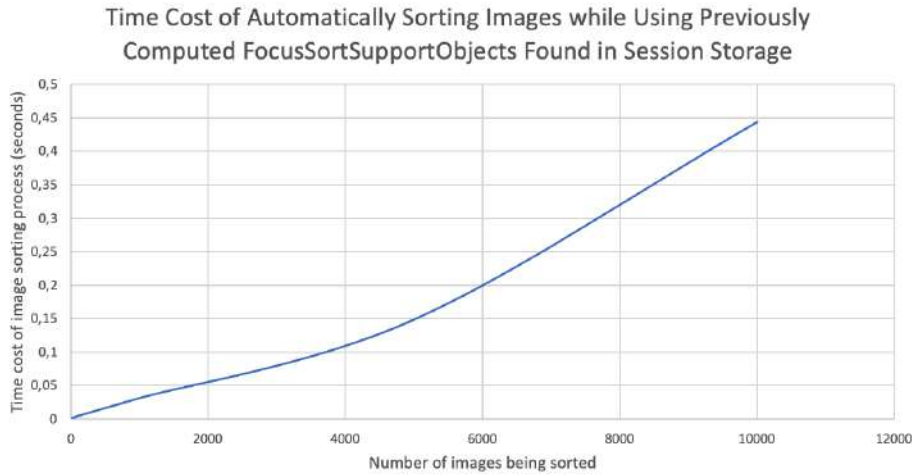
[6] *FocusSortSupportObject* (previously defined in Section 3.2.4), is a class which contains the numeric value used to represent the overall focus level of an image. This value is essential for the purposes of automatically sorting the images contained in a group, as it serves as the application's current metric for image quality.



(a)



(b)



(c)

Figure 4.9: Plots representing the results obtained by the practical tests performed to validate the time complexity of the function used to automatically sort images.

The relative time costs between the three scenarios also seem consistent with what was expected, since sorting images using *FocusSortSupportObjects* found in the application's session storage is substantially faster than doing so with *FocusSortSupportObjects* found in the application's persistent storage, which is then also substantially faster than doing so while computing and creating a *FocusSortSupportObject* for every image during the course of the function's execution.

We can also see that the time taken to load the *FocusSortSupportObjects* from the IndexedDB database is much higher than the time cost of loading the same number of images (as shown in Figure 4.6). For instance, while loading 10000 images from the IndexedDB database took around 4.5 seconds, loading 10000 *FocusSortSupportObjects* took around 840 seconds.

One proposed explanation for this difference lies in how these objects are loaded from the application's persistent storage:

- While loading images, only a single request is made from the database, in which we provide the id of the parent element of the image (e.g., a folder or similarity group), and the IndexedDB then simply retrieves every image in the object store which is associated with that parent id;
- On the other hand, while loading *FocusSortSupportObjects*, we cannot differentiate them based on a common parent id (since the images can be moved between groups at any point), and therefore we must make as many requests from the IndexedDB database as the number of images whose *FocusSortSupportObject* is being requested (or, in other words, as the number of images which are being sorted).

While these high loading times seem egregious at first sight, it is worth mentioning that it is expected that this operation (i.e., sorting the images contained in a similarity group) would only be called upon with a much lower number of images, since we expect similarity groups to normally contain only tens of images at most. Nevertheless, this is still an issue worth exploring in future developmental efforts, so as to further improve the application's temporal performance.

5

Final Considerations

5.1 Conclusions

In this work, we set out to develop a web-based version of the photo|uniq application for desktop systems which would help its users to identify similar images and provide them with tools which would facilitate and improve the process of comparing these similar images, so that they might make more informed decisions regarding whether or not to keep these similar photos.

In order to do this, we first began by establishing the key functionalities of our application, analysed some of its competing solutions, and researched some essential web development and image analysis concepts which would prove invaluable for the purposes of implementing the program.

After this, it came the time to start the development process. For these purposes, we began by creating a non-functional prototype of its UI using Axure RP 9, which allowed us to establish the basis for what this interface would be, while simultaneously maintaining a fast and nimble development process, and without the need to program anything.

We then proceeded to the development of the application itself, using TypeScript [86] and the React.js library [23]. For these purposes, we implemented the main application component (which is responsible for the generation and delivery of the application's UI to the user), the photo processing component (which handles the image analysis necessities of the application), and the data management component (which is responsible for the loading and storing of the application's data, which in turn allows for the user's progress to be consistently stored between sessions).

Even though the currently implemented version of the application runs solely on the user's browser, its architecture was implemented with a special focus on abstraction of the different components of the application, in order to allow for a higher separation of concerns, and making it so future development efforts can more easily and readily modify or re-implement these different components (in particular, the photo processing and data managements components), without needing to additionally change the implementation of the front end/interface portion of the application.

We then performed some essential testing and validation of the developed application. We did this by verifying the functional validity and time complexity of some of its key functionalities, as well as by performing some moderated user tests, in which users outside the development efforts were put in contact with the application, and were asked to give their feedback and opinions regarding their experiences with it, as well as any suggestions they might have which could be used to further improve the application. We also used these user tests as an opportunity to calculate the application's SUS, CSAT, and NPS, as better ways to more accurately and objectively measure how the application's users truly feel about it.

While the developed application is currently functional and mostly complete, we also identified certain facets which could be improved and additional functionalities which can be added to the current solution, so that it may provide a more complete product, and with a better UX. These points will be presented and discussed in the next section.

5.2 Future Work

When it comes to the future work that should be done in order to improve the photo|-uniq web-based application further, there are a few elements which we feel should take precedence over the other potential general improvements and additions that could be made:

- First, we propose the adaptation of the web application for mobile devices, such as smartphones and tablets, which would improve the application's cross-platform capabilities, and greatly increase the size of its potential user base.

- Additionally, we recommend the implementation of a remote server infrastructure, or at the very least a back end that could be run either remotely or locally on the user's computer. This back end could be used to host the application's photo processing and data management components, which would in turn allow the application to be much less reliant on WebAssembly and the browser client's specific implementation, while simultaneously allowing for more complex image analysis operations, which could potentially improve both the performance and correctness of some of the application's key functionalities (i.e., it's automatic image grouping and sorting capabilities).

- We also emphasize the necessity for the implementation of a proper user account system with additional authentication and security measures, as these essential facets of the application are currently unexplored (even though the application's front end is prepared to be integrated with some such elements). After doing so, we could then implement additional functionalities which take advantage of the existence of several users (e.g., sharing elements such as photos, workspaces, and albums between users).

- We also consider it relevant to implement certain settings which might be considered to be relevant to improve the UX, such as support for different languages, a usable darkmode which the user could toggle if desired, among others.
- We should also explore the idea of linking the photo|uniq application with cloud photo storage services (i.e., iCloud, Google Photos and Dropbox).
- There were also some potentially interesting functionalities and improvements to the UI of the application that were identified during the development process of the application or by the users who had contact with its current version[1], but which could not be added to the current version of the application due to time constraints. These ideas should be developed upon and presented to additional potential users in order to verify if they would be worthy additions to the application, and if so, they should be implemented at a later date.
- Additional work should also be done in order to iron out the compatibility issues that were identified in Section 4.1.2, as well as any other compatibility problems that might arise due to the additions and improvements that were previously mentioned in this section.

Of course, each and any of these improvements would also require further testing and validation to be performed in order to verify the correctness and stability of the application, as well as to see how its usability and user satisfaction progresses with these changes.

[1] These user-identified improvements are presented in Section 4.2.2.

Bibliography

[1] *About - OpenCV*. 2021. URL: https://opencv.org/about/ (visited on Feb. 11, 2021) (cit. on p. 125).

[2] *AI powered visual search engine | pixolution*. URL: https://pixolution.org/ (visited on July 8, 2020) (cit. on pp. 108, 109, 111, 112).

[3] A. Alves. "Best Photo Selection". MA thesis. Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal: NOVA School of Science and Technology, 2015 (cit. on pp. 29, 120).

[4] A. Atalaia. *Vue.js vs React: We Built an App on Both Frameworks*. 2020. URL: https://www.imaginarycloud.com/blog/vue-js-vs-react-an-app-on-both-frameworks/ (visited on July 20, 2020) (cit. on pp. 14, 16).

[5] *Awesome Duplicate Photo Finder - Find and Remove Duplicate or Similar Images*. 2019. URL: https://www.duplicate-finder.com/photo.html (visited on July 8, 2020) (cit. on pp. 108, 109, 111).

[6] *Axure RP 9 - Axure*. 2021. URL: https://www.axure.com/blog/category/axure-rp-9 (visited on Sept. 15, 2021) (cit. on pp. xv, 38).

[7] *Base64 | MDN*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Base64 (visited on Sept. 16, 2021) (cit. on p. xv).

[8] *Basic Concepts - Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB (visited on July 9, 2020) (cit. on pp. 20, 21).

[9] *Beautiful Free Images & Pictures | Unsplash*. 2021. URL: https://unsplash.com/ (visited on May 27, 2021) (cit. on p. 121).

[10] *Big-O Algorithm Complexity Cheat Sheet*. 2012. URL: https://www.bigocheatsheet.com/ (visited on Aug. 30, 2021) (cit. on p. 86).

[11] *Browser storage limits and eviction criteria - Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria (visited on July 9, 2020) (cit. on pp. 21, 22).

[12] M. Calonder et al. "BRIEF: Binary Robust Independent Elementary Features". In: *Computer Vision – ECCV 2010*. Ed. by K. Daniilidis, P. Maragos, and N. Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792. ISBN: 978-3-642-15561-1 (cit. on p. 116).

[13] *Can I use... Support tables for HTML5, CSS3, etc.* URL: https://caniuse.com/#search=localstorage (visited on July 9, 2020) (cit. on p. 19).

[14] *Can I use... Support tables for HTML5, CSS3, etc.* URL: https://caniuse.com/#search=indexeddb (visited on July 9, 2020) (cit. on p. 21).

[15] J. Cryer and the Huddle development team. *Resemble.js : Image analysis and comparison*. URL: https://rsmbl.github.io/Resemble.js/ (visited on July 8, 2020) (cit. on pp. 108, 109, 111, 112).

[16] *digikam - Features*. URL: https://www.digikam.org/about/features/ (visited on July 8, 2020) (cit. on pp. 8, 108–112).

[17] *Duplicate Photo Finder - Find Duplicate and Similar Photos*. URL: https://www.duplicatephotocleaner.com/ (visited on July 8, 2020) (cit. on pp. 108, 109, 111, 112).

[18] *Duplicate Photo Finder Software - Image Comparer*. URL: https://www.imagecomparer.com/ (visited on July 8, 2020) (cit. on pp. 108, 110–112).

[19] *Duplicate Photos Fixer Pro for - Mac | Windows | iOS | Android*. URL: https://www.duplicatephotosfixer.com/ (visited on July 8, 2020) (cit. on pp. 108, 109, 111, 112).

[20] *Facebook*. 2021. URL: https://www.facebook.com/ (visited on Apr. 9, 2021) (cit. on p. 43).

[21] M. Felismino. "Conceção e Desenvolvimento de uma Aplicação Android para Eliminação Assistida de Fotografias Repetidas". MA thesis. Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal: NOVA School of Science and Technology, 2019 (cit. on pp. 4, 5).

[22] *GitHub - angular/angular: One framework. Mobile & desktop.* URL: https://github.com/angular/angular (visited on Feb. 8, 2021) (cit. on p. 14).

[23] *GitHub - facebook/react: A declarative, efficient, and flexible JavaScript library for building user interfaces.* URL: https://github.com/facebook/react (visited on Feb. 8, 2021) (cit. on pp. 13, 97).

[24] *GitHub - vuejs/vue: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.* URL: https://github.com/vuejs/vue (visited on Feb. 8, 2021) (cit. on p. 15).

[25] *Google*. 2021. URL: https://www.google.com/ (visited on Apr. 9, 2021) (cit. on p. 43).

[26] M. Hassaballah, H. Alshazly, and A. Ali. "Analysis and Evaluation of Keypoint Descriptors for Image Matching". In: Jan. 2019, pp. 113–140. ISBN: 978-3-030-02999-9. DOI: 10.1007/978-3-030-03000-1_5 (cit. on pp. 115–117).

