Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Data bachelor	Vårsemesteret, 2019 Åpen / Konfidensiell
Forfatter: Jacob Vetle Kvinnsland Ihle Thomas Osaland	Jacob Ihle, Thomas Osaland (signatur forfatter)
Fagansvarlig: Erlend Tøssebro Veileder(e): Erlend Tøssebro	
Tittel på bacheloroppgaven: GUI for å vise store histologiske bilder Engelsk tittel: GUI for displaying large histological images	
Studiepoeng: 20	
Emneord:	Sidetall: 97 + vedlegg/annet: 3 Stavanger, 14.05/2019 dato/år

Forside for bacheloroppgaven
Det teknisk-naturvitenskapelige fakultet

Terminology list	
.scn format	Image format developed by Leica. Multi-layered image.
Aperio Imagescope	Leicas' own software for viewing .scn images.
Cookie	Data sent from a website to be stored on the clients web browser.
Deep Zoom (.dzi)	Image format developed by microsoft. Multi-layered image.
Histological Image	Large images used to study the tissues of animals and plants.
OpenSeadragon	Javascript library for viewing Deep Zoom images.
Openslide	An interface for reading "whole-slide images".
ORM	Object Relational Mapper. Programming technique that effectively creates a virtual object database within a programming language, such as Python.
SQLAlchemy	Python library used for accessing SQL databases.
Tag	Short description linked to an object.
Tile	Small image that represents a part of a much larger image.
Viewer	The OpenSeadragon viewer that the user interacts with.
Viewport	Part of Viewer that handles coordinate-related functionality(zoom, pan, rotation etc.)
Werkzeug	A Web Server Gateway Interface library, primarily used in this application for password hashing.
Whole-slide images	Glass slides containing tissue samples(usually) that have been digitalized.
XHR	XMLHttpRequest, a standard way to update the web page with information from the server without having to reloading the page.
XML	Extensible Markup Language. A set of rules of how to format a document to make it both human- and machine-readable.

1 Project Description

The task given to the group was developing a web application for viewing and annotating large histological images. Histological images are microscopic images of tissues and is commonly used in the study of diseased tissue.

The main functions of the website is for it to be easy to access histological images from a database, access tagged areas on the images and also create your own. One should also be able to search the images after specific tags so that finding different tissue damages is made easier.

The task was given by the Biomedical data Analysis laboratory at the university of Stavanger, with Rune Wetteland and Professor Kjersti Engan as contractors, while Erlend Tøssebro acted as supervisor.

Video demonstration:
https://drive.google.com/file/d/1YOOuZL8yHl_8YqY_-WqaXDBNBJzV1G-x/view?usp=sharing

2 Introduction

2.1 Technology choices

2.1.1 Python

When deciding what program language to use, it was immediately suggested by Rune to use python, as he had previous experience handling histological images using python. After some research it was discovered that there are a rather limiting amount of technologies that have supports for these types of images. Since the Deep Zoom format was developed by microsoft, there exists libraries for C#, but those libraries are only for the windows operating system, and since this application is meant to run on a linux machine, it was ruled out as a possibility. That made python the only choice left, at least that the group also had some experience with.

2.1.2 Flask

Flask is a Back-end web framework written in python. It is classified as a micro-framework. This classification is given to frameworks with little to no dependencies on external libraries. That means it has no database abstraction layer, form validation or any other such component. Which makes it easy to use the tools one personally wish to use. In total, Flask has two dependencies, Werkzeug and jinja2, both developed by the same entity as Flask, the Pallets team.[1,2,3] There exists some other web frameworks for python, like django and bottle, but Flask was the only one the group had used before so the choice was rather simple.

2.1.3 Jinja2

Jinja2 is a web templating engine for python, modeled after Django, but created to provide expressions more similar to python. Web templating engines are used for generating HTML with information often received from the server. Jinja is also developed by the same team that created Flask, which made it a natural choice.

2.1.4 OpenSeadragon

OpenSeadragon is “An open-source, web-based viewer for high-resolution zoomable images, implemented in pure JavaScript, for desktop and mobile.” [4] It provides a powerful API with a wide range of tools for navigating the image, overlays and much more.

2.1.5 OpenSlide

OpenSlide is a C library that provides a simple interface to read whole-slide images. It provides a consistent and simple API for reading files from multiple vendors.[5]
The library has a python binding which will be used in this project. It includes a DeepZoom generator to allow for accessing e.g. “.scn” images as if it was a “.dzi” file without having to convert it.

2.1.6 MySQL / SQLAlchemy

The database is a MySQL relational database and SQLAlchemy which is an object-relational mapper (ORM), that is used as a connection between python and the database. Which makes it possible to create sql queries in an object oriented programming language. SQLAlchemy also simplifies the database aspect by auto escaping the queries made to the database, as well as making sure each transaction is isolated.[6]

2.1.7 JavaScript

For client side functionality Javascript with JQuery was the preferred choice. A main reason being that a of lot the logic was activated when users interacted with different parts of the GUI. And JQuery excels when it comes to event handling.

2.1.8 HTML/CSS

HTML and CSS is the universally accepted way to build and style a website, and combining it with JavaScript makes a compelling way to create websites.

2.2 List of dependencies

- Python libraries
    - Flask
    - Flask-SQLAlchemy
    - SQLAlchemy
    - Flask-Login
    - openslide-python
    - mysqlclient
    - mysql-connector
- Javascript
    - OpenSeadragon
    - OpenSeadragon-scalebar
    - OpenSeadragon-canvas-overlay
- Other
    - Openslide distribution package for unix based operating systems / Visual C++ for Windows | https://openslide.org/download/
    - Libmysqlclient20
- Browsers
    - Google chrome version 70 and later.

2.3 Working Method

There was a discussion in the beginning if there was any benefits in using KanBan or Scrum, but they are generally used in larger groups (3-9) and for full time workers. As there was only 2 working on the project, and neither dedicating 100% work hours on it, there was not found any substantial benefits. Instead the next feature to be implemented was decided in the beginning of each week. This way the group always had a short term goal to work against.

Special thanks to:

Erlend Tøssebro as advisor and liaison between the bachelor group and the Biomedical laboratory.

Theodor Ivesdal for helping with installing all necessary dependencies on the unix server, allowing our server to run through an inhouse apache server, and giving us a MySQL database to work with.

3 Theory

In this chapter, a brief introduction is given to key concepts that will form an understanding basis for the project's implementation.

3.1 Histological images

Histological images are very large images, often multiple GB in size, used in Histology, which is a branch of biology that studies tissues of plants and animals. The images consists of many copies of the same image, with different resolutions on top of each other, like a pyramid.

