ABSTRACT

Facetag, The Image Managing Service for the Leukocoria Detection Project

Vaclav Cibur, M.S.

Mentor: Greg Hamerly, Ph.D.

This paper describes my work in improving the image managing application for the leukocoria detection project. This application allows volunteers to upload photos of their children affected by retinoblastoma. Researchers then use it to tag eye positions in those images and review them. The resulting eye crops are then exported for use by other applications. I have developed a new version of the application using the Django web application framework and the PostgreSQL database engine. I have also designed an elaborate deployment scheme using the Docker framework and deployed the application using it.

The output of my work will be used to improve the leukocoria detection algorithm. Leukocoria is a bright white reflection occurring in eyes of children suffering from retinoblastoma, a type of cancer, when subject to a direct light source like a camera with flash. This white reflection is easily detectable by a computer algorithm, whose accuracy can be improved by analyzing more images.

Facetag, The Image Managing Service for the Leukocoria Detection Project

by

Vaclav Cibur, B.S.

A Project

Approved by the Department of Computer Science

_____

Gregory D. Speegle, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Project Committee

_____

Greg Hamerly, Ph.D., Chairperson

_____

Gregory D. Speegle, Ph.D.

_____

Bryan F. Shaw, Ph.D.

Accepted by the Graduate School
May 2016

_____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my great appreciation to my advisor and mentor Dr. Greg Hamerly, who has always stood by my side, guided me to significantly improve the application and helped me get through the whole time I spent here. My sincere thanks also go to Dr. Brian F. Shaw and Dr. Gregory Speegle for being willing to be members of my committee. I would also like to thank all my friends and family who stood by my side every day motivating me to work hard. Last but not least, I am very grateful to everyone who participated in the usability study and provided me with very important feedback.

CHAPTER ONE

Introduction

The main goal of this project is to introduce Facetag, a new and improved version of the image managing service for the automatic leukocoria detection project. It will have all the necessary features allowing users to utilize natural photographs obtained from volunteers to create training and testing data for the Convolutional Neural Network (CNN) and thus improving the accuracy and precision of the leukocoria detection algorithm.

Retinoblastoma is a rare type of cancer that begins in the retina of a human eye. It almost exclusively affects young children and is the most common malignant cancer of the eye in children. Although most patients in developed countries survive this cancer, they may lose their vision or need to have the eye removed (Rivas-Perea et al. 2014).

Leukocoria, also known as white pupillary reflex is an abnormal white reflection from the retina of the eye and is the main symptom of retinoblastoma. On photographs taken using a flash, instead of the familiar red-eye effect leukocoria can cause a bright white reflection in an affected eye (DEMIRCI et al. 2001). This white reflection is easily detectable by a computer algorithm and is the essential feature in the leukocoria detection project.

CRADLE is a mobile application which can either scan through your image gallery, or live scan the camera feed for signs of leukocoria and therefore possible retinoblastoma or other diseases and it is the main part of the leukocoria detection project. It uses a special algorithm to detect faces and eyes and each eye image is then passed into a CNN to determine the probability of a given eye showing leukocoria. If

the probability is higher than a set threshold, the user of the application gets notified and can follow up with a medical professional.

The custom CNN used in CRADLE is complicated and requires a precise set of parameters to guarantee a certain level of accuracy and precision in detecting leukocoria. In order to find the ideal parameters we have to analyze a lot of eye images both with and without signs of leukocoria. We were lucky enough to get tens of thousands of photos from parents using CRADLE and other volunteers, who offered to have their images used for research purposes.

Getting the images however, is only the beginning. What follows is a long and intricate process. Some of the images could be too blurry, or have explicit content and need to be filtered. After that we have to manually find positions of eyes on each photograph and every eye needs to be labelled leukocoric or healthy. The output of this process are cropped images of eyes with appropriate labels used to train and test the CNN. However, there is a large database of images that need to be processed this way with no application to make this process easy and intuitive. My project focuses on maintaining the image collection and on improving and streamlining the described workflow to make it as easy and as fast as possible to put the images we have to good use.

CHAPTER TWO

Previous work

Throughout the life of the leukocoria detection project, there was a need to develop a web application that would allow researchers from Baylor University to upload images, tag eye positions in them, diagnose those eye positions and export training and testing data for the CNN in an efficient manner.

There was never an application that would efficiently combine all of these features, but there were two applications that each tackled a subset of the required features. I will briefly describe what those two applications look like, what they do and what are their flaws.

2.1 Image Tagging

The Image Tagging application was the first try to help organize the eye tagging procedure. It was a PHP application with a MySQL database and only offered basic functionality.

2.1.1 Database

As you can see in Figure 2.1, the database didn't really capture much information and is not normalized. The `tags_log` table doesn't have a primary key and associations between tables do not use foreign key constraints. Even though it seems that the model has some additional information about the images and tags, e.g. `angle` and `orientation` attributes in the `tags` and `tags_log` tables, those attributes had `null` values in all rows. So the only real information stored are the coordinates and label of the eye tags and a path to the corresponding image.

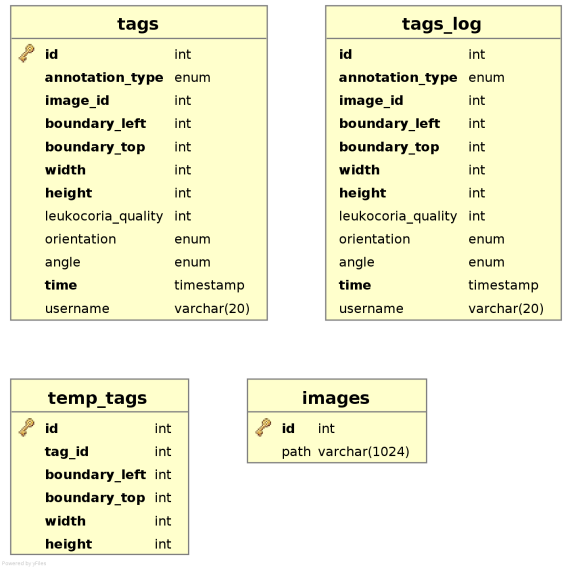

Figure 2.1. Database model of the Image Tagging application

2.1.2 Conclusion

Although the Image Tagging application was definitely a step in the right direction it does not provide nearly enough functionality to support the entire workflow and was therefore enhanced into a new version.

2.2 New Leuko

New Leuko is a new and improved version of the Image Tagging application. It still uses PHP with a MySQL database, but is more robust and offers enhanced functionality.

2.2.1 Database

The New Leuko database has definitely improved compared to its Image Tagging counterpart. Among the main differences, we notice that there is a significant amount of image metadata being stored. Specifically we are storing data extracted from the image files' EXIF headers. Those are important, because they can help us categorize leukocoria cases, e.g. by location, by time. Flash information can help us better understand under what circumstances the white eye reflection occurs. We

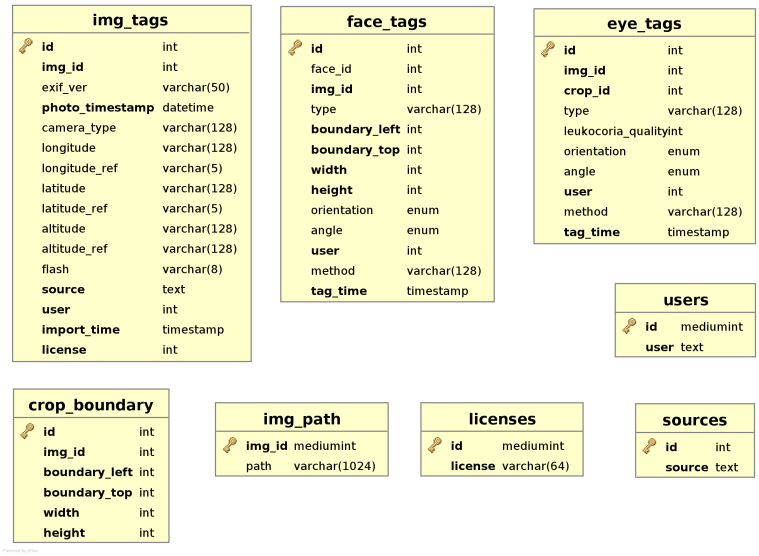

Figure 2.2. Database model of the New Leuko application

also separate images based on their source, which is the name of the parent of the leukocoria affected child in most cases and their license, which helps us determine where a given picture can be used.

The database model is a lot more normalized, but still isn't perfect. Data from the `crop_boundary` table is duplicated in the `face_tags` table and could easily be replaced with a foreign key relationship. Relationships between tables are still not governed under foreign key constraints. One relationship in particular stands out, the `img_tags` table has a `source` attribute of type `text`, but it contains ids from the `source` table, so it should definitely be a foreign key relationship of type `int`. We can still see `orientation` and `angle` attributes, but they are still not supported in the web application itself, in other words, they are never set nor used.

2.2.2 User Interface

The user interface allows us to move forward and backwards through all images simply based on their database ids and tag and label eye positions. There is still a lot left to be desired.