[27] *idb - npm*. 2021. URL: https://www.npmjs.com/package/idb (visited on Feb. 2, 2021) (cit. on pp. 68, 72, 73, 128, 129).

[28] *IndexedDB API - Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (visited on July 9, 2020) (cit. on pp. 20, 21).

[29] *Introduction to localStorage and sessionStorage*. URL: https://alligator.io/js/introduction-localstorage-sessionstorage/ (visited on July 8, 2020) (cit. on p. 19).

[30] *iPhone 11 - Technical Specifications - Apple*. URL: https://www.apple.com/iphone-11/specs/ (visited on July 8, 2020) (cit. on p. 2).

[31] *jsZip - npm*. 2021. URL: https://www.npmjs.com/package/jszip (visited on May 27, 2021) (cit. on pp. 67, 69, 128, 133).

[32] A. Kryzhanovska. *Pros and Cons of Building Single Page Applications in 2019*. 2019. URL: https://gearheart.io/blog/pros-and-cons-building-single-page-applications-2019/ (visited on July 22, 2020) (cit. on pp. 11, 12).

[33] A. Kun. *Understanding Camera Exposure: ISO, Aperture, and Shutter Speed Explained*. 2019. URL: https://www.exposureguide.com/exposure/ (visited on May 30, 2021) (cit. on p. 122).

[34] E. Lamprecht. *The Difference Between UX And UI Design - A Layman's Guide*. 2019. URL: https://careerfoundry.com/en/blog/ux-design/the-difference-between-ux-and-ui-design-a-laymans-guide/ (visited on July 8, 2020) (cit. on p. 16).

[35] P. LePage. *Storage for the web*. 2020. URL: https://web.dev/storage-for-the-web/ (visited on July 9, 2020) (cit. on pp. 21, 22).

[36] S. Leutenegger, M. Chli, and R. Y. Siegwart. "BRISK: Binary Robust invariant scalable keypoints". In: *2011 International Conference on Computer Vision*. 2011, pp. 2548–2555. DOI: 10.1109/ICCV.2011.6126542 (cit. on pp. 114, 116).

[37] *LocalStorage, sessionStorage*. 2020. URL: https://www.javascript.info/localstorage (visited on July 9, 2020) (cit. on p. 19).

[38] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* (2004). URL: http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf (cit. on pp. 114, 119).

[39] *MacBook Pro 13-inch - Technical Specifications - Apple*. URL: https://www.apple.com/macbook-pro-13/specs/ (visited on July 8, 2020) (cit. on p. 2).

[40] E. Macpherson. *The UX Honeycomb: Seven Essential Considerations for Developers*. 2019. URL: https://www.medium.com/mytake/the-ux-honeycomb-seven-essential-considerations-for-developers-accc372a398c (visited on July 8, 2020) (cit. on p. 17).

[41] E. Mair et al. "Adaptive and Generic Corner Detection Based on the Accelerated Segment Test". In: *Computer Vision – ECCV 2010*. Ed. by K. Daniilidis, P. Maragos, and N. Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–196. ISBN: 978-3-642-15552-9 (cit. on p. 116).

[42] S. Mallick. *Homography examples using OpenCV (Python / C ++)*. 2016. URL: https://learnopencv.com/homography-examples-using-opencv-python-c/ (visited on May 22, 2021) (cit. on p. 119).

[43] F. Mano. "A Computing and Storage Server Infrastructure for a Mobile Application". MA thesis. Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal: NOVA School of Science and Technology, 2019 (cit. on pp. 4, 5, 63).

[44] A. McPeak. *What's the True Cost of a Software Bug?* 2017. URL: https://crossbrowsertesting.com/blog/development/software-bug-cost/ (visited on July 16, 2020) (cit. on p. 23).

[45] D. Miller. *UI vs UX: What's the Difference?* 2012. URL: https://www.webdesignerdepot.com/2012/06/ui-vs-ux-whats-the-difference/ (visited on July 8, 2020) (cit. on p. 16).

[46] P. Morville. *User Experience Design*. 2004. URL: http://semanticstudios.com/user_experience_design/ (visited on July 8, 2020) (cit. on p. 17).

[47] Z. Mughal. *Browsers support and limitation towards PWA*. 2019. URL: https://medium.com/@zaffarabbasmughal/browsers-support-and-limitation-towards-pwa-2e6c58cd16d5 (visited on July 9, 2020) (cit. on p. 21).

[48] M. Muja and D. G. Lowe. "Fast approximate nearest neighbors with automatic algorithm configuration". In: *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009, pp. 331–340 (cit. on p. 117).

[49] *MVC Framework - Introduction*. URL: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm (visited on July 23, 2020) (cit. on p. 10).

[50] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993. ISBN: 9780125184052 (cit. on p. 18).

[51] *OpenCV - Feature Matching*. 2021. URL: https://www.docs.opencv.org/master/dc/dc3/tutorial_py_matcher.html (visited on May 22, 2021) (cit. on pp. 59, 62, 117, 118).

[52] *OpenCV - OpenCV*. 2021. URL: https://opencv.org/ (visited on Apr. 5, 2021) (cit. on p. 124).

[53] *OpenCV: Build OpenCV.js*. 2021. URL: https://docs.opencv.org/master/d4/da1/tutorial_js_setup.html (visited on Apr. 6, 2021) (cit. on p. 131).

[54] *OpenCV: cv::FlannBasedMatcher Class Reference*. 2021. URL: https://docs.opencv.org/master/dc/de2/classcv_1_1FlannBasedMatcher.html#details (visited on May 25, 2021) (cit. on p. 117).

[55] *OpenCV: Introduction to OpenCV.js and Tutorials*. 2021. URL: https://docs.opencv.org/master/df/d0a/tutorial_js_intro.html (visited on Feb. 2, 2021) (cit. on pp. 68, 73, 93, 125, 128, 130).

[56] *opencv.js*. 2021. URL: https://docs.opencv.org/master/opencv.js (visited on Apr. 6, 2021) (cit. on pp. 72, 131).

[57] J. Pech-Pacheco et al. "Diatom autofocusing in brightfield microscopy: a comparative study". In: *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000*. Vol. 3. 2000, 314–317 vol.3. DOI: 10.1109/ICPR.2000.903548 (cit. on pp. 61, 124).

[58] N. Periyapperuma. *How I solved the LocalStorage 'QuotaExceededError: DOM Exception 22'*. 2015. URL: https://medium.com/@nisalperi/how-i-solved-the-localstorage-quotaexceedederror-dom-exception-22-b69db46f0cee (visited on July 9, 2020) (cit. on p. 20).

[59] S. Pertuz, D. Puig, and M. García. "Analysis of focus measure operators in shape-from-focus". In: *Pattern Recognition* 46 (Nov. 2012). DOI: 10.1016/j.patcog.2012.11.011 (cit. on pp. 123, 124).

[60] *Pros and Cons of an MVC Framework*. URL: https://jhmediagroup.com/blog/2009/02/11/pros-and-cons-of-an-mvc-framework-2/ (visited on July 23, 2020) (cit. on p. 11).

[61] *Questions tagged [angular]*. URL: https://stackoverflow.com/questions/tagged/angular (visited on Feb. 8, 2021) (cit. on p. 14).

[62] *Questions tagged [reactjs]*. URL: https://stackoverflow.com/questions/tagged/reactjs (visited on Feb. 8, 2021) (cit. on p. 13).

[63] *Questions tagged [vue.js]*. URL: https://stackoverflow.com/questions/tagged/vue.js (visited on Feb. 8, 2021) (cit. on p. 15).

[64] T. Rascia. *How to Use Local Storage with JavaScript*. 2017. URL: https://www.taniarascia.com/how-to-use-local-storage-with-javascript/ (visited on July 8, 2020) (cit. on p. 19).

[65] *React vs Angular vs Vue.js: A Complete Comparison Guide*. 2018. URL: https://medium.com/front-end-weekly/react-vs-angular-vs-vue-js-a-complete-comparison-guide-d16faa185d61 (visited on July 20, 2020) (cit. on pp. 13, 15, 16).

[66] *react-sortable-hoc - npm*. 2021. URL: https://www.npmjs.com/package/react-sortable-hoc (visited on Feb. 2, 2021) (cit. on pp. 68, 73, 128, 131).

[67] *react-zoom-pan-pinch - npm*. 2020. URL: https://www.npmjs.com/package/react-zoom-pan-pinch (visited on Feb. 2, 2021) (cit. on pp. 65, 69, 73, 128, 132).

[68] J. Reis. *Angular vs React: A Comparison of Both Frameworks*. 2020. URL: https://www.imaginarycloud.com/blog/angular-vs-react/ (visited on July 20, 2020) (cit. on pp. 14, 15).

[69] E. Rosten and T. Drummond. "Machine Learning for High-Speed Corner Detection". In: *Computer Vision – ECCV 2006*. Ed. by A. Leonardis, H. Bischof, and A. Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. ISBN: 978-3-540-33833-8 (cit. on p. 116).

[70] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544 (cit. on pp. 58, 62, 114, 116).

[71] J. Sauro. *Measuring Usability with the System Usability Scale (SUS)*. 2011. URL: https://measuringu.com/sus/ (visited on Mar. 18, 2021) (cit. on pp. 24, 25, 75).

[72] *serve - npm*. 2021. URL: https://www.npmjs.com/package/serve (visited on June 30, 2021) (cit. on p. 84).

[73] A. Sharma. *Why You Should Use TypeScript for Developing Web Applications - DZone Web Dev*. 2018. URL: https://dzone.com/articles/what-is-typescript-and-why-use-it (visited on July 8, 2020) (cit. on p. 13).

[74] A. Silva. "Energy Optimization of OpenCV Algorithms for Android". MA thesis. Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal: NOVA School of Science and Technology, 2020 (cit. on pp. 115, 116, 118).

[75] R. Silva. *What Is 4K Resolution? Overview and Perspective of Ultra HD*. 2020. URL: https://www.lifewire.com/4k-resolution-overview-and-perspective-1846842 (visited on July 8, 2020) (cit. on p. 2).

[76] *Single-page application vs. multiple-page application*. 2019. URL: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58 (visited on July 22, 2020) (cit. on pp. 11, 12).

[77] T. Smith. *Tom Smith (_shootthephoto_) | Unsplash Photo Community*. 2021. URL: https://unsplash.com/@_shootthephoto_ (visited on May 27, 2021) (cit. on p. 121).

[78] D. Spanic. *Man in gray long sleeve shirt and white pants sitting on green grass field during daytime photo – Free Yoga Image on Unsplash*. 2021. URL: https://unsplash.com/photos/JFl-QLawHX4 (visited on Aug. 18, 2021) (cit. on p. 154).

[79] *Stack Overflow Developer Survey 2020*. 2020. URL: https://insights.stackoverflow.com/survey/2020 (visited on Feb. 12, 2021) (cit. on p. 27).

[80] *System Usability Scale online with analytics | usabiliTEST*. 2021. URL: https://www.usabilitest.com/system-usability-scale (visited on Mar. 18, 2021) (cit. on pp. 24, 25, 75).

[81] S. A. K. Tareen and Z. Saleem. "A comparative analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK". In: *2018 International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*. 2018, pp. 1–10. DOI: 10.1109/ICOMET.2018.8346440 (cit. on pp. 115–117).

[82] *Test of localStorage limits/quota*. URL: https://arty.name/localstorage.html (visited on July 9, 2020) (cit. on p. 20).

[83] *Tricentis Software Fail Watch Finds 3.6 Billion People Affected and $1.7 Trillion Revenue Lost by Software Failures Last Year*. 2018. URL: https://www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year/ (visited on July 16, 2020) (cit. on p. 22).

[84] *ts-image-processor - npm*. 2021. URL: https://www.npmjs.com/package/ts-image-processor (visited on May 16, 2021) (cit. on pp. 68, 128, 130).