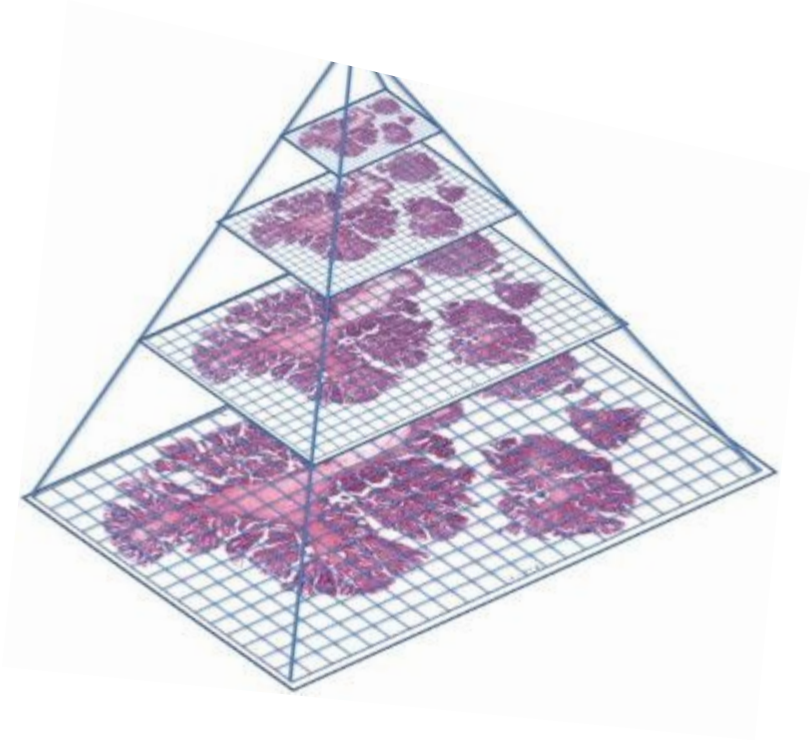


Image text: Visual representation of a histological image structured as a pyramid.[7]

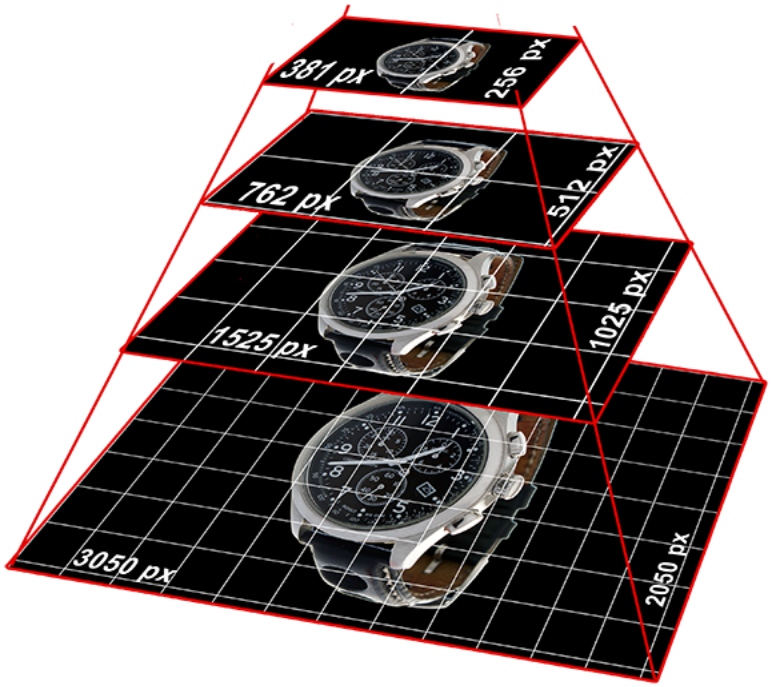


Image text: Visual representation of the increase in number of tiles in each layer. [8]

Each image layer is tiled. This means that the layers consists of a large number of smaller images(tiles), where each tile is 256x256 pixels in size. These are combined to form the full image layer. This means that the whole image does not need to be loaded. Individual tiles can be loaded as they are needed. Which greatly reduces the amount of data that has to be loaded when viewing the image.

3.2 File Formats

Given the size of histological images, it is generally not beneficial to render the entire image. On a web application this becomes even more true, as loading images that are Gigabytes in size over the internet can be very slow. When accessed on computers locally from the university, this will not pose a huge problem, as the access points for the wifi has a 1Gbit/s connection to the server. But since the application is also supposed to be accessed remotely the speeds may be severely reduced.

Even ignoring the limitations of internet speeds, not having to load the entire image in memory greatly reduces the load on both the server and the client.

The way we solve this is by only downloading the tiles being viewed. Subsequent tiles are downloaded as the user pans to (or zooms into them); animations are used to hide any jerkiness in the transition.[9]

With the pyramid like structure, it gives the possibility of creating sparse images. Sparse images makes it possible for images to have more resolution in different parts of an image. Both the Deep Zoom and BigTiff file formats supports sparse images.

3.2.1 Deep Zoom

The Deep Zoom file format (.dzi) is developed by microsoft and is an XML based approach to the image type described above. One oddity that result from this file format, is that it starts counting levels at 1x1. This means there is generated 9 (2^8 <= 256) layers containing the same image at slightly higher resolutions. There may exists a practical reason for needing such low resolution images, but none comes to mind.

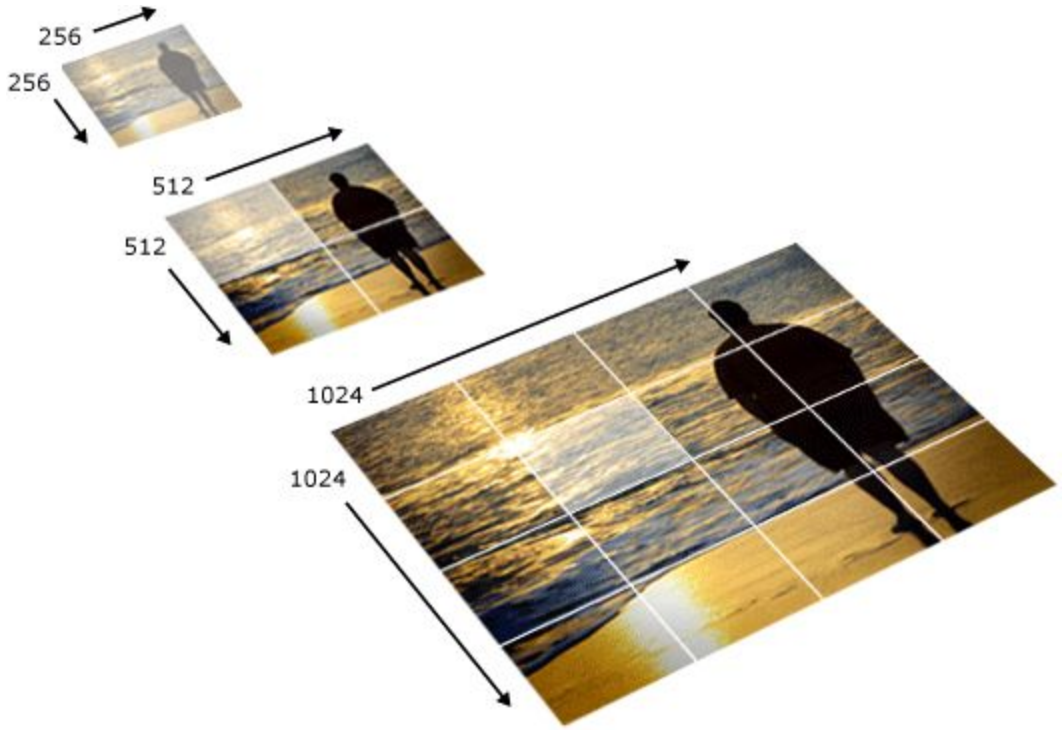
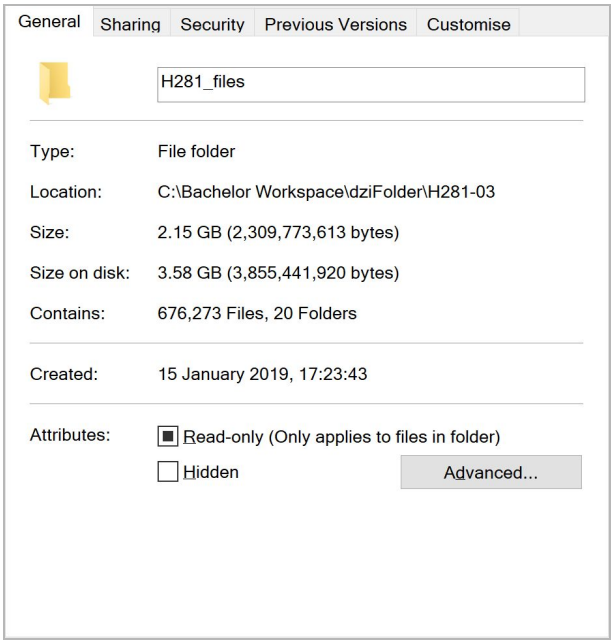