2.2.3 Conclusion

We can clearly see progress from the Image Tagging application, but the functionality is still limited to basic eye tagging and labelling and we are still missing important features like a user friendly image upload, a user role hierarchy and management of existing licenses and sources. These tasks were done using complicated scripts and by direct database access, which very much limits the usability to a small group of people.

CHAPTER THREE

Requirements Analysis

This chapter focuses on analyzing the requirements and limitations of this application. The functionality of this application is mostly based on the combination of the two legacy applications with information gathered from years of working with them and the underlying data.

3.1 Project Goals

Analyze the current version of the image managing service for the automatic leukocoria detection project. Analyze and describe its current and future requirements. Design and implement a new and improved version of the application using state of the art approaches in web application development.

The web application should in particular allow users to tag eye positions in pictures and allow multiple users to classify possible leukocoric eyes. It should also allow certain users to review eye positions and leukocoria classifications. There should be enhanced user management distinguishing user roles. Other important features include incorporating new images with source attributions and tagging images for training purposes.

3.2 Requirements Specification

This section goes in depth to describe any and all requirements and limitations of the application. It includes all the functionalities the finished project should have and how they should look like.

3.2.1 User Roles

The application will be accessed by users having the following roles, shown in the order of the number of features they have access to.

3.2.1.1 Anonymous user. An anonymous user is a user that does not have a user account or has not logged into the system. She can access the home page of Facetag which has some basic information about leukocoria and what the website is for.

This role should also allow parents of kids with leukocoria or anybody else to upload photographs and provide information about those photographs, e.g. name, age, sex and any known related diseases of the leukocoria affected child and also some information about the contributor herself in case she may want to be contacted in the future. There are some limitations on the size and quantity of photos being uploaded at once. Images obtained this way will be subject to review before being moved further in the workflow.

3.2.1.2 Uploader. Any role beginning with this one requires a user to have an account. And to be able to access all her features, the user has to log in to the website.

A user with this role will have access to a more advanced upload page then the anonymous user. This page will have all the features of the anonymous upload page without limitations and it will also allow more control over the attributes. An uploader can select the source of a given set of images from already existing ones and also add new sources. Each set of images also has a specific license under which the photos are shared and a priority setting, which will affect how fast the image will get through the entire workflow.

3.2.1.3 Exporter. The exporter role allows anyone with it to access images that have been approved for the use in neural network training and testing. More specifically it enables you to see all approved eye tags with labels of approved images and additional information associated with each image. This role is intended to be

used to export training and testing data that has successfully made it through the entire workflow.

3.2.1.4 Tagger. A user with the tagger role will be able to access work queues of images based on certain criteria, e.g. priority. She can navigate through those queues back and forth and access all images in them. On a specific image page she can see any relevant image information, draw tags around the eyes of the people in that photo and assign a label (leukocoric, healthy). She can move, resize or delete any of the tags she previously drew or change their labels. However, she doesn't have access to other users' tags and labels.

3.2.1.5 Reviewer. Reviewers can access all functions of any of the previous roles and also have access to a dedicated reviewing part. It also consists of many work queues based on different properties, similarly to the tagging queues mentioned above. They too can see any relevant image information and do any of the things they can do on the tagging page. But in contrast to the tagging page, they can see and change other users' tags. They can also either accept or reject any of the tags of a given photo, thus making the tags either be ready for export, or just stay unused.

3.2.1.6 Administrator. Administrators have access to everything any of the other roles do. On top of that they can manage the sources, licenses and diseases that are being associated with images, they are in charge of creating user accounts, assigning them roles and assigning permissions to the roles. There will also be an activity log accessible by administrators, which will show the order of all timestamped events and their associated information.

3.2.2 Functional Requirements

This section briefly covers some of the most important functions the end product should facilitate.

3.2.2.1 Image rotation. Most cameras these days have an orientation detection feature built in telling the camera whether a photo was taken in portrait or landscape mode. However, cameras do not rotate the image data itself before saving, that would be too costly. They instead save the image data in the same way every time and add the orientation information into a special header of the final image file. They use what is called an Exchangeable image file format (EXIF) to store the image data and additional information. The rotation data gets written into the EXIF header of each file along with other useful information, e.g. location, time, flash etc.

For our purposes however we want eye images to be rotated upright, because that leads to the highest accuracy and precision. That should ideally be done not by using any special tags, but in the image data itself. Therefore, all photographs that get uploaded into the system should be rotated according to the EXIF attribute and have their EXIF rotation attribute removed to mitigate any confusion.

Although, this method should theoretically guarantee that all images are oriented correctly, sometimes cameras get the orientation attribute wrong and the photograph has to be corrected manually. Hence, another feature of the system should allow users to rotate newly uploaded images to fix that problem.

3.2.2.2 Eye position tagging. Eye position tagging is the main task this application should facilitate, therefore it should be fast and intuitive. Users should be able to draw tags on images that haven't been tagged yet and associate a label (healthy, leukocoric) with each tag. It should be as easy as possible to draw very precise tags even on giant images, therefore there should definitely be a zoom function that would allow that.

Users can see and make changes to the tags they previously made until they are reviewed. However, they cannot see or change other users' tags. This allows us to let multiple people draw tags on the same eye in an unbiased way from which we can later pick the best ones.

3.2.2.3 Reviewing eye positions. After all the eyes have been tagged on an image, all of the tags should be reviewed by more experienced users to eliminate the chance of any imperfect tags being used further. The review function will be very similar to the tag function, it should also allow drawing new tags, changing existing tags and modifying labels. But unlike the tag function which only shows the current user's tags, the review function shows all tags drawn on a photograph by any user.

Each of those tags should then be subject to review and either accepted or rejected. If any of the tags get accepted, that image and all its accepted tags are then used further to create training and testing data for the neural network.

3.2.3 Non Functional Requirements

Apart from the functions the application should facilitate, there are also many non functional requirements, which I will describe in this section.

(1) Use the python programming language for the implementation of the application. Also choose an appropriate web framework.

(2) Design a reusable and easy to use model for deploying the application. We should have two versions of the application, the first one will be a stable production version and the second one will be a development version used to test new features.

3.2.4 Full Workflow Example

I would like to dedicate this section to summarize and describe the entire workflow each photograph should go through in this application.

(1) A photograph gets uploaded to the system by either an anonymous uploader or a logged in user. It gets rotated automatically.

(2) If the photo was uploaded anonymously, it has to be approved before it moves further.

(3) The image is now accessible in many different queues ready to be tagged by multiple users.

(4) Many different users, as they move through their tagging queues, will come upon this image. Each of them will tag positions of every human eye and assign a label to each tag. The label can either be healthy or leukocoric.

(5) After a certain time, a reviewer will login to the system and start going through her review queues, until she gets to our photo. She will review all tags on that image by either accepting or rejecting each one.

(6) If all of the tags get reviewed and any of them get accepted, the image is categorized as ready to be exported. At this point, the tags on that photo should stay unchanged.

(7) After that, the image and all its associated tag and label data can be exported and used in other applications. Most likely it will be used to create training and testing datasets for the CNN.

3.3 Domain Model

This section will summarize all the classes of the domain model and the relationships between them. This model is one of the results of the requirements analysis and serves as the basis for creating a database schema. You can see its graphical representation in Figure 3.1.

3.3.1 User

This class represents a user of this application. Each user has a username and password pair which she uses to login into the system. Each user also has roles which have different permissions. The user authentication system is a lot more complex and will be discussed more in depth later.

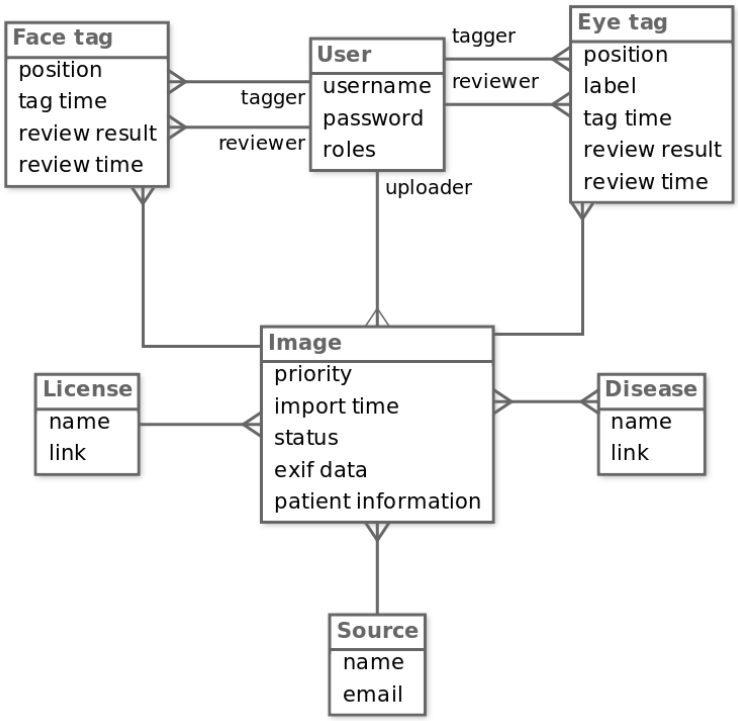

Figure 3.1. Domain model

3.3.2 Source

Source stores information about where a given image comes from. It should have a name and an email address of the person who contributed that image.

3.3.3 License

The license class describes the actual license under which a given image was obtained. We need to keep track of that in order to know when and where we can use a given photo. Most licenses have websites that describe them in depth, that is why we have a link attribute in addition to the name attribute.

3.3.4 Disease

As is obvious from the name, this class represents a disease which can be found on an image. We care about its name and a link to a website which has information about that disease.

3.3.5 Image

This class represents one image. It needs to store a path to the image or the image data itself. Other parameters are the image's priority and time of import.

The status parameter is really important and it helps us keep track of which part of its life cycle the image is in. We distinguish three states:

- The RED state represents a newly uploaded image by an anonymous user that needs to be approved.
- Images with the YELLOW state are in the process of being tagged and reviewed.
- GREEN state images represent those that have been fully tagged and reviewed and at least one tag has been approved. This also means that all approved tags on those images can now be used for export.

We also store some additional EXIF data extracted from the image file, e.g. photo time, location, flash, etc. and basic information about the person in the image. That can be the disease they are suffering from, their age, sex and name. Each image is associated with a particular user who uploaded it, diseases that occur in it, the license it was obtained under and the source.

3.3.6 Eye Tag

An eye tag represents the position of one eye in a given photograph. Each tag also has a label which represents whether the eye is healthy or leukocoric. Another important attribute is the time when it was modified.

We also store review information with each tag. If it hasn't been reviewed yet, those attributes will be empty. But after it has been reviewed, the review result and time are kept. The review result can be accepted or rejected.

Each eye tag is associated with its tagger, that is the user who created it. Also if it is reviewed, it is associated with the user who reviewed it.

3.3.7 Face Tag

A face tag represents the position and location of a face in a given image, it contains information about the time it was created, reviewed and the result of that review.

CHAPTER FOUR

Design And Implementation

Based on the gathered requirements and the domain model I designed the application itself. This chapter contains a detailed description of the final design of the application and all of its components.

Firstly I will talk about the technologies we used to develop and run Facetag. Later on, I will describe how Facetag consists of multiple layers providing services for each other and I will define each layer and how it interacts with the others.

4.1 Used Technologies

The code base of this application is written mostly in the Python programming language based on the requirements. However to build the application I have used several frameworks and libraries to make the development easier and to take advantage of any existing and proven code.

4.1.1 Python Version

There are currently two actively developed version branches of Python available. Although version 2 is receiving less and less updates recently, there is still a broader spectrum of supported libraries. However I still decided to go for the more modern version 3, because it is being worked on currently and most probably will be in the future too. I also made sure that all of the libraries and frameworks this application was going to be using were supported in Python 3.

4.1.2 Web Framework

The next important step was to choose a web framework the application is going to run on. After long and thorough research and advisement, I decided to choose the Django web framework (Bendoraitis 2014), because it has the largest

community behind it and also a very detailed and comprehensive documentation. I also found that it is being used by several well known websites including Pinterest, Instagram, Mozilla, Bitbucket, etc.

4.2 Database

Selecting an appropriate database engine was an important decision determined by several factors. This subsection will talk in depth about what the main deciding factors were.

4.2.1 Image data storage

The previous versions of the image data managing application stored the image data directly in the file system. However, during the development of this application, a need to unify the images with other data we store arose. Therefore, after careful consideration and discussing this with Dr. Gregory Speegle, who is the database expert of our department, we decided to store raw image data in the database along with all the other information.

This way, all of the data this application works with is stored within a single database and is not dependent on any other source. It also makes it very easy to do backups of the data, where we only need to export one database into a file.

4.2.2 Database engine

Both legacy databases used MySQL as their database engines, but it caused some problems, mainly because the dialect that was used doesn't support foreign key constraints. However, I am aware of the importance of using foreign keys and I think it greatly improves maintainability of the database, therefore I wanted to steer away from MySQL.

Also, when we take into consideration that we want to store raw image data in the database, the choice of the database engine is fairly limited. We decided to use the PostgreSQL database engine mainly because it handles raw data storage really

well, supports foreign keys and it is also one of the best documented database engines available.

4.2.3 Database schema

The database schema is almost identical to the domain model mentioned above, however it contains all the attributes and their data types. Figure 4.1 shows the main part of the entity-relationship diagram without additional tables responsible for user authentication. Figure 4.2 shows tables related to user authentication missing in the previous figure. The authentication system and its related tables are provided by the Django framework (Elman and Lavin 2014).

The following text talks about important tables and attributes, or the ones not mentioned in the domain model in Figure 3.1 above.

4.2.3.1 Image. The `image` table is essential for this application and it stores all data we have associated with a particular image. It contains a special `data` attribute of type `bytea`, which is where the raw image data itself goes, as already discussed above. It also has some attributes that help us present the photographs in a certain order: `import_time` and `priority`. Attributes `exif_version`, `photo_time`, `camera_type`, `flash` and location information (`altitude`, `longitude` and `latitude`) help us better categorize images and understand when, where and under what circumstances leukocoria occurs.

Information gathered about the affected child from her parents is stored in `child_of_interest_age`, `child_of_interest_name`, `child_of_interest_sex` and `comment`. Those can guide us to understand how different diseases have different refractory effects on eyes and also get some statistics about leukocoria and its affected population.

4.2.3.2 Eye tag. Each row in the `eye_tag` table represents a tagged eye. To precisely describe the position of one eye, we need four values: `top` and `left` tell

Figure 4.1. Database schema

us how far the eye starts from the top and left edge of the photo respectively, while `width` and `height` give us the dimensions of the eye tag in relation to the original photo size. We also keep track of the assigned label, to distinguish whether the eye is leukocoric or healthy.

After a tag is reviewed, we store relevant review information: `review_date`, `review_comment` and most importantly `review_result`, which tells us whether the tag is good and usable for export. Each tag eventually ends up with two associated

users, one being the user who created the tag and the other one the user who reviewed it.

4.2.3.3 Face tag. The `face_tag` table is almost identical to the `eye_tag` table, with the exception of not having the `label` attribute and the obvious difference of representing face positions.

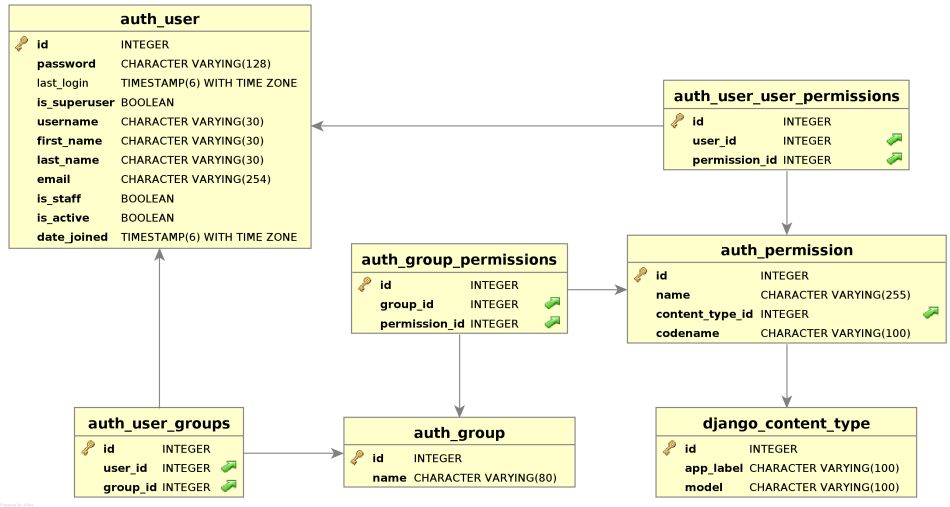

Figure 4.2. Authentication and authorization part of the database schema

4.2.3.4 Authorization and authentication tables. The `auth_user` table stores information about all Facetag users. Most importantly, each user has a `username` and a `password` to be able to login. Passwords are stored in a secure hashed form.

Each user can have specific permissions assigned directly by using the `auth_user_user_permissions` table, but this is not how the authorization system is used in Facetag. We can also assign permissions to users through user groups. That is why the `auth_user` table has a relationship with the `auth_group` table (by means of the `auth_user_groups` table). User groups map directly to the user roles mentioned in previous chapters. And similarly to users, they can be assigned permissions by

utilizing the `auth_group_permissions` table and that is the main way of assigning permissions in Facetag.

4.3 Model Layer & ORM

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.(Django 2016a)

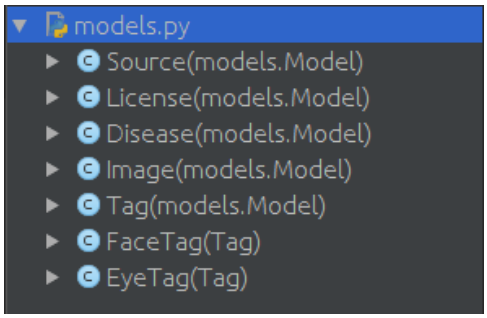

Figure 4.3. Model classes

Figure 4.3 contains all model classes of Facetag. Almost all of them map to their corresponding database tables, with a few exceptions. The `Tag` class is only a logical parent of `FaceTag` and `EyeTag` classes. It contains common attributes shared between them to minimize code duplication and it doesn't map to a database table. `FaceTag` and `EyeTag` classes are the ones mapped to database tables and each can specify its own additional attributes.

All model classes inherit their functionality from the `django.db.models.Model` class, which enables Django to automatically create the database schema and also expose a database-access API. This API allows us to retrieve information from the database in a simple and comprehensive manner similar to writing SQL queries, but much easier. Everything is done in Python and the resulting objects are actual instances of model classes mapped directly to the database. Retrieved objects can be

changed in many ways and if they are saved, all the changes are propagated to the database. This allows us to access and manipulate all data in the database in a clean object oriented way and without having to type any SQL queries. We can still do that if we want to, but it is necessary only in some extreme cases.

4.4 Views

This application has many different views. A view is a dedicated webpage which shows some specific content and allows specific actions with that content. In this section I will describe some of the most important views of this application, what they are used for and how can a user interact with them. A view typically uses model classes to retrieve some data from the database, does some operations on that data and serves it to the user in a way defined by a template.

4.4.1 Upload Views

There are two distinguished upload views as already mentioned above. They allow you to upload sets of images with additional information. During upload, each image is automatically rotated based on its EXIF (Exchangeable image file format) rotation attribute to be upright and the EXIF attribute is removed. We also extract some other useful EXIF information and save it into the database with each photo.

4.4.2 Check Images View

After a photograph gets uploaded into our system by an anonymous user, its status is changed to red, signifying it needs to be verified before it is allowed into the tag/review workflow. This verification step is facilitated by the check images view. It shows all images with the red status one by one. After an image is shown, the user has two options. She can approve it, which changes its status to yellow, or she can delete the image, which irreversibly removes the image from the database.

4.4.3 Tag View

The tag view shows a single photograph based on the current queue. Queues will be discussed later. The task for the user is to correctly tag and label each eye and move onto the next image. However, a user can also go back to images she already tagged, in which case all already existing tags are shown and can be modified as the user pleases. Every tag can be moved, resized and deleted. Any changes the user makes to the tags can either be saved to the database, or reverted to the original state.

To make it easier for users to draw tags around small eyes on a large image this view has a zoom function which enables zooming in and out of the entire image. There is also a small window that constantly shows a zoomed in version of the image around the position of the user's cursor if hovering over the image itself, or a zoomed in version of a tag, if hovering over one.

If no other users have tagged the current image yet, users also have an option to rotate the image by 90 degree intervals to make it upright, in case that the auto-rotation feature failed for some reason. It is important to note that doing this deletes all tags associated with that photo. That's why if there are some already existing tags created by other users on the current photo, the rotation feature is disabled, because it would delete their tags.

4.4.4 Review View

The review view builds on top of the tag view, by adding more functionality. This time, tags from all users of the current image show up and the task of the reviewer is to assign a review result to each tag. There are two possible review results: accepted and rejected. The reviewer makes sure that each tag is done the right way around an eye and that the assigned label corresponds to the condition of the eye. Finally, when a user reviews all tags of an image and at least one of them was accepted, that image's status changes to green marking it as eligible for export.

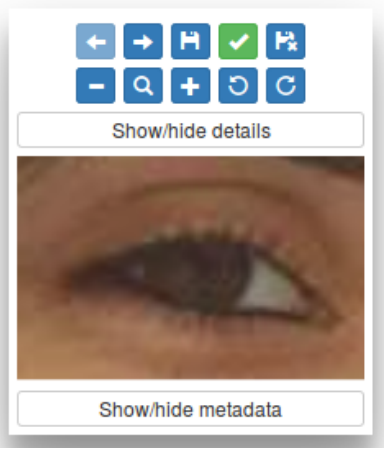

Figure 4.4. Tool window on the review page

A reviewer can modify or delete any of the tags the same way as in tag view and can also create new tags. The rotation feature from tag view is kept, however it gives reviewers more power, because they can rotate any image, while keeping in mind that all of its tags are going to get removed. The review page also has a special shortcut function, which marks all tags as accepted, saves the reviews and redirects to the following image, to accelerate the process for reviewers.

Figure 4.4 shows the main tool window used to control the review page. The buttons of the first row from left to right are: previous image, next image, save changes, accept all and save and revert changes. Second row allows users to zoom out, reset zoom, zoom in and rotate counter-clockwise and clockwise. The Show/hide details button allows users to minimize the tool window and only show the top buttons. Under it is the already mentioned zoom window, which zooms in on tags or the image. And finally the Show/hide metadata button shows additional information we have about the image.

4.4.5 Tag And Review Queues

Both the tag and review views are built on top of a system of queues. This means that each time you want to tag or review, you have to select one of many

queues available. After you select one, it shows you the first image of that queue and you can start working. When you are done with the first image, you click the next image button, which redirects you to the next image within that queue. And you can keep going as long as there are images in the selected queue.

Each queue represents a certain order of a subset of all the images. For example, we have the Tag by priority queue, which shows images with the highest priority first. Another example is the Recently reviewed by you queue, which would show a reviewer the images they most recently reviewed first. Although it is clear from those two examples that the review and tag workflows have different queues, there are some that are shared by both. Each queue is mapped onto a database query written using Django's ORM. This makes the queue system maintainable as it is fairly straightforward to create new queues.

4.4.6 Activity Log

The activity log is a useful new feature, that lets administrators see all the actions of all users in one place. Many of the actions users can do on Facetag are timestamped. For example each time a new tag is created or updated, each time an image gets uploaded and each time a tag is reviewed. All of those actions have a time associated with them and the activity log takes advantage of that by showing all actions in a chronological order beginning with the newest.

Each row in the activity log represents one of the actions mentioned above and contains all relevant information. For example when it occured, which user triggered it and the image in question. Each row has a link which takes you to the review page of the image it affected. Also each username and image mentioned in these actions is a link which filters only the events associated with it.

By default, the activity log shows all actions that happened in the system, but you can also filter the events by certain criteria. You can choose to only see actions

done by a given user, or ones affecting a specific image, which can be very helpful in tracking how long it takes for an image to get through the entire workflow.

4.4.7 Other Management Views

There are also several administrator only views which allow viewing, creating, updating and deleting of certain data items. Those are designed to make it easier to manage data in different tables. You can manage sources, licenses, diseases, users, user groups and permissions this way. For each of those tables, you can display a list of the items, each of which has links to edit and delete individual rows and also create new ones. This approach is a great improvement over the old application, which did not have any functions allowing the user to do this except for direct database access.

4.4.8 Profile View

Each user also has a profile view, which allows her to see her assigned groups and change her personal information.

4.5 Forms

Django forms provide a way for the application to get input from users, for example by entering text, selecting options, etc. They are fully defined in Python in one place including the validation of all fields and the form as a whole. This provides a very good separation of concerns, because in order to slightly change the behavior of a form, we only have to change it in one place and it applies on all layers of the application wherever the form is used. In Facetag, forms are used in the upload view and also in management views for creating and updating objects.

Forms are validated by both the user's browser before submitting and the server after submitting. The validation, in both places, is done based on the same rules specified in the form to prevent any misalignment.

4.6 Templates

Templates are Django's way for dynamically generating HTML code. A template consists of static parts of the HTML as well as some special syntax describing how dynamic content will be inserted (Django 2016b).

For example let's say we have a page with a heading that is always going to be the same, but under it is a list of items, which will be dynamically obtained from the database. This is exactly what the Django template language facilitates.

Each view has an associated template, which it populates with dynamic data. The template processes all the data it was given and generates an HTML document, which is then displayed to the user.

4.7 URL Dispatcher

The URL dispatcher is an important part of Facetag. It contains all possible URLs provided by Facetag and maps them to specific views. Whenever a user sends a request for a specific URL to Facetag, it is first compared to all rules of the dispatcher and if there is a match, the control is given to the appropriate view.

Some URLs may contain dynamic elements, for example an identification number of a specific object. Luckily, the URL dispatcher allows us to have rules which support that by specifying regular expressions for each dynamic part of the URL. Those dynamic values are then passed to the corresponding view, which uses them to load specific dynamic data from the database.

4.8 Application Architecture Summary

Figure 4.5 (Diksha 2016) shows how all the above described layers interact.

(1) User sends a request to one of Facetag's URLs.

(2) URL Dispatcher finds the matching URL rule and passes control to the corresponding view together with any parameters.

(3) View needs to obtain some dynamic data, so it calls the model.

(4) Model queries the underlying database for the data requested.

(5) Database retrieves the requested data.

(6) Model passes the data back to the view.

(7) View does some calculations on that data and passes it into a corresponding template.

(8) Template renders the HTML code using the data it was given.

(9) Response with the HTML data is sent back to the user.

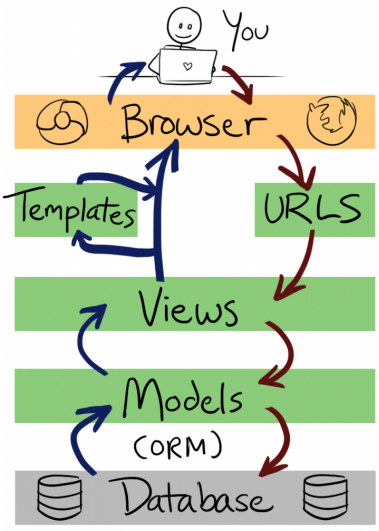

Figure 4.5. Overview of a Django application architecture

CHAPTER FIVE

Deployment

Previous versions of Facetag have been running on internal servers of the Department of Engineering and Computer Science at Baylor University. The infrastructure is already built and has been enhanced for the new Facetag application and my goal is to use this infrastructure to deploy Facetag.

This chapter will describe in depth how Facetag is deployed and the technologies it is using for that.

5.1 Docker

Docker is a project which automates the deployment of applications inside software containers. It provides an additional layer of abstraction and automation of operating-system-level virtualization. It allows independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

5.1.1 Image

A docker image is essentially a snapshot of a Linux operating system, its memory and state.

5.1.2 Container

A container in docker is created when you run a docker image. It is basically a sandbox environment behaving like a running virtual operating system. Many containers can be spawned off a single image and changed in different ways. However, when you remove a container, all changes done within it are lost. You can save the changes done to a container by committing it into an image.

5.1.3 Conclusion

The docker structure allows for a very simple development and testing lifecycle. When new features are developed for Facetag, they need to be tested before used in production. This can be achieved by simply running a new container from the Facetag image and test the new features within it. After everything has been tested thoroughly, we can simply mark the development container as production and remove the old production one.

Also, docker allows us to separate Facetag into multiple containers, each of which is responsible for a single task, therefore achieving a separation of concerns, which is an essential property of any good modern application. Those are the two main reasons that led to using docker as our deployment strategy.

5.2 Docker And Facetag

Facetag is deployed using a structure of docker images and containers. This section is dedicated to the definition and description of that structure. This application comprises of three images. You can find the deployment diagram in Figure 5.1.

5.2.1 The DB Store Container

The DB Store container is a data volume container. It doesn't need to be running, because it doesn't provide any services. It actually provides us with a persistent data volume mounted from the host system. When creating this container, we specify a path to a directory we want to mount, which effectively binds this directory to this container and we can later add mounted volumes from this container into other containers.

5.2.2 The DB Image

The DB Image is a light version of the Ubuntu operating system with PostgreSQL installed as our main database engine. Whenever we spawn a production container out of this image, we mount a volume from the DB Store container, which

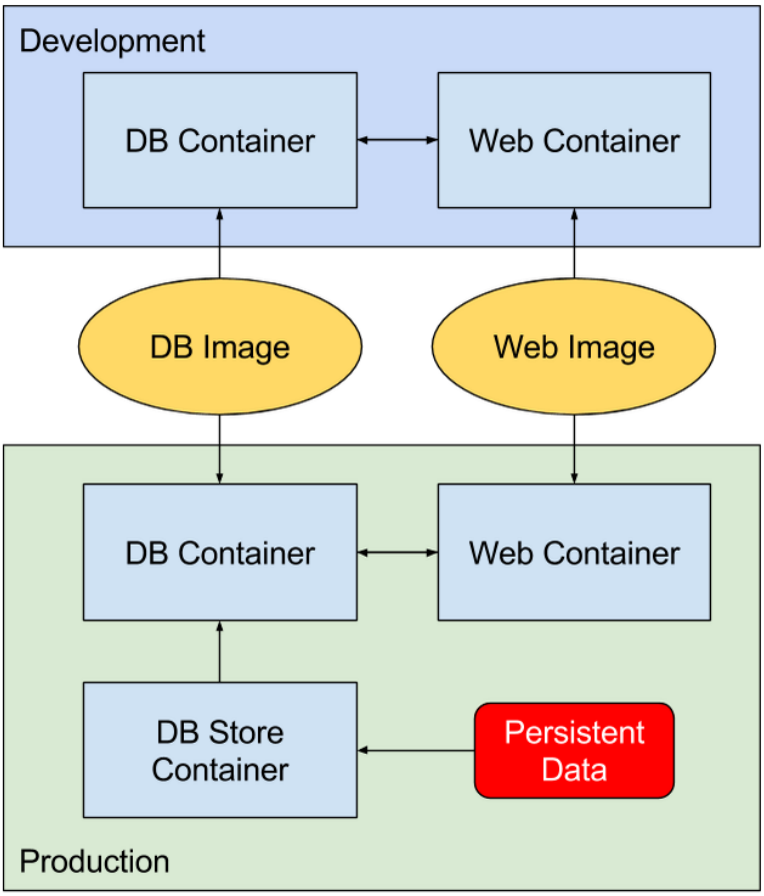

Figure 5.1. Deployment diagram

is then used by the database engine to store data in a persistent manner. Therefore even if we delete this running container, our data stays persisted.

The production container spawned out of this image is solely dedicated to run PostgreSQL and no other processes. This results in a good level of the separation of concerns.

5.2.3 The Web Image

The Web Image is also based on a light version of Ubuntu and it has the Apache2 web server installed. To make our website working, we have to spawn a container out of this image. After we do that, the container starts up the Apache2 web server, which is then ready to process requests from different users.

This container is also linked to the DB container, which allows them to interact with each other. We use this to allow the Web container to communicate with the database. Therefore, whenever a request comes in to the web server, it can do all the necessary communication with the database and produce a response.

5.2.4 Development Containers

Everything described above is how the production works. However, we also need a way of testing new features without affecting our stable production version. It works similarly to production. We have separate containers for the database and the web server, however we don't mount any data from the DB Store image, because that is strictly production data that cannot be affected. That is why the database container of the development version stores all data within it in a non persistent fashion. Therefore, if we removed the development database container, all data would be lost. However, that is not a problem, because it is only intended for development and should never contain any important sensitive data that is not stored elsewhere.

CHAPTER SIX

Usability study

This application's main purpose is to allow its users to complete certain tasks in a fast and efficient manner. That is why it is really important to get a lot of user feedback and make changes in order to allow for an intuitive and fluent experience. We organized a usability study for this application, where we asked for volunteers from the Department of Computer Science to come and help us improve the application.

6.1 Volunteers

We tried to select volunteers of different levels of computer literacy, which can help us better understand the varying approaches people take to complete a certain task an everyone should have a slightly different view on what should be improved and how. We gathered users in pairs for 30 minute testing sessions. Each volunteer had a different set of assigned roles, to better understand different work flows. In total, we had 12 volunteers participate in the usability study divided into 6 groups.

6.2 Test Scenarios

Before each session began, all data got deleted from the database and we populated it with initial data. Initial data consisted of a list of users with assigned roles to allow our volunteers to login.

We provided the first tester with a set of image files in a designated folder and gave her a list of tasks:

(1) Go to the main page of Facetag.

(2) Select the Upload option.

(3) Choose all the provided files.

(4) Fill in your name and email address.

(5) Accept the license and upload the images.

After all the images were uploaded, we asked the other person to login to Facetag as a tagger and do the following tasks:

(1) Select the Check images option.

(2) Go through all of the images.

(3) Approve any relevant images.

(4) Delete any irrelevant images.

When the second person was done, we asked the first tester to login as a tagger and do the following tasks:

(1) Go to the Tag by priority queue.

(2) Go through all of the images.

(3) Create tags around all eyes in each photo.

(4) Assign a proper label to each tag.

After the tagger was done, we asked the second person to login as a reviewer and finish the following tasks:

(1) Go to the Needs review queue.

(2) Go through all of the images.

(3) Accept all tags that are correctly around peoples' eyes and have the correct label.

(4) Reject any tag that is either wrongly labelled, or doesn't contain the whole eye in a reasonable crop.

Whenever both testers were done with their assigned tasks, we asked them to provide any feedback on the process and any suggestions they would have on improving the experience.

6.3 User Feedback

This section covers some of the main remarks users had after working with the application for some time and the ways we addressed them.

(1) One user complained about the username field not being automatically focused on the login screen. We changed the form, so that the username field is now in focus when the page loads and users can start typing their usernames immediately.

(2) Multiple users complained that they weren't sure how to control the application at first and how leukocoric and healthy eyes are supposed to look like. We added a user manual to this paper for users to be able to quickly get up to speed. Also the help page will have a quick guide on how to control the website and examples of leukocoric and healthy eyes and how they should be tagged.

(3) Some volunteers said they would prefer to be able to resize the tags on all edges and corners as opposed to just the bottom right corner in the current version. After careful consideration, we decided not to add this feature, because it would clutter each tag with too many different interactive areas and it would be hard to use them all.

(4) A tester had problems with creating tags on images where a person was standing far from the camera, because the eyes were too small. The tester was then informed about the zoom feature and started using it. We also decided to add information about all of these functions into the help page.

(5) Many users complained that after tagging all eye positions, they had to click twice to save and move to the next image. We added a shortcut button that saves all changes and redirects to the next image.

(6) One volunteer was disappointed that after failing form validation on the upload page, all fields were populated to the values the user had previously put in, except for the selected files. Those had to be reselected. After thorough research, we found out that browsers do not allow prepopulating file input fields for security reasons and therefore there is nothing we can do about it.

6.4 Time analysis

Later we looked into the database and used the timestamp data we gathered to analyze how much time users spent on different tasks, giving us an insight into how long it would take on average to accomplish those tasks. Table 6.1 shows the average times it took users to tag and review one image.

Table 6.1. Average time to accomplish important tasks per one image.

Task	Average time per image
Tag	24.96s
Review	18.76s

It is also interesting to note that there is no obvious learning curve in the time it took users to tag and review individual images. The average times it took users to tag and review images in the beginning is not significantly different from the times it took them towards the end of the experiment. Figure 6.1 and Figure 6.2 show how long it took users to tag and review individual images. Each colored line represents times of one user.

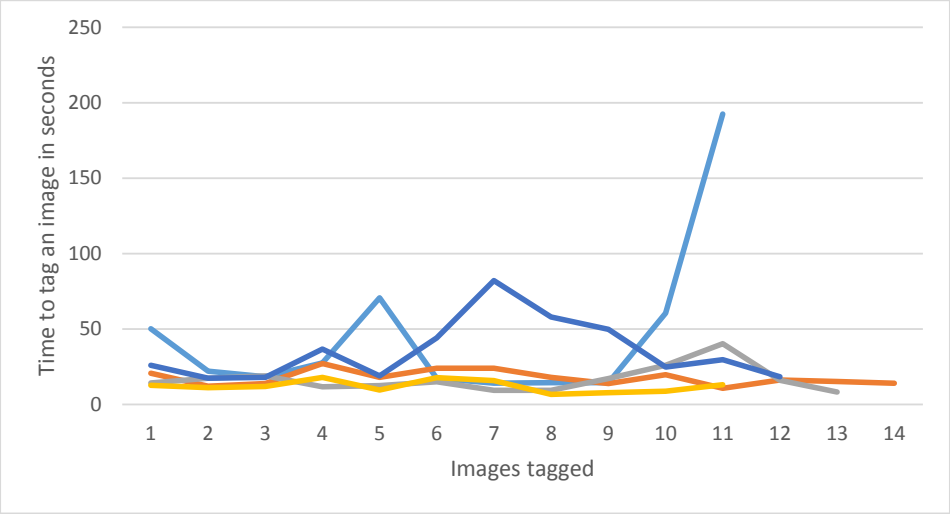

Figure 6.1. Times to tag each image by different users

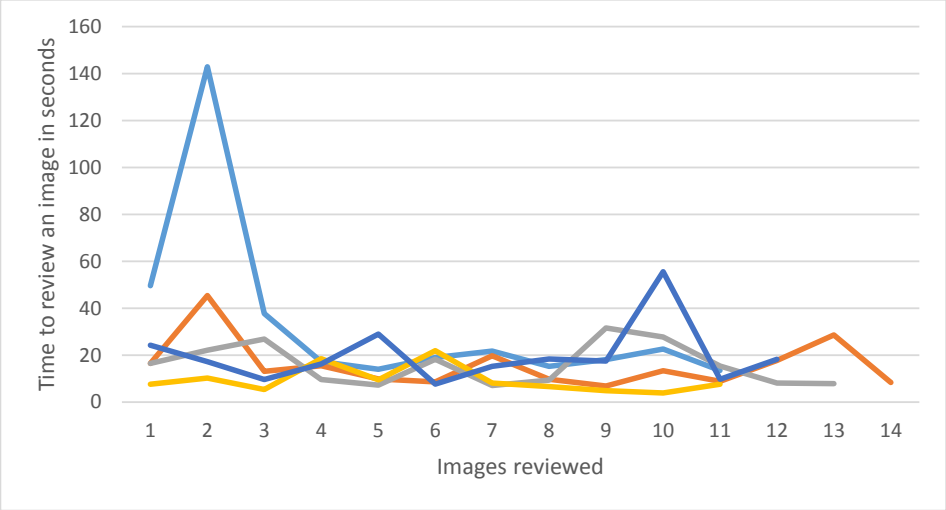

Figure 6.2. Times to review each image by different users

CHAPTER SEVEN

Future work

There are certain pieces of data we gather in the database for most images, without using them anywhere in the application itself. That is mostly because we either do not know how exactly to use the data yet, or we do not have enough resources to take advantage of it.

7.1 Patient Information

For example we are gathering a lot of information about the persons in these images, their age, name, sex and the eye diseases they have. This could potentially be used in the future to improve the neural network and allow us to not only detect leukocoria, but calculate the probabilities of various different diseases, based on the susceptibility of a certain sex or an age group.

7.2 Image Metadata Statistics

The previously mentioned information together with the GPS location information for each image and the time the photos were taken could be used to create some really interesting statistics. For example, we could find regions that have the most patients with leukocoria. What age is most susceptible to it and how those features change over time. A statistical dataset of this magnitude could help with the prevention and early detection of leukocoria, if we were to find that it is more likely to occur in certain regions and climates for example.

CHAPTER EIGHT

Conclusion

In this project, I have analyzed the current version of the image managing service for the Automatic leukocoria detection project. I have analyzed and described its current and future requirements. The web application distinguishes several user roles, which dictate the functions that user has access to.

Anybody can upload sets of photos with source attributions and appropriate license information. Some users are able to upload sets of photos, with more detailed information and source definition. Other users can use the application to tag eye positions on those photos and classify each eye position with a corresponding diagnosis label. More experienced users can review and adjust those eye positions and leukocoria classifications to be able to be used for training and testing purposes.

I have designed and implemented a new and improved version of the image managing application which meets all the requirements using state of the art approaches in web application development. The finished application has been successfully deployed on local virtual servers using a custom Docker deployment schema and all data from the legacy versions of this application has been migrated to the new version to be used further. The functionality of the application has been thoroughly tested by a group of volunteers, whose feedback was used to make some changes thereby making the application more usable and intuitive.

APPENDICES

APPENDIX A

User manual

This appendix is intended as a guide for anybody using Facetag. It shows how to perform basic tasks and has examples of both good and poor eye tags.

A.1 Upload

Facetag has two separate image uploading pages. In this section, I will only focus on the more advanced one, available to authenticated users. In order to upload images, make sure to login and then simply navigate to the appropriate option in the top navigation bar. The page is divided into three sections. The first section is where you select which files you want to upload. You can simply click the box and select the files using your browser's file selection function, or you can drag and drop them.

The second section contains mandatory information about the uploaded images. You choose a source for these images, which is usually a parent of a leukocoria affected child. If the source doesn't exist yet, you can add a new one by clicking "Add source" below. You also select a license and the priority with which the images will be processed.

The last section tells us important information about the actual person in these images. If there is more people, we usually focus on the person with leukocoria. In this section, you can fill out the name, age, sex and any additional comments.

After everything is filled out, you can click "Upload" and the images will start uploading to the server. This may take a couple minutes depending on your internet connection speed. After this is done, all the images are ready to be tagged and reviewed.

A.1.1 Public Upload

The public upload is almost identical to the private one with a few distinctions. You can't set priority for your images. Also, you don't get to directly choose a source and a license. Instead, you fill out your name and email address and the system tries to match these to the database. If there is a source that matches, the application adds these images to that source, otherwise it creates a new source. Instead of selecting a license, you have the option to agree to a preselected one.

Also, after the images get uploaded, they have to go through a validation process before being able to be tagged and reviewed. This process will be discussed in the next section.

A.2 Checking Images

After images get uploaded from anonymous users, they have to be checked to make sure they abide by our rules and can be used further. There is a designated section on the website to do just that.

Clicking on the "Check images" option in the main navigation bar shows the first unchecked image. Now you have two options, you can approve or delete the image. If the image is approved, it can be tagged and reviewed. Deleting the image removes it from the database. After choosing one of the options, the next unchecked image is shown and the process repeats as long as there are images to check.

A.3 Tagging Eye Positions

The tagging process is based on different work queues. A queue in this context is a list of images to be tagged, sorted in a certain order. After choosing one of these queues, the image in the front of the queue is shown. A user would then tag the image and move to the next one in the queue. This process repeats for as long as the user wants to or until she reaches the end of the queue.

The general goal here is to tag the positions of all human eyes in each photo and associate a label with each crop. The label can be healthy or leukocoric. You can find an example of a healthy eye in Figure A.1 and an example of a leukocoric eye in Figure A.2. A leukocoric eye should have a distinct white reflection in the retina. The red eye reflection that sometimes occurs is perfectly normal and should be labelled as healthy.

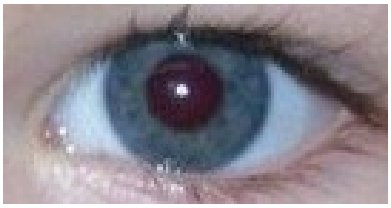

Figure A.1. Example of a healthy eye

Figure A.2. Example of a leukocoric eye

To create a new tag, click on the image wherever you want the crop to have its top left corner and whilst keeping the mouse button pressed, move the cursor to the bottom right corner of the desired crop and let go. After the tag has been created, it can be moved around by clicking inside of it and moving the cursor around and also resized by dragging the bottom right corner. Tags can be deleted by clicking the trash bin icon that appears in the top right corner when being hovered over.

To aid the tagging process, a user can zoom in to be able to tag the eye positions more precisely. She can also zoom out or reset the zoom to its original

state. All of this is done through a designated tool window depicted in Figure4.4. That window also allows moving back and forth within a queue and saving and reverting changes done to tags. Lastly, it displays a zoomed in crop of the area surrounding your cursor and additional metadata about the current image.

It is important to make sure that the eye is cropped right, we want to be able to see the whole eye, not just the retina, but the crop shouldn't be much wider than the eye either. You can use the previously shown crops as good examples. An example of a bad crop can be found in Figure A.3. The problem there is that the top portion of the eye is cropped out, the crop should encompass the whole eye.

Figure A.3. Example of a poorly cropped eye

There may also arise a situation, where the image is not oriented properly. In that case, if the feature is available, a user should always rotate it so that it is oriented properly. It is important to note here that doing so deletes any existing tags that have already been made. The feature is disabled if some other user has already placed tags on that image.

A.4 Reviewing Tags

Reviewing is intended to be done by more experienced users to make sure that all crops are accurate and all labels correct. Similarly to tagging, reviewing allows going through images in different queues. The queues in general work exactly like in tagging, but there may be different ones.

The purpose of reviewing is to go through previously tagged images and either accept or reject every tag. A reviewer can do the exact same things with each tag as a

tagger and can assign the extra review result label. The goal here is to make sure all eyes are tagged and all tags are correct. Tags can also be adjusted before accepting them to make sure they are perfect. After an image is reviewed and at least one of its tags has been accepted, that image is marked in the database to signify that it can now be used for export.

The tool window in the review process is almost identical to the one found in the tagging process. The only addition is a shortcut which accepts all tags found on the current image, saves the review results and redirects you to the next image in the current queue.

APPENDIX B

Deployment guide

This chapter describes how to setup new docker images from scratch, how to run those images into production and development containers and how to setup a local development machine.

B.1 Image Setup

This section assumes that we want to create a clean fresh setup of Facetag with an empty database. We are using docker as our main engine for all the servers.

B.1.1 Database Images

Start from a basic Ubuntu image. Run the image in a new container using the following command.

```
docker run -d -it --name facetag-db ubuntu
```

Then use the following command to get into the container's terminal.

```
docker exec -it facetag-db bash
```

And from there run the following commands.

```
apt-get install postgresql
service postgresql start
sudo -u postgres psql
```

This gets you into the `psql` environment. Run the following commands within it.

```
create database facetag;
create user facetag with password 'l3uk0';
grant all privileges on database facetag to facetag;
```

Exit the `psql` environment using `Ctrl+D`. And edit the following file.

```
/etc/postgresql/9.3/main/pg_hba.conf
```

Make the following changes to it.

```
# IPv4 local connections:
host all all all md5
```

Edit `/etc/postgresql/9.3/main/postgresql.conf` like this:

```
listen_addresses = '*' # what IP address(es) to listen on;
```

Run the following command and then exit the facetag-db container terminal.

```
service postgresql restart
```

After all of that is done we can proceed to setting up the web image.

B.1.2 Web Image

Again, start from a basic Ubuntu image. Run the image in a new container using:

```
docker run -it -d --link facetag-db:facetag-db -v
  /usr/local/facetag/:/facetag-certs:ro --name facetag-web ubuntu
```

Then use the following command to get into the container's terminal:

```
docker exec -it facetag-web bash
```

And from there run the following commands:

```
apt-get install python3-pip
pip3 install django
apt-get install git
ssh-keygen -t rsa
```

This generates an RSA key pair. In order to be able to use git in this container, you have to add the public key into your profile on Bitbucket that is associated with the Facetag repository. After you've done that, you can resume with the following commands.

```
git clone git@bitbucket.org:vcibur/facetag.git
pip3 install django-crispy-forms
pip3 install django-debug-toolbar
apt-get install python-psycopg2
apt-get install libpq-dev
pip3 install psycopg2
```

```
pip3 install django-multiupload
pip3 install piexif
apt-get install libffi-dev
pip3 install cffi
apt-get install libjpeg-dev
apt-get install libfreetype6-dev
pip3 install jpegtran-cffi
pip3 install pillow
pip3 install python-magic
pip3 install django-simple-captcha
pip3 install django-ratelimit
python3 /facetag/manage.py migrate
```

We need to generate a secret key (you can learn more about it by googling: secret key django) and place it into `/etc/secret_key.txt`. The location of the file can be changed in project settings and is only used in production (not development). We also need to add the HOST line in `/facetag/facetag/settings.py` in order for the web container to be able to access the database:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
       'HOST': 'facetag-db',
        'NAME': 'facetag',
        'USER': 'facetag',
        'PASSWORD': 'l3uk0',
        'ATOMIC_REQUESTS': True,
    },
}
```

B.1.2.1 Apache. Install apache and navigate to its configuration directory.

```
apt-get install apache2
cd /etc/apache2/
```

Create a file in the `sites-available` directory called `facetag.conf` with the following content:

```
WSGIPythonPath /facetag

<VirtualHost *:80>
```

```
Alias /static/ /facetag/facetagapp/static/

<Directory /facetag/facetagapp/static>
Require all granted
</Directory>

WSGIScriptAlias / /facetag/facetag/wsgi.py

<Directory /facetag>
<Files wsgi.py>
Require all granted
</Files>
</Directory>

ErrorLog ${APACHE_LOG_DIR}/facetag/error.log
CustomLog ${APACHE_LOG_DIR}/facetag/access.log combined

</VirtualHost>
```

Now we need to install mod-wsgi into Apache.

```
apt-get install libapache2-mod-wsgi-py3
```

Create a folder for Facetag logs:

```
mkdir /var/log/apache2/facetag/
chgrp adm /var/log/apache2/facetag/
```

Remove the default apache application and reset apache:

```
rm /etc/apache2/sites-enabled/000-default.conf
service apache2 restart
```

B.1.2.2 HTTPS configuration. Add an SSL configuration file for facetag in the `sites-available` directory called `facetag-ssl.conf` with the following content:

```
<VirtualHost *:443>
    ServerName facetag.ecs.baylor.edu
    ServerAdmin Vaclav_Cibur@baylor.edu

    # Django Application
    Alias /static/ /facetag/facetagapp/static/
    <Directory /facetag/facetagapp/static>
        Require all granted
    </Directory>
```

```
    WSGIScriptAlias / /facetag/facetag/wsgi.py

    <Directory /facetag>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>

    SSLEngine on
    SSLCertificateFile /facetag-certs/leuko.ecs.baylor.edu.crt
    SSLCertificateKeyFile /facetag-certs/leuko.ecs.baylor.edu.key
    SSLCACertificateFile /facetag-certs/leuko-ca-bundle.crt

</VirtualHost>
```

Enable SSL by running the following commands.

```
a2enmod ssl
a2ensite facetag-ssl
```

B.1.2.3 Postfix. We need postfix to send password reset emails. To install postfix and other mail utilities run this command in the facetag-web container:

```
apt-get install mailutils
```

Towards the end of the installation, it will ask you some questions. In the first selection, select Internet Site and in the second screen type in: facetag.ecs.baylor.edu. Now we need to edit the configuration file a little bit:

```
nano /etc/postfix/main.cf
```

Change `inet_interfaces` like this:

```
inet_interfaces = localhost
```

Restart postfix to apply all the configuration changes.

```
service postfix restart
```

Now it is also important to have the appropriate settings in place in the Facetag project configuration files (`settings_development.py`, `settings_production.py`). The email configuration should look something like this:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = '127.0.0.1'
EMAIL_PORT = 25
```

This concludes setting up the web image and we may now proceed with the final steps.

B.1.3 Final Steps

We should commit, stop and remove both of the containers we just finished setting up into images. Those images will later be used to create the production and development environments.

```
docker commit facetag-web facetag-web
docker stop facetag-web
docker rm facetag-web

docker commit facetag-db facetag-db
docker stop facetag-db
docker rm facetag-db
```

B.2 Running Containers

This section describes how to run and setup the production and development containers from existing docker images and an existing database.

B.2.1 Production

This section describes how to setup the production environment with existing docker images and a database.

Firstly, we create a new facetag-dbstore container. We specify a volume in the directory where postgresql stores data, so the directory is mounted from the host file system. This container is a pure data container and doesn't need to be running for us to be able to use the data.

```
docker create -v /var/lib/postgresql/9.3/main --name facetag-dbstore
  facetag-db /bin/true
```

Now we spawn a new facetag-db container from the same image, but we tell it to mount the volume from the facetag-dbstore container.

```
docker run -it -d --volumes-from facetag-dbstore --name facetag-db
  facetag-db
```

Then we connect to the facetag-db container and run start the postgresql service:

```
docker exec -it facetag-db bash
service postgresql start
```

Now we can spawn a new facetag-web container like this (maps inner port 443 to outer port 443 and inner port 80 to outer port 80):

```
docker run -it -d -p 80:80 -p 443:443 --link facetag-db:facetag-db
  -v /usr/local/facetag/:/facetag-certs:ro
  --name facetag-web facetag-web
```

But after that we also have to get into the container and start apache and postfix.

We also have to make sure to comment/uncomment the proper settings file selection. Files `/facetag/facetag/wsgi.py` and `/facetag/manage.py` contain such settings.

```
docker exec -it facetag-web bash
service apache2 start
service postfix start
```

The application should now be running and listening for requests on the domain: `leuko.ecs.baylor.edu`.

B.2.2 Development

This section focuses on how to setup the development version of facetag and how to fill its database with a random sample of data from the production database. We start by creating a database development container from an existing database image.

```
docker run -it -d --name facetag-db-dev facetag-db
docker exec -it facetag-db-dev bash
sudo -u postgres psql
```

Then, run the following commands within the psql console.

```
create database facetag;
create user facetag with password 'l3uk0';
grant all privileges on database facetag to facetag;
```

Now exit the psql console and the container and run the following commands to create the web server container. The port number 8080 determines which port will the development version of the application be available at.

```
docker run -it -d -p 8080:443 --link facetag-db-dev:facetag-db
  -v /usr/local/facetag/:/facetag-certs:ro
  --name facetag-web-dev facetag-web
docker exec -it facetag-web-dev bash
python3 /facetag/manage.py migrate
service apache2 start
```

We also have to make sure to comment/uncomment the proper settings file selection. Files `/facetag/facetag/wsgi.py` and `/facetag/manage.py` contain such settings.

Now we are going to export data from the production database container.

```
docker exec -it facetag-db bash
mkdir /dbdump
pg_dump -t auth_group -t auth_group_permissions -t auth_permission
  -t auth_user -t auth_user_groups -t auth_user_user_permissions
  -t disease -t license -t source -U facetag -h localhost -W facetag
  > /dbdump/dump.sql
sudo -u postgres psql
```

And run the following commands within the psql console.

```
\c facetag;
BEGIN;
CREATE TEMP TABLE imgs (id int);
INSERT INTO imgs SELECT id FROM image ORDER BY RANDOM() LIMIT 1000;
COPY (SELECT * FROM image WHERE id IN (SELECT id FROM imgs)) TO
  '/dbdump/images.sql';
COPY (SELECT * FROM eye_tag WHERE image_id IN
```

```
  (SELECT id FROM imgs)) TO '/dbdump/eyetags.sql';
COPY (SELECT * FROM face_tag WHERE image_id IN
  (SELECT id FROM imgs)) TO '/dbdump/facetags.sql';
COPY (SELECT * FROM image_diseases WHERE image_id IN
  (SELECT id FROM imgs)) TO '/dbdump/diseases.sql';
ROLLBACK;
```

The number 1000 in the previous set of commands determines how many images will get exported from the production database and it can be changed at will. Exit the production database container and run the following commands to copy the exported files into the development database container. The files will also be copied to you current directory in the process.

```
docker cp facetag-db:/dbdump/ .
docker cp dbdump/ facetag-db-dev:/dbdump/
rm -r ./dbdump/
```

Now we can go back to the production database container and remove the dump files.

```
docker exec -it facetag-db bash
rm -r /dbdump/
```

After that is done, we can exit the container and run the following commands to enter the database development container.

```
docker exec -it facetag-db-dev bash
cat /dbdump/dump.sql | psql -U facetag -h localhost -W facetag
sudo -u postgres psql
```

And finally run the following commands within the psql console to import the remaining data.

```
\c facetag;
BEGIN;
COPY image FROM '/dbdump/images.sql';
COPY eye_tag FROM '/dbdump/eyetags.sql';
COPY face_tag FROM '/dbdump/facetags.sql';
COPY image_diseases FROM '/dbdump/diseases.sql';
SELECT setval('image_id_seq', (SELECT MAX(id) FROM image));
SELECT setval('eye_tag_id_seq', (SELECT MAX(id) FROM eye_tag));
SELECT setval('face_tag_id_seq', (SELECT MAX(id) FROM face_tag));
```

```
SELECT setval('image_diseases_id_seq',
  (SELECT MAX(id) FROM image_diseases));
COMMIT;
```

At this point the development version of facetag is configured and ready to be accessed at: `leuko.ecs.baylor.edu:8080`.

B.2.3 Settings file selection

After starting any of the development or production web containers, we have to make sure to load the right settings file. There are three settings files:

- Development
- Production
- Local development

The first two are used on the server in docker containers and the last one is used on local machines of developers.

The selection of the settings file that will be used is done in two places in the code:

- `manage.py`
- `facetag/wsgi.py`

We have to make sure to comment out the two settings files we are not using and uncomment the one we are using following the directions in the above mentioned files' code comments. Also make sure to restart apache after changing any settings.

B.3 Local Development

To setup a local machine for development, you will not be using docker. I recommend using a virtual machine with Ubuntu installed. You have to install all the required libraries whose list can be seen in the Web Image setup section. You also need to setup and run a postgresql database and edit the local development settings file to reflect your local database settings. It is also important to uncomment the local development settings file import in `manage.py` and comment out the other

ones. I recommend using `python3 manage.py runserver` to run the server locally. To develop new changes I recommend the `PyCharm` IDE which has good support of Python and Django.

B.4 Maintenance

This section contains information on how to achieve some selected tasks in Facetag.

B.4.1 Disable Public Upload

There may be a time when we need to disable the public upload feature because of a high number of requests. To be able to disable it easily I have added a simple setting in `/facetag/facetag/settings.py`. To disable or enable public upload, simply open the file and find the following line of code.

```
PUBLIC_UPLOAD = True
```

The public upload is disabled if the value of this variable is `False`. To turn it back on again, simply change the value of the variable back to `True`. After changing the value of this variable in any way, we have to restart apache for the change to take effect. That can be done by running the following command.

```
service apache2 restart
```

B.4.2 Deployment

This section describes how to deploy new changes to the code into existing development and then production containers.

B.4.2.1 Deploying to Development. Firstly, we should deploy any new code changes into the development web container. To do so, enter the container using the following command.

```
docker exec -it facetag-web-dev bash
```

At this point, you should pull the new version of the code from git and do any other necessary changes like installing new libraries etc.

```
cd /facetag/
git stash
git pull
git stash pop
python3 /facetag/manage.py migrate
service apache2 restart
```

We are using git stash, because we want to save the settings file selection, which should be the only difference compared to the original git version. After this is done, please exit the container.

At this point we should thoroughly test the application to make sure everything works as it should. Please test not only the new features, but also any other part of the application that might have been affected.

B.4.2.2 Committing the image. We need to run the following set of commands in order to commit the development web container into its original image and to run the development container from it.

```
docker commit facetag-web-dev facetag-web
docker stop facetag-web-dev
docker rm facetag-web-dev
docker run -it -d -p 8080:443 --link facetag-db-dev:facetag-db
  -v /usr/local/facetag/:/facetag-certs:ro
  --name facetag-web-dev facetag-web
docker exec -it facetag-web-dev bash
service apache2 start
exit
```

B.4.2.3 Deploying to Production. After the changes have been deployed to development, thoroughly tested and committed to the web image, we can push them into production. To do that, run the following commands. Please keep in mind that all the steps in this section should be done in a short amount of time, because the main production website will be unavailable throughout the process.

```
docker stop facetag-web
docker rm facetag-web
docker run -it -d -p 80:80 -p 443:443 --link facetag-db:facetag-db
  -v /usr/local/facetag/:/facetag-certs:ro
  --name facetag-web facetag-web
docker exec -it facetag-web bash
python3 /facetag/manage.py migrate
```

At this point we also have to make sure that the proper settings file is selected, which is thoroughly discussed above. After that is done, we finish by running the following commands.

```
service apache2 start
service postfix start
exit
```

After this is done, we should make sure that everything works and test the application thoroughly.

BIBLIOGRAPHY

Bendoraitis, A. (2014). *Web Development with Django Cookbook*. Packt Publishing Ltd.

DEMIRCI, H., C. L. SHIELDS, J. A. SHIELDS, S. G. HONAVAR, and R. C. EAGLE (2001). Leucocoria as the presenting sign of a ciliary body melanoma in a child. *British Journal of Ophthalmology 85*(1), 110–110.

Diksha (2016). Implementing mtv model in python django — blog. [Online; accessed 4-April-2016].

Django (2016a). Models — django documentation — django. [Online; accessed 4-April-2016].

Django (2016b). Temnplates — django documentation — django. [Online; accessed 4-April-2016].

Elman, J. and M. Lavin (2014). *Lightweight Django*. " O'Reilly Media, Inc.".

Rivas-Perea, P., E. Baker, G. Hamerly, and B. F. Shaw (2014). Detection of leukocoria using a soft fusion of expert classifiers under non-clinical settings. *BMC ophthalmology 14*(1), 1.