[85] M. Tsarouva. *The Pros & Cons of Single Page Applications (SPAs)*. 2019. URL: https://www.itechart.com/blog/pros-cons-of-single-page-applications/ (visited on July 22, 2020) (cit. on pp. 11, 12).

[86] *TypeScript - JavaScript that scales*. URL: https://www.typescriptlang.org/ (visited on July 8, 2020) (cit. on pp. 13, 97).

[87] *Using IndexedDB - Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB (visited on July 9, 2020) (cit. on p. 21).

[88] *uuid - npm*. 2021. URL: https://www.npmjs.com/package/uuid (visited on Feb. 2, 2021) (cit. on pp. 68, 128).

[89] L. Vieira. *HTML5 Local Storage Revisited*. 2015. URL: https://www.sitepoint.com/html5-local-storage-revisited/ (visited on July 8, 2020) (cit. on pp. 19, 20).

[90] *Web Storage Support Test*. URL: http://dev-test.nemikor.com/web-storage/support-test/ (visited on July 9, 2020) (cit. on p. 20).

[91] *WebAssembly | MDN*. URL: https://developer.mozilla.org/en-US/docs/WebAssembly (visited on May 23, 2021) (cit. on p. xv).

[92] *What is CSAT and how do you measure it?* 2021. URL: https://www.qualtrics.com/experience-management/customer/what-is-csat/ (visited on Mar. 18, 2021) (cit. on pp. 25, 76).

[93] *What is Net Promoter*. 2021. URL: https://www.netpromoter.com/know/ (visited on Mar. 18, 2021) (cit. on p. 25).

[94] *What is Net Promoter Score (NPS)? Everything you need to know*. 2021. URL: https://www.qualtrics.com/uk/experience-management/customer/net-promoter-score/ (visited on Mar. 18, 2021) (cit. on p. 25).

[95] *Why MVC Architecture?* 2017. URL: https://medium.com/@socraticsol/why-mvc-architecture-e833e28e0c76 (visited on July 23, 2020) (cit. on p. 11).

[96] L. Wiese. *Grayscale photography of man in scoop-neck top photo – Free Portrait Image on Unsplash*. 2021. URL: https://unsplash.com/photos/d-MfHM-jHwc (visited on Aug. 18, 2021) (cit. on p. 154).

[97] B. Williams. *How To Use White Balance In Photography*. 2020. URL: https://bwillcreative.com/how-to-use-white-balance-in-photography/ (visited on May 30, 2021) (cit. on p. 123).

[98] *Window.localStorage - Web APIs*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (visited on July 8, 2020) (cit. on pp. 19, 20).

[99] *YouTube*. 2021. URL: https://www.youtube.com/ (visited on Apr. 9, 2021) (cit. on p. 43).

[100] A. Z. *What's the Difference Between Single-Page and Multi-Page Apps*. 2018. URL: https://rubygarage.org/blog/single-page-app-vs-multi-page-app (visited on July 22, 2020) (cit. on pp. 11, 12).

A

Competing Solutions Detailed Analysis

This appendix contains a detailed analysis of how some other solutions similar to the photo|uniq application handle some of the key steps of the application's workflow, in particular image upload, similar image identification, image quality grading, image comparison, and clearing the user's storage space, as well as how they handle offline work. These findings are summarized in Table A.1, and are discussed below and in Section 2.2.

A.1 Image Upload/Import

While looking at the identified competing solutions, we can identify a few different strategies used to upload/import images into the application.

The most basic way to do this is by directly providing images to the application, either by dragging and dropping them or selecting them when prompted from the computer's file system. This is the method used by Resemble.js [15] and pixolution [2] (however, as Resemble.js [15] is a simpler application, only used to compare two images at a time, it only allows the user to upload two images maximum).

This uploading process is made more efficient in other applications, allowing the user to select folders contained in the computer's file system, after which the application will import every picture contained within them. This is the case with digiKam [16], Awesome Duplicate Photo Finder [5], Duplicate Photo Cleaner [17] and Duplicate Photos Fixer [19].

Some applications also accept photographs from sources other than the file system. Pixolution [2] allows the upload to occur by providing a CSV file containing image URLs and optional metadata like keywords or ids. Duplicate Photos Fixer [19] is also capable of using an iPhotos or Mac Photos Library, much like Duplicate Photo Cleaner [17]. However, Duplicate Photo Cleaner [17] can also receive pictures directly from a smartphone, an Adobe Lightroom catalog, a Picasa Album, or even from a Corel PaintShop Pro catalog. Duplicate Photo Cleaner [17] does not allow the comparison of images that were uploaded from different sources.

Image Comparer [18] does not specify in its website how this import process is done.

A.2 Identify Similar Images

Another important feature for the proposed application is the ability to identify similar pictures from within a given set of images.

Here, looking at the analysed competitors, we can observe several levels of comparison using different types of input and output.

Resemble.js [15] is an extremely simple application to which one can only provide two images for comparison, and therefore it only informs its user of the difference in pixels between the two provided images (in percentage format). It does, however, allow the user to choose from five different comparison modes, has an option for resizing the images so that they are compared using the same dimensions, and allows the user to choose which parts of the images to compare, by selecting the dimensions for a bounding box and an ignored box, and even allows the user to determine a particular color to ignore (by providing RGBA values).

Awesome Duplicate Photo Finder [5] does not allow the user to insert any input other than the folders containing the pictures to be compared and what file types to consider. After the analysis, this application informs the user of the similarity level (in percentage format) between pictures from the provided folders that were considered similar (showing two pictures at a time).

Pixolution [2] allows its user to either upload a new image or select an image that was previously uploaded, and then displays the existing images from the application library that the algorithm considered similar, as well as their similarity ratio (ranging from 0 to 1, with 1 being an exact match). It also provides the user with a slider that can be used to select how strict the image similarity should be, from loose to exact.

DigiKam [16] is similar to pixolution [2], in the sense that it allows the user to select a single image from its library and it then provides the users with the images from the library that were considered similar (along with a similarity ratio). It is however different, in the sense that it allows the user to control the similarity level using a range of percentages (for instance, from 90% to 100% similarity).

Duplicate Photo Cleaner [17] is also similar to Awesome Duplicate Photo Finder [5], as it also identifies similar pictures from the provided sources (without the user having to select a single image to compare) while allowing the user to select which file formats to consider, but it provides the user with a few input options to allow them to be more selective with the images to compare, such as an image similarity threshold (controllable using a slider that displays the selected threshold in percentage format), and optional inputs defining the range of file sizes to consider. It then provides the user with a percentage value representing the similarity level of the pictures that were considered similar.

Duplicate Photo Cleaner [17] also contains another mode that allows the user to select a specific image or just a section of an image, and the application then detects images that also contain similar sectors to that image.

Duplicate Photo Fixer Pro [19] compares the provided pictures in two different modes:

exact match, which only detects those images which are the exact same, and similar match, that can be used detect similar images. In this second mode, the user can select how high the matching level should be (from low to high), select the size of the bitmap that is used to compare the images, and narrow down the images that should be considered by selecting a threshold for the maximum difference in timestamp and GPS location of the creation of the two photographs. The application will then show the images that were considered similar (or exactly alike) in separate groups, with no additional comparison information (e.g., similarity ratios), with one of the images being considered as the original (based on the user-provided priority of a few parameters such as capture date, file format, picture resolution, file size, among others).

Image Comparer [18] allows its user to select an image from its gallery, and will then display the top ten most similar images from the ones that were previously provided. The application also contains another mode in which the user selects a similarity threshold, and the application will then display all the image pairs whose similarity ratios were considered to be above the given threshold (in percentage format). This application also allows the user to select if the images should be modified when looking for similarities, by rotating them in different degrees (90º, 180º, or 270º), as well as flipping them horizontally and/or vertically.

A.3 Image Quality Grading

One of the most important features of the photo|uniq application is the ability to automatically grade the objective quality of an image when compared to the ones that are considered as similar, so that the user can better decide which images to keep or delete.

None of the identified competitors has the capacity to automatically grade the quality of the uploaded images, be it on an individual level or in regards to those considered similar.

On a more manual level, digiKam [16] allows its users to provide an image with a grading of 1 to 5 stars, and it also provides the user with some metadata information that can be used to better determine this grading. However, it is reliant on the capacity of the user to interpret this metadata in a meaningful and correct way.

A.4 Image Comparison

One other essential functionality that is provided by the photo|uniq application is its ability to assist its users with the manual comparison of two or more images that were considered similar. As explained in Section 2.1.2, this is done through the use of homography matrix assisted side-by-side image comparison that automatically synchronizes the zoom and panning of both compared images so that they always display the same motif.

A few different tools and options are provided by the identified competing apps to deal with this particular issue.

Some applications have a mode that displays both of the pictures that are being compared on the screen side by side, or one above the other. This is the case with Resemble.js [15], Awesome Duplicate Photo Finder [5], digiKam [16], Duplicate Photo Cleaner [17] and Image Comparer [18]. However, all the applications except Awesome Duplicate Photo Finder [5] and Duplicate Photo Cleaner [17] have a few more tools for this purpose.

Resemble.js [15] also displays the differences between the compared images by showing the user a third image that serves as a visual representation of the differences in the two images, highlighting the pixels that were considered different between then in a bright pink or yellow colour.

DigiKam [16] has synchronized zoom and panning capacities, making it so when the user zooms or scrolls on one of the images, that modification is also replicated on the same X and Y coordinates of the other image (allowing for an easier comparison of specific parts of the images). This synchronization is limited however, since it only synchronizes the absolute image coordinates on both images, with no additional techniques or algorithms being used to ensure that the same motif is shown on both images.

Image Comparer [18] has a special option that highlights the differences between the two images when displaying them side-by-side, by drawing rectangles on top of the sections that were considered different in the images being compared.

The applications that were not mentioned above (pixolution [2] and Duplicate Photo Fixer Pro [19]) do not have side-by-side picture comparison. Instead, pixolution [2] just lists the pictures that were considered similar to an image selected by the user, and Duplicate Photo Fixer Pro [19] agglomerates the images that were considered similar into groups, with one of them being considered the original.

A.5 Clearing Storage Space

After comparing the images and analysing then, the user should be able to select which images should remain in the system, and which should be deleted. It is also important that the state changes that were performed in the application are also reflected in the source from where the images came from, so that only the photos that were selected to remain in the system continue occupying the user's storage space.

Looking at the analyzed competitors, we can see that some of them do not have this functionality at all, and do not allow the user to choose to delete or keep any of the uploaded pictures, either from the application's library or the local computer's memory. This is the case with Resemble.js [15] and pixolution [2]. Still, all the remaining solutions have some way to deal with this issue.

In the case of Awesome Duplicate Photo Finder [5], the application allows the user to either delete or move images to another folder, with those changes being automatically reflected in the local computer. However, the application only lets the user process

one pair of duplicate images at a time, which is an inefficient process if the application identifies several duplicates at once.

DigiKam [16] allows the user to delete images from its library, or move them in between different albums. However, it is unclear in the app features list as described in its web site if these changes are replicated in the local file system.

Image Comparer [18] allows its users to select several pictures at once (either manually or automatically by the application), and then copy, delete or move them (directly affecting the local file system).

Duplicate Photo Fixer Pro [19] lets the user mark several pictures at once, and then delete these pictures from the local file system or move them to a different folder/album. These pictures can either be marked manually or automatically by the application (in this situation, the application simply marks all the pictures that were identified as duplicates).

Duplicate Photo Cleaner [17] also lets the user select several images at once, be it manually or through an automatic function similar to that of Duplicate Photo Fixer Pro [19], but with more selection options (such as selecting all the originals, selecting all the duplicates that have low resolution, among others). After selecting the images, the user can then move or delete them from the local file system. There is also an undo button that can be pressed by the user if they have made a mistake, which will make it so the previous action (or actions, if the user presses it several times) is taken back (this does not work if the user chooses to delete the images permanently from the computer, as opposed to just moving them to the Trash or Recycle Bin).

A.6 Offline Capabilities

Another relevant functionality is the capacity for the application to work while temporarily disconnected, i.e. to perform its operations to some extent while it does not have an active internet connection.

Here, there is an important distinction to be made regarding the analysed competing solutions: some of them are web applications, while the others are desktop applications.

In the web application front, two competing options were considered: Resemble.js [15] and pixolution [2]. Resemble.js is completely functional when the computer does not have any internet connection, as long as the web page has been previously loaded. On the other hand, pixolution is not functional at all, as it relies on being able to process requests and accessing an image database that is not kept on the local computer's storage space.

The remaining competitors are all desktop applications, which gives them the ability to be less reliant on an internet connection (as they can use the local computer's processing power to execute their functionalities). All these applications are completely functional without the use of any internet connection, because they only need to analyse and modify files that can be accessed by the same computer that runs them.

Table A.1: Functionalities and capabilities of the competing solutions.

Application	Platform	Image Upload/ Import	Identification of Similar Images	Image Quality Grading	Image Comparison	Storage Space Clearing	Offline Compatible
Awesome Duplicate Photo Finder	Windows	Folders	Identify from all images	✗	Side-by-side	✓	✓
digiKam	Linux, macOS, Windows	Folders	Must select one image	Basic manual grading	Side-by-side, synchronizing zoom and pan	✓	✓
Duplicate Photo Cleaner	macOS, Windows	Folders, libraries from other sources	Identify from all images	✗	Side-by-side	✓	✓
Duplicate Photo Fixer Pro	Android, iOS, macOS, Windows	Folders, libraries from other sources	Identify from all images	✗	List similar images	✓	✓
Image Comparer	Windows	Unspecified	Identify from all images	✗	Side-by-side, highlighting differences	✓	✓
pixolution	Web Application	Individual images, sets of image URLs	Must select one image	✗	List similar images	✗	✗
Resemble.js	Web Application	Individual images[a]	Must provide both images	✗	Side-by-side, highlighting differences	✗	✓

[a] Up to two images at a time.

B

Image Analysis

As explained in Section 2.1, in order to deliver some of its planned key functionalities, the photo|uniq application needs to analyse the images that are uploaded to it, in order to detect similar images, measure their technical quality, and calculate the homography matrices that can be used to translate point coordinates from one image to another.

In this appendix, we will elaborate on the more technical steps that must be taken to make these functionalities work, as well as introduce OpenCV, the library that is used by the photo|uniq application to perform these steps. The summarization of our decisions when it comes to how the photo|uniq web-based application will handle these image analysis requirements is presented in Section 2.4.

B.1 Image Keypoints and Descriptors

In the context of image analysis, keypoints are points of interest contained in an image that stand out from among their neighbors. When detected using certain algorithms (e.g., Scale Invariant Feature Transform (SIFT) [38], ORB [70] and Binary Robust Invariant Scalable Keypoints (BRISK) [36]), keypoints are resistant to image transformations such as translation, rotation, and differences in scaling, or even distortions caused by changes to the image perspective.

Because of this, when looking at two similar images that show the same motif without any extreme changes between them, it is usually possible to identify the same keypoints on both images. This is visually displayed in Figure B.1, which showcases the matching keypoints found in two similar images that show the same motif captured with different perspectives and orientation.

While detecting image keypoints using certain keypoint detection algorithms, we are also provided with a descriptor for each detected keypoint, which contains the information required to describe and classify this keypoint, so that it may be used for our implementation purposes (such as, for instance, keypoint correlation between two different images). The combination of an image keypoint and its associated descriptor is called an image feature.

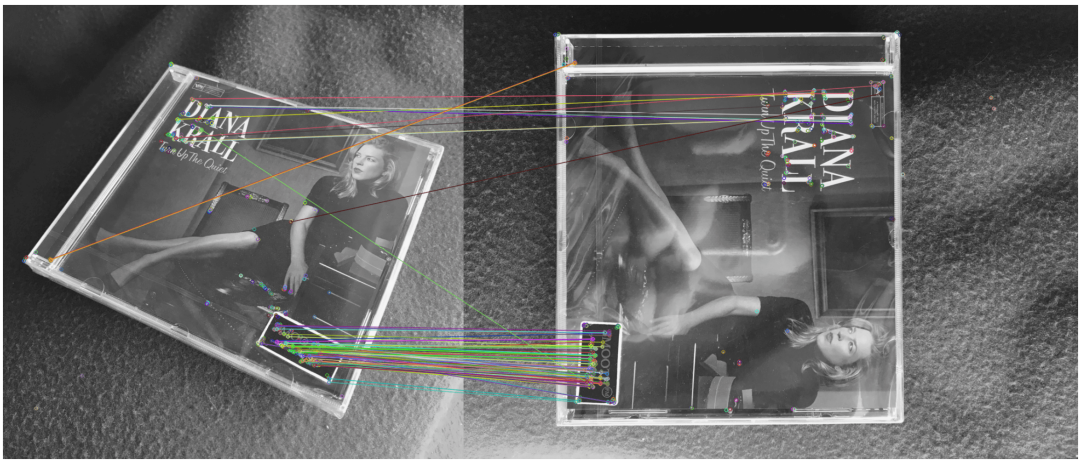


Figure B.1: Visual display of the matching keypoints found in two similar images.

Keypoints and descriptors are relevant concepts to the photo|uniq application since, in theory, one can correlate the keypoints detected in two images to see if these images are similar to each other. It is also by correlating image features that we are able to estimate the homography matrices that are used to translate image point coordinates between similar images, in a process which is further elaborated in Section B.2.

In order to find an image's set of features, one simply has to apply a feature detection algorithm to that image, which will then return the identified set of keypoints, as well as their associated descriptors. These keypoints and descriptors can then be used by the application to deliver its functionalities to its users.

There are two main types of image features that should be differentiated when considering the image analysis needs of the photo|uniq web-based application: those which use **floating-point descriptors**, and those which use **binary descriptors**.

The main difference between these two types of descriptors is made explicit in their names: floating-point descriptors store the description of their associated keypoints using floating-point variables, while binary descriptors do this same thing using binary values [26]. Because of this, floating-point descriptors tend to be more complex and require more computational power and time to identify and correlate, as well as more storage space to store [26, 81, 74].

On the other hand, their higher complexity can often be used to generate more specific and detailed descriptors, which might produce more accurate results than those provided by the binary descriptors [26].

Since the current version of the photo|uniq web-based application's image processing component[1] runs solely on the user's web browser, and this work's use case deals with potentially hundreds of high-resolution images, it was important to take temporal and computational costs into account when deciding the feature detection algorithm of choice. In light of this, we chose to focus our attention in the feature detection algorithms which

[1] Which is further elaborated upon in Section 3.1.

use binary descriptors, in order to improve the execution times and computational costs of our solution.

After making this distinction, we considered two feature detection algorithms for usage in the context of the photo|uniq web-based application: ORB [70] and BRISK [36].

These two algorithms were chosen over other binary descriptor based options due to their high temporal and computational efficiency, as well as their appropriately accurate results when it came to feature matching, as shown in the evaluations performed by Hassaballah *et al.* [26], Tareen & Saleem [81], and Silva [74].

ORB [70] is a feature extraction algorithm which detects and describes image features using rotation resistant versions of the Features From Accelerated Segment Test (FAST) keypoint detector [69], and Binary Robust Independent Elementary Features (BRIEF) descriptor [12].

In their work, Rublee *et al.* built upon FAST and BRIEF to improve their resistance towards image rotation, and in doing so defined oFast (oriented FAST) and rBRIEF (rotation-aware BRIEF).

ORB also employs scale pyramids of the analysed images to detect features on multiple scales, which make its detected feature sets resistant to image scale changes.

Through these improvements and the already existing advantages of FAST and BRIEF, ORB boasts high computational and temporal performance, while being invariant to image rotation and resistant to noise and limited affine transformations (including scale changes).

BRISK [36] is a feature detection algorithm which uses a modified version of the Adaptive and Generic Corner Detection Based on the Accelerated Segment Test (AGAST) feature detection algorithm [41] in conjunction with the FAST corner score, to detect and filter corners contained in an image, while simultaneously using the s score provided by FAST to build a scale-space pyramid. This pyramid is then used to select image features while taking into consideration multiple different scales, thus making BRISK more resistant to changes in image scale.

Additionally, BRISK also identifies the characteristic direction of every detected keypoint, in order to achieve rotation invariance.

These presented mechanisms make it so that, much like ORB, BRISK is also considered to be invariant to image rotation and resistant to limited affine transformations (including scale changes).

In order to decide between these two feature detection algorithms, we analysed the results obtained by Silva [74], which were especially relevant to our use case, as his work was also developed with particular focus towards the photo|uniq application. In his work, when comparing the results obtained by applying ORB and BRISK to the problem of feature detection, Silva found that ORB had a lower execution time, had less CPU usage, and had a lower energy consumption than BRISK while detecting image features.

These results are also consistent with those found by Tareen & Saleem [81], who found that of all their tested feature detection algorithms, ORB had the lowest computation time

during the process of feature detection. Additionally, Tareen & Saleem also found that while testing ORB in a bounded state (or, in other words, when limiting the number of features that ORB could find to a maximum of 1000 per image), ORB also had lower feature matching and image matching times.

Hassaballah *et al* had different results in their analysis of these two algorithms, finding that in their tests, BRISK was 2.34× faster than ORB at extracting features from images.

An additional relevant point that should be acknowledged when comparing these two algorithms is that ORB's feature descriptors have half the size of those used by BRISK, with ORB's descriptors occupying 32 bytes per detected feature, and BRISK's occupying 64 bytes per detected feature [26, 81].

After seeing these results, we decided to use ORB to detect image features in the current version of the photo|uniq web-based application's image processing component[2], due to its apparent lower energy, temporal, and storage cost.

After obtaining the keypoints and descriptors of two images, we can attempt to correlate them them in order to try to find matching points between these two images.

To correlate keypoints obtained from two images, we feed the two sets of descriptors associated with these keypoints to a feature matching algorithm, which will then return a set of matches (or, in other words, matching keypoints).

Two feature matching algorithms were considered for the image processing needs of the photo|uniq web-based application: a Fast Library for Approximate Nearest Neighbors (FLANN) based matcher, and a Brute-Force based matcher [51], both provided by the OpenCV library[3].

FLANN [48] is a software library containing a collection of different Nearest Neighbour based algorithms (in particular, the *randomized kd-tree* and *hierarchical k-means tree* algorithms), which are optimized for fast searching in large datasets, as well as for high dimensional features [51].

The library is also designed to be able to automatically identify which algorithm and parameters for this algorithm are more appropriate for a given dataset, thus improving efficiency for each specific dataset, while simultaneously removing the the need for the developer to study up on and choose a particular algorithm and parameters during the development process. The library also allows for manual selection of which algorithm to use, as well as manual parameterization of the algorithm, if this option is preferable.

The FLANN based matcher provided by the OpenCV library [51] works by first training an index on a set of "training descriptors", and then using this index in conjunction with the nearest neighbours based algorithms provided by FLANN, to find the best matches for every descriptor [54].

[2]The application image processing component is further elaborated upon in Chapter 3, and particularly in Section 3.1.

[3]OpenCV will be further discussed and introduced in Section B.4.

As its name implies, the **brute-force based matcher** provided by the OpenCV library [51] takes a brute-force approach to the problem of feature matching. This essentially means that, for every feature that is being matched, the brute-force based matcher will compute a distance value between that feature and every feature that was detected for the second image. After calculating all the distances, the algorithm will then select the closest feature (or, in other words, the one whose distance has the lowest value) to be a match with the originally selected feature.

When attempting to match image features which contain binary descriptors, this distance value is usually calculated using the Hamming distance measure, which measures distance by verifying in how many positions the two image descriptors contain different values from each other.

The decision between which feature correlation algorithm should be used in our implementation was also informed by the results of Silva's analysis of these two algorithms in the context of the photo|uniq application concept [74].

In his work, Silva tested using the FLANN and brute-force based matchers (which will henceforth be called FLANN and Brute-force respectively, so as to improve this document's readability) to correlate image features detected by the ORB feature detection algorithm, and reached the following conclusions [74]:

- While testing in a desktop environment, Brute-force's execution speeds consistently fall in the 20-30 millisecond (ms) range, while FLANN's execution speeds are typically much slower, ranging between 150 and 165 ms.

- FLANN seems to be more discriminant with what is considered a matching keypoint than Brute-force, and is thus considered to produce more accurate results.

- Image resolution seems to have no impact in the execution time of either algorithms, with both of them having consistent execution times while handling scaled down versions of images (ranging from 10% to 90% of the image's original resolution).

- FLANN seems to be very slightly less costly in terms of energy consumption when compared to Brute-force.

- Brute-force appears to be more compatible with the image features that are identified by the ORB feature detection algorithm than the FLANN based matcher. This is due to the fact that Brute-force is capable of using the Hamming distance measure to compare binary descriptors, while FLANN (which was designed to compare floating-point descriptors) is not, and requires alterations to be made to the features detected by ORB so as to make them become floating-point descriptors.

With these conclusions in mind, and since the brute-force based matcher seems to be much faster than the FLANN based matcher, and is more compatible with the image

features detected using ORB (without requiring any alterations to be made to this algorithm), we chose to use the brute-force based matcher for the purposes of image feature correlation in the photo|uniq web-based application.

However, even after the process of feature correlation is performed, it is still possible that these returned matches are not all correct, with some of them possibly being bad (or false) matches. We can further narrow down these matches into good matches through the use of the ratio test proposed by Lowe in the same paper where he introduced SIFT [38].

Lowe's ratio test works by assuming that every image point will only have at most one matching point in a second image, which means that if we try to match it to two points, then the worst match will be an example of random noise. Therefore, if the first (and best) match is different and better than the second (and worst) match by a sufficiently large margin, we can consider this first match to not be random noise, and therefore a good (or valid) match. Using this logic, we consider a match to be good if the distance between the two matching keypoints is lower than the distance between the second closest match multiplied by a defined ratio with a value contained between 0 and 1 (for instance, 0.8).

After narrowing down the matches into good matches, we can then use this set of good matches to verify the similarity of the two images and, if they are identified as similar, calculate the homography matrices that can be used to translate point coordinates between them.

B.2 Homography Matrix

A homography matrix is a 3×3 matrix that represents a mapping of points from one plane to another. These matrices can be estimated using a set of at least four matching points contained in any two planes [42] (such as, for instance, the matching keypoints found through the process of keypoint correlation that was described in Section B.1).

Homography matrices allow us to translate an image point's coordinates to the point coordinates that display that same point's motif on a similar image. This translation is done though the use of matrix multiplication between the homography matrix and a 3×1 matrix created using the coordinates of the original point.

For instance, in order to translate a 2D point with (x,y) coordinates to an equivalent 2D point with (x',y') coordinates using a previously estimated homography matrix (H), we simply need to follow the formula shown below [42]:

$$H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

This technique is extremely useful in the context of the photo|uniq application, since it is through its use that we are able to perform the side-by-side comparison of similar images with zoom-and-pan synchronized in the image's motifs, and not its coordinates.

While comparing two images A and B, if the user performs a pan operation on A, three steps occur:

1. The application creates a 3×1 matrix using the image coordinates of the central point of image A's display (which contains a certain motif);
2. The application multiplies a previously estimated homography matrix (used to translate points from A to B) with the matrix which was created in the previous step. This calculation results in a 3×1 matrix containing the coordinates of that same motif on image B;
3. The application makes it so the central image coordinates that are displayed on image B are the ones that were found through the previously described multiplication, thus making sure that the two images are simultaneously displaying the same motif.

B.3 Measuring Image Quality

The automatic grading of the objective quality of images is one of the key functionalities of the photo|uniq application concept, as it allows the user to easily sort the images contained in a similarity group based on their quality.

Doing this allows users who are not very versed in the technical side of photography to have a better understanding of how their similar photographs are measured against each other, which will in turn help them make decisions about which similar images should be kept or not.

There are several image characteristics and metrics which can be taken into account while attempting to measure the objective quality of an image. In his work, Alves [3] proposed a workflow which should be followed when analysing the relative quality of similar images, which is composed of four different steps:

- Image focus analysis or, in other words, an analysis performed to see if the image is blurry or out of focus (an example of corruptions related to image focus can be seen in Figure B.2);
- Detection of motion blur, or in other words, a verification to see if the image contains any blurring caused by the photographed objects being captured while moving (e.g., the photo shown in Figure B.3);
- Detection of image noise or, in other words, grain in the image which is not integrated with the remainder of the picture, and thus corrupts the image viewing experience (such as with the rightmost image shown in Figure B.4);
- The analysis of the colors present in the image, in order to determine its level of exposure (with an underexposed image usually being composed of darker colours,

Figure B.2: Comparison between a properly focused and a blurry (or unfocused) image.

Figure B.3: Example of image containing motion blur. Photo by Tom Smith [77] on Unsplash [9].

Figure B.4: Example of image corruptions related to noise. In this presented case, the noise on the rightmost was artificially increased.

and an overexposed image being composed of brighter colours, as shown in Figure B.5), as well as the general temperature of the colours which make up these images (an example of the same image being shown with different general colour temperatures can be seen in Figure B.6).

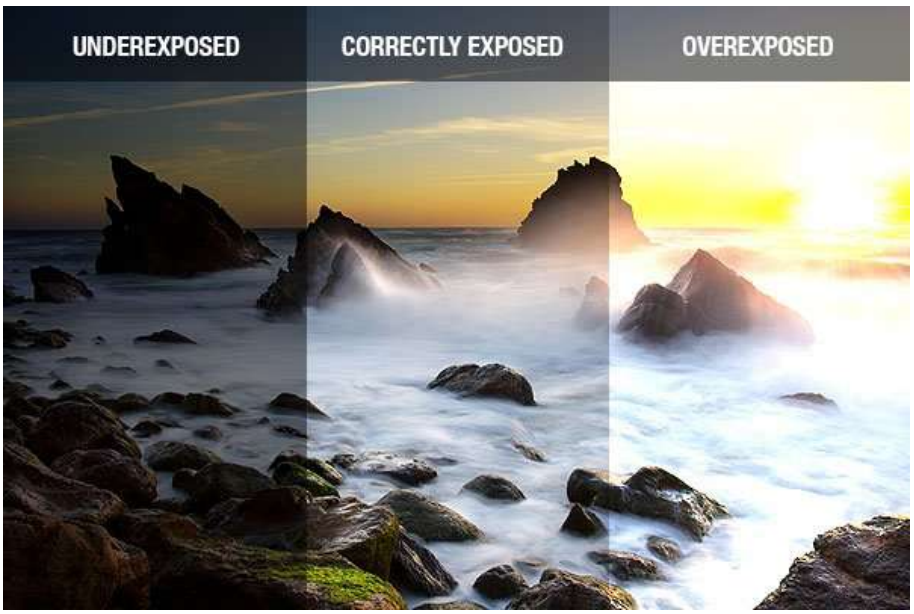


Figure B.5: Example of the same image being shown with different exposure levels, and how this difference affects the visual display of the images themselves [33].

Since the main focus of the work associated with this dissertation was to implement

Figure B.6: Example of the same image being shown with different colour temperatures (through the alteration of this image's white balance), and how this difference affects the visual display of the images themselves [97].

the main application component of the photo|uniq web-based application[4], most of the development time and effort was put into the implementation of this component, with only some effort being drawn to the implementation of the photo processing component.

Because of this, in the current version of the photo|uniq application, the automatic analysis of an image's objective quality was implemented using a simplified version of the previously presented workflow, in which only the first step of the analysis, the image focus, was taken into account for the purposes of ascertaining the quality of the images.

While this version of the image quality analysis process is not ideal, it was decided that it was sufficiently effective for now. This way, a functioning (albeit simple and incomplete) version of the image analysis process was implemented, which allowed us to continue to focus on the development of the main application component. In the future, further development time and effort might be put into improving this workflow in a future work, so as to make it more complex and comprehensive in its image quality analysis capacity.

That being said, some consideration still had be made in order to decide which focus detection algorithm should be used for the implementation purposes of the current version of the photo|uniq web-based application.

below Competing Solutions Detailed Analysis1. Categoria/perfil dos destinatários / Requisitos obrigatórios:

In their analysis and comparison of the results provided by 36 different focus measuring operators, Pertuz *et al.* [59] reached a conclusion that Laplacian-based operators have the best overall performance in terms of focus detection in normal imaging conditions

[4]The three components of the photo|uniq web-based application are identified and discussed in Section 3.1.

(i.e., when the image is not artificially worsened by adding noise or reducing contrast or image saturation).

Taking these results into account, we chose to use a Laplacian-based algorithm to measure image focus levels in our implementation of the photo|uniq web-based application.

The Laplacian is a second derivative operator which, when applied to an image, will highlight sections of the image where rapid changes in pixel intensity happen (which are typically associated with edges contained in that image).

Laplacian-based methods of computing an image's focus level work based on the assumption that blurrier images contain less edges than images with a higher focus level (since blurrier images tend to contain less extreme changes in pixel intensity level). Through this assumption, it becomes possible to compute a photo's focus level by applying the laplacian operator on that photo, which will return an image where the edges of the original photo are highlighted. We can then use the resulting image to generate a metric through which we can compare the amount of edges in one photo to the amount of edges in another one (and therefore, compare the focus level of two pictures).

Pech-Pacheco *et al.* [57] proposed a method that measured the focus level of images by calculating the variance of the image resulting from the application of the laplacian operator to these images. When using this method, a higher variance is associated with more edges, and therefore with a more focused image.

Looking at a specific example, if we use this method to analyse the focus level of the two images shown in Figure B.2, we obtain a focus level of 1500 regarding the leftmost image, and a focus level of 1.85 on the rightmost image (which is consistent with these two images, since the rightmost image is a much blurrier image than the leftmost image).

It is worth mentioning that the laplacian-based methods of detecting an image's focus level will detect any sudden changes in pixel intensity, and not just those associated with edges contained in the image. Because of this, these methods are particularly vulnerable to image containing a lot of visual noise (such as the one shown to the right in Figure B.4), which also may cause a sudden change in pixel intensity.

Nevertheless, laplacian-based methods have been shown by Pertuz *et al.* [59] to have the best overall performance in normal image conditions, and are thus still a viable option for the focus detection needs of the photo|uniq web-based application.

B.4 OpenCV

Open Source Computer Vision Library or OpenCV [52] is a multi-platform open source computer vision and machine learning software library originally developed by Intel that contains more than 2500 optimized algorithms that can can be used to provide an application with the ability to deliver several different image analysis and computer vision functionalities, such as detecting an image's keypoints, correlating these keypoints, calculating homography matrices, face detection and recognition, camera movement

tracking, among others [1]. It is written natively in C++, but also has Python, Java, and MATLAB interfaces [1].

There also exists a JavaScript binding called OpenCV.js [55] which contains limited support for a selected subset of the OpenCV functions and algorithms. This binding runs directly in the user's browser through the use of WebAssembly, and can be used to call and execute this limited subset of functions and algorithms in the front end of a web or web-based application, without any need to implement a local or remote back end.

While there are definite advantages associated with the use of OpenCV.js, there are also some disadvantages.

On the one hand, it allows a web application to perform several image analysis operations (e.g., keypoint detection and correlation, homography matrix estimation, among others) directly in the browser, which saves a lot of development time and effort since, as mentioned before, it avoids the need to implement a local or remote back end to handle these image analysis requests. Running directly in the client browser also means that the application is capable of performing these operations without having the need to connect to a remote server, which might potentially reduce (or even remove) the application's reliance on an internet connection.

On the other hand, it is a very heavyweight file, occupying 8.6 MB in its default state[5], which might severely increase the application size. Additionally, running the image analysis and computer vision operations provided by OpenCV directly in the user's browser might also put some strain on their computer, since it does not relegate these heavy operations to a remote back end infrastructure. OpenCV.js also has a somewhat incomplete documentation, and sometimes requires a trial and error approach to software development, which also negatively impacts the development efforts and experience.

Nevertheless, we found OpenCV.js to be an extremely valuable tool in allowing the photo|uniq web-based application to achieve its image analysis requirements, and thus used this library in the context of developing this work.

[5] We mention a default state here since it is possible to add additional OpenCV modules and functions to OpenCV.js other than the default ones, which will change the file's size.

C

Generic Functionalities of the photo | uniq application

This appendix serves to present and discuss the functionalities of the photo|uniq application that are both not particularly relevant nor distinct enough from their equivalents found in other web or web-based applications (as opposed to the photo|uniq application's more specific and unique functionalites, which are presented in Section 3.4).

C.1 Sign Up and Log In

We decided that it was more relevant for the work at hand to first dedicate our efforts to the development of the application's key functionalities (or, in other words, the functionalities that revolved around image analysis and identification of similar photos, as well as image quality comparison). Because of this, the currently existing facets of the application that deal with user-related matters are mostly visual, with limited functionalities and verification. Further work and time will be dedicated to these parts of the application when an infrastructure that is capable of dealing with these authentication and registration needs is implemented, upon which the already developed views will the integrated with this infrastructure.

In the current state of the application, the user will encounter the log in page whenever they access the application. At the time of the writing of this dissertation, this page is little more than a visual barrier, and the user will be able to access the application by using any combination of username and password that is not composed of empty strings.

Similarly to this, the sign up page is also little more than a visual display that will not do anything with the information that the user supplies in the fields of username, email or password, and will simply redirect the user to the log in page when they press the submit button, as long as none of the input fields are empty and the two fields used to input the password contain equal strings.

No verification is performed on the user-submitted data other than that which was explained in the two previous paragraphs.

C.2 Page History Navigation

One of the main drawbacks caused by developing a web-based single page application is that, since the application is composed by a single web page, the user cannot use the browser's back and forward buttons to quickly navigate between the pages that they have previously visited. In order to compensate for this disadvantage, we implemented a functionality to allow the users to navigate between the most recently accessed "pages" in the photo|uniq web-based application (thus, providing an alternative way to navigate their "page history" while using the application).

In order to do this, we first needed to implement a class that could be used to identify the pages that were being accessed, so that we could store objects of this class for later access. These objects contain information regarding the type of page that is being accessed, as well as the information of the photo|uniq element (if any) that is being used to access that menu (for instance, if the user had accessed the page related to a similarity group found using the path "workspace1/folder3/Similarity Group 1", then the object related to that page would have its functionality field be equal to "Similarity_Group", its workspace field be equal to "workspace1", and so on, and so forth). This way, if the user later returns to a page that had already been accessed previously, the application simply needs to read this information in order to dynamically generate the view associated with that page.

We also needed to decide how many of these objects the application could store at a time. Up to 20 of these objects are stored by the application in the current version of the application. However, this number can be altered by changing a constant in the code in order to store information related to either less or more menus.

It is also possible that, through the use of the application's functionalities, certain pages that had been previously accessed stop being valid. This happens when the user causes a change to the data related to the application's session and/or persistent state (e.g., when an element is deleted, or the photos contained in a group are promoted to the gallery). In these situations, it does not makes sense to allow the user to re-visit any previous pages associated with these elements, since doing so would not be consistent with the changes performed by the user to the application state, and would not even work because, since the pages are generated dynamically, if the element in question is no longer there, then the user would only see an empty page.

Several functions were implemented that "purge" these invalid pages from the page history. Therefore, when a change is made to the application data that might cause a page stored in the history to be invalid, we simply need to call these functions and provide them with the parameters that can be used to identify the invalid pages, so that the functions themselves can then remove them from the history.

D

External Packages and Libraries

As mentioned previously in Section 3.5, seven external packages and libraries were used to complement the capabilities of the base TypeScript language and React library for the development purposes of the photo|uniq web-based application: UUID [88], IDB [27], ts-image-processor [84], OpenCV.js [55], React Sortable HOC [66], react-zoom-pan-pinch [67], and JSZip [31].

This appendix will serve to provide a short introduction to each of these packages and libraries, followed by an explanation of how they were used to enhance the application's capabilities.

D.1 UUID

As discussed previously in Section 3.2, the photo|uniq application deals with several different elements, from workspaces and albums to user uploaded photos (which by themselves have the potential to reach a count in the order of the hundreds or even thousands of elements). While there are several different possible methods that could be used to identify these elements, one of the simplest ones is to assign a unique id to them. For these purposes, we integrated UUID [88] (a JavaScript package registered in npm that allows for the generation of unique IDs) with the source code of the photo|uniq web-based application.

This way, whenever a new element that needs to have a unique id (e.g. a workspace photo or similarity group) is added to the application, a unique string id is generated using UUID [88] and stored in the object that will be used to represent the element in the application's persistent and session data. This id can then be used to search and load this element when necessary, as well as to create relationships between two different elements (for instance, the id of the parent workspace of a similarity group is stored in the object that represents this similarity group, as discussed in Subsection 3.2.1).

D.2 IDB

It was decided early on in the planning stages of the photo|uniq web-based application that the application should strive to provide its users with as much support for offline work as possible. For these purposes, it was important to be able to store data in the user's local browser storage, so that the application could be relatively self-sufficient when dealing with data that had previously been requested from the application's data management and photo processing components, which had the potential of being set up and run in a remote server infrastructure.

Through our research, we reached the conclusion that there were two main solutions that allowed us to achieve this without resorting to the implementation of a local back end: LocalStorage and IndexedDB.

However, both of these options had their shortcomings, with the most relevant ones being an extremely limited storage space when dealing with LocalStorage (which would not be able to store all the elements that the photo|uniq application required) and a more complex and less than developer friendly API when dealing with IndexedDB (which also had the risk of potentially making the application bulkier, more verbose, and prone to bugs). As it was clear that the only functional option among the two considered ones was IndexedDB, the next step was to try to mitigate the issues that were identified with this API. It was for these purposes that IDB [27] was used in the photo|uniq web-based application's front end.

IDB [27] is a TypeScript package registered in npm that provides some improvements to the usability of IndexedDB, among which the most relevant ones are a more developer friendly API and the integration of IndexedDB with TypeScript Promises (which allow for a higher and much more straightforward control over the asynchronous nature of IndexedDB).

IDB was an invaluable tool in the efforts towards making the photo|uniq web-based application's front end store values in IndexedDB as easily and cleanly as possible, and at the same time provided the option to (when necessary) wait for these values to be retrieved from the database before continuing the execution of the application's functionalities.

D.3 Ts-image-processor

In certain specific situations, the photos uploaded to the application will need to be listed to the user, such as when displaying the contents of an element that contains photos (as discussed in Subsection 3.3.3), or to show the sibling photos of a photo or photos that are being displayed to the user (as shown in Subsections 3.3.5 and 3.3.6).

Trying to show multiple images in this fashion has the potential to increase the time needed to fully generate the page shown to the user, especially if dealing with several high resolution images (which is consistent with the use case of the photo|uniq application).

In situations such as these, the browser will need to dynamically resize the images to thumbnail size when generating the page. This requires some time and computation power, and has the potential to drastically worsen the user experience.

In order to mitigate the negative effects of these photo listing operations, we used ts-image-processor [84] to generate thumbnail versions of the uploaded photos, which could then be displayed when listing photos, this way removing the need to dynamically resize the shown images.

Ts-image-processor [84] is a TypeScript package registered in npm that provides some image processing operations such as image resizing and sharpening, that can be used for the purposes of generating an image thumbnail.

In the context of the photo|uniq application, this package is used during the process of uploading images to the application. Whenever a workspace photo object is created, a method is called that uses some of the functions provided by the ts-image-processor package to resize and sharpen the uploaded photo, in order to create its thumbnail, which can then be later used to display that photo in smaller resolution when needed.

This thumbnail is also re-used when creating and listing the deleted version of workspace photos, as well as the gallery photos which are created by the promotion of favorite photos into the photo|uniq gallery.

D.4 OpenCV.js

It is essential to have access to image analysis capabilities in order to provide some of the functionalities of the photo|uniq application to its users (namely, the automatic grouping of similar images, the automatic sorting of images based on their technical quality, and the side-by-side comparison of similar images assisted by homography matrices).

The lack of an existing back end infrastructure capable of providing these capabilities meant that we had to make a decision early on in the development process to either implement this back end ourselves, or perform these operations in the application's front end itself. For these purposes, and choosing to focus on the second of the two presented options, we chose to integrate our application with OpenCV.js.

OpenCV.js [55] is a JavaScript binding of OpenCV that allows some of the many functions and methods of the complete OpenCV Library to be called and executed directly in the user's browser (or, in other words, in the front end portion of a web/web-based application). Through these available functions, it becomes possible for the application's front end to contain the infrastructure required to deliver the image analysis capabilities that are essential to the proper functioning of the photo|uniq application.

In order to access these functionalities in the front end of a web-based application, the application developers need to have an "opencv.js" file accessible to this front end during the application's runtime[1]. After doing so, the functions, objects and methods provided

[1] The developers can get access to a "opencv.js" file by either directly downloading the prebuilt version

by OpenCV.js can be accessed by the application during its runtime through the loading of this file (for instance, through a call to the JavaScript *require* function).

In the current version of the photo|uniq web-based application, and as was previously mentioned in Section 3.1, OpenCV.js was used in the implementation of a TypeScript class in order to allow the application to deal with its required image analysis needs.

D.5 React Sortable HOC

One of the goals of the photo|uniq application is to help its users select which photos among several identified similar ones should be kept or deleted. For these purposes, it is important to allow the user to manually change the order of the photos contained in similarity groups, in order to help them sort these photos by their desired criteria, be it objective quality, personal preference, or even a mixture of both.

During the process of UX and UI development, it was decided that, if possible, we would aim to make it so the manual sorting of these photos could be performed by dragging them and placing them in the desired place among the similarity group's photo list.

React Sortable HOC [66] is a JavaScript package registered in npm that provides two functions (*SortableContainer* and *SortableElement*) that can be used together to turn any JSX element into a draggable element, with support for horizontal and vertical lists, and even grids of elements.

This package was an ideal fit for the desired UX of manually sorting the photos in a similarity group, since it provided a clean and simple way to make the photos draggable, while simultaneously making the UI extremely smooth and responsive. Furthermore, the support for grids of elements is essential to the photo|uniq application, since the lack of a defined upper limit to the number of photos contained in a similarity group means that it is possible for the list of photos to occupy several rows.

In order to use React Sortable HOC [66] to turn a UI element displayed by the photo|uniq web-based application into a draggable element, we simply need to use the *SortableContainer* function to create a container in which the elements displayed within it can be dragged (e.g. a photo list), and then fill this container with the desired draggable elements (e.g. the photos) that can be created using the *SortableElement* function. Both of these functions return a JSX element, which can be integrated with the remainder of the photo|uniq web-based application in a very simple and quick fashion, thus immensely simplifying the development process of this UI.

that is provided by the OpenCV Team [56], or building it themselves using the instructions that are provided in the OpenCV docs [53].

D.6 React-zoom-pan-pinch

As previously mentioned in Section 1.2, in today's world it is an extremely common occurrence to have photo resolution be higher than screen resolution. Because of this, in order to display these high resolution photos in lower resolution screens, the photos can either be shown in their true resolution, which means that the whole photo cannot be simultaneously shown on screen, or they can be scaled down, in which case the photo will not be shown in its true quality, potentially hiding imperfections and relevant details.

This is a particularly relevant issue when considering the use case of the photo|uniq application, as the true quality of the photographs is an essential metric to be considered when trying to decide which ones of several similar images should be kept or not.

Because of this, when developing the UI of the photo visualization and photo comparison pages, it was necessary to allow the user to be able to see both the complete images on screen (to be able to visualize and consider them and their entirety) and also to be able to zoom in on these images and visualize their details in higher scale, in order to analyse how certain particularly relevant motifs are represented in the photographs.

To deal with these presented concerns, we decided to utilize react-zoom-pan-pinch in the development of the photo|uniq web-based application. React-zoom-pan-pinch [67] is a TypeScript package registered in npm that allows the developer to make it so any JSX element has zoom and panning capabilities, while simultaneously allowing a lot of customization and limitation to these capabilities, so that they may better suit the specific needs of the application.

This high potential for customization is an essential requirement for the implementation of the photo comparison page's UI, since the use of homography matrices to maintain focus on the same motif in both of the compared photos meant that it was necessary for the implemented solution to be able to change the panning of one photo based on the output of the application of a homography matrix with the point coordinates of the point that is being displayed on the center of the other photo.

In order to use react-zoom-pan-pinch [67] to turn a JSX element into a zoomable and pannable element, the developer simply needs to use a function (*TransformWrapper*) and a React Component (*TransformComponent*) that are both provided by the react-zoom-pan-pinch package.

TransformWrapper is a function that returns a JSX element that will serve as a wrapper for an instance of *TransformComponent*. This function can be customized through the use of several different optional parameters.

TransformComponent is a React Component that, if contained in the JSX element returned by a call to *TransformWrapper*, can be zoomed and panned by the actions of a user (i.e. zooming in or out via a scrolling or pinching motion, as well as panning via clicking and dragging).

Therefore, if the developer wishes to allow a user of their application to zoom and pan on an image, they simply need to display this image inside of a *TransformComponent*

that is contained in the JSX element returned by a call to *TransformWrapper*.

In the context of the photo|uniq web-based application, we used react-zoom-pan-pinch to implement the UI of the photo visualization and photo comparison pages.

In the case of the photo visualization pages, this is done by simply following the previously explained steps, and essentially containing an *img* element representing the displayed main image inside of a *TransformComponent* that is contained in the JSX element resulting from a call to *TransformWrapper*.

The implementation of the photo comparison page also followed the same basic idea behind these steps, but it was a more complicated and complex process, requiring two different calls to *TransformWrapper* and two different *TransformComponent* components to be used (one contained in each of the resulting JSX elements that were returned by *TransformWrapper*). Some extra parameters were also provided to the calls done to *TransformWrapper* in order to both allow for the direct manipulation of the scale and translation values used for the zoom and panning of the images, as well as to slightly alter the functions that were called when the user executed the actions that would cause a zoom or panning effect. This implementation is further described in Subsection 3.3.6.

D.7 JSZip

As previously explained in Section 3.2, the elements contained in the photo|uniq gallery are built around a specific structure. Looking at this structure from a top down perspective:

1. The photo|uniq gallery may directly contain albums;
2. The albums may directly contain album folders and gallery photos;
3. The album folders may directly contain gallery photos.

Because of this, when implementing the function to download the contents of the photo|uniq gallery into the user's local file system[2], we decided to attempt to replicate the same general structure in the downloaded files.

In theory, this could be easily done by interpreting the albums and album folders of the gallery as being file system directories (or folders), and the gallery photos as being image files contained in these folders.

In order to integrate this presented path structure with the function used to download the user's photo|uniq gallery contents, we used JSZip [31] to create and edit a zip file to follow the presented translation scheme, so that it may be later downloaded by the user.

JSZip [31] is a JavaScript package registered in npm that provides several functions and classes that allow web and web-based application to create, read, and edit zip files directly in JavaScript and Typescript.

[2]This function is presented and explained in Subsection 3.4.6.

We use four functions provided by the JSZip library in the current version of the photo|uniq web-based application:

- The *JSZip()* object constructor, which creates an empty instance of a JSZip object. This constructor is used to create the blueprint of an empty zip archive, which may be then populated with the elements contained in the photo|uniq gallery.
- The *JSZip.folder(name)* function, which creates a folder with the given *name* inside of the instance of JSZip which called the function, and returns another JSZip instance which points to the root of the created folder. We use this function to create folders inside of the JSZip object with the names of the albums contained inside of the photo|uniq gallery, as well as the album folders contained inside these albums.
- The *JSZip.file(path, content)* function, which creates a file with the given *content* (e.g. a text or image file) in the given relative *path* (e.g. "filename.txt" or "directory/filename.txt"). This function is used to add the gallery photo files into the folders that represent their parent album and album folders in the blueprint of the zip archive which will later be generated.
- The *JSZip.generateAsync(options)*, which asynchronously generates a zip archive using the JSZip instance which called it, while taking into account the provided *options*. These options allow the developer to, for instance, define what type of archive will be generated (e.g. a blob or a Base64 string). This function is called after all the gallery elements have been stored inside a JSZip instance, in order to generate a zip archive of blob type that contains these elements. After the zip archive has been generated, the download of this archive can then be triggered, so that the user may save it to their file system.

E

First Dataset of Images Used for Functional Validation of Image Grouping Function

Figure E.1: Automatic Grouping Functional Validation Dataset 1 - Image 1.

Figure E.2: Automatic Grouping Functional Validation Dataset 1 - Image 2.

Figure E.3: Automatic Grouping Functional Validation Dataset 1 - Image 3.

Figure E.4: Automatic Grouping Functional Validation Dataset 1 - Image 4.

Figure E.5: Automatic Grouping Functional Validation Dataset 1 - Image 5.

Figure E.6: Automatic Grouping Functional Validation Dataset 1 - Image 6.

Figure E.7: Automatic Grouping Functional Validation Dataset 1 - Image 7.

Figure E.8: Automatic Grouping Functional Validation Dataset 1 - Image 8.

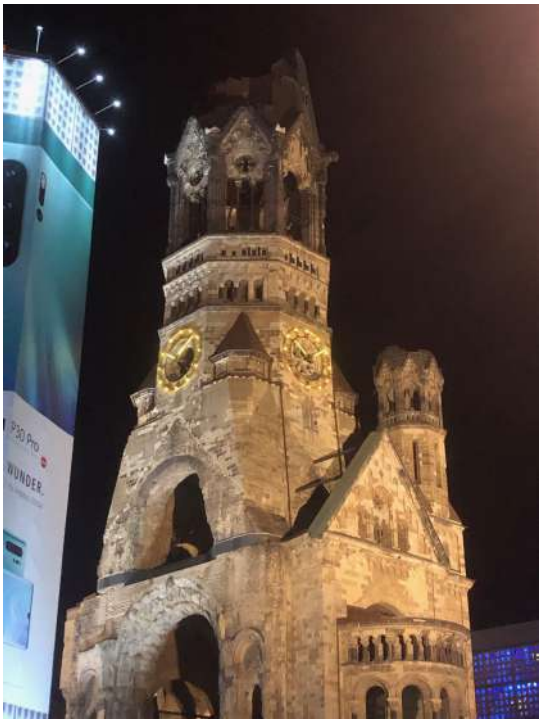


Figure E.9: Automatic Grouping Functional Validation Dataset 1 - Image 9.

Figure E.10: Automatic Grouping Functional Validation Dataset 1 - Image 10.

Figure E.11: Automatic Grouping Functional Validation Dataset 1 - Image 11.

Figure E.12: Automatic Grouping Functional Validation Dataset 1 - Image 12.

Figure E.13: Automatic Grouping Functional Validation Dataset 1 - Image 13.

Figure E.14: Automatic Grouping Functional Validation Dataset 1 - Image 14.

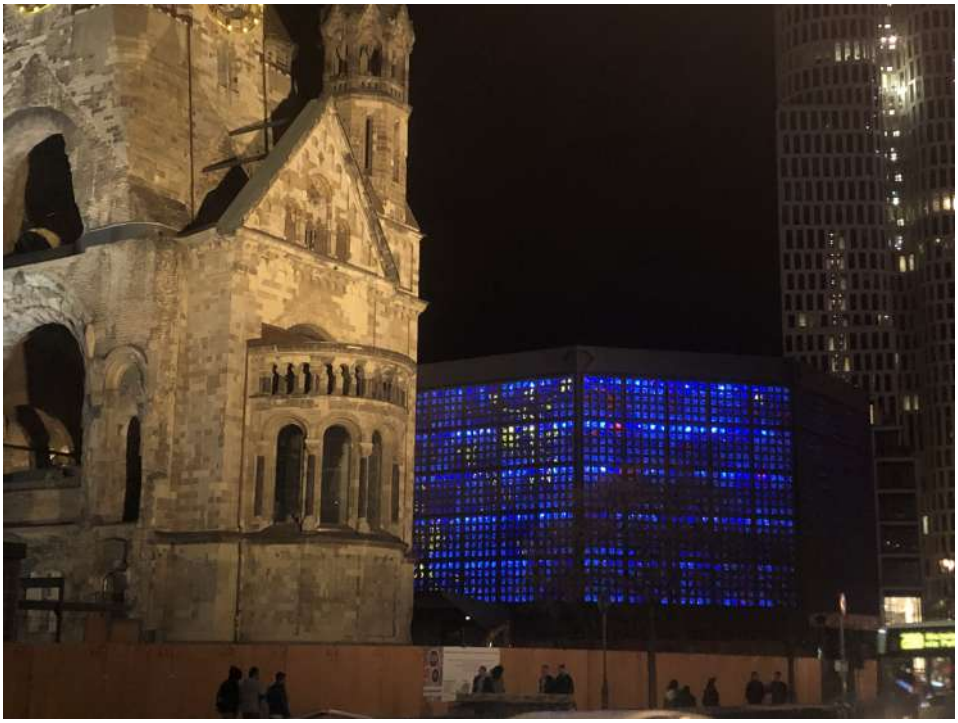


Figure E.15: Automatic Grouping Functional Validation Dataset 1 - Image 15.

F

Second Dataset of Images Used for Functional Validation of Image Grouping Function

Figure F.1: Automatic Grouping Functional Validation Dataset 2 - Image 1.

Figure F.2: Automatic Grouping Functional Validation Dataset 2 - Image 2.

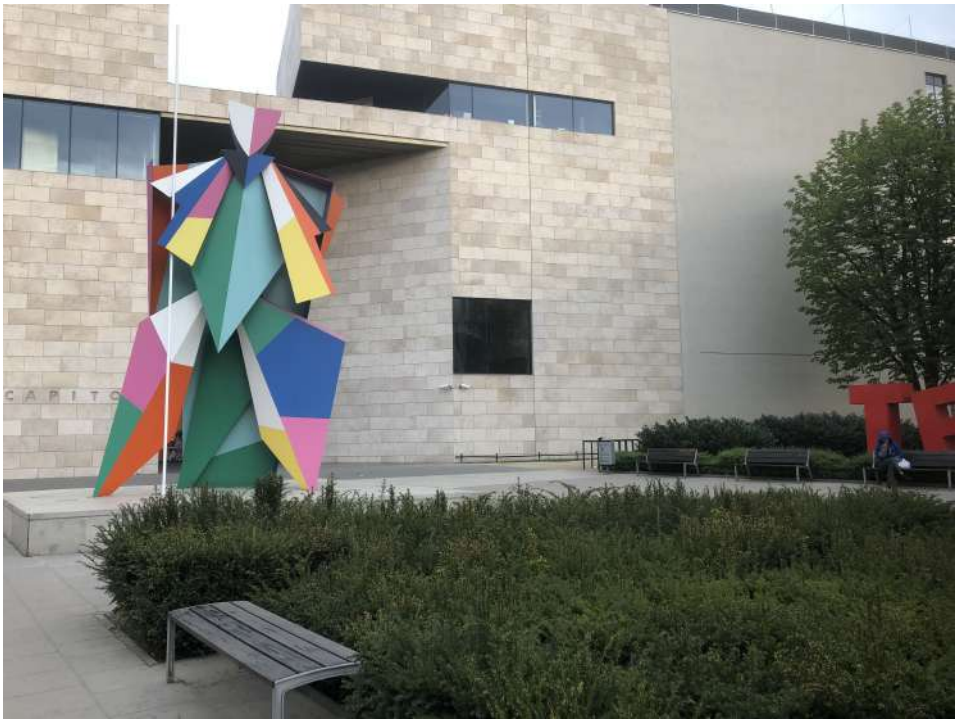


Figure F.3: Automatic Grouping Functional Validation Dataset 2 - Image 3.

Figure F.4: Automatic Grouping Functional Validation Dataset 2 - Image 4.

Figure F.5: Automatic Grouping Functional Validation Dataset 2 - Image 5.

Figure F.6: Automatic Grouping Functional Validation Dataset 2 - Image 6.

Figure F.7: Automatic Grouping Functional Validation Dataset 2 - Image 7.

Figure F.8: Automatic Grouping Functional Validation Dataset 2 - Image 8.

Figure F.9: Automatic Grouping Functional Validation Dataset 2 - Image 9.

Figure F.10: Automatic Grouping Functional Validation Dataset 2 - Image 10.

Figure F.11: Automatic Grouping Functional Validation Dataset 2 - Image 11.

Figure F.12: Automatic Grouping Functional Validation Dataset 2 - Image 12.

Figure F.13: Automatic Grouping Functional Validation Dataset 2 - Image 13.

Figure F.14: Automatic Grouping Functional Validation Dataset 2 - Image 14.

Figure F.15: Automatic Grouping Functional Validation Dataset 2 - Image 15.

Figure F.16: Automatic Grouping Functional Validation Dataset 2 - Image 16.

Figure F.17: Automatic Grouping Functional Validation Dataset 2 - Image 17.

Figure F.18: Automatic Grouping Functional Validation Dataset 2 - Image 18.

G

Images Used for the First Test of Functional Validation of Image Sorting Function

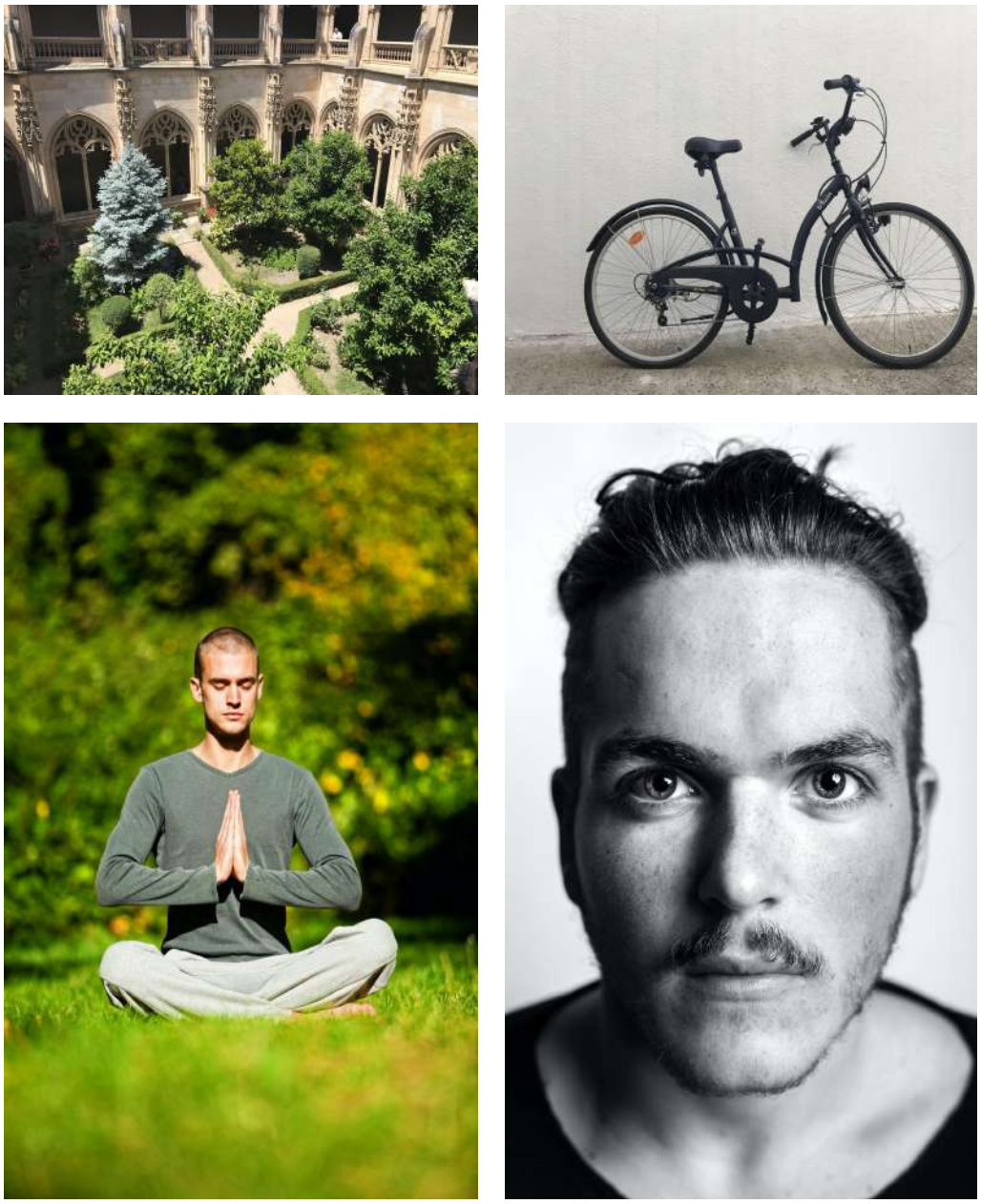


Figure G.1: Unaltered versions of the images used for the first test of the functional validation of the photo|uniq application's image sorting function [78, 96].

H

Images Used for the Second Test of Functional Validation of Image Sorting Function

Figure H.1: Images contained in the first group that the testers were asked to sort based on objective quality.

Figure H.2: Images contained in the second group that the testers were asked to sort based on objective quality.

Figure H.3: Images contained in the third group that the testers were asked to sort based on objective quality.

Figure H.4: Images contained in the fourth group that the testers were asked to sort based on objective quality.

I

Big O Time Complexity Estimation Support Tables

As discussed in Section 4.3.1, the performance of four functions was tested for the purposes of performance validation of the photo|uniq application. One of the steps of this performance validation was the estimation of the Big O time complexity of these functions by looking at their source code.

This appendix contains tables which present the reasoning behind these estimations, by showing the several steps that make up these functions and the estimated time complexities of these steps.

Table I.1: Time complexity of the image upload function.

	Step of Function	Upload to Workspace	Upload to Workspace Folder
1.	Create Empty array	O(1)	O(1)
2.	Get id of the workspace with the given name	O(w)	O(w)
3.	Verify if the workspace id is valid	O(1)	O(1)
4.	Verify if the images are being uploaded to workspace folder	O(1)	O(1)
5.	Get Id of the workspace folder with the given name	Does not occur	O(f)
6.	Verify if the workspace folder id is valid	O(1)	O(1)
7.	For every image being uploaded (repeats every following step up to and including step 9)	O(n)	O(n)
8.	Create image thumbnail	O(1)	O(1)
9.	Create workspace photo object	O(1)	O(1)
10.	Add photos to session storage	O(1)	O(1)
	Total Function Time Complexity	O(w+n)	O(w+f+n)

[1] The function receives three parameters: the name of the parent workspace of the images that are being uploaded, the name of the parent workspace folder of the images that are being uploaded (if applicable), and the images themselves.
[2] The **w** variable refers to the number of workspaces existing in the application.
[3] The **f** variable refers to the number of folders contained in the parent workspace of the folder to which the images are being uploaded to.
[4] The **n** variable refers to the number of images being uploaded.

Table I.2: Time complexity of the image loading function.

	Step of Function	**Loading images into non-deleted similarity group**	**Loading images into other lowest level element**	**Loading images into mid or highest level element**
1.	Get parent similarity group	O(g)	Does not occur	Does not occur
2.	Load from IndexedDB array of images contained in element with the given id	O(n log (t))	O(n log (t))	O(n log (t))
3.	Sort array obtained from IndexedDB	O(n)	Does not occur	Does not occur
4.	Filter array obtained from IndexedDB	Does not occur	Does not occur	O(n)
5.	Set value in photo map	O(1)	O(1)	O(1)
6.	Set value in component state	O(1)	O(1)	O(1)
	Total Function Time Complexity	O(g + n log(t))	O(n log(t))	O(n log(t))

[1] The function receives one parameter: the id of the element whose images are being loaded.
[2] The **g** variable refers to the number of groups contained in the group's parent workspace folder.
[3] The **n** variable refers to the number of images which are retrieved from the IndexedDB object store.
[4] The **t** variable refers to the total number of images stored in the accessed object store (which will always be higher than or equal to **n**).

Table I.3: Time complexity of the image grouping function.

	Step of Function	Time Complexity
1.	Get image features of every image which will be compared	O(f log(i))
2.	Initialize array of arrays with an empty array for each existing group	$O(g_p)$
3.	For every ungrouped image, (repeats every following step up to and including step 7)	O(n)
4.	Read value from map	O(1)
5.	Deserialize descriptors of ImageInfo object	O(1)
6.	For every group representative, verify similarity when compared to ungrouped image, and store number of good matches and index of group if it has the highest number of good matches so far	O(g)
7.	Push index of image into array of arrays created in step 2	O(1)
8.	For the image features of every analysed image, delete the OpenCV object associated with their descriptors	O(f)
9.	Create empty array of arrays	O(1)
10.	For each ungrouped image, place the image in the correct indexes of the array of arrays created in previous step	$O(g \times i_{max})$
11.	Return array of arrays	O(1)
	Total Function Time Complexity	O((f log(i)) + (n × g))

[1] This time complexity was estimated while ignoring variables such as number of detected image features and image resolution.

[2] The function receives two parameters: an array containing the ungrouped images, and an array containing the existing group representatives (the images which represent the already existing groups).

[3] The **n** variable refers to the number of ungrouped images which are being grouped through the execution of the function.

[4] The **f** variable refers to the number of images whose features are being correlated (or, in other words, the sum of **n** with the number of group representatives which will be analysed during the execution of the function).

[5] The **i** variable refers to the total number of *ImageKeypointDetectorInfoObjects* (see Section 3.2.4) which are stored in the application's persistent storage.

[6] The **g** variable refers to the number of groups which are contained in the parent workspace folder after the end of the function's execution, with $\mathbf{g_p}$ referring to the number of groups at the start of the function's execution (**g** is always higher than or equal to $\mathbf{g_p}$).

[7] The $\mathbf{i_{max}}$ refers to the maximum number of images which will be moved into a group due to the execution of this function ($\mathbf{i_{max}}$ is always lower than or equal to **n**).

Table I.4: Time complexity of image sorting function. The concept of *FocusSortSupportObject* is presented in Section 3.2.4.

	Step of Function	**Time Complexity**
1.	Store all image ids in an array	O(n)
2.	Create map containing as many *FocusSortSupportObject* for each image id stored in the previously created array as can be found in the application's session storage	O(n)
3.	Retrieve from IndexedDB object store and set in map created in step 2 as many *FocusSortSupportObjects* as can be found in the object store for each image which is not already represented in the map	O(s log(f))
4.	If there are still images which do not have an associated *FocusSortSupportObject*, every following step until step 9 happens	O(1)
5.	For every provided image (repeats every following step up to and including step 9)	O(n)
6.	If image id does not exist in map, every following step until step 9 happens	O(1)
7.	Convert image into grayscale, calculate its laplacian, and calculate variance of laplacian	O(1)
8.	Create *FocusSortSupportObject* and set it in map created in step 2	O(1)
9.	Delete OpenCV objects used for the calculation of the image's *FocusSortSupportObject*	O(1)
10.	Create array from values found in map created in step 2	O(n)
11.	Sort array created in previous step by image focus level	O(n log(n))
12.	Iterate through sorted array and store image ids in string array	O(n)
13.	Return sorted array of image ids	O(1)
	Total Function Time Complexity	O(s log(f) + n log(n))

[1] This time complexity was estimated while ignoring image resolution as a relevant variable.
[2] The function receives one parameter: an array containing the images to sort.
[3] The **s** variable refers to the number of images whose *FocusSortSupportObject* cannot be found in the application's session storage.
[4] The **f** variable refers to the total number of *FocusSortSupportObjects* stored in the application's IndexedDB database.
[5] The **n** variable refers to the number of images which are being sorted.