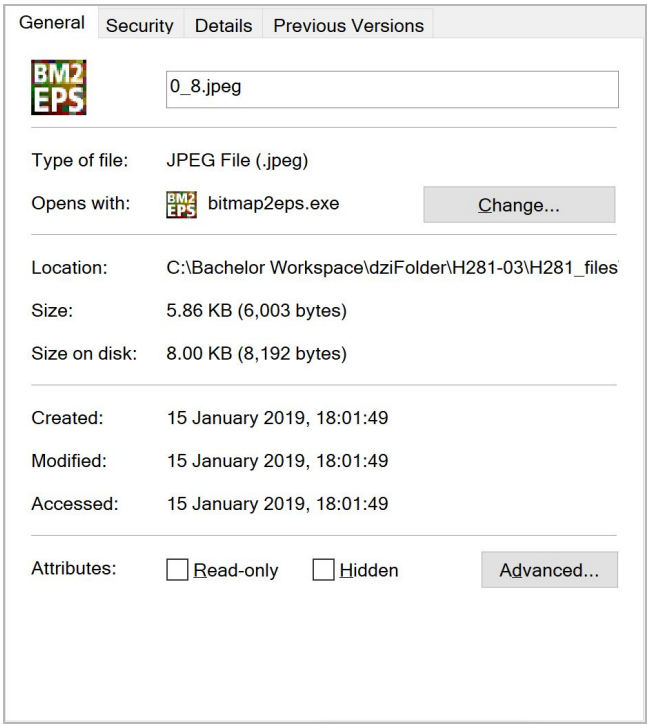
Image Text: If, for example, you zoomed in to see only the highlighted middle part of the image, Deep Zoom only loads the highlighted tiles, instead of the entire 1024x1024 image.[10]

For storage, each level is stored in a separate folder. All levels are stored in a folder with the same name as the .dzi file with the extension removed and "_files" appended to it.



An advantage of using such small image sizes (256x256) is that it is very efficient when it comes to data usage, such as in the a web application. Since the program only has to load the relevant pixels to a very high precision. A drawback from using so small files is that the size on disk becomes quite a lot larger than the actual size of the files.

In the previous picture it is shown that an image that was originally 1.29GB in size have been converted to .dzi format. As can be seen, the new size is now 2.15 GB. Over 60% larger than the original size. In the same picture we also see that the size on disk is 60.1% larger than the actual size of the files(1.43 GB wasted). This is because the operating system organize disk size in clusters, these clusters are limited to be in power of two's (2^n), and normally limited within 4 - 16 kB. This means that if a file is 3kB in size, it will be allocated a block that is 4kB in size since it will not fit in a block with size 2kB, and 4 kB is the next cluster size the operating system can allocate.[11]



To give an example, a specific image tile was selected. Here we can see that only 6 003 bytes was used out of 8 192 bytes allocated on disk. And with a total number of 676 273 files, there is quite a lot of wasted storage space.

One can also approximate the average amount of space that is lost in this manner by using the equation *(cluster size)/2 * (number of files)*. Which if used on the H281_files folder, gives us 1.29GB of wasted space, which is 90,2% of the actual wasted space.

In total, the size goes from 1.29GB to 3.58GB, an increase of 278%. With the 1 100 images found on the unix server, converting every single image to .dzi is not a very practical solution.

3.2.2 Tiff file format

Tagged Image File Format (TIF) is used for storing bitmap images. TIFF files are organized into three sections: the Image File Header(IFH), the Image File Directory(IFD) and the bitmap data. Along with JPEG and PNG, it is a popular format for deep-color images, but due to it's 32-bit offset, it is limited to 4GB.

The TIFF IFH contains three fields of information and is a total of only eight bytes in length:

Offset	Datatype	Value
0	Word	Byte order indication
2	Word	Version number (always 42)
4	Unsigned Long	Offset to first IFD

Because of this tag based structure, TIFF files can act as a container for other file formats. This is achieved by having the IFH point to the IFD, which contains information about the image, as well as pointers to the actual image data. One of the pieces of information found in the IFD is the color model used (RGB, CMYK, etc), this lets TIFF use the proper type of color map (lookup table) when displaying the image.[12]

3.2.3 BigTIFF

BigTIFF came about as the 4 GB limit of TIFF started to become insufficient. BigTIFF closely resembles TIFF, but uses a 64 bit offset instead of the earlier 32. This gives BigTIFF the ability to store much larger files (theoretically up to 16 Exabytes), while keeping the benefits of TIFF, like support for most color models and making it easy for developers to extend TIFF libraries to BigTIFF.

Offset	Datatype	Value
0	Word	Byte order indication
2	Word	Version number (always 43)
4	Word	Bytesize of offsets Always 8 in BigTIFF, it provides a nice way to move to 16byte pointers some day. If there is some other value here, a reader should give up.
6	Word	Always 0 If there is some other value here, a reader should give up.
8	Unsigned 8Byte	Offset to first IFD

3.2.4 Leica´s SCN file format

Leica's .scn format is an image with tiles structured like a pyramid, much like the Deep Zoom format, but instead of storing each tile as an individual file and each layer as a folder, it is saved in a single BigTiff image.[13] This makes handing the images for any operating system much easier, as the need for indexing thousands, if not millions of images is removed.

3.3 Slides

3.3.1 Microscope slides

A microscope slide is a thin flat piece of glass, typically 25 x 75mm and about 1 mm thick, used to hold objects for examination under a microscope.

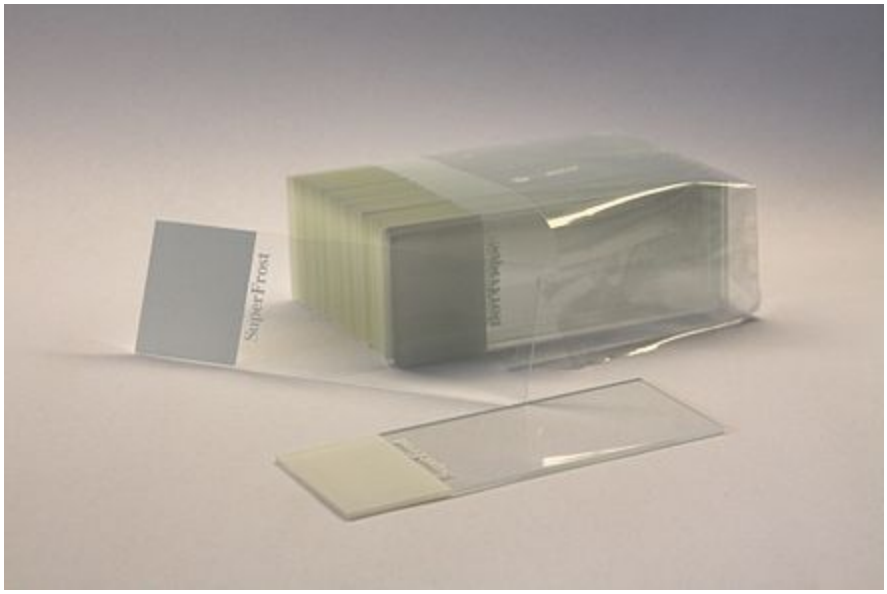


Image text: A set of standard of 25 x 75 mm microscope slides.[14]

3.3.2 Whole Slide Images

Whole slide images, are digitalization of the aforementioned microscopic slides. It aims to emulate conventional light microscopy in a computer system. Utilizing specialized scanners (Leica SCN400 in this case) to generate large digital images of the physical slides. And then use specialized software, like the web application described in this thesis, to view and analyze the images.

To create the file formats (.dzi, .scn etc.) used in the whole slide images, the scanners utilizes a robotics-controlled motorized slide stage to obtain a large number of image frames, which are assembled together into a mosaic pattern and stitched together into a single massive image.

Whole slide image platforms have the potential to improve diagnostic accuracy, increase workflow efficiency, balance workloads, better integrate images with information systems, and financially enhance return on investment. [15]

3.4 VIPS

VIPS is a fast and very memory efficient, open source image processing software package. It can do parallel work on up to 64 processor cores and has support for 64-bit operating systems, which means it is not limited by the 2GB file size limit of 32-bit operating systems. This makes it particularly good for working with large images, such as Deep Zoom and .scn images. [16]

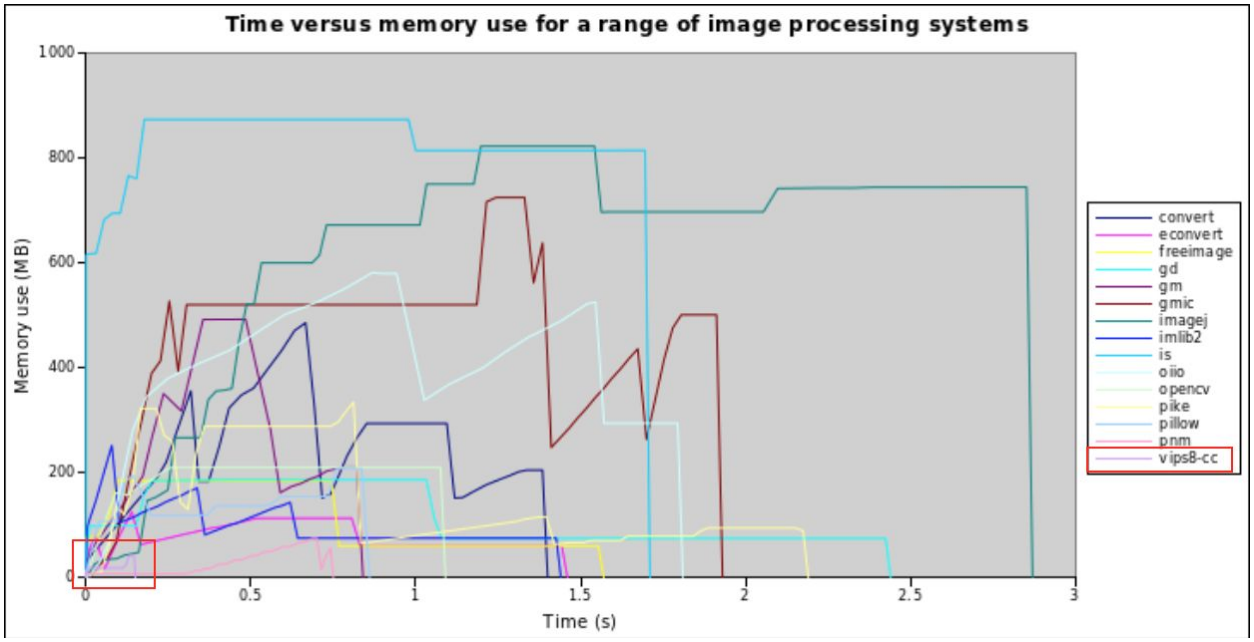


Image text: The image above demonstrates how much more efficient VIPS is to load images. The test was done loading a 5000 by 5000 pixels large 8-bit TIFF image.[16]

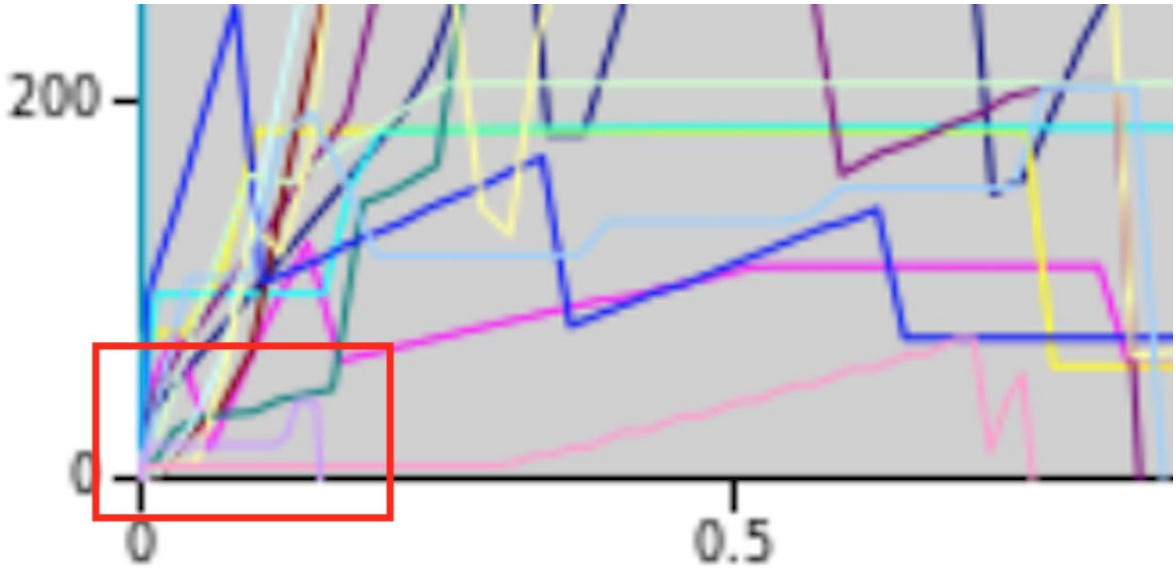
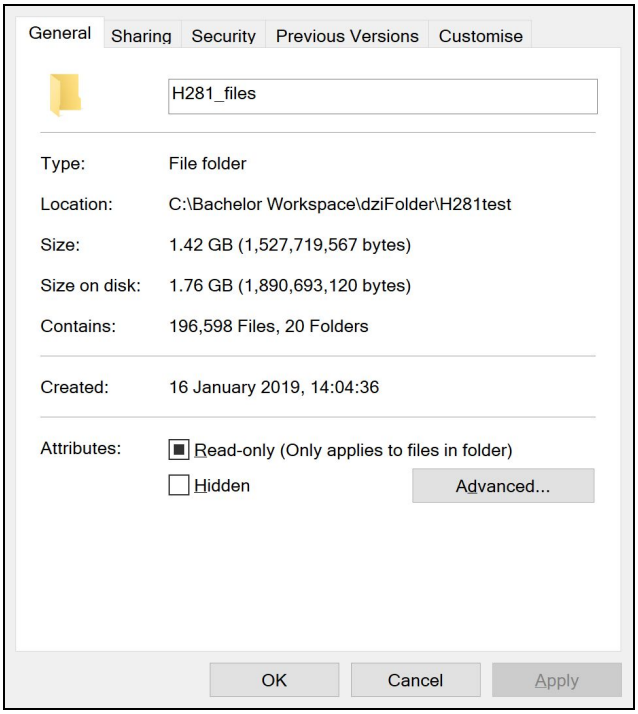


Image text: Closer look at the area of interest from the graph above. Showcasing the speed of VIPS compared to other image processing softwares.[16]

The software contains two main parts, libvips and nip2. libvips is the image processing library and nip2 is the graphical user interface. Development of VIPS startet during a research project funded by the EU in 1989-1993 called VASARI. After its discontinued funding, VIPS has been further built upon and maintained by several other research groups, and is still being maintained today.[16,17]

3.4.1 Converting with Vips

There is little documentation on the .scn file format, which leads to few tools for the format. Early in development, it was figured out how to use vips to convert the .scn files to .dzi, which is supported by OpenSeadragon. Here, the problem with a large amount of files as described in 3.2.1 arose.



Writing a small python program that looks for files smaller than 1900 bytes, a number that was determined after looking at different kinds of images in the foldes and seeing what kind of sizes the empty images had, found 479 676 files. And by modifying the program to delete the files found, the folder now contains 196 598 files, which in turn reduces the total size used by 1.82 GB. Another change is the relative space that is wasted, it went from over 60% to about

24%, this must mean that in general, the images with information has sizes closer to binary sizes. Luckily there was a solution that does not involve converting the images.

3.4.2 Transitioning to OpenSlide

By leveraging the OpenSlide API in this project, it was no longer necessary to convert all .scn images to the .dzi format. Individual tiles being requested by the web application can now be individually extracted from the .scn image as needed. Lowering storage and memory requirements for the hosting machine, and saving a lot of time by eliminating the need for complete conversion.

3.5 Security

The histological images being accessed through the application discussed in this thesis consists of confidential health data. Which ensured that security was a high priority throughout development.

When creating a web application there are a lot of security concerns one need to think about. On the front end, cross site scripting and SQL injection attacks are two prominent attack vectors that are common to all websites and needs to be guarded against. In the backend how one store information is also of the out most importance, in case there is a data breach. Keeping things up to date is another important aspect of security.

3.5.1 Cross site scripting

Cross site scripting (XSS) enables attackers to inject client side scripts into web pages viewed by other users. Simply put, an attacker can insert js code (<script></script>) into any text field on the website where the browser will read that as the js code it is and run it.

There are many subtle ways XSS can be implemented, depending on the website. Take for example a social network that haven't properly secured their website against XSS, here other users can access your profile, if you insert some javascript code on your profile, it will not be visible to users, but it will be read by the browsers which will in turn run it. This for example means that all cookies stored on the other user's browsers can potentially be read by that script, and sent back to you, even though you are not supposed to have access to that information.

Another example, that is more applicable to the web application described in this thesis is the ability to access the back end server. As explained in a later chapter if a client sends a request to the server url https://*histology.ux.uis.no/searchTags* with the proper information, the server

will return a dictionary of every image that contains this tag. If the website was not properly protected against such attacks, anyone could potentially have access to sensitive information.

The relatively easy solution is to escape all special characters that can be used to create scripts, one example being converting the “<” and “>” characters into “&lt” and “&gt” respectively. &lt and &gt are rendered on webpages the same was as < and > but are handled very differently by the web browser.

Flask configures jinja2 to escape all values unless explicitly told otherwise.[18] With one exception, that being links, e.g <a href = “ {{ value }} “> [19]

3.5.2 SQL injection attack

SQL injection attack is another code injection attack like XSS, but is targeted towards databases. Nefarious SQL statements are inserted into an entry field for execution, it exploits a security vulnerability where user inputs are incorrectly filtered for string literal escape characters. SQL injection attacks allow attackers to spoof identity, tamper with existing data, disclose all data in a system or even destroy the data or make it otherwise unavailable.[20] Luckily SQLAlchemy auto escapes all strings put through its predefined methods.[21]

3.5.3 Forward secrecy

Forward Secrecy is an encryption style known for producing temporary private key exchanges between clients and servers. For every individual session initiated by a user, a unique session key is generated. If one of these session keys is compromised, data from any other session will not be affected. Therefore, past sessions and the information within them are protected from any future attacks. [22]

If forward secrecy is not implemented, malicious actors can intercept and collect encrypted packages and wait for the servers private key to be compromised. All collected data packets can then be decrypted. A good example of why forward secrecy is important is the recent heartbleed bug. Heartbleed allowed attackers to get access to the private keys of affected servers. Enabling the attackers to decrypt data from all client sessions.

3.5.4 HTTPS

Hypertext Transfer Protocol Secure(HTTPS) is an extension of the Hypertext Transfer Protocol(HTTP) used for secure communication between client and server. It is often referred to as HTTP over Transport Layer Security(TLS - Preceded by SSL) as the TLS protocol is used to encrypt HTTP packets before sending them to their destination.
In addition to encryption, TLS provides two more layers of protection.

- Data integrity: Data cannot be modified or corrupted during transfer without being detected.
- Authentication: Protects against “man in the middle attacks” by proving that the user communicates with the intended website.

Even with all these protection layers it’s still possible to execute a “man in the middle” attack on connections using HTTPS. On HTTPS, detection of this type of attack is dependant on the user's ability to spot a fraudulent certificate and the browsers ability to warn the user about untrusted certificates. Modern browsers will always warn the user if it detects an untrusted certificate on the target site.[23]

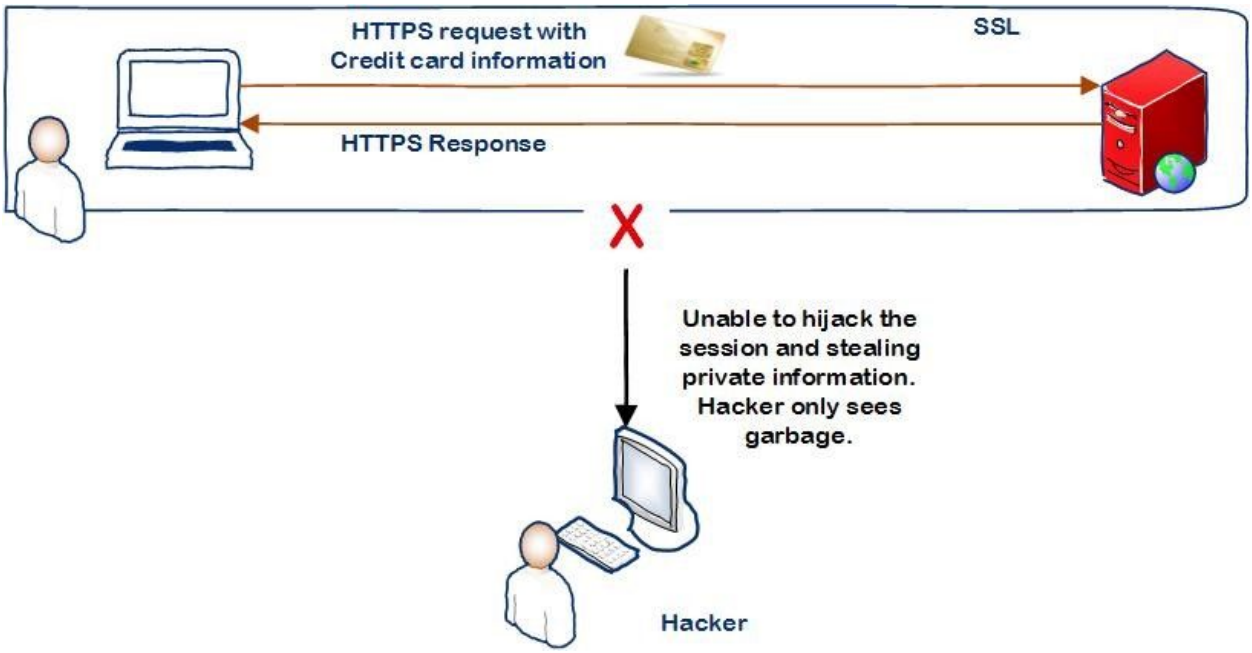


Image text: The image illustrates a “man in the middle” attack where a hacker hijacks another users session with the server. In this case, because HTTPS is being used, any data the hacker manages to collect are encrypted and useless to the hacker.[24]

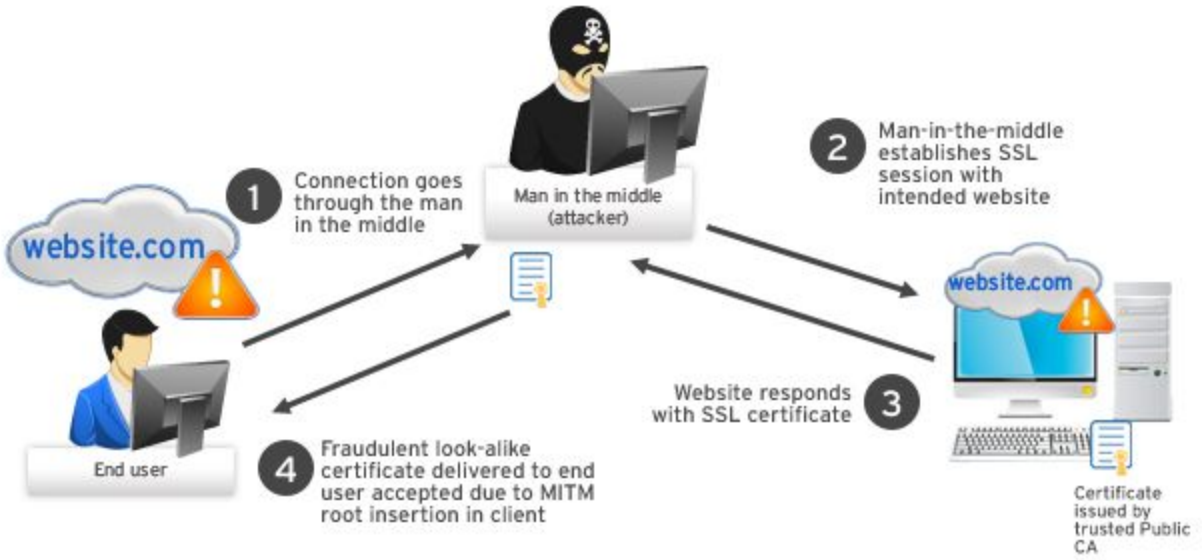


Image text: The image illustrates a more sophisticated version of the "man in the middle" attack. [25]

3.5.5 Login

Login provides user session management, as well as security. By implementing login it becomes easy to restrict access to the whole server except for the login screen. Restricting access was implemented using Flask-login. Flask-Login is a Flask extension designed to make user management easier for the developer as it creates a easy way to restrict access to different parts of the website. For example, unless a user is logged in, the only part of the web application they have access to is the login screen.

The text fields in the login screen are also protected against SQL-injection attacks with SQLAlchemy. This is necessary as the contents of the text fields are connected to the database, which automatically makes them a target for SQL attacks. And combined with the other security implementations, primarily HTTPS and password hashing, the login screen has a fair amount of security. Which is needed as it is the gateway between the server and the open networks it is connected to.

3.5.6 Cryptographic Hashing

A cryptographic hash function is a special class of hash functions that has certain properties which makes it suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size, and is designed to be a one-way function, meaning that it is infeasible to invert. The only way to recreate the input data from an ideal cryptographic

hash functions output is to attempt a brute-force search of possible inputs to see if they produce a match.[26]

An ideal cryptographic hash has five main properties:[27,28]

1. It is deterministic, meaning the same message always results in the same hash
2. It is quick to compute the hash for any given message
3. It is infeasible to generate a message from its hash except by trying all possible messages.
4. A small change to a message should completely change the hash value
5. It is infeasible to find two different message with the same hash value

3.5.6.1 Common problems

A problem with all hashing functions is the concept of the pigeonhole principle. It describes that if there is n number of possible hashes, m number of possible messages to hash, and m > n. There must exist some messages that creates the same hash, which in hashing is called collision. But given a high enough number of possible hashes, this will not be a problem, for the foreseeable future.

Another common weakness with all cryptographic hashes is the potential of a dictionary attack. A dictionary attack uses the fact that all hashes, given two identical inputs, will give the same output. This creates a problem when a lot of people use the same password (password1, qwerty, letmein, etc). Thus making it possible to collect the most commonly used passwords, run them through the hashing algorithm, get the output, and just search the database after password hashes that match the found results.

3.5.6.2 Rainbow Tables

A rainbow table is a table of precomputed cryptographic hash functions usually used for cracking passwords. It is used as a tradeoff for using more storage space and less processing time than a brute-force attack, and can be used efficiently against multiple different hashing functions

3.5.6.3 MD5

MD5 is a message-digest algorithm that used to be widely used hash function producing a 128-bit hash value. It's designed to be used as a cryptographic hash function, but has been found to suffer from extensive vulnerabilities such as collision and rainbow tables.

In 1996 the first collision were found in the compression function used for MD5[29], and in 2005 researches were able to create pairs of PostScript documents, [30] and X.509 certificates.[31]

This means MD5 cannot be used for Digital Signatures, as it is now possible to create documents that are different but contains the same hash.

As for when it comes to rainbow tables, there are multiple websites online where you can either download rainbow tables to use against MD5 encryption, or enter single hashes into the website to get the plaintext.[32–34]

3.5.6.4 SHA-1

SHA-1 (Secure Hash Algorithm) is similar to MD5, a cryptographic hash function that produces a 160-bit hash value instead of 128. And like MD5, has been found to contain certain vulnerabilities.

In 2005 it was shown that SHA-1 is not collision free. Bruce Schneier also discussed the possibility of well funded opponents being able to build machines that could do enough operations fast enough to break any SHA-1 hashes in a relatively short time, given moore's law, that would be substantially easier today than when it was discussed in 2005.

In 2017 CWI Amsterdam and Google had managed to create two dissimilar PDF files which produced the same SHA-1 hash.[35,36] And in 2015, the US National Institute of Standards and Technologies(NIST) updated their policies and said SHA-1 should no longer be used for applications that require collision resistance.[37]

3.5.6.5 SHA-2

SHA-2 is the successor of SHA-1 and is designed by the United States National Security Agency (NSA). SHA-2 consists of different hash functions with hash values 224, 256, 384, 512, where SHA-256 and SHA-512 are hash functions computed with 32 and 64 bit respectively, and SHA-224, and SHA-384 being truncated versions of SHA-256 and SHA-512 respectively.[38]

SHA-2 (256bit) is listed by NIST as the minimum recommended hashing function to use for cryptographic hashing.[39] That does not necessarily mean that it is completely secure. SHA-2, like SHA-1 and MD5 is based on the Merkle-Damgård hash function and as such, is vulnerable to length extension attacks. But length extension attacks are not a vulnerability when it comes to hashing passwords, so it is of no concern to the current application.

3.5.6.6 Salting

The way to solve the problem of dictionary attacks is by salting the passwords. Salting passwords means adding some random data to the password strings, and run the new string through the hash function used. This means that even if some people might use the same passwords, there is no way of knowing by looking at the hash. Ofc this entails that each salt is

unique for each password. If the salt was identical and an attacker found what the salt was, one would simply hash the most commonly used passwords with the given hash, and one could again use dictionary attacks.

3.5.6.7 SHA-3

SHA-3 is the latest member of the SHA family, released by NIST in august of 2015[40]. It is internally different from earlier SHA hashes as it is no longer built upon Merkle-Damgård. Instead it is built using the Keccak algorithm and is therefore not susceptible to Length extension attack, like earlier SHA algorithms. [40,41]

The main reason SHA-3 is not used in the current application is availability, SHA-2 is such a widely used hashing function and is the default hash used by werkzeug in Flask. And since there is no security concern, no rainbow tables and no collisions, the extra work required to implement SHA-3 made it a low priority.

4 Construction

This chapter explains the steps taken to solve the tasks given by the biomedical laboratory. How the web application is able to display the histological images, how information is stored, and how drawing was implemented, as well as how security was implemented to keep the information secure.

4.1 Graphical User Interface

The GUI is a rather simplistic design with most of the functionality on either the header or the footer. This leaves the entire middle section available to display the image. Which is the main point of the application.

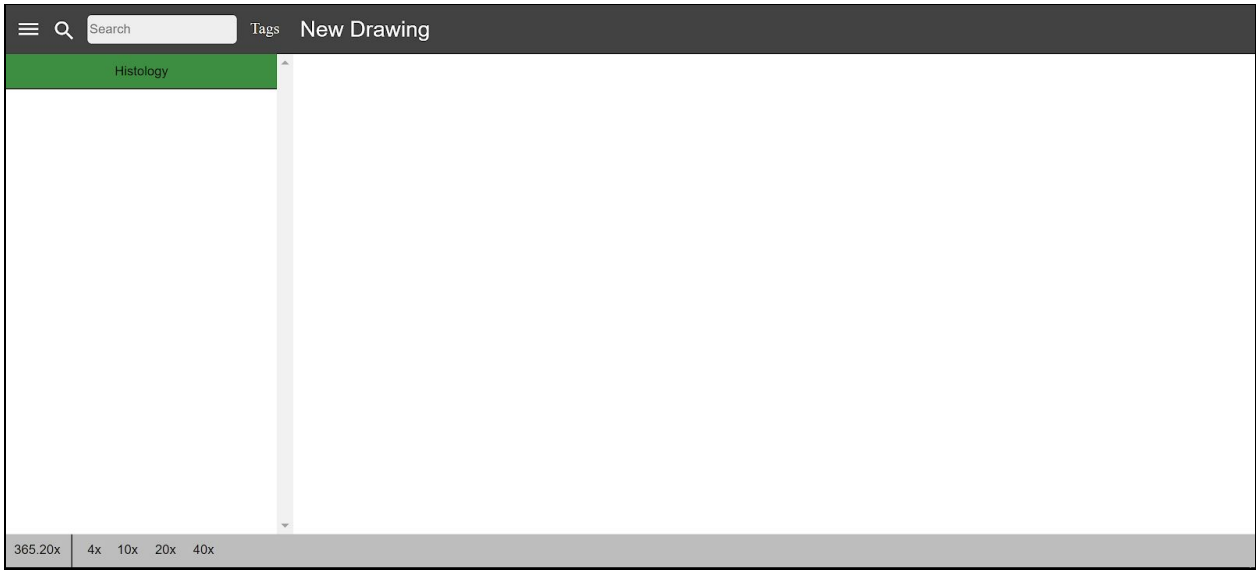


Image Text: Front page shown right after logging in.

When first logged in, the most common action would be to choose an image. So the image list (left middle section of the above image) will be visible from the start. This is an expandable list where each layer is expanded individually. The green rows are representing different folders, while the lowest layer is the actually loadable images and are shown with white background. This is to make it easy to distinguish from folders and images when traversing the imagelist.

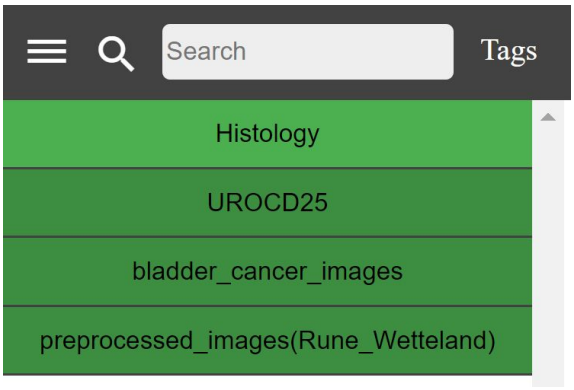


Image text: The initial image list with one layer rendered.

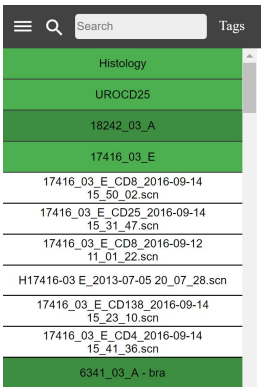


Image text: The image list expanded down to the first layer of images.

Once an image is chosen, the image will be loaded into the main part of the website, with any pre existing drawings rendered with the image.

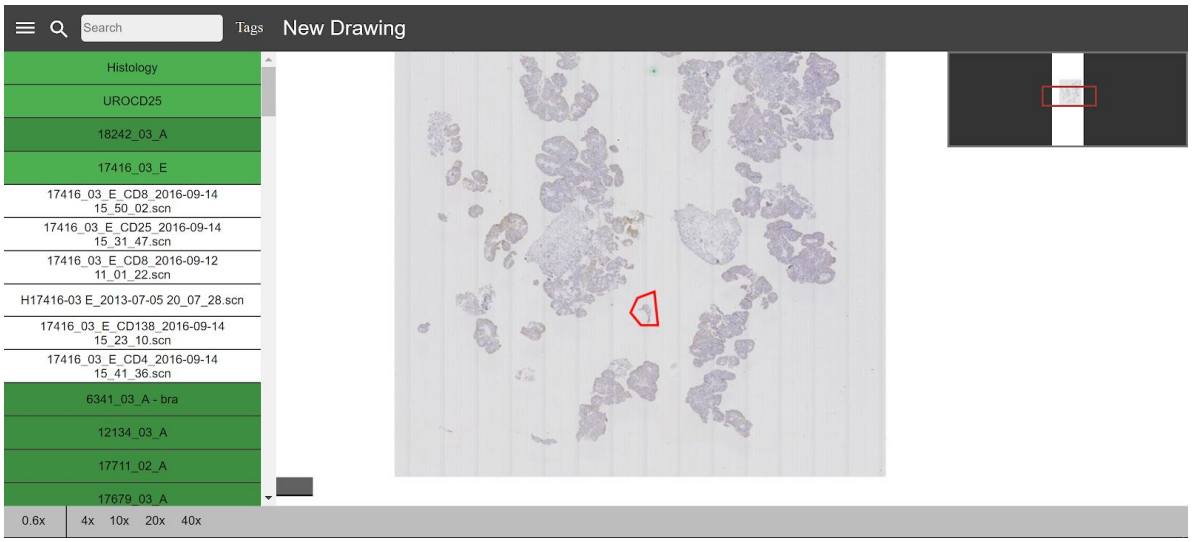


Image Text: Front page with image loaded, with an example of a drawing shown in the middle.

Image Text: Zoomed in example to a marking that also better displays zoom level and scale bar.

When zooming in, the thickness of the lines will remain consistent no matter the level. This makes the drawings look rather crude when fully zoomed out, but makes sure they are still visible. The other option would be for them to get smaller when zoomed out, which in return would make them near invisible when zoomed far out.

In the top left corner, there is a menu accessible with a couple of functions:

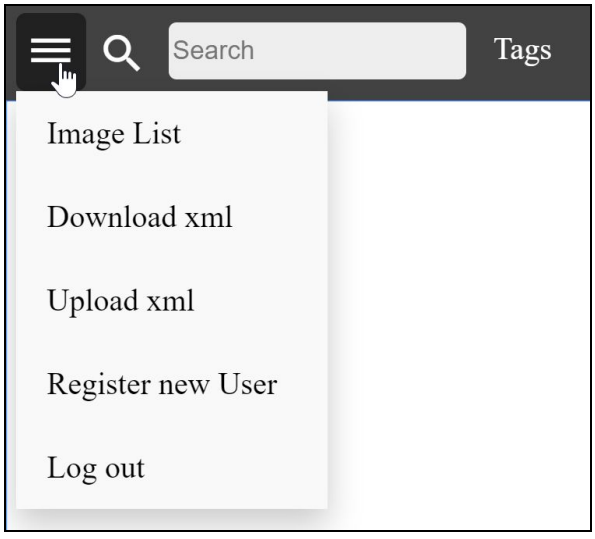


Image Text: List on the top left corner of the main page.

```
<div id="menu" class="dropdown">
   <button id="menuicon" class="material-icons menu">dehaze</button>
   <div class="dropdown-content">
       <a id="imageList">Image List</a>
       <a id="DownloadXML">Download xml</a>
       <a id="UploadXML">Upload xml</a>
       <a href="register">Register new User</a>
       <a href="logout" id="Logout">Log out</a>
   </div>
</div>
```

Html code: HTML code for the menu. Each id is used with JQuery to initiate different code/functionality.

imageList is used for hiding/showing the image list. Which is accomplished by toggling the visibility of the entire object with id *imageExplorer*. Except for the initial *imageExplorer* element, the entire HTML code for the image list is generated server side and then rendered with jinja.

```
<div id="imageExplorer">
   {{ imageList|safe }}
</div>
```

Html code: Initial HTML code for image explorer, before it is populated by jinja.

The |*safe* filter is used to tell the template engine that the content given is safe to render directly without any auto-escaping. This is necessary because browsers are unable to read HTML code which has been string escaped.

```
$("#imageList").click(function () {
   $("#imageExplorer").toggle();
});
```

Javascript code: Toggles the imageExplorer id that is the root element for the entire image list.

Download XML is rather self explanatory. Once pressed, (if an image is selected) an XML file will be downloaded with all the coordinates in a specified structure that is compatible with Aperio ImageScope. This makes it possible to download an XML locally and import it to Aperio.

The web application also supports having XML files uploaded to it from the client. It needs to be the same format as when downloaded. This means it is possible to create drawings on Aperio and upload those markings. Since the application will be used by multiple users, this makes it much easier to collaborate and share drawings.

The “register new user” button sends the user to a new page where a new user can be created. This is only possible to do if you are logged in as an admin. If a “regular” user tries to enter this page, they will be redirected to a page that tells them they are not authorized.

The log out button log the user out of the website. How this is done is further explained in 4.9.1.

As for the design aspect, the menu is normally hidden but shown on hover with the background color also changed to give the user some feedback. This method is used both for the icon it self and each row in the menu.

In the footer at the bottom left corner there are added zoom tools, the leftmost tells the current zoom level. While the 4x, 10x, 20x and 40x are buttons that can be pressed to zoom to their respective levels.

Image text: Zoom level and zoom buttons.

```
$("#zoom4x").on("click", function () {
   viewer.viewport.zoomTo(4);
});
```

JavaScript code: Event handler for the 4x zoom button. The logic is identical for every button, where only the number changes.

4.2 Database

The database is created using MySQL. