Fábio Miguel Martins Mano

Bachelor of Science

A Computing and Storage Server Infrastructure for a Mobile Application

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser:	João Lourenço, Assistant Professor, NOVA University of Lisbon
Co-adviser:	Fernando Birra, Assistant Professor, NOVA University of Lisbon

Examination Committee

Chair:	Prof. Henrique Domingos, FCT-NOVA
Rapporteur:	Prof. Rui Nóbrega, INESC
Member:	Prof. João Lourenço, FCT-NOVA

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2019

A Computing and Storage Server Infrastructure for a Mobile Application

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Para o meu avô.

Acknowledgements

In first place, I'd like to thank FCT and, in particular, the Department of Informatics, for providing me the tools and knowledge to reach this step in my academic journey.

I'd also like to express my gratitude for my advisers, Prof. João Lourenço and Prof. Fernando Birra, for the endless support and motivation during the entire dissertation process.

A special thank you to the friends that I made along the way, that have helped me through many long days and endless nights. Although it already has been a very long road, I know these were only the first few years in a friendship that will last a lifetime. Thank you for being the best friends that I could ever ask for.

Finally, the most important acknowledgement of all: to my parents, for supporting me through every step of the way and being there for every moment, good or bad. Without you, none of this would be possible. Every achievement and success that I've had is because of your support, your guidance, your patience and your strength. Thank you for believing in me. Thank you for everything.

Abstract

The expansion of digital photography has turned a once expensive task into an easily accessible activity. Particularly, the dissemination of smartphones gave us the ability of taking multiple pictures at no cost, which was just not possible in the past with traditional film cameras.

With the current tools, we can use our smartphones to capture moments that we wish to revisit or share with our friends. However, due to having no associated cost, we can easily take multiple photos of the same motive, ending up with several similar pictures. This makes it difficult to share them with our friends and peers, as it is bothersome to navigate through many identical photos. While a simple solution would be to simply delete the repeated photos, this is often a difficult process: how can we be sure which of the photos is the best one? For this reason, we end up keeping more than one photo, which perpetuates the issue.

This work is part of a larger project that aims to solve this problem. The project is composed of a mobile app and a server infrastructure. Due to processing capabilities and energy restrictions, the mobile device off-loads the image pre-processing to the remote server infrastructure, in order to extract measurable technical features.

With this work we plan to design and provide a server infrastructure that will be able to assist the mobile app (and the photographer) in the process of removing duplicate photos, by providing services that offer features like similar image detection, image scoring and easing the process of selecting the best one that shall be kept while the others may be safely removed. The server infrastructure shall handle requests efficiently, in order to allow users with large photo galleries to remove their duplicate photos.

Keywords: Digital Photography, Server Infrastructure, Web Services, Web Frameworks, Image Processing

Resumo

A expansão da fotografia digital transformou uma tarefa com custos elevados numa actividade bastante acessível. A disseminação dos *smartphones* deu-nos a possibilidade de tirar várias fotografias sem nenhum custo, o que não era possível no passado com as máquinas de rolo tradicionais.

Com as ferramentas existentes actualmente, podemos usar os nossos *smartphones* para capturar momentos que queremos relembrar ou partilhar com os nossos amigos. Contudo, uma vez que não existe custo associado, podemos tirar várias fotografias do mesmo motivo, ficando com várias fotografias muito semelhantes. Isto torna dificil o acto de partilhar estas fotografias, uma vez que é desagradável navegar por várias imagens idênticas. Uma solução simples seria simplesmente apagar as fotografias repetidas, mas este não é um processo fácil: como podemos ter a certeza qual delas é a melhor? Por este motivo, acabamos por manter mais do que uma fotografia semelhante, o que perpetua o problema.

Esta dissertação é parte de um projecto que pretende resolver este problema. O projecto é composto por uma aplicação móvel e uma infraestrutura de servidores. Devido a capacidade de processamento e restrições energéticas, a aplicação transfere o trabalho de processamento para a infraestrutura de servidores, por forma a extrair dados relevantes para comparação.

Com esta dissertação, pretendemos desenhar e fornecer uma infraestrutura de servidores capaz de assistir a aplicação móvel e o fotógrafo no processo de remoção de fotografias duplicadas, facilitando o processo de selecção da melhor imagem a manter e permitindo que as restantes sejam facilmente removidas. A infraestrutura será capaz de tratar pedidos de forma eficiente, permitindo que utilizadores com grandes galerias de fotografias possam utilizar o sistema para remover as suas imagens duplicadas.

Palavras-chave: Fotografia Digital, Infraestrutura de Servidores, Serviços Web, Frameworks Web, Processamento de Imagem

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Context

Since the dissemination of smartphones, digital photography has become a highly accessible activity. In the past, film size limited the number of photos that could be taken, which made us choose our motives carefully. Reaching the film limit either meant that we would have to stop taking pictures, or replace the film, incurring in an additional cost. Nowadays, smartphones allow us to take a virtually infinite amount of pictures, where the only limiting factor is storage space. This limit is easily extended with a plethora of services such as *Google Drive*, *Flickr* or even *Instagram*, meaning that we can upload our pictures and safely delete them from our smartphones.

This dissemination lead photography, which was a previously costly activity, to become part of our everyday lives. On high traffic days, 25 million photos are uploaded to Flickr [2], which shows how massively widespread this activity actually is. This number, however, only includes the pictures that were shared, meaning that the number of pictures actually taken was likely much higher, as not only do we keep some photos private, but we also tend to take several pictures of the same motive in order to choose the best one.

1.2 Problem Statement

The accessibility of photography leads to a large amount of pictures taken, including repeated photos of the same motive.

This is illustrated in Figure 1.1, in which we can clearly distinguish five different motives pictured repeatedly: the dog, the piano, the trees, the mountain river and the butterfly.

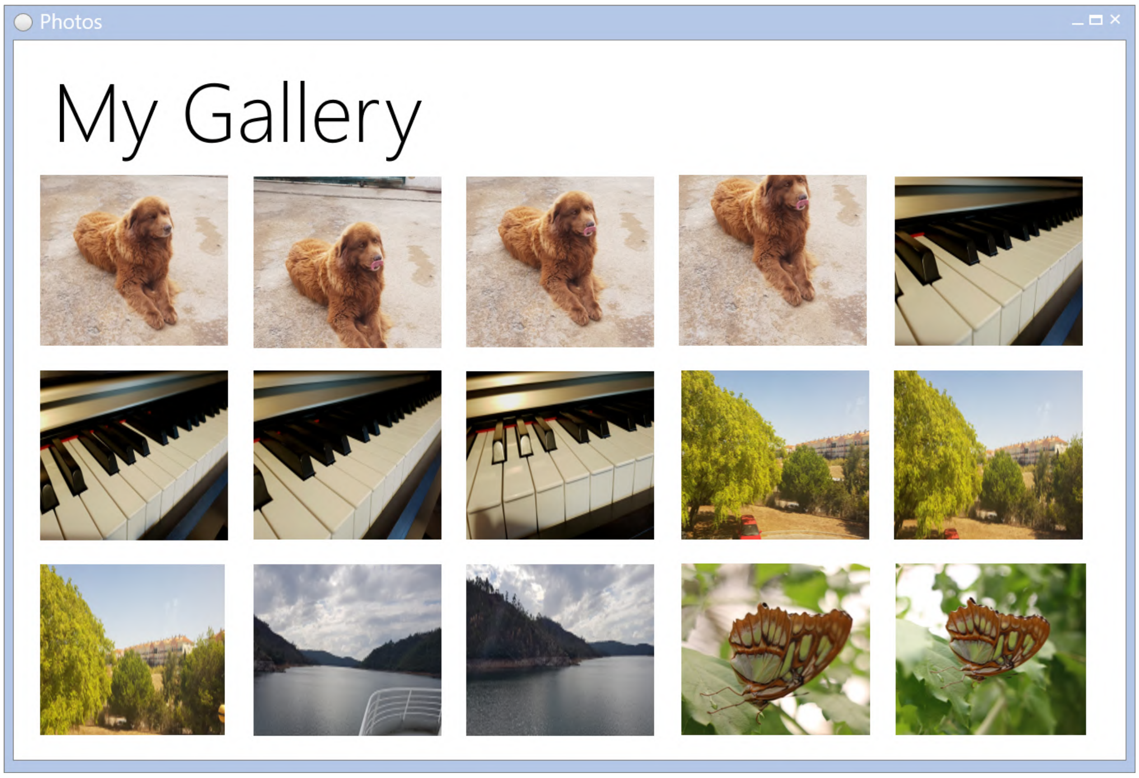


Figure 1.1: Photo album with duplicate or similar photos.

One of photography's purposes is to capture moments, so that we can revisit them or share those moments with others. However, by taking several duplicate photos, this experience can easily become more bothersome than enjoyable: when showing, for example, pictures from a vacation in Paris, nobody would appreciate iterating through several identical pictures of the Eiffel Tower. Additionally, resolving this issue can currently be a cumbersome process. Since the pictures are so similar, the difficulty of picking a single one is very high, meaning that we end up keeping more than a single photo due to being uncertain about which one is the best.

This dissertation is part of a project that aims to assist the photographer in the process of removing duplicate photos. The project consists of two large components: the *photo|uniq* app and a server infrastructure. The *photo|uniq* app is a mobile application that provides an interface for the removal of duplicate photos, album creation and side-by-side comparison of duplicates. The app offloads image processing work to a server, which uses existing libraries to establish relations between photos. This dissertation is focused exclusively on the server component of the project, which will be detailed throughout the following chapters.

1.3 Proposed Solution

The solution proposed by this dissertation is the design, specification and implementation of the server infrastructure. This infrastructure will be a distributed system in which different servers perform different roles in order to achieve the purpose of the system.

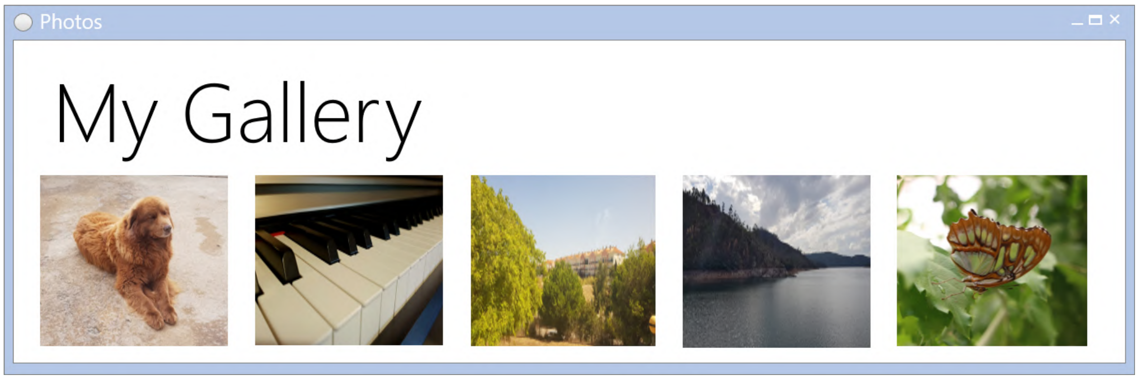


Figure 1.2: Photo album with the best photo of each motive.

The system should be capable of assisting the user to transform his photo album from the one seen in Figure 1.1 to the one depicted in Figure 1.2, in which every duplicate photo was removed and only the best picture of each motive was kept. As such, the proposed solution is a system capable of reaching this end by performing the necessary steps to do so:

1. Identify groups of similar photos within an album;
2. Propose a ranking from best to worst photo within each group;
3. Provide image correlation methods to improve the manual photo comparison experience.

While steps 1 and 2 are automatically performed by the server, the user should always maintain the capability of overriding the established groups or rankings, as the objectiveness of the algorithms used in the solution may not match the user's subjective opinion. This is further reinforced by step 3, which provides the user with the tools to manually compare photos using a client application via homography matrices, further detailed in Section 3.5.

Figure 1.3 depicts the general expected workflow of the proposed solution. The user interacts with a client application which, in turn, interacts with the server infrastructure. When processing results are available, such as, for example, a specific photo ranking for a similar group of pictures, those results are made available to the user via the client application.

To enable the interaction with external client applications, the server infrastructure will provide a set of webservices to perform the required tasks.

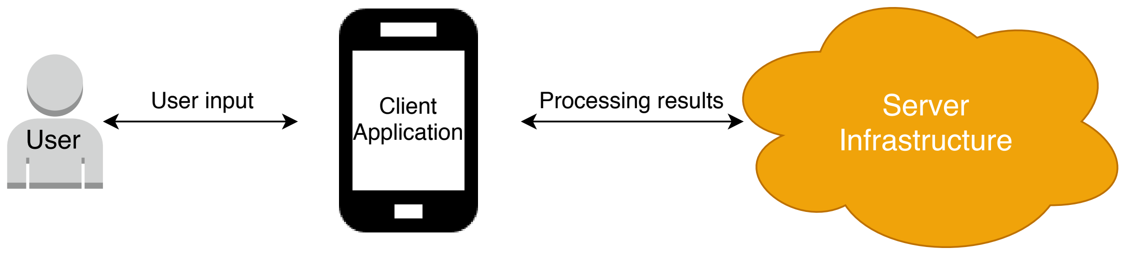


Figure 1.3: General workflow of the photo|uniq system.

With this proposed solution, users can rely on automatic processes to effortlessly clear their photo galleries of duplicate photos, reducing the strain of manually comparing groups of several identical pictures. Should the user choose not to rely on the automatic process, the proposed solution also provides tools to aid in the manual comparison, thus extending its value proposition to both use cases.

1.4 Document Structure

This document's structure is detailed in the following list:

- Chapter 1: Introduction — refers to this chapter, where we present the context of this dissertation, the problem description and the general approach for our proposed solution;
- Chapter 2: Related Technologies — this chapter presents a description of technologies and concepts relevant in this dissertantion's context, as well as a comparison and critical analysis of the presented technologies;
- Chapter 3: Architecture and Implementation — this chapter describes in detail the approach taken to develop the proposed solution, presenting the system architecture and the steps taken for its implementation;
- Chapter 4: Validation — this chapter presents a series of tests and their respective results, in which the system's performance and correctness is evaluated and described;
- Chapter 5: Conclusion and Future Work — this chapter draws a conclusion from the research and system developed, and highlights a path for the future work regarding this dissertation's context.

Chapter 2

Related Concepts and Technologies

In this chapter, we identify and describe topics that are in some way related to this dissertation. We approach some concepts that serve as a base for this dissertation, as well as the key technologies used in its development.

2.1 Web Application and Server Development

Nowadays, application development is aided by a multitude of tools, processes and frameworks that enhance, in some way, the development process. Whether by enabling better solutions or simply accelerating the process, these tools provide foundations on which applications can be built.

Web applications and servers, specifically, can benefit from making use of existing web development frameworks, which are detailed in the sections below.

2.1.1 Web Development Frameworks

A web framework is a collection of tools and language modules with the purpose of accelerating the development of web applications. In order to enable this acceleration, web frameworks generally provide a high-level environment, relieving the need of implementing lower level functionalities.

Web frameworks typically implement features such as URL routing and templating, which are two important functionalities in a web application.

URL routing allows applications to serve content, such as web pages or documents, that are not stored in a file inside the application's structure. This allows web applications to use descriptive and user-friendly URLs, e.g., `https://example.com/users/edit` instead of `https://example.com/EditUser.html`. Templating allows an application's output, whether it is a web page or a document, to be associated with a data model. This enables content to be generated dynamically, relieving the need of manually implementing repeated items. Listing 2.1 illustrates how templating can be used: the web page holds a data model with a list of users, which is iterated in the page itself (line 2) and each element is placed inside an unordered list. Additionally, each user in the list also contains data, which is used for the URL and displayed text (line 3). This feature is simple, but it is also very useful to develop custom content individually tailored to a collection of items, such as the user list we've seen previously, without the need to implement every possible case manually.

```
1  <ul>
2  {% for user in users %}
3    <li><a href="{{user.url}}">{{user.username}}</a></li>
4  {% endfor %}
5  </ul>
```

Listing 2.1: Templating example [1].

There are numerous frameworks available for many different languages, each with its own strengths and purposes.

In this dissertation, we will focus on Python-based web frameworks.

2.1.1.1 Full-stack Frameworks

A full-stack framework is a type of web framework that provides tools for the development of all levels of a web application. This means that a full-stack framework can be used to handle both front and back-end development, ranging from serving web pages to accessing data in a database.

Typically, full-stack frameworks provide templating and URL routing mechanisms, as detailed in Section 2.1.1. These mechanisms are associated with the front-end component, as they are focused on displaying and accessing data.

Regarding the back-end, full-stack frameworks generally also provide a set of tools that help in this component's development. One of those tools is an Object-Relational Mapper (ORM), which is a layer that allows access to a database's contents in an object-oriented fashion. This allows applications to seamlessly integrate data retrieval and updating in their workflow, as data can be accessed as if it were a regular data structure within the application itself, such as a list or a map, without the need to use querying languages such as SQL to access data.

Despite being able to fully implement a web application, full-stack frameworks can be used as only either a back-end or front end framework. Applications can use these fully-featured frameworks to implement back-end components, and use front-end frameworks to implement the respective component of the application.

An example of a full-stack Python web framework is *Django* [3]. Out of the box, *Django* provides URL routing, templating, object-relational mapping, form generation and an administration panel, to cite only a few of the default features. *Django* was built with the goal of enabling fast application development. This makes *Django* a solid choice for many web applications, given its extensive functionality without additional configuration. *Django* is used in many largely known websites, such as Instagram [4], which further reinforces the quality and scalability of this framework.

Another example is *TurboGears*, a framework that aims at scaling with the complexity of the project [5]. This means that the framework is robust enough to handle complex projects, but can also be used to solve simpler problems with ease. Like *Django*, *TurboGears* includes an object-relational mapper, with the addition of multi-database support. Additionally, *TurboGears* also offers a transaction system, which handles database concurrency without additional configuration.

Both *Django* and *TurboGears* are based on the Model-View-Controller paradigm, which is an architectural pattern used commonly when developing user interfaces [6]. In this pattern, there are three main components: the model, the view and the controller. The model is an abstract representation of the data, which can be anything from a simple integer or string to a complex object. The view is a representation of any type of data, and can be a web page, a table or a chart. Finally, the controller is the connector between the view and the model, converting the model data into its visual representation to be displayed in the view.

2.1.1.2 Other Frameworks

Not all frameworks are aimed at full-stack development. In fact, some frameworks focus on either the client-side or server-side, providing specialized tools for each of those components.

An example is *Flask*, a microframework that aims to provide a constraint-free development environment [7]. Unlike *Django* or *TurboGears*, *Flask* does not provide many features out of the box. Instead, it focuses on offering a clean slate, without forcing the user to adopt any specific technology. This means that *Flask* is a very simple framework, but provides modules that can be used to scale into a fully-fledged web application.

Despite its simplicity, a default *Flask* installation provides routing and a templating language based on *Django*'s templates, meaning that it is capable of building simple applications out of the box and can be extended to develop complex and reliable solutions.

2.1.1.3 Comparison

As seen in Sections 2.1.1.1 and 2.1.1.2, there are a number of python web-frameworks that offer extensive functionalities. In this section we establish a comparison between the three mentioned frameworks: *Django*, *TurboGears* and *Flask*. The defined criteria are the following:

- Routing — routing is the support for retrieving the right server method given a specific URL;

Table 2.1: Python oriented web-framework comparison.

	Django	TurboGears	Flask
Routing	Yes	Yes	Yes
Templating	Yes	Yes	Yes
Database Support	Yes	Yes	No
Full-stack	Yes	Yes	No
GitHub Stars	39400	271	41800

- Templating — templating refers to a language that allows the creation of web views or web pages while maintaining access to the model data;
- Database Support — refers to whether the framework natively supports database access and management or not;
- GitHub Stars — the number of stars of each framework on GitHub. This number serves as an indicator of the size of the developer community using each framework.

Table 2.1 shows a comparison between the three depicted frameworks. In Section 2.1.1.4 we expand this comparison.

2.1.1.4 Discussion

The three frameworks implement routing by default. This means that the effort of serving the right web page in each URL is done by the framework, freeing the developer of this task.

Templating is also universal to the three frameworks, with each one implementing its own templating engine. *Django* and *TurboGears* implement custom templating engines, while *Flask* uses *Jinja 2*, a templating engine based on *Django*.

Regarding project structure, we evaluated the folders and structure required in order to build a project in each framework. Both *Django* and *TurboGears* require a complex folder structure, where each file and folder plays a specific role in the project. *Flask* does not require any structure, and the whole project can exist in a single file if necessary.

In terms of database support, *Django* and *TurboGears* natively support relational databases through object-relational mappers. *Flask* does not have this built-in support, but provides extensions that can be used for the same effect. None of the three frameworks have built-in support for non-relational databases, but both *Django* and *Flask* provide extensions that do so while *TurboGears* does not.

Regarding support community, both *Django* and *Flask* have a large amount of users, with 39,400 and 41,800 stars[1] on GitHub, respectively. This means that the support community is likely to be able to help in the solution of problems within the respective frameworks. In this regard, *TurboGears* is significantly behind, with only 271 stars.

[1]As of the 10th of February, 2019.

It is also important to note that there is a significant difference regarding the complexity of developing an application in each framework. *Flask* is, by far, the most simple of the three presented examples, as it is free of constraints regarding project structure and, in an edge case, an entire application can be built in a single file. However, this simplicity may cause some setbacks when trying to develop complex applications. With *Django* and *TurboGears* this is not the case, as these frameworks require projects to follow a specific structure. This additional complexity, however, is alleviated through an extensive documentation and community, in which Django beats *TurboGears* by far.

2.1.2 Security

When providing services over the internet, we are exposing resources that may be potentially misused if there are no protections in place. To provide a layer of security, it is necessary to understand the authenticity of incoming requests and if the requesting client should be able to access the requested resource. As such, we chose to analyze existing security alternatives through two different perspectives: authentication and authorization.

Authentication is how requesting clients prove their identification. As an example, when we login to our email account, we are authenticating our identity by inputting our email and password combination, thus proving that we are indeed the owners of our accounts. In web services, this can be done in the same way: before accessing a resource, the client may be required to prove his identity via a previous request containing his username and password.

Authorization is the second step in this security process: after we verify that the client is indeed who they claim to be, we must check if this client should be able to access a specific resource. Using the same email account example above, after I login into my account, I should be authorized to access my own emails, but not someone else's.

Regarding authorization, OAuth 2.0 is the industry-standard protocol [8]. OAuth 2.0 is a protocol that provides a workflow for request authorization. This protocol works by checking if incoming requests are signed with an access token, which can be obtained via several different methods. These methods are called grant types, and each method specifies what is required in order to obtain an access token [9]. A common grant type is the password grant, in which the client requests an access token by providing his password [10]. After the access token is issued, every request must be signed with this token, which will provide access to the resources accessible by the requester.

The resulting access token may also have different formats. Two common formats are Basic and Bearer. Basic tokens are a simple string containing the username and password encoded in base 64, but not encrypted. Bearer tokens improve upon the Basic tokens, by encrypting the data that they contain. [11].

2.2 Image Analysis and Correlation

Image analysis is a widely studied process that occurs across several fields, such as computer vision or machine learning. As such, there are many tools available to aid with image processing related tasks. The sections below identify concepts that are key to this dissertation and enumerates tools used to work with such concepts.

2.2.1 Image Quality Assessment

Image quality can be a difficult property to assess. We can demonstrate this by inquiring subjects on which is the best photo among a set of similar photos and, generally, different subjects will have different opinions on which photo presents the best quality. This means that perceived image quality is a subjective measure.

As such, subjective image quality assessment methods exist, in which groups of observers provide a subjective opinion on the quality of an image. The list below details some of these methods [12]:

- **Single stimulus** — a single image is displayed and the observer rates the image using a previously defined scale;
- **Double stimulus** — two similar images are displayed, which are evaluated separately by the observer using the same scale;
- **Forced choice** — two similar images are displayed, but rather than assigning a score to each image, the observer must choose a single image as the one with highest quality;
- **Similarity judgement** — two similar images are displayed, but additionally to assessing which has the highest quality, the observer is also requested to indicate how different the two images are, using a previously defined scale.

These methods, however, are not viable solutions for real time applications that require almost instant feedback on an image's quality. Additionally, these methods are also costly both on time and resources, as they require a significant amount of human participants to provide meaningful results.

Another approach is to establish an objective quality measure that does not rely on opinion. Using the combination of visual attributes of an image, it is possible to extract an objective quality measure[2] [13].

The following list enumerates some objective image quality assessment methods [14]:

1. **Full-reference (FR)** — a reference image is provided and assumed to have perfect quality, while assessed images are compared to the reference in order to extract a quality assessment;

[2] The objective quality measure may not match its subjective counterpart, as observers may not agree that the objectively highest quality image is, indeed, the best image.

2. **Reduced-reference (RR)** — features are extracted from reference and test images, from which a quality assessment is generated;

3. **No-reference (NR)** — no reference image is provided, and the quality assessment is performed exclusively based on the test image's features.

In the context of this dissertation, we are interested in the NR methods, which do not require users to select a reference image in order to function. One of such methods is the Blind/Referenceless Image Spatial Quality Evaluator, or BRISQUE [15].

BRISQUE works by identifying regular features from natural images, which are images captured by a camera that are not subject to post processing, and attempts to quantify the loss of these features using localized luminance coefficients. This leads to a quantifiable score, which serves as an objective measure of an image's quality.

2.2.2 Similar Image Detection

To identify similarity between two images, there are several methods available, each with different approaches to the problem. In this section, we identify two solutions, namely the Mean Squared Error (MSE) method and the Structural Similarity Index (SSIM) [16].

Starting with MSE, this method follows the mean squared error formula below [17]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2 \tag{2.1}$$

In detail, this formula subtracts the values of each pixel between two images (y_i and y_i'), and computes the sum of the squared differences (or errors) between them. Finally, dividing the result by the number of pixels yields the mean, hence the name mean squared errors. Generally, each pixel is identified by its color values of red, green and blue and its intensity. By converting images to black and white before calculating the MSE, the process is simplified to use only the intensity property of each pixel. The resulting value is a number that starts at 0.0 (perfect similarity) and has no upper bound, meaning that the higher it is, the less similar the two images are.

Figure 2.1 depicts two similar images that were compared using the MSE method, yielding a value of 560.17. Using two different images, such as the ones depicted in Figure 2.2, the MSE value is much higher, at 17050.08, indicating that the images are significantly different.

An alternative method is SSIM [16], which is a much more complex algorithm with potentially better results, but also heavier in terms of processing power required in comparison to MSE.

Figure 2.3 provides a simplified overview of the steps taken by the SSIM algorithm. SSIM measures the luminance and contrast of both images and establishes a comparison between both images, taking into account their structure as well. The final result is a

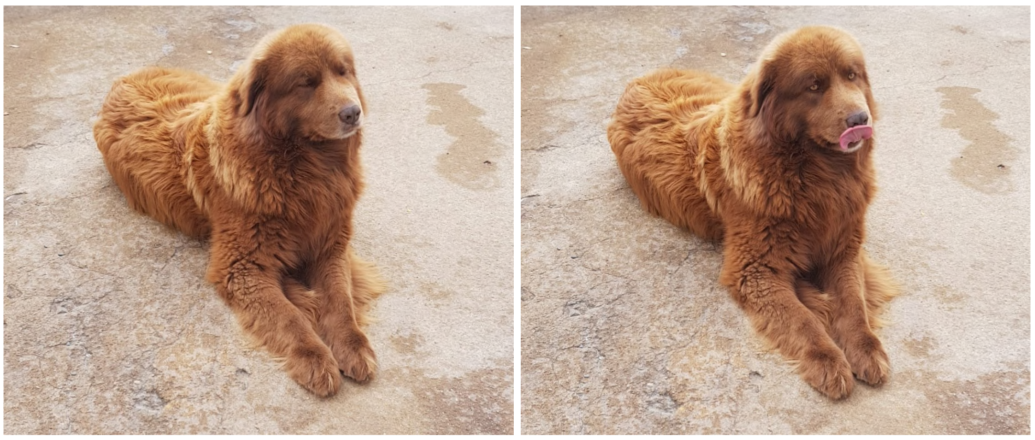


Figure 2.1: Similar photos with a mean squared error value of 560.17.

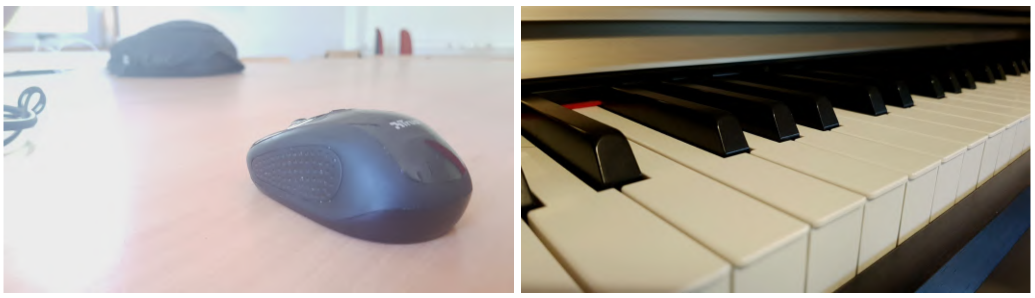


Figure 2.2: Different photos with a mean squared error value of 17050.08.

measure, where 1.0 indicates perfect similarity and lower values indicate differences in the images.

It is also interesting to note that SSIM can be used as a full-reference quality assessment method: if we compare two images where one is established as the perfect reference, the resulting similarity value indicates how different the test image is from the reference, thus providing a quality measure which can be read as how much worse the tested image is, when compared to the perfect reference.

Figure 2.4 depicts two images with a SSIM value of 0.88, which indicates that they are highly similar. Using two different images, such as depicted in Figure 2.5, the lower value of 0.35 indicates that the images are significantly different from one another.

2.2.3 Image Correlation

Image correlation refers to the process of identifying points within an image and determining where those points would be in a similar image. This process is split into two separate concepts: keypoint detection and homography matrix.

Keypoint detection consists in the identification of relevant points within an image, i.e., points whose properties make them distinguishable from their neighbouring points. There are several algorithms for keypoint detection, from which we highlight two in the list below:

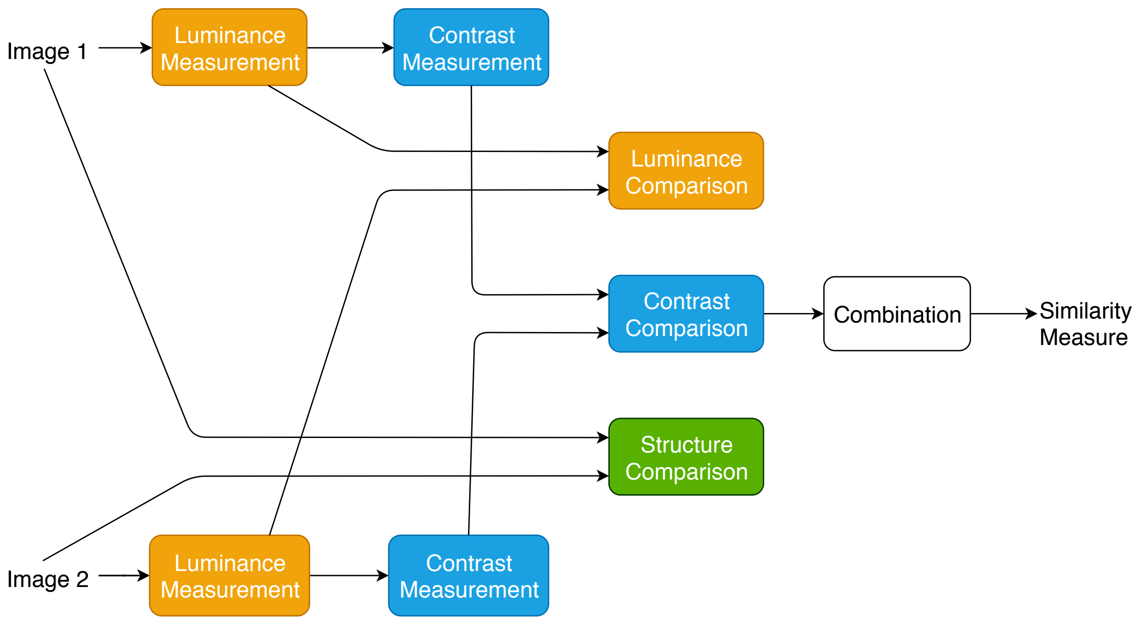


Figure 2.3: SSIM algorithm steps. Adapted from [16] (simplified).

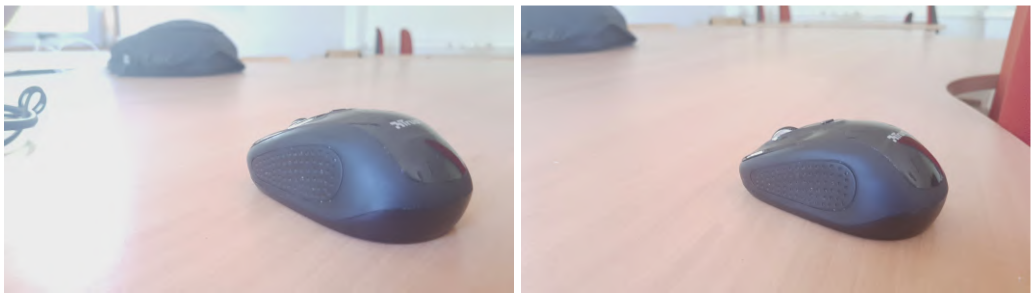


Figure 2.4: Similar photos with a structural similarity value of 0.88.

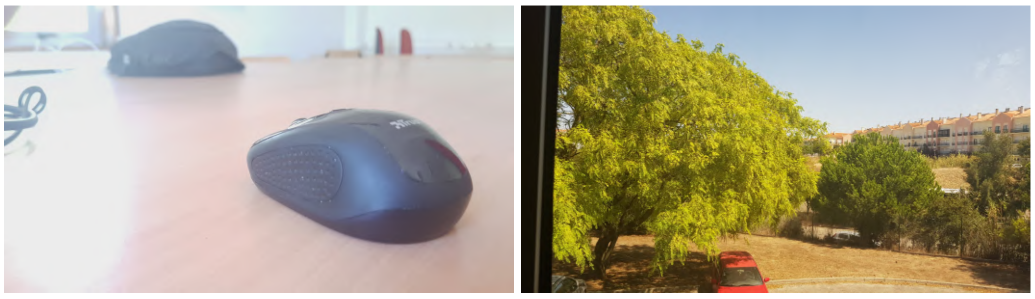


Figure 2.5: Different photos with a structural similarity value of 0.35.

- **Scale-Invariant Feature Transform (SIFT)** [18] — this algorithm starts by identifying potential keypoints by finding local extrema points across the image. Then, low contrast and edge points are removed, as they are considered points of low interest. The remaining points are then assigned an orientation, in order to account for image rotation. Finally, each point is assigned a descriptor, which is a structure containing information about the point's neighbourhood.

- **Speeded Up Robust Features (SURF)** [19] — this algorithm aims to be a faster version of SIFT by stripping the algorithm down to its essential components. This algorithm shares similar steps to SIFT, but speeds up the potential keypoint detection process by using faster approximation methods, thus increasing the performance of the keypoint detection.

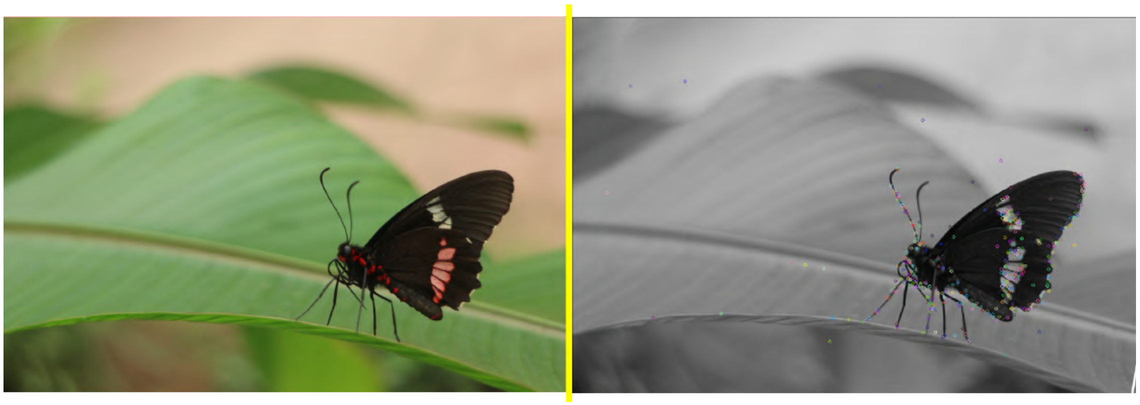


Figure 2.6: Keypoint detection results (right) based on the original image (left).

Both algorithms have similar output: a set of keypoint structures, composed of their coordinates, orientation, descriptors and other features. Figure 2.6 shows a visual representation of the keypoint detection results: the image on the left is the original image, while the image on the right contains the points (or areas) identified as keypoints. This output is used in the second step of the image correlation process, the homography matrix.

The homography matrix is the correlation between two images' keypoints. Given two similar images, it is likely that they will share at least a subset of their keypoints. However, each image may have a different set of transformations, meaning that each keypoint's coordinates will not be the exact same in both images. The homography matrix establishes this correlation, enabling the transformation of an image's keypoints into another's.

Figure 2.7 depicts a visual representation of this transformation. In the figure, the two photos share the same motive, but they are not completely identical: the image on the right is slightly tilted and panned in comparison to the image on the left. Applying the homography matrix to the previously detected keypoints, a correlation is obtained, as depicted in the figure: the green lights connect the keypoints on the left photo to its correspondent keypoint in the right photo.

Figure 2.8 shows another example, this time with a more drastic transformation: the image on the left is smaller and contained within the image on the right, while both images also have different transformations between each other. As with the previous example, the green lines draw the correlation between the keypoints on the left and the keypoints on the right. Additionally, in this example, we can identify a white border

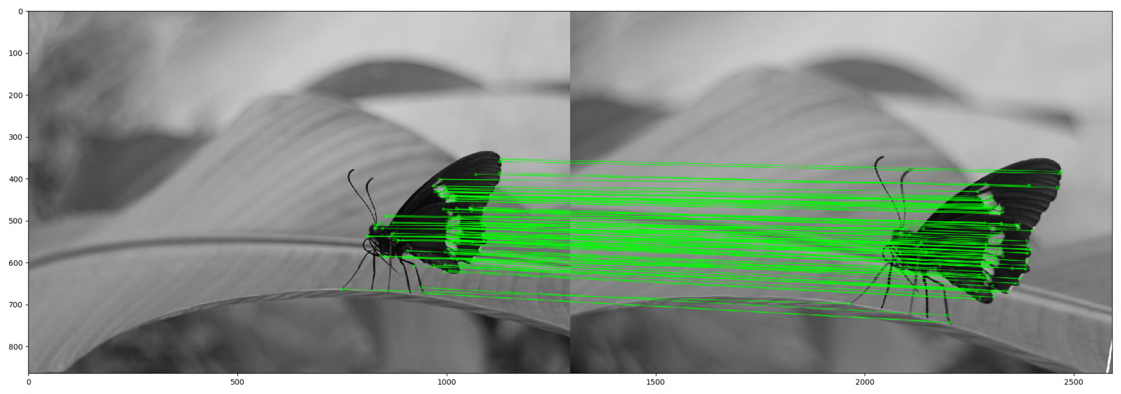


Figure 2.7: Keypoint correlation between two images.

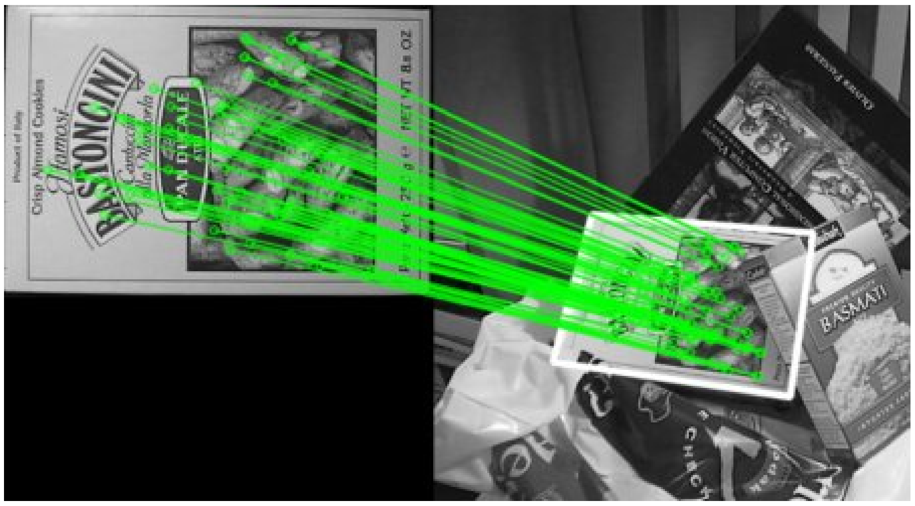


Figure 2.8: Homography matrix application. Taken from [20].

along the outline of the box on the right image: this is also a product of the homography matrix, which identifies the boundaries of the first image within the second image. This is also visible in Figure 2.7, altough to a much smaller extent, in the bottom right corner of the right image.

Chapter 3

Architecture and Implementation

In this chapter, we specify the architecture and data model for the server infrastructure, illustrate the workflow of the photo|uniq system and describe the steps taken to implement the proposed architecture.

3.1 Architecture

To implement the proposed system, we designed a distributed system with multiple servers and databases, each with a specific role.

Figure 3.1 depicts the proposed architecture, as well as the interactions between elements of the photo|uniq system. Each component is described in detail in the paragraphs below.

The first component is the client application, which is external to the work developed in this dissertation. This component can be any type of application that is able to interact with a REST API and that is capable to interpret and handle its results.

The Storage Server is the component responsible for the storage of the system's data, which includes photos, similarity groups, albums and user data. This server keeps track of the photo structure, i.e., photos within similarity groups and similarity groups within albums, and while it is capable of storing the photos, the server is built in a way that it would, in theory, be possible to simply store a link, allowing the photos to be stored elsewhere (i.e., different storage systems such as Google Drive, or image hosting services such as imgur or Flickr).

Since the system requires the client application to upload photos to the storage server, we chose to make these two components interact with each other. This is the only component that interacts with the client applications, which simplifies the connection process (the application maintains a single connection URL rather than several different URLs

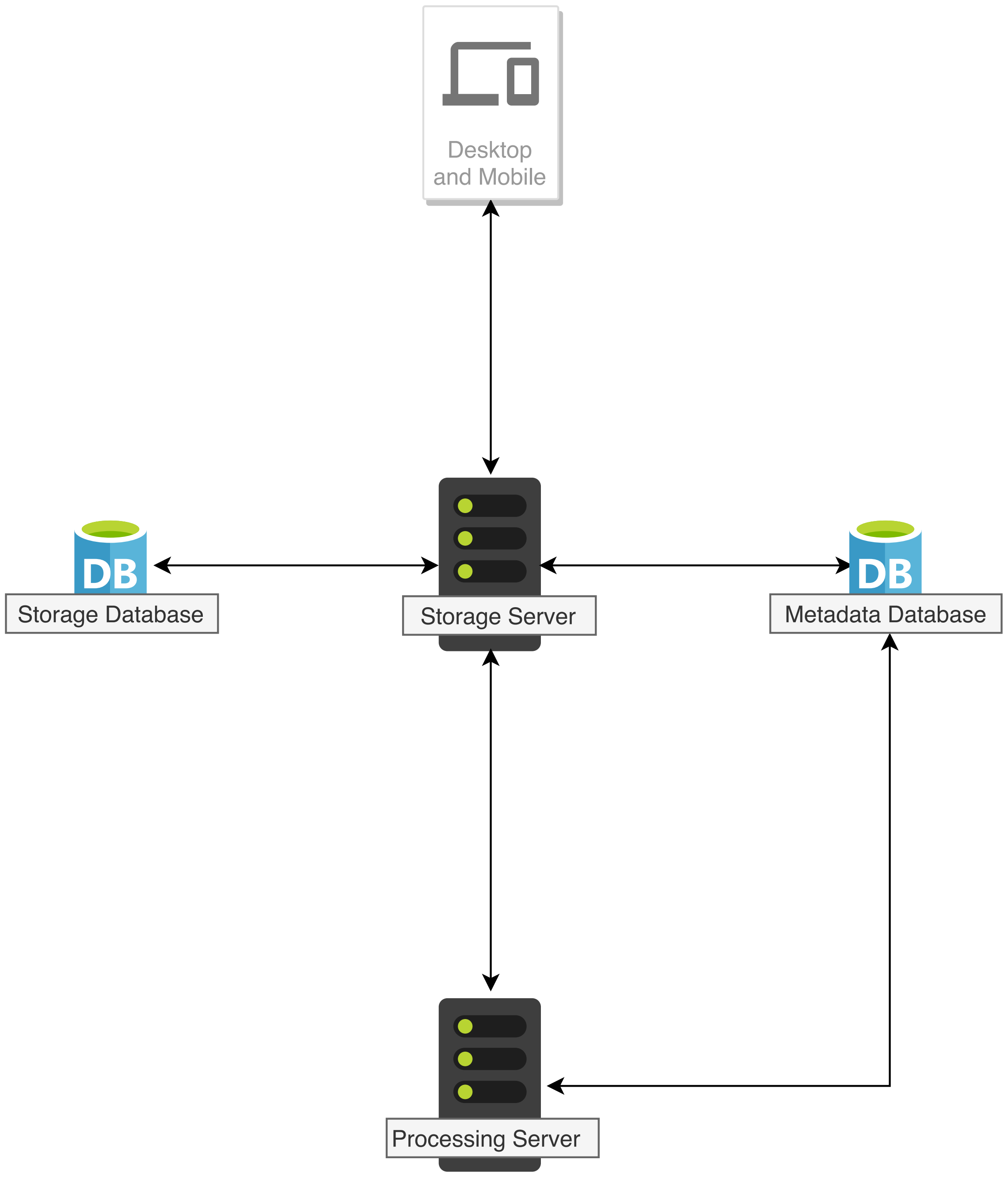


Figure 3.1: photo|uniq system architecture.

for different purposes). However, the main reason for this interaction is to accelerate the photo upload process: rather than bouncing the photo across several servers, the application sends it directly to the server responsible for the photo's storage. Being the single entry point for the application, the Storage Server is responsible for handling all requests. If the requested data is available in one of the databases, the storage server simply returns it to the requesting client. Otherwise, the request is forwarded to the server capable of performing it.

The Processing Server is the component responsible for image processing requests.

This includes calculating keypoints, homography matrices, image resizing, similarity group extraction and image scoring. This server works in a multi-threaded fashion, which means that it handles processing requests in separate threads in order to minimize the waiting time for simultaneous requests.

To support the system, we rely on two distinct databases. The first one is the Storage Database, which is directly associated to the Storage Server, containing data related to the system and its files. The second is the Metadata Database, which contains data relative to the photos themselves. The purpose of this database is to keep relevant data without relying on the photos themselves. This means, for example, that we may store a photo's keypoints in this database and later, when a homography request arrives, we can simply use the already calculated keypoints, without ever requiring to access the photo again, thus accelerating the processing time. Additionally, this database separation allows us the possibility of using external image storage services while maintaining the required elements within the system itself (such as homography matrices and keypoints).

3.2 Data Model

As mentioned in Section 3.1, we chose to store the system's data across two different databases: Storage Database and Metadata Database.

The Storage Database is responsible for holding data relative to the structural hierarchy of the system's photos: photos within groups[1] and albums, and groups within albums.

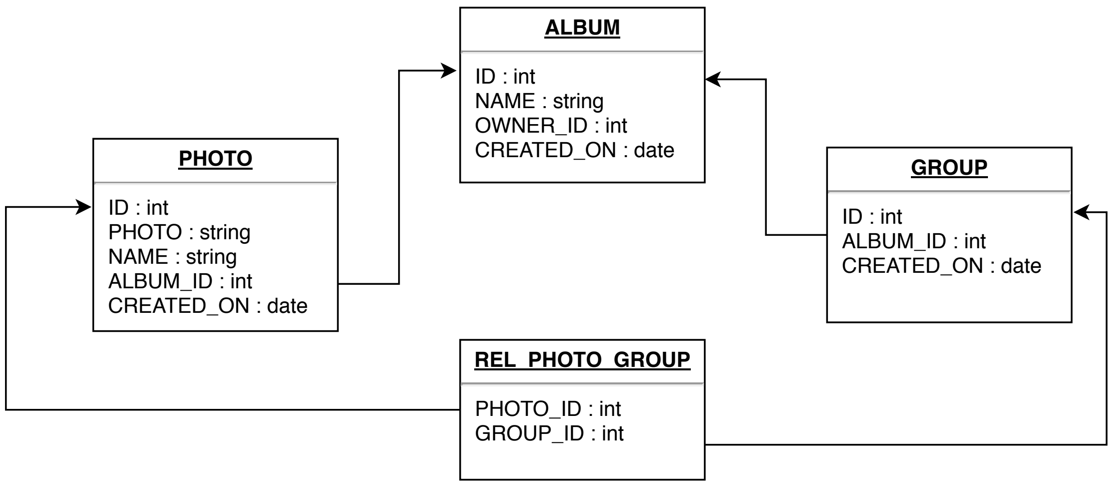


Figure 3.2: Storage data model.

Figure 3.2 illustrates the data model for the Storage database. This model is composed of four different tables, three of which represent entities and one of which represents a relation. The `PHOTO`, `ALBUM` and `GROUP` tables represent the respective entity within the

[1]Groups, or similarity groups, are sets of photos that are duplicate or very similar to each other. This concept is further detailed in the sections below.

system, whereas the REL_PHOTO_GROUP table represents a many-to-many relation between photos and similarity groups, i.e., a group may contain many photos and a photo may belong to several different similarity groups.

The second database is the Metadata database, responsible for holding data which the system requires in order to perform its image processing tasks.

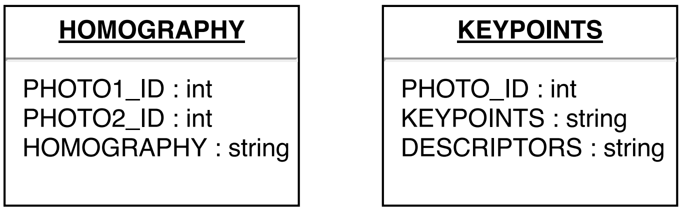


Figure 3.3: Metadata data model.

Figure 3.3 displays the model in which such data is organized. The HOMOGRAPHY table contains the calculated homography matrices between pairs of two photos, and the KEYPOINTS table holds the detected keypoints and keypoint descriptors for each photo.

The goal of this database separation is to enable future integration with external image hosting services. By keeping the storage database separated, we may, in the future, be able to replace it with, for example, Google Photos, while keeping the required homography and keypoints tables within the system.

3.3 System Workflow

As a multi-part system, the photo|uniq workflow crosses both the mobile application and the server infrastructure. While the user only interacts with the mobile application, image processing tasks occur in the servers before the results are available. In this section, we describe the workflow of the system and the available interactions between each component.

Using the mobile application, the user can perform a series of tasks, some of which require server interaction, and some of which do not. Photo browsing tasks, such as listing images from the user gallery, are performed independently by the mobile application, while tasks that require image processing are performed with the assistance of the server infrastructure.

The first step in the photo|uniq workflow is the photo upload to the server infrastructure, required by the system in order to perform any image processing tasks. This can be done via the mobile application by either selecting individual photos to upload or selecting an entire album.

With the photos stored in the server infrastructure's database, the user can now use the main features of the mobile application, which are related to assisted duplicate photo removal. The server infrastructure can then assist the user by automatically extracting groups of similar photos. Figure 3.4 depicts this process in a sequence diagram. However, after this is done, if the user does not agree with the results, it is possible to manually override the extracted groups using the mobile application.

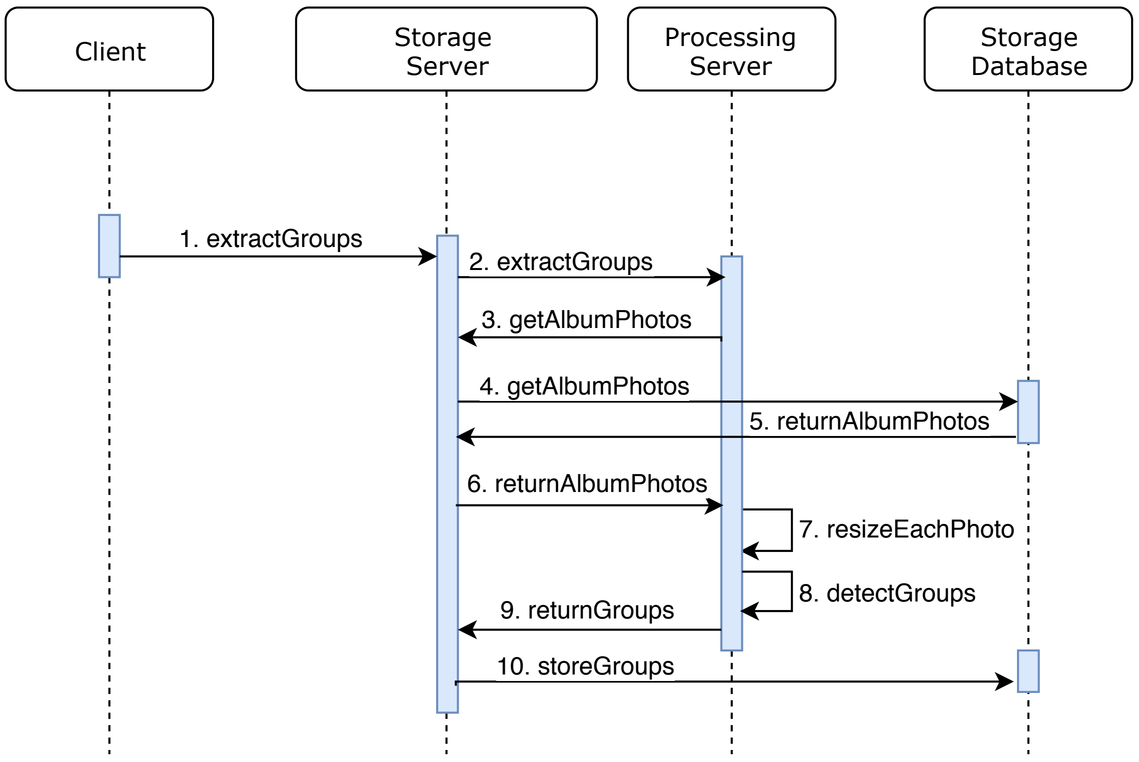


Figure 3.4: Sequence diagram for similarity group extraction.

After establishing similarity groups, the user can then proceed to analyze photos within those groups, either by comparing them manually using the mobile application's features or relying on the server infrastructure's automatic photo ranking, which orders the photos from best to worst in terms of quality. The automatic process follows the steps in Figure 3.5's sequence diagram and is further detailed in Section 3.5.3.3. Using the automatic method, the user can then observe the results and make intended changes within the mobile application. Otherwise, the user may also choose to manually compare images, which is another mobile application feature that relies on server infrastructure support.

This is done via the detection and calculation of the photos' keypoints and homography matrices, which are further detailed in Section 3.5.3.4 and Section 3.5.3.5, respectively, and allows the user to obtain a side by side view of both photos focused on the same motive.

Figure 3.6 displays an example without the use of the homography matrix, in which both sides are focused on the same area of the photo itself, but not on the same motive.

This difference is highlighted in Figure 3.7: the dashed red square depicts the incorrect area for comparison, illustrated in Figure 3.6, while the bright orange square represents the correctly focused area.

Figure 3.8 depicts an example using the photo|uniq mobile application, in which

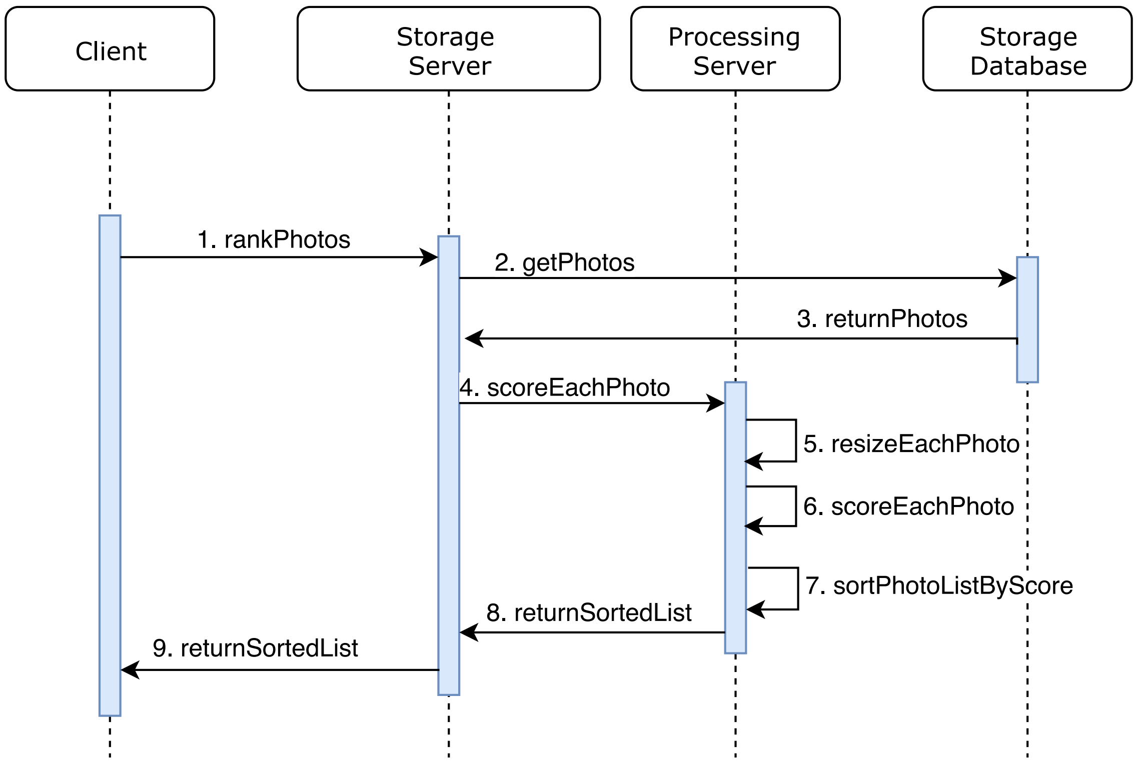


Figure 3.5: Sequence diagram for photo ranking.

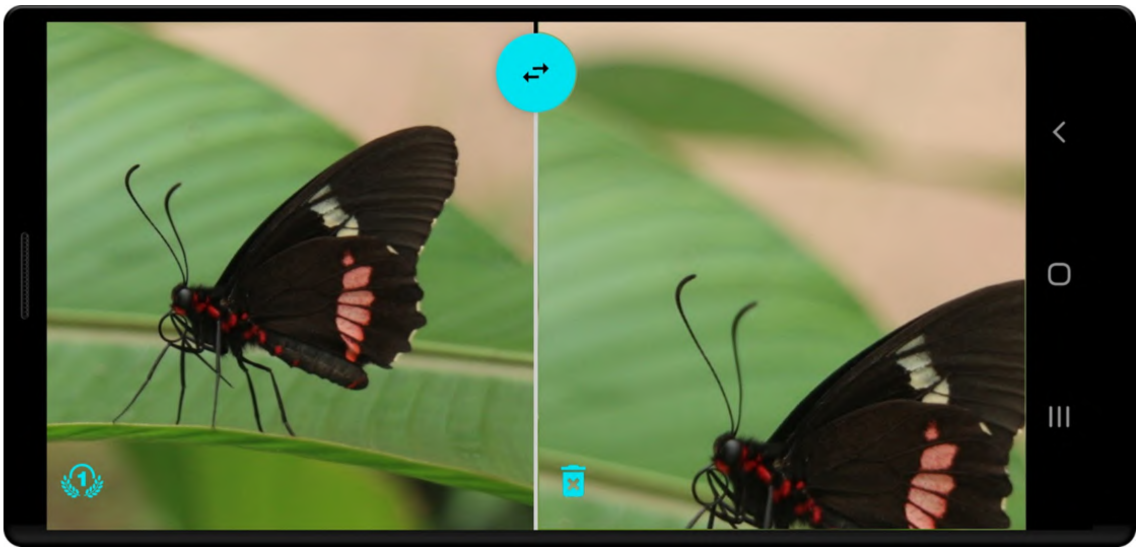


Figure 3.6: Photo comparison focused on the same area of the image, but not on the motive.

photos are displayed side by side and, despite having slightly different transformations among themselves, they can be panned or zoomed in/out while always maintaining focus on the same motive within the photo itself, rather than simply focusing on the same area of the image. In the mobile application, it is also possible to highlight specific points in one of the photos, which will then be highlighted in the correspondent location in the

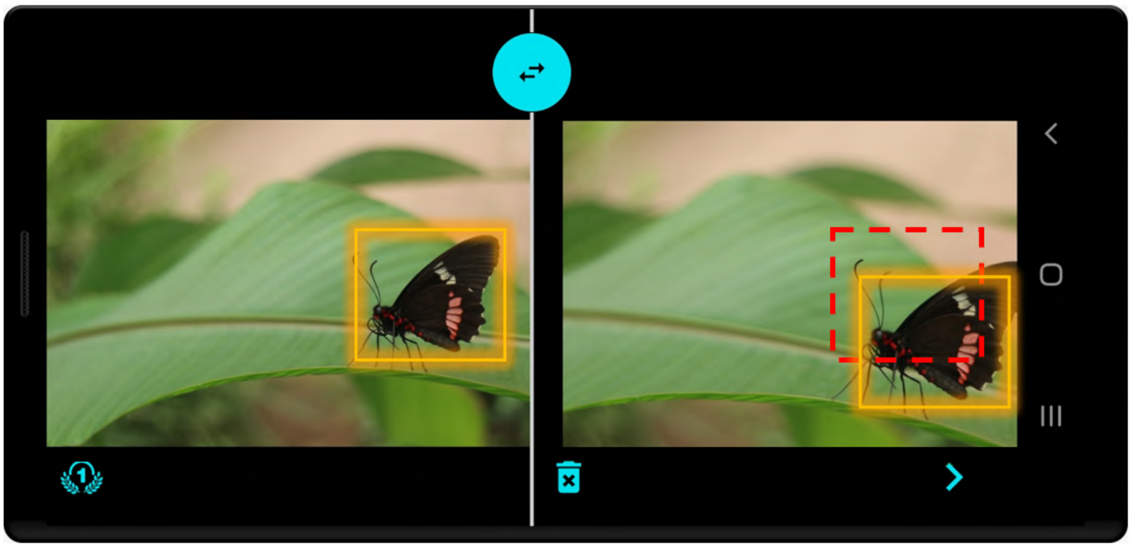


Figure 3.7: Different focus areas.

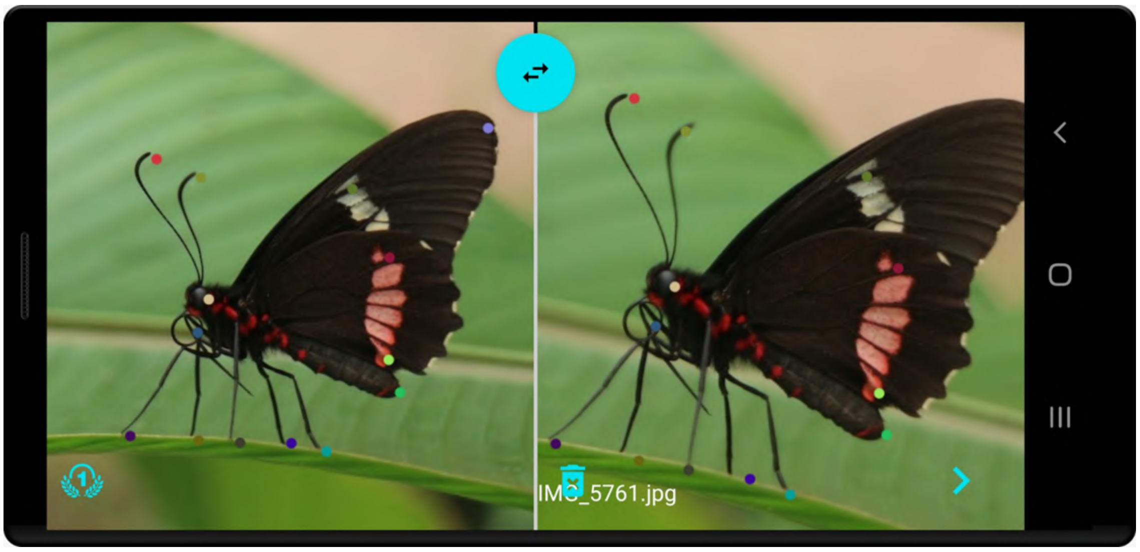


Figure 3.8: Photo comparison focused on the motive, with some highlighted points.

other image.

After analyzing the results, the user may then make the intended changes and delete the duplicate photos via the mobile application, removing them from the entire system.

3.4 Web Services

To interact with the server, we defined a set of web services to run within the photo|uniq system. As stated in Section 3.1, the Storage Server acts as an entry point for the entire system and, as such, it must be capable of handling every request. However, this server is not responsible for the execution of every type of request and, as such, it delegates processing tasks to the Processing Server. The list below specifies the high level web services defined for the macro system. In the cases where a JSON object is used, the object properties are sample values for each field.

- **Create Album** - Creates a new album.
  `Method: POST`
  `URL: /api/album`
  - Receives: JSON object with album data:

```
    {
        "NAME": "sample␣name",
        "OWNER_ID": 1
    }
```

  - Returns: Integer representing a new album ID
- **Delete Album** - Deletes an existing album and all photos within the album.
  `Method: DELETE`
  `URL: /api/album`
  - Receives: Integer representing the album ID
  - Returns: n/a
- **Rename Album** - Changes an existing album's name.
  `Method: PUT`
  `URL: /api/album`
  - Receives: JSON object with album data:

```
    {
        "ID": 1,
        "NAME": "sample␣name",
        "OWNER_ID": 1
    }
```

  - Returns: n/a
- **Upload Photo** - Uploads a photo to an album.
  `Method: POST`
  `URL: /api/photo`
  - Receives: JSON object with photo data:

```
    {
        "ALBUM_ID": 1,
        "PHOTO_NAME": "sample␣name",
        "PHOTO_BYTES": "/9j/4AAQSkZJ..."
    }
```

  - Returns: Integer representing the photo ID

- **Delete Photo** - Deletes an existing photo.
  `Method: DELETE`
  `URL: /api/photo`
  - Receives: Integer representing the photo ID
  - Returns: n/a
- **Get Photos** - Returns a list of photos, filtered by album.
  `Method: GET`
  `URL: /api/photo_list`
  - Receives: Album ID
  - Returns: JSON object with a list of photos:

```
    {
        [
            "PHOTO_ID": 1,
            "PHOTO_BYTES": "/9j/4AAQSkZJ..."
        ],
        [
            "PHOTO_ID": 2,
            "PHOTO_BYTES": "/9j/4AAKSZJ..."
        ],
        ...
    }
```

- **Create Group** - Creates a new similarity group inside an album.
  `Method: POST`
  `URL: /api/group`
  - Receives: Integer representing the album ID
  - Returns: Integer representing the new group ID
- **Delete Group** - Deletes an existing similarity group (does not affect photos within the group)
  `Method: DELETE`
  `URL: /api/group`
  - Receives: Integer representing the group ID
  - Returns: n/a
- **Change Photo Group** - Moves a photo from one similarity group to another.
  `Method: POST`
  `URL: /api/move_photo`

  - Receives: JSON object with the photo ID, new and old groups:

```
{
   "PHOTO_ID": 1,
   "OLD_GROUP_ID": 5,
   "NEW_GROUP_ID": 6
}
```

  - Returns: n/a

- **Extract Groups** - Automatically extracts similarity groups from an existing album.
  `Method: GET`
  `URL: /api/group/extract`
  - Receives: Integer representing the album ID
  - Returns: JSON object with photo IDs and the groups to which they were assigned:

```
{
   [
      "GROUP_ID": 1,
      "PHOTOS": [15, 18, 21, ...]
   ],
   [
      "GROUP_ID": 2,
      "PHOTOS": [16, 17, 19, 20, ...]
   ],
   ...
}
```

- **Get Homography** - Returns the homography matrix between two photos.
  `Method: GET`
  `URL: /api/homography`
  - Receives: Integers representing the IDs of two photos
  - Returns: Homography matrix in string format

- **Get Keypoints** - Returns the keypoints of the given photo.
  `Method: GET`
  `URL: /api/keypoints`
  - Receives: Integer representing the photo ID
  - Returns: Photo keypoints in string format

- **Get Ordered Photos** - Returns images within a similarity group sorted from best to worst quality.
  `Method: GET`
  `URL: /api/order_photos`

- Receives: Integer representing the group ID
- Returns: JSON object with ordered list of photo IDs from best to worst quality:

```
    {
        "RANKING": [15, 19, 20, 17, 21, 16, 18, ...]
    }
```

3.5 Implementation

To implement the photo|uniq system in its entirety, we made use of different tools and frameworks. This section describes each component and the steps taken to implement the architecture, data model and web services detailed in the previous sections.

3.5.1 Databases

The system is composed of two databases, which were both implemented in a similar fashion. Due to the relational nature of the data model, we chose to implement it using a relational database. The chosen management system was MySQL, due to its ease of usage and natural integration with the remaining system components. The two databases, StorageDB and MetadataDB, are very simple, composed only of tables with relations between them, as there was no need for additional structures such as views or triggers. Annex I contains the SQL creation scripts for both databases.

3.5.2 Storage Server

The storage server is the single entry point for the photo|uniq system. As such, it is responsible for handling every request that reaches the system. This section details how this server was implemented and how it performs its tasks.

3.5.2.1 Request Handling and Storage

The storage server is required to handle every incoming request to the system. This server must be able to forward requests that it is not able to handle, such as processing requests, while also being able to perform its own storage related tasks.

To implement the storage server, we chose to use the Django framework, mentioned in Section 2.1.1.1. This framework was chosen due to its known reliability and ease of usage, as well as being a solid fit with the server's requirements (i.e., request handling and database interaction).

Django allows direct integration with one or more databases, which then permits us to interact with database entries as objects. This simplifies the database access code and significantly accelerates the development process.

Listing 3.1: Django database settings.

```
1  DATABASES = {
2      'default': {
3          'ENGINE': 'django.db.backends.mysql',
4          'HOST': '{host}',
5          'NAME': 'StorageDB',
6          'USER': '{username}',
7          'PASSWORD': '{password}'
8      },
9      'metadb': {
10         'ENGINE': 'django.db.backends.mysql',
11         'HOST': '{host}',
12         'NAME': 'MetadataDB',
13         'USER': '{username}',
14         'PASSWORD': '{password}'
15     }
16 }
```

Listing 3.1 shows Django's database settings which, in photo|uniq, are configured to make use of two distinct databases. From lines 2 through 8 we define the default database, the StorageDB, which will be used whenever we do not provide a specific name when interacting with the databases. From lines 9 through 15 we define the secondary database, the MetadataDB. In lines 3 and 10, we instruct Django to use the MySQL database engine, which is one of many other engines available in the framework. This configuration enables us to interact with database objects as if they were data structures within the code base, removing the necessity of writing SQL queries or prepared statements.

To implement the database interactions, we defined a model class that Django can interpret and establish the necessary connections between code elements and database entries.

Listing 3.2: Django model class example.

```
1  from django.db import models
2
3  class Photo(models.Model):
4      id = models.AutoField(db_column='ID', primary_key=True)
5      photo = models.TextField(db_column='PHOTO', blank=True, null=True)
6      created_on = models.DateTimeField(db_column='CREATED_ON', blank=True, null=True)
7      updated_on = models.DateTimeField(db_column='UPDATED_ON', blank=True, null=True)
8      album = models.ForeignKey(Album, models.DO_NOTHING, db_column='ALBUM_ID', blank=True,
           ↪ null=True)
9      name = models.CharField(db_column='NAME', max_length=300, blank=True, null=True)
10
11     class Meta:
12         managed = False
13         db_table = 'PHOTO'
```

Listing 3.2 shows an example of one of such models. In line 3, we specify the class name, which will be the name used to reference this object within the code. From line 4 onwards, we specify the class properties, specifying details such as primary and foreign keys, and making use of Django's data types (e.g. DateTimeField on lines 6 and 7) to aid in the database integration process. In line 13, we specify the database table that corresponds to this class. This enables Django to either create the table if it does not yet exist, or to simply associate an existing table with this class.

To handle the web service component of the system, we used a Django extension named Django REST Framework. This extension provides features focused on the implementation of RESTful web services, such as serializers and routers, while keeping the remaining Django features available. In our system, serializers are used to enable the conversion of database query results into JSON objects, which greatly facilitates the communication not only between the storage and processing servers but also with the client applications.

Listing 3.3: Django REST framework serializer example.

```
1 from rest_framework import serializers
2 from webserver.api.models import Photo
3
4 class PhotoSerializer(serializers.HyperlinkedModelSerializer):
5     class Meta:
6         model = Photo
7         fields = '__all__'
```

Listing 3.3 shows an example of the Photo object serializer. Line 6 specifies the model to be used by this serializer, which, in this case, is the Photo model previously seen in Listing 3.2. Line 7 indicates that we want every field to be serialized. This allows us to send and receive Photo objects between servers while keeping track of every field, but we could easily define a more specific set of fields to serialize.

Finally, we handle routing using the Django REST Framework router, which works by specifying the intended route name and handler function.

Listing 3.4: Django REST framework router example.

```
1 router = routers.DefaultRouter()
2 router.register(r'albums', views.AlbumViewSet)
3
4 urlpatterns = [
5     path('', include(router.urls)),
6     path('homography/', views.homography_get),
7     ...,
8 ]
```

Listing 3.4 shows two ways of establishing these routes. Lines 1 and 2 are the Django REST Framework way, which defines a router and associates a ViewSet, which is a group of related views, to that route. Line 6 is the default Django way, which associates a route with a single view function. This view function is then responsible for handling each type of request (e.g., POST, GET, PUT, DELETE) within itself. In this dissertation, the implementation makes use of both systems: ViewSets are used for requests that may have more than one request type, such as Photos (GET photo, POST photo, DELETE photo), and regular views are used for simple requests with only one request type, such as Homography (GET homography).

3.5.2.2 Data Management

To manage the data it stores, the storage server makes use of the web service features described in the previous sections.

Photo management is achieved by defining a ViewSet, which handles photo creation, editing and deletion. Most of the work is performed behind the scenes by Django REST Framework, simplifies the development by handling the creation, edition and deletion using pre-implemented functions within the framework itself. However, we developed some extensions which enable us, for example, to obtain all photos filtered by album ID. An important extension was done to implement the photo upload process: using the pre-implemented create function as a base, we extended its functionality to return the ID of the created object, enabling the client application to receive the ID of the photo that was just uploaded for future access.

Album management is also managed via a single ViewSet, with one particular exception: deleting an album also deletes every photo within the album. As such, this triggers a set of operations that must be performed in order to prevent foreign key errors within the database: first, the homographies and keypoints of those photos must be deleted, then, the photos must be removed from any similarity groups, and finally, the photos themselves must be deleted. This was implemented as an extension of Django REST Framework's `perform_destroy()` function, which is called whenever an object is deleted.

Similarity Group management does not make use of ViewSets, but rather multiple distinct views. This was implemented in this way due to the additional complexity introduced by a photo potentially belonging to multiple groups. As such, the Photo-Group relation must be mantained in a separate database table, rather than a simple foreign key such as the album ID. This would lead to the necessity of overwriting every function within a ViewSet and, as such, we chose to implement each view function separately.

Group creation is simple, due to the fact that the group does not have any properties besides its ID and album ID. When adding photos to a group, we establish an entry in the Photo-Group relation table, which specifies the photo ID and group ID. The opposite happens when removing a photo from a group: we simply remove the photo ID and group ID entry from the table. It is also possible to move photos from one group to another, which happens by receiving the old and new group IDs and making the change in the relation table. Finally, deleting a group results in all photos being disassociated with that group.

These similarity group management functions work manually, meaning that the group contents is managed by someone who is manually adding or removing photos between groups. An automatic function is available, which is executed by the processing server and will be detailed below, in Section 3.5.3.2.

3.5.3 Processing Server

The processing server is responsible for every image processing request that reaches the system. It communicates exclusively with the Storage Server and the metadata database. In the sections below, we describe the processing server's main functions.

3.5.3.1 Request Handling and Storage

The processing server interacts exclusively with the storage server and the metadata database. This means that every request will occur within the system's controlled environment and, as such, there is no need for a fully-fledged full-stack framework, such as Django, to handle incoming requests and database accesses.

For this reason, we chose to implement the processing server using the Flask framework, described in Section 2.1.1. This framework allows us to quickly setup a server structure without unnecessary features for the processing server, such as Django's ORM for database accesses or complex serializers.

With Flask, instead of using an ORM, we handle database access using prepared statements. In this case, prepared statements are easy to maintain due to a single database connection (the metadata database), which, in turn, is a very simple database composed only of two tables and no relations between them. As such, we dismiss the use of a heavier solution such as Django's ORM in favor of a lighter-weight framework.

To handle incoming requests, Flask provides a simpler form of routing which works by annotating functions with the intended route.

Listing 3.5: Flask routing example.

```
1 @app.route('/api/homography/', methods=['GET'])
2 def getHomography():
3     ...
```

Listing 3.5 illustrates an example of Flask's routing method. Line 1 shows an annotation inserted before a function. This annotation contains the route that we intend to associate to this function, and the HTTP method we intend to handle. In this case, the `getHomography()` function would be executed whenever a GET request was sent to `/api/homography/` within the processing server URL.

To accelerate the system and avoid potentially long waiting times from the mobile application, we chose to implement the processing server's tasks asynchronously, meaning that whenever possible, we execute processing tasks separately from the main execution thread. By launching a separate thread for each processing task, we are able to handle multiple incoming requests simultaneously and provide immediate feedback before the processing work is completed. In these cases, the processing server executes the requested work and rather than returning it to the caller, stores it in the Metadata database, which will then be queried by the storage server until it contains the required results.

3.5.3.2 Similarity Group Extraction

When attempting to compare photos, a certain degree of similarity is required in order to establish an adequate comparison. As an example, figure 3.9 depicts two completely different pictures from which we do not expect a comparison to be established.

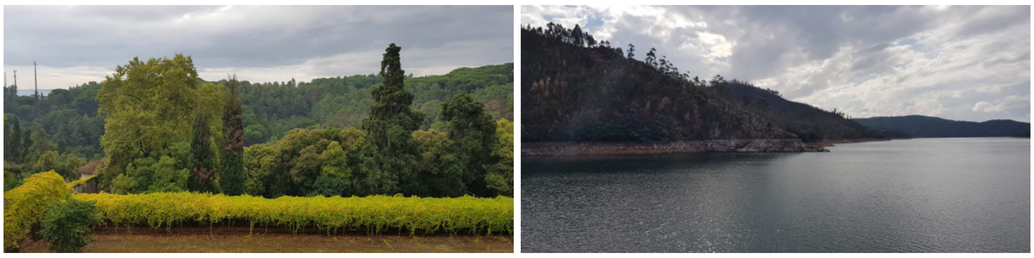


Figure 3.9: Photos with different motives.

Figure 3.10, however, depicts two relatively similar photos loosely focused on the same motif. This is an example of the expected degree of similarity when comparing photos.

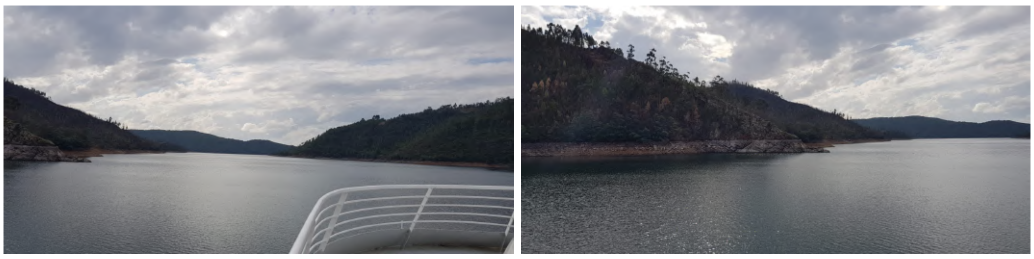


Figure 3.10: Photos with similar motives.

These similar pictures can be organized in similarity groups. The photo|uniq system provides web services to create and delete similarity groups, as well as moving photos between groups. This means that users can manually identify similar photos and establish similarity groups as intended.

However, the system also provides an automatic way of establishing similarity groups. This can be done via two different methods: structural similarity or mean squared error. Using scikit's `compare_ssim` function, we can obtain the structural similarity[2] between two pictures and obtain a similarity index. This value is a number ranging from 0 to 1, with the highest value being the most similar. The server applies this function to every

[2]The structural similarity is a measure of the similarity between two images. This also serves as a quality measure, which works by using the first image as a perfect reference and then evaluates the quality of the second image based on the first one.

pair of pictures within an album and, when the similarity index exceeds a certain threshold, establishes a similarity group with those pictures. The second method, mean squared error, works by subtracting the pixel intensities of each image. The resulting value indicates how similar the images are: if the result is 0.0, the images are identical; if the result is greater than 0.0, there are differences. This method tends to be difficult to evaluate due to the fact that slightly different images may produce very large mean squared error values. However, we chose to implement both methods as the mean squared error is significantly faster while still providing satisfactory results, as described in Chapter 4, and structural similarity is still available for situations where comparison accuracy is more important than elapsed time.

This provides an effortless way for users to organize pictures in similarity groups, while maintaining the ability of manually changing the automated grouping. The automatic group generation is implemented by comparing all pictures between themselves. This results in a similarity matrix, of which an example is provided below, using the structural similarity method:

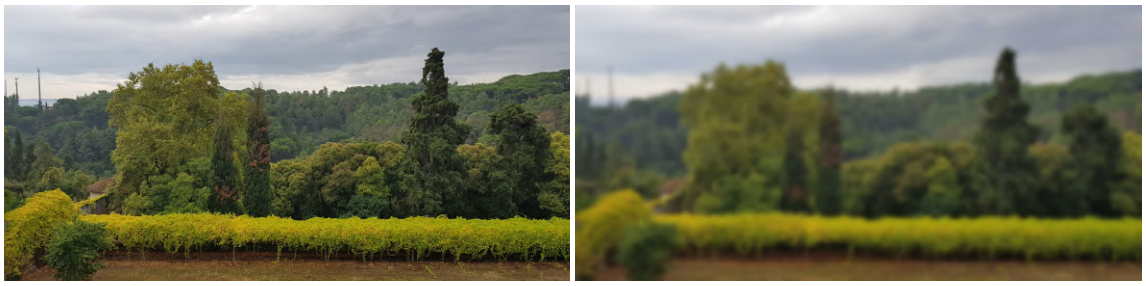
$$M = \begin{bmatrix} 1.0 & 0.55 & 0.49 & 0.45 & 0.43 \\ 0.55 & 1.0 & 0.43 & 0.46 & 0.45 \\ 0.49 & 0.43 & 1.0 & 0.46 & 0.41 \\ 0.45 & 0.46 & 0.46 & 1.0 & 0.59 \\ 0.43 & 0.45 & 0.41 & 0.59 & 1.0 \end{bmatrix} \tag{3.1}$$

The displayed matrix represents the comparison of the photos within an album. The dimension of the matrix represents the number of photos being compared. In this case, the 5×5 matrix indicates that the album was composed of five pictures. Each entry represents a comparison between two photos, e.g., the entry $M_{23} = 0.43$ indicates that the mean structural similarity between photos 2 and 3 is 0.43. It is important to mention that this comparison is commutative, i.e., comparing photos 2 and 3 is the same as comparing photos 3 and 2. As such, $M_{23} = M_{32} = 0.43$, which means that we can calculate only half of the matrix, saving processing time. Naturally, comparing a photo with itself yields a mean structural similarity of 1.0, e.g., $M_{33} = 1.0$.

To extract the similarity groups from the matrix, we establish a minimum threshold, above which we consider photos to be similar. This threshold is a configurable parameter in the photo|uniq server. The group extraction process works by first iterating every line and, for each line, creating a group with the photo correspondent to the matrix index whose value is above the threshold. In the matrix above, setting, for example, a threshold of 0.5, this would produce the following groups:

$$(1,2)(2,1)(4,5)(5,4)$$

The server then excludes the duplicate groups (e.g. $(2,1)$ and $(5,4)$) and creates the remaining ones in the storage database, which will allow them to be used as regular



(a) Photo with BRISQUE score of 56.7.

(b) Photo with BRISQUE score of 70.6.

Figure 3.11: Image quality assessment using BRISQUE.

similarity groups. After being created, these groups can be manually edited using the similarity group management web services available in the photo|uniq storage server.

3.5.3.3 Photo Scoring

As seen in Section 3.4, one of the provided web services returns an ordered list of similar photos, sorted from best to worst quality. While image quality is, generally, a subjective measure, we can apply image processing techniques to identify and score features that objectively represent an image's quality value.

A possible technique would be to use the `compare_ssim` function described in Section 3.5.3.2 which, as mentioned earlier, measures the structural similarity between two images. As such, in order to sort a set of pictures from best to worst using `compare_ssim`, we would need to manually specify the best image and compare every other image with the best one. This, however, defeats the purpose of the photo|uniq concept: the user should not be required to manually specify the best photo.

As such, it is clear that the solution requires a blind quality assessment, i.e., the quality measure must be able to assess the quality of an image without requiring another image as reference.

To this end, we chose to use PyBRISQUE, which is a Python implementation of the Blind/Referenceless Image Spatial Quality Evaluator (BRISQUE) [15]. The BRISQUE algorithm identifies properties that occur whenever an image presents distortions or luminance alterations that do not appear in natural images [3].

The result of this algorithm is a value between 0 and 100, where 0 represents the best quality and 100 the worst quality.

Figure 3.11 depicts two similar images, where the left image is a photo taken with a camera and the right image is the same photo with a blur effect applied in post-processing. Naturally, we expect the image on the right to have a higher quality and it is indeed the case, having a BRISQUE score lower than the blurred image.

[3]The BRISQUE algorithm considers natural images any image that is captured by an optical camera which is not subject to artificial or accidental distortions.

3.5.3.4 Keypoint Detection

Keypoints are specific points within an image that are highly distinctive, which means that even if the image is distorted or transformed, these keypoints remain the same. This is useful for several purposes, but, in the context of this dissertation, keypoints are particularly useful for the calculation of the homography matrix, detailed in Section 3.5.3.5.

To detect keypoints within an image, we chose to use OpenCV's implementation of the Speeded-Up Robust Features (SURF) algorithm. This algorithm is a speeded-up version of previous keypoint detection algorithms [19], which is interesting in the context of a end-user focused application.

Keypoint detection is a costly task in terms of processing time. This time is further increased by the photo size and pixel density, which is relatively high with current equipment (smartphones and consumer cameras). This may lead to high memory usage when detecting keypoints and even, in some cases, lead to out-of-memory errors. To minimize this, we chose to resize the photos before applying the SURF algorithm. This allows us to maintain control over the maximum dimensions of pictures while performing heavy processing tasks, thus minimizing the chance of unexpectedly high resource consumptions.

In the photo|uniq system, we try to minimize keypoint detection by storing the results in the database. Whenever a keypoint calculation is requested, the first step is to check whether the database already contains the keypoints for the requested image. If it does, the keypoints are returned. Otherwise, the detection process continues normally.

3.5.3.5 Homography Calculation

The homography matrix, as seen in Chapter 2, represents the correlation between two images' keypoints. Specifically, the homography matrix describes the necessary operations to transform one image's keypoints into another's. In photo|uniq, the homography matrix is used in the client application, for example, to manually compare two photos while focusing on the same motif. This allows the comparison to be focused on the same point even if the images have different transformations between them.

To implement this, we use the detected keypoints (whether from the database if they already exist, or newly detected if they do not), as described in section 3.5.3.4, and apply OpenCV's keypoint matching functions, which associate an image's keypoints with another's. However, we chose to work with resized images in order to alleviate the processing requirements. This means that the resulting homography matrix may not match the original photos, as the original and resized scales may not be the same.

To fix this, we first established a fixed value for the resized images. We established that the larger dimension would be resized to a maximum threshold, and the other dimension would be appropriately resized in order to maintain aspect ratio. For example, if we resized a 1920×1080 image, we would first take the larger dimension (1920) and resize it to our maximum value of, for example, 1024, and would appropriately resize the second dimension (1080) while maintaining aspect ratio, which would result in a 1024×576

image. Then, we store the scaling factors for each image in the homography matrix calculations using the following formula

$$r_1 = \frac{size}{max(width_1, height_1)}, \quad r_2 = \frac{size}{max(width_2, height_2)}$$

where r_i represents the scaling factor for each image, $max(width_i, height_i)$ represents the largest dimension between the width and the height and $size$ represents the size to which we intend to resize to. Finally, after obtaining the matrix for the resized images, we apply the following operations to the result

$$H = \begin{bmatrix} H_{11} \times r & H_{12} \times r & \frac{H_{13}}{r_1} \\ H_{21} \times r & H_{22} \times r & \frac{H_{23}}{r_1} \\ H_{31} \times r_2 & H_{32} \times r_2 & H_{33} \end{bmatrix}$$

where H is the resulting homography matrix and $r = \frac{r_1}{r_2}$. In short, this scales the homography matrix of the resized images so that it may be applied to the original images.

Figure 3.12 depicts the homography matrix calculation process in its entirety, which is detailed in the list below. Some steps, such as photo retrieval from the storage database, were ommitted for readability purposes.

1. The client application requests the homography matrix between two images, providing their IDs to the storage server;
2. The storage server checks whether the homography was previously calculated, by consulting the Metadata database;
3. The metadata database returns the homography if it exists, otherwise returns nothing. If it exists, the process skips to step 16 and returns the homography to the client application. Otherwise, the process continues to step 4;
4. The storage server requests the homography calculation from the processing server;
5. The processing server requests the first image from the storage server;
6. The storage server returns the first image to the processing server;
7. The processing server requests the second image from the storage server;
8. The storage server returns the second image to the processing server;
9. The processing server checks whether the keypoints for the first image were previously calculated;
10. The metadata database returns the keypoints if they exist, otherwise the processing server calculates them and stores them in the metadata database;

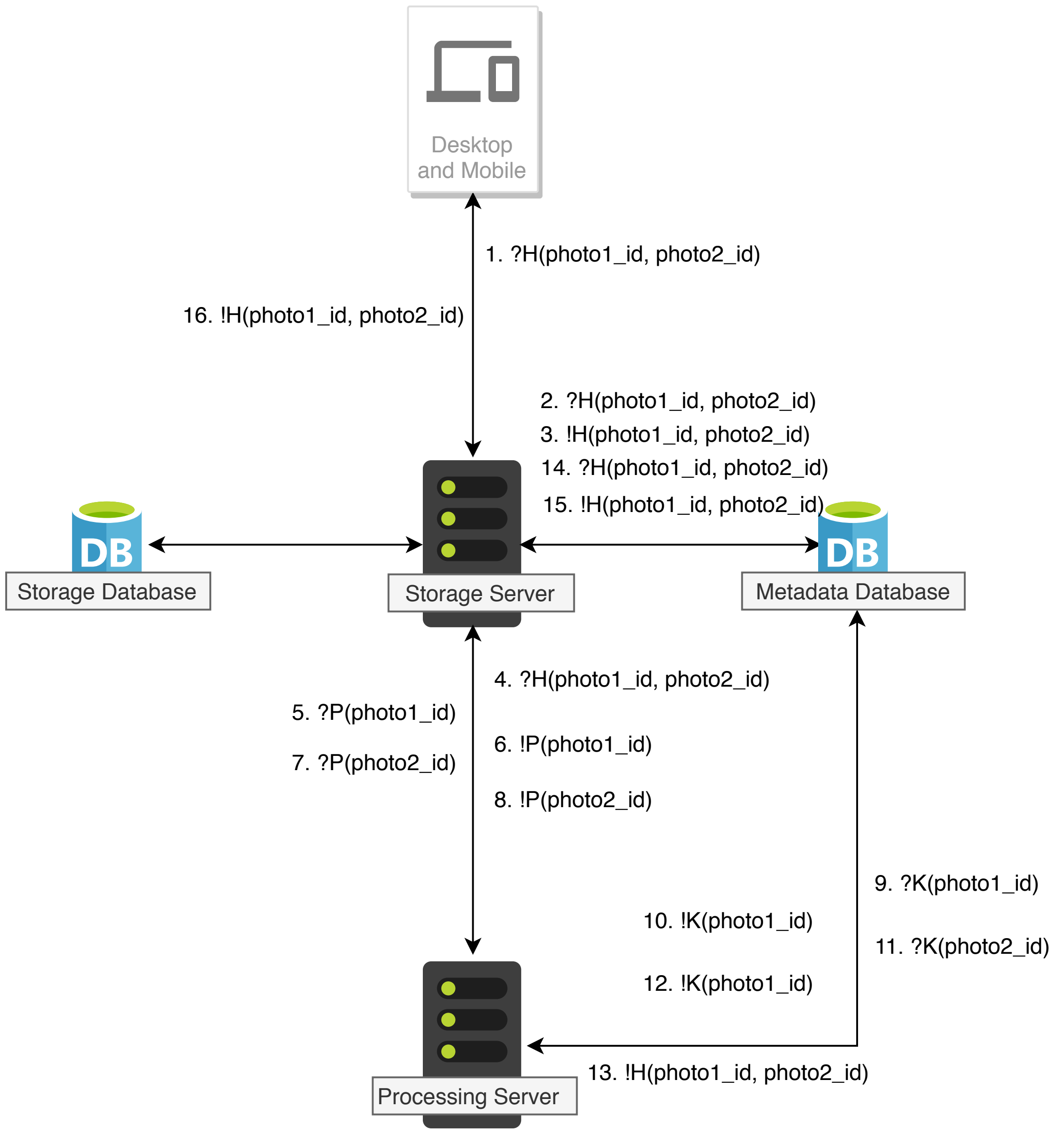

Desktop and Mobile
1. ?H(photo1_id, photo2_id)
16. !H(photo1_id, photo2_id)
2. ?H(photo1_id, photo2_id)
3. !H(photo1_id, photo2_id)
14. ?H(photo1_id, photo2_id)
15. !H(photo1_id, photo2_id)
DB
Storage Database
Storage Server
DB
Metadata Database
4. ?H(photo1_id, photo2_id)
5. ?P(photo1_id)
7. ?P(photo2_id)
6. !P(photo1_id)
8. !P(photo2_id)
9. ?K(photo1_id)
10. !K(photo1_id)
11. ?K(photo2_id)
12. !K(photo1_id)
13. !H(photo1_id, photo2_id)
Processing Server

Figure 3.12: Homography matrix calculation process.

11. The processing server checks whether the keypoints for the second image were previously calculated;

12. The metadata database returns the keypoints if they exist, otherwise the processing server calculates them and stores them in the metadata database;

13. The processing server calculates the homography matrix and stores it in the metadata database;

14. The storage server, after step 3, continuously checks the metadata database on a regular interval on whether the homography matrix was already calculated;

15. After being stored by the processing server, the storage server obtains the newly calculated homography matrix;
16. The storage server returns the homography matrix to the client application.

Chapter 4

Validation

In this chapter, we will validate the developed system in terms of accuracy (i.e., if the computed results match the expected results) and execution speed. To do so, we implemented a test application that acts as an external client. The sections below describe this application, as well as the executed tests and results.

4.1 Testing Environment

To simulate the client application, we developed a simple Python application which calls the web requests that are available in the system. This application makes use of the `requests` library, which provides simple syntax for HTTP requests. Additionally, this application measures the elapsed time of each request, providing an accurate representation of the temporal cost of each operation.

In the sections below, we list a set of tasks which will be performed using resized[1] and non-resized images in order to compare the time and accuracy with each image size. The tests were performed with the full system running within the same machine in order to eliminate network latency issues during testing. The specifications of the testing machine are as follows:

- CPU: Intel Core i7-6700HQ @ 2.60GHz
- RAM: 8GB
- GPU: NVIDIA GeForce GTX 950M
- OS: Ubuntu 18.04.02 LTS

[1]The resized images will be scaled to different width values while keeping the same aspect ratio.

To test the system, we used a set of 16 pictures taken with the same device (a Samsung Galaxy S8 smartphone). All photos were taken with the same dimensions: 4096×2268 pixels. To run the tests, the pictures were uploaded to a previously existing empty album, taking an average of 10.2 seconds for the upload of the 16 pictures.

4.2 Similarity Group Extraction

To validate similarity group extraction, we tested the Extract Groups web service defined in Section 3.4 using the two different methods described in Section 3.5.3.2: structural similarity index and mean squared error.

To evaluate the correctness of the results, we manually identified groups of similar pictures and compared them with the automatically generated groups. Besides correctness, we also measured the elapsed time of each method.

Figure 4.1 illustrates the manually formed groups with which the results will be compared to.

The similarity group extraction service was called by the testing application and the elapsed times of each method can be found in Figure 4.2. The X axis contains the images' width in pixels, while the Y axis contains the elapsed time, in seconds. The blue line represents the elapsed time for each image using the SSIM method, while the orange line represents the elapsed time for each image using the MSE method.

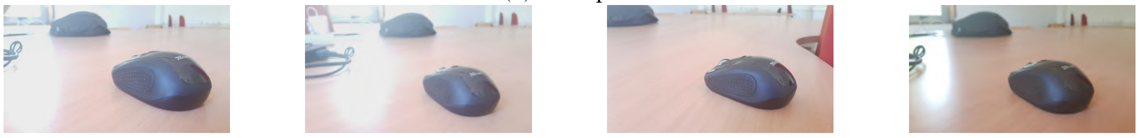
The graph shows that in the performed tests, MSE is always faster than SSIM. For 256 pixels wide images, the difference is almost negligible. As the image size increases, the MSE line remains relatively flat, while the SSIM line grows by a large factor. This is particularly noticeable when the image size grows from 1024 to 2048 pixels and 2048 to 4096 pixels.

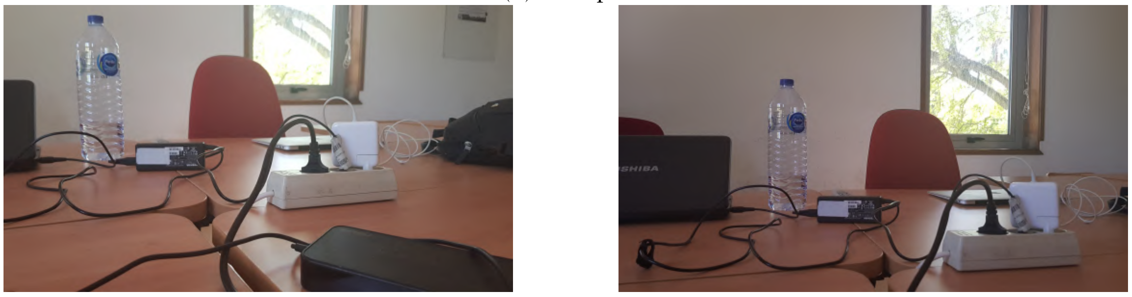
With this graph, we can assess that elapsed time is heavily affected by image size when using SSIM, but is relatively unimportant when using MSE, as the line remains close to constant. However, as image size increases to larger values, a slight growth is noticeable, as seen in the graph between the 2048 and 4096 values.

Regarding the correctness of the results, we measured the number of photos correctly and incorrectly assigned to similarity groups. When resizing images to very small values, such as 256 and 512 pixels of width, too much detail is lost. As such, the similarity algorithms fail to identify significant differences between the images and classify them into a single similarity group, which does not serve the system's purpose. As such, we established the value of 1024 pixels as the comparison for the correctness of the results, as this value provides enough detail to clearly identify groups of similar images.

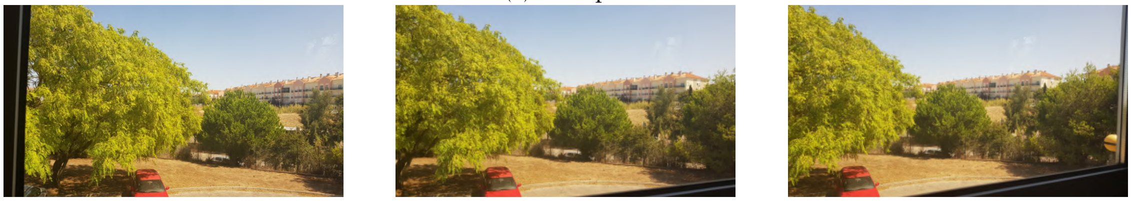
Figure 4.3 shows the results of this test: the X axis represents each method's name for full-sized and resized (to 1024 pixels of width) images, and the Y axis represents the number of photos. The green bars represent the number of correctly assigned photos and the orange bars represent the number of photos assigned to wrong groups.

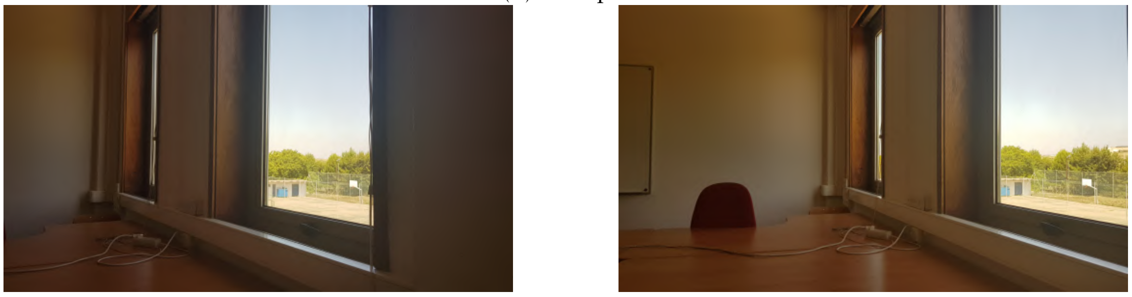
(a) Group 1.


(b) Group 2.



(c) Group 3.


(d) Group 4.


(e) Group 5.

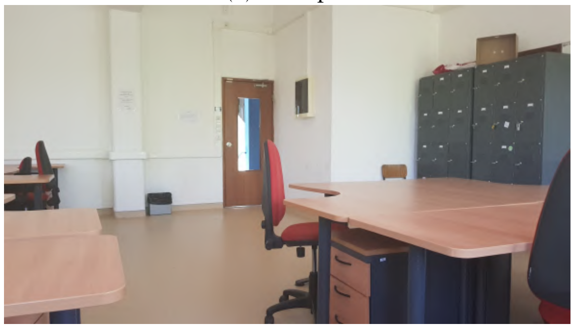

(f) Group 6.

Figure 4.1: Manually formed similarity groups.

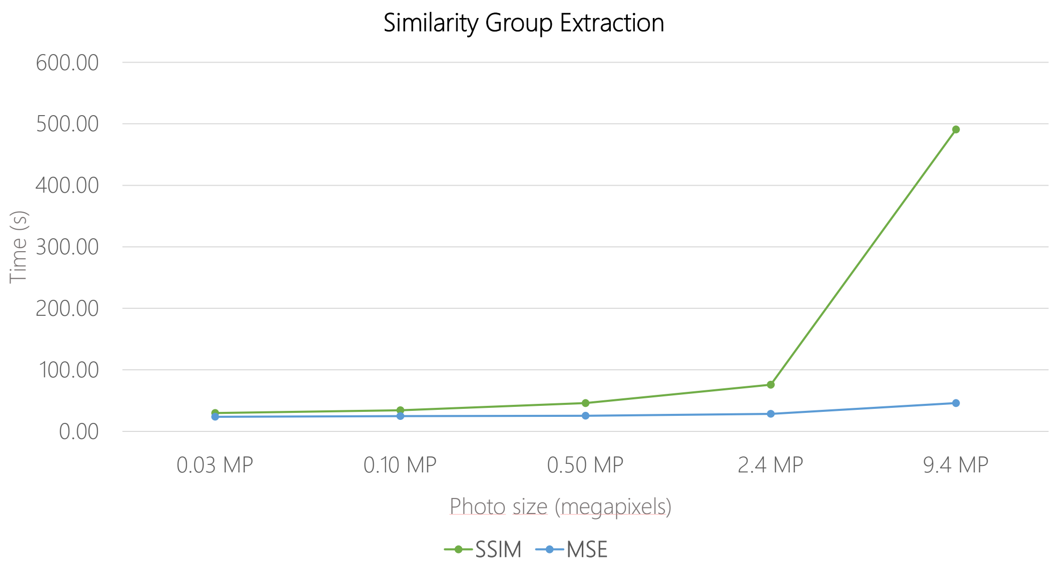


Figure 4.2: Similarity group extraction elapsed time.

We defined a correct group as a group that is composed of photos that we consider similar, such as the groups in Figure 4.1. Naturally, an incorrect group is a group that does not match this criteria. It is also important to note that the sum of correct and incorrect photos for each method is not equal to the total number of photos. This happens because a photo may belong to several similarity groups, or may simply not be assigned to any groups if it is not considered similar to any other photos.

The results show that the MSE method identifies more similar photos than SSIM. While MSE assigns a larger number of correct photos, it also incorrectly classifies a larger amount of photos into the wrong groups. SSIM, on the other hand, identifies a smaller number of correct photos, but the number of wrong results is also significantly lower. Resizing the images does not seem to have a strong impact in the correctness of the results, as the only difference in the two methods is a single correct group assignment in the SSIM method.

Taking the elapsed time into consideration, and given that the end user has the possibility of manually correcting the results, MSE is likely to be the better alternative. The longer execution time of SSIM is not practical for a end-user focused application, and MSE, while providing a significant number of incorrect results, still offers a large amount of correctly assigned photos with a relatively fast execution time.

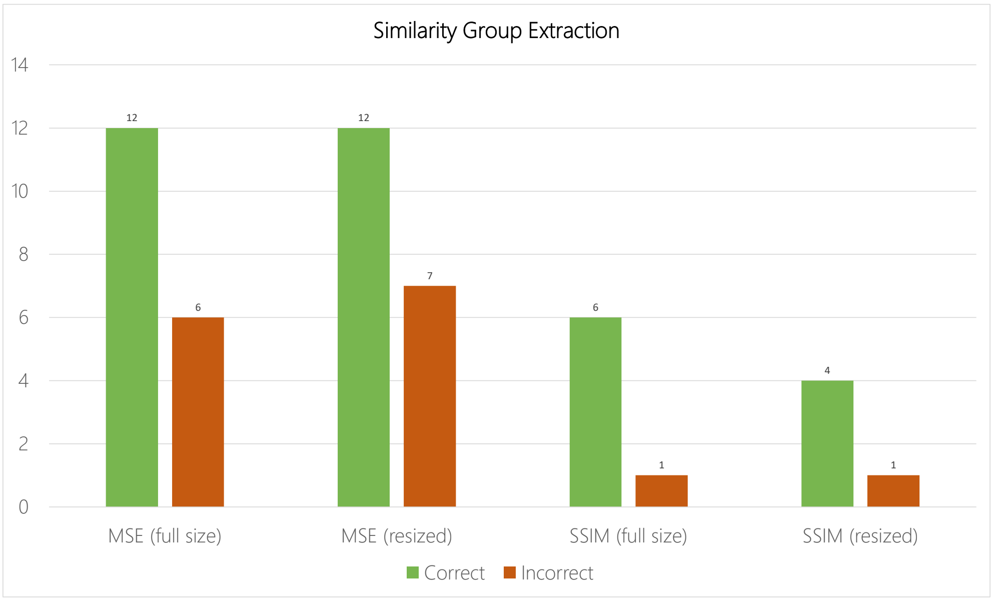


Figure 4.3: Similarity group extraction correctness.

4.3 Photo Scoring

To evaluate photo scoring, we tested the service using one of the similarity groups defined in Figure 4.1, specifically Group 2, which is composed of four similar photos.

Figure 4.4 depicts the elapsed time of the photo scoring function for full sized and resized photos. The X axis represents the photo widths and the Y axis represents the elapsed time, in seconds. As expected, photo scoring is much faster with smaller images, increasing significantly as the image size also increases.

Figure 4.5 shows the score assigned to each image. The X axis represents the photo widths and the Y axis represents the BRISQUE score. Each coloured bar represents a different image. With this test, we intend to identify the image dimensions which more closely approximate the original image's (the image with 4096 pixels of width) score. We do this not by examining the score value directly, as an image and its resized version are, effectively, different images with different scores, but by evaluating the resulting order. For example, for 512 pixels of width, the BRISQUE score ranks Image 4 as the best image (highest score) and Image 1 as the worst. However, using the original images, the highest scoring image is Image 3, which means that 512 pixels of width was not a good enough approximation. In this case, the best result would be the one produced using

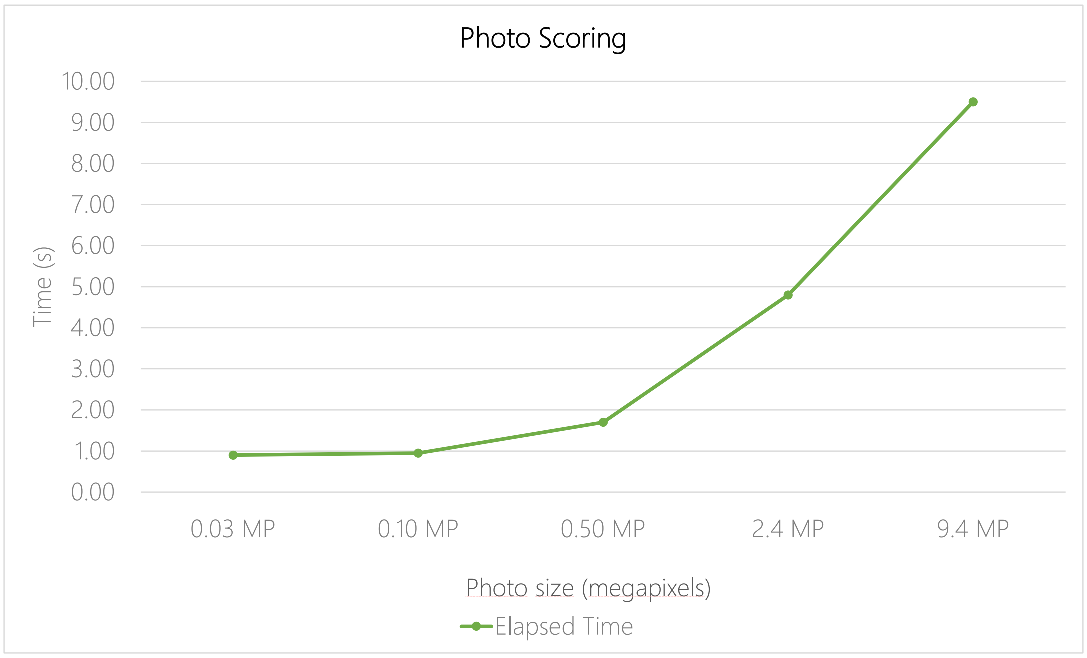


Figure 4.4: Elapsed time for photo scoring.

2048 pixels of width, as the resulting ranking is the same as the one produced with the original images.

Naturally, it is impossible to determine which of the obtained rankings is, in fact, the correct one, as image quality is a subjective measure. As such, we chose to favor methods that more closely approximate the original images while providing a decent speed up, which in this case would be using 2048 pixels wide images. This width provides similar results and takes approximately half of the execution time in comparison to the original images, which is an acceptable result for a end-user focused application.

4.4 Keypoint Detection

Keypoint detection occurs in the system whenever a homography matrix is requested. This means that if the system did not previously detect the keypoints for one or both of the photos involved in the homography, it will have to detect them when requested. As such, we tested keypoint detection by measuring the elapsed time for the execution of the keypoint detection in two photos.

Figure 4.6 shows the elapsed time for keypoint detection with differently sized images. The X axis represents the photo dimensions and the Y axis represents the elapsed time,

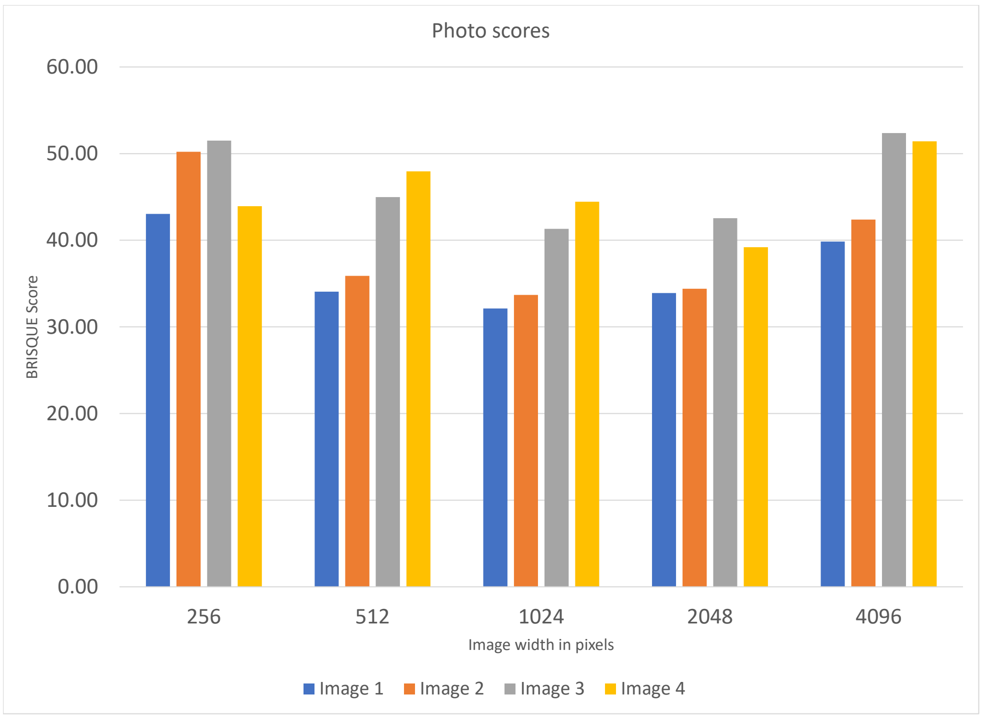


Figure 4.5: BRISQUE score for differently sized images.

in seconds. As expected, keypoint detection is much faster with smaller photos. Up until 1024 pixels of width, the elapsed time is close to constant, with only a slight increase. After 1024 pixels, the elapsed time grows to more than double.

4.5 Homography Calculation

As mentioned in Section 4.4, calculating the homography matrix requires the detection of keypoints. The homography calculation, being mostly arithmetic operations, is relatively fast, meaning that most of the elapsed time is taken by the keypoint detection. To measure the impact of image size in homography matrix calculation, we removed the time taken by keypoint detection from the results.

Figure 4.7 depicts the measured elapsed time for homography calculation. The X axis represents the image dimensions and the Y axis represents the elapsed time, in seconds.

The curve in the graph shows that the homography matrix calculation takes longer for larger images, as expected. However, even the highest elapsed time, ocurring for images with 4096 pixels of width, is relatively small, taking approximately 0.23 seconds of execution time. The line is relatively flat up until 1024 pixels of width, where it grows to almost double the elapsed time and again at 2048 pixels, where the process takes

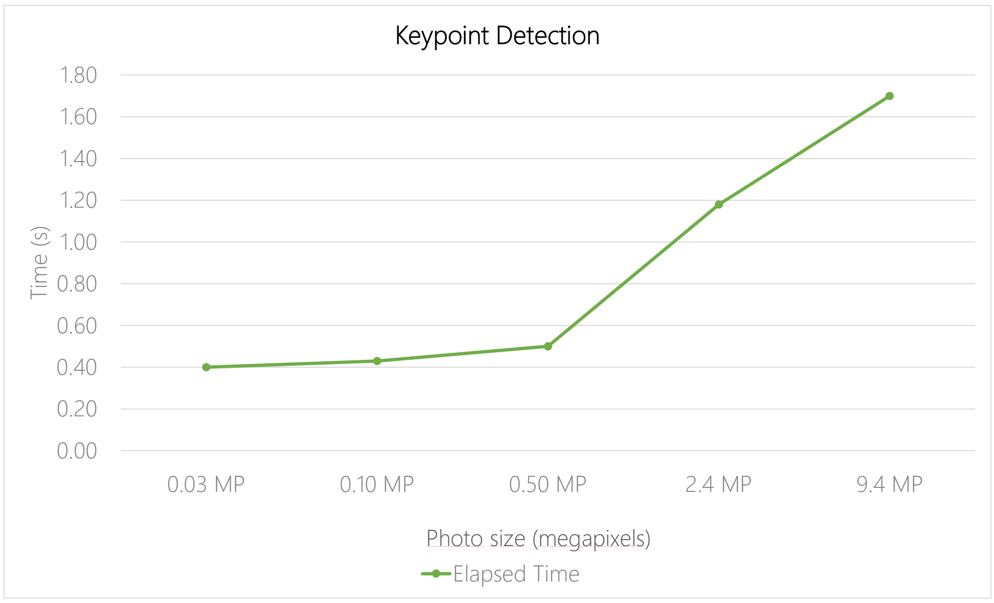


Figure 4.6: Elapsed time for keypoint detection.

almost four times longer.

Although homography matrix calculation is a relatively fast process, multiple requests can quickly lead to a larger load for the system's processing ability and, as such, we should attempt to minimize the impact of each small task. In this case, 1024 pixels of width seems to be the ideal image, as the additional elapsed time in comparison with 256 and 512 pixels is almost negligible, and the next dimension, 2048, takes significantly longer to execute.

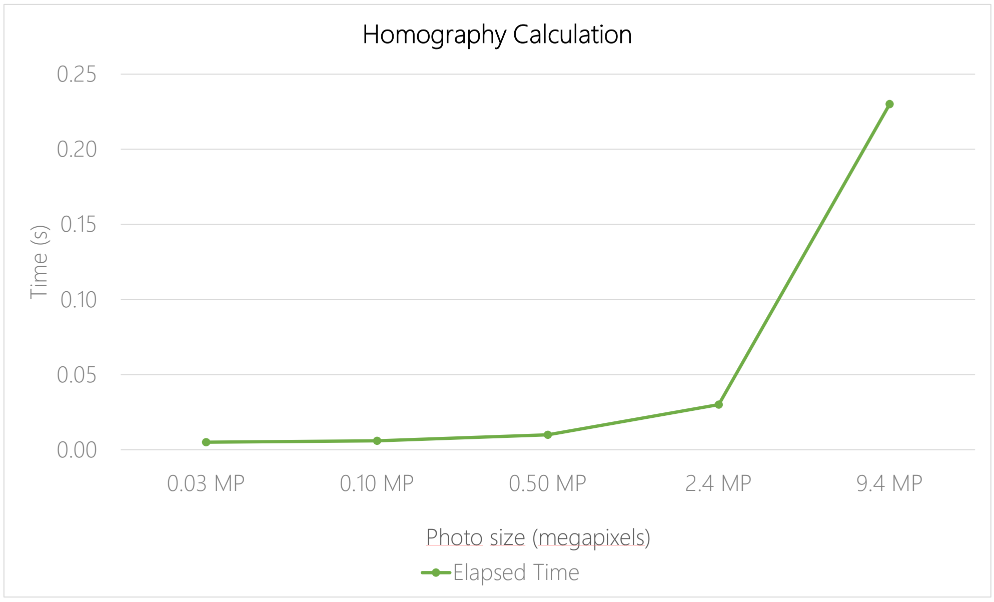


Figure 4.7: Elapsed time for homography calculation.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

In this dissertation, we identified the duplicate photo problem, described in Section 1.2, and developed a server infrastructure that, in partnership with the photo|uniq mobile application, solves it. As part of the photo|uniq system, we developed the server system architecture taking into account the integration with the existing mobile app and, in the future, other possible client applications. With the architecture defined during this dissertation's preparation phase, we managed to implement the system without major drawbacks, which was a great benefit in terms of the quality of the final product.

The developed work is an image processing system structured across two servers, Storage and Processing, where the former is accessible to outside applications via a set of web services and the latter is only accessible within the system itself, with data storage being handled by two distinct databases. This structure gave us a simple connection point with the mobile application, which is only required to keep track of a single server's address, and perform request handling and redirecting within the system itself.

Regarding the implementation, it is important to note the impact of using external frameworks, such as Django and Flask. Using these framework's tools as a baseline for the development of both servers, we managed to greatly speed up the initial process of setting up database connections and request routing, allowing us to focus more on the image processing aspect which, in turn, led to a better final product.

During the validation phase, as detailed in Chapter 4, we gathered that the methods that provided the best results were often too slow to use in a live application. However, we managed to circumvent this issue by implementing alternative methods that while not as accurate, provide satisfactory results with a much faster execution time, such as MSE versus SSIM (in Section 3.5.3.2) for similar image detection or SURF versus SIFT for keypoint extraction (in Section 4.4).

To summarize, we believe that the work developed in this dissertation, in tandem with the photo|uniq mobile application, offers a viable solution for the presented problem, while providing a solid server infrastructure able to be integrated within external client applications that are capable of calling and handling web service requests.

5.2 Future Work

As future work, an important step would be the expansion of the server infrastructure to handle user authorization and authentication. In its current state, the system's data is not associated with a specific user, meaning that if it were a live system, every user would have access to every photo. While the system's data model attributes user ownership to albums, this is not yet implemented in the servers.

Another step would be integration with external applications. During development, the system was tested with the photo|uniq app, which managed to successfully access the server infrastructure's services. However, it was not possible to fully test the system after the server infrastructure was complete, which means that the entire photo|uniq system would benefit from a set of exhaustive tests that cover all functionalities.

Additionally, it would be interesting to develop a web or desktop version of a client application for the server infrastructure, which would extend the possible user base of the system.

Finally, another step would be using an external storage service in place of the storage database currently in the system. In theory, the system's architecture was developed in such a way that an external storage service could be put in place of the storage database, but some work would still be required to make sure the necessary connections are in place.

Bibliography

[1] *Welcome | Jinja2 (The Python Template Engine)*. 2014. URL: http://jinja.pocoo.org/ (visited on 02/18/2019).

[2] A. Russell. *A Year Without a Byte | code.flickr.com*. en. URL: http://code.flickr.net/2017/01/05/a-year-without-a-byte/ (visited on 10/13/2019).

[3] *Django Documentation*. 2018. URL: https://media.readthedocs.org/pdf/django/latest/django.pdf (visited on 02/18/2019).

[4] I. Engineering. *What Powers Instagram: Hundreds of Instances, Dozens of Technologies*. Instagram Engineering. 2011. URL: https://instagram-engineering.com/what-powers-instagram-hundreds-of-instances-dozens-of-technologies-adf2e22da2ad (visited on 02/18/2019).

[5] *TurboGears Documentation*. 2018. URL: https://media.readthedocs.org/pdf/turbogears/next/turbogears.pdf (visited on 02/18/2019).

[6] G E. Krasner and S. Pope. “A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80.” In: *Journal of Object-oriented Programming - JOOP* 1 (Jan. 1998).

[7] *Flask Documentation*. 2017. URL: https://media.readthedocs.org/pdf/flask/latest/flask.pdf (visited on 02/18/2019).

[8] *OAuth 2.0 — OAuth*. URL: https://oauth.net/2/ (visited on 10/11/2019).

[9] *OAuth 2 Simplified*. URL: https://aaronparecki.com/oauth-2-simplified (visited on 10/11/2019).

[10] *OAuth 2.0 Password Grant Type*. URL: https://oauth.net/2/grant-types/password/ (visited on 10/11/2019).

[11] *OAuth 2.0 Bearer Token Usage*. URL: https://oauth.net/2/bearer-tokens/ (visited on 10/11/2019).

[12] R. K. Mantiuk, A. Tomaszewska, and R. Mantiuk. “Comparison of Four Subjective Methods for Image Quality Assessment.” en. In: *Computer Graphics Forum* 31.8 (Dec. 2012), pp. 2478–2491. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2012.03188.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2012.03188.x (visited on 09/22/2019).

[13] J. P. Hornak. *Encyclopedia of imaging science and technology. Vol. 2 Vol. 2.* English. OCLC: 53225285. New York: J. Wiley, 2002. ISBN: 9780471647386 9780471443391. URL: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=99009 (visited on 09/22/2019).

[14] Z. Wang. "Applications of Objective Image Quality Assessment Methods [Applications Corner]." In: *IEEE Signal Processing Magazine* 28.6 (Nov. 2011), pp. 137–142. ISSN: 1053-5888. DOI: 10.1109/MSP.2011.942295. URL: http://ieeexplore.ieee.org/document/6021857/ (visited on 09/22/2019).

[15] A. Mittal, A. K. Moorthy, and A. C. Bovik. "No-Reference Image Quality Assessment in the Spatial Domain." In: *IEEE Transactions on Image Processing* 21.12 (Dec. 2012), pp. 4695–4708. ISSN: 1057-7149, 1941-0042. DOI: 10.1109/TIP.2012.2214050. URL: http://ieeexplore.ieee.org/document/6272356/ (visited on 09/22/2019).

[16] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. "Image Quality Assessment: From Error Visibility to Structural Similarity." en. In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861. URL: http://ieeexplore.ieee.org/document/1284395/ (visited on 09/22/2019).

[17] A. Rosebrock. *How-To: Python Compare Two Images.* en-US. Sept. 2014. URL: https://www.pyimagesearch.com/2014/09/15/python-compare-two-images/ (visited on 09/22/2019).

[18] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints." en. In: *International Journal of Computer Vision* 60.2 (Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: http://link.springer.com/10.1023/B:VISI.0000029664.99615.94 (visited on 09/22/2019).

[19] H. Bay, T. Tuytelaars, and L. Van Gool. "SURF: Speeded Up Robust Features." In: *Computer Vision – ECCV 2006.* Ed. by A. Leonardis, H. Bischof, and A. Pinz. Vol. 3951. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 9783540338321 9783540338338. DOI: 10.1007/11744023_32. URL: http://link.springer.com/10.1007/11744023_32 (visited on 09/22/2019).

[20] *Feature Matching + Homography to find Objects — OpenCV-Python Tutorials 1 documentation.* URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html (visited on 09/22/2019).

ANNEX I

Annex 1 - Database Creation Scripts

This annex contains the creation scripts for the system's databases. The system relies on several other tables due to framework requirements, which were hidden for readability purposes.

Listing I.1: MetadataDB table creation script.

```
--
-- Table structure for table `HOMOGRAPHY`
--

DROP TABLE IF EXISTS `HOMOGRAPHY`;
CREATE TABLE `HOMOGRAPHY` (
  `PHOTO1_ID` int(11) NOT NULL,
  `PHOTO2_ID` int(11) NOT NULL,
  `HOMOGRAPHY` longtext,
  PRIMARY KEY (`PHOTO1_ID`,`PHOTO2_ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Table structure for table `KEYPOINTS`
--

DROP TABLE IF EXISTS `KEYPOINTS`;
CREATE TABLE `KEYPOINTS` (
  `ID` int(11) NOT NULL,
  `PHOTO_ID` int(11) DEFAULT NULL,
  `KP` longtext,
  `DES` longtext,
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Listing I.2: StorageDB table creation script.

```
--
-- Table structure for table `ALBUM`
--

DROP TABLE IF EXISTS `ALBUM`;
CREATE TABLE `ALBUM` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `OWNER_ID` int(11) DEFAULT NULL,
  `NAME` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=latin1;

--
-- Table structure for table `GROUP`
--

DROP TABLE IF EXISTS `GROUP`;
CREATE TABLE `GROUP` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `CREATED_ON` datetime DEFAULT NULL,
  `UPDATED_ON` datetime DEFAULT NULL,
  `ALBUM_ID` int(11) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `fk_GROUP_1_idx` (`ALBUM_ID`),
  CONSTRAINT `fk_GROUP_1` FOREIGN KEY (`ALBUM_ID`) REFERENCES `ALBUM` (`ID`) ON DELETE NO
     ↪  ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=90 DEFAULT CHARSET=latin1;

--
-- Table structure for table `PHOTO`
--

DROP TABLE IF EXISTS `PHOTO`;
CREATE TABLE `PHOTO` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `PHOTO` longtext,
  `CREATED_ON` date DEFAULT NULL,
  `UPDATED_ON` date DEFAULT NULL,
  `ALBUM_ID` int(11) DEFAULT NULL,
  `NAME` varchar(300) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `fk_PHOTO_1_idx` (`ALBUM_ID`),
  CONSTRAINT `fk_PHOTO_1` FOREIGN KEY (`ALBUM_ID`) REFERENCES `ALBUM` (`ID`) ON DELETE NO
     ↪  ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB AUTO_INCREMENT=40 DEFAULT CHARSET=latin1;

--
-- Table structure for table `REL_PHOTO_GROUP`
--
```

```

DROP TABLE IF EXISTS `REL_PHOTO_GROUP`;
CREATE TABLE `REL_PHOTO_GROUP` (
  `PHOTO_ID` int(11) NOT NULL,
  `GROUP_ID` int(11) NOT NULL,
  PRIMARY KEY (`PHOTO_ID`,`GROUP_ID`),
  KEY `fk_REL_PHOTO_GROUP_2_idx` (`GROUP_ID`),
  CONSTRAINT `fk_REL_PHOTO_GROUP_1` FOREIGN KEY (`PHOTO_ID`) REFERENCES `PHOTO` (`ID`) ON
      ↪ DELETE NO ACTION ON UPDATE NO ACTION,
  CONSTRAINT `fk_REL_PHOTO_GROUP_2` FOREIGN KEY (`GROUP_ID`) REFERENCES `GROUP` (`ID`) ON
      ↪ DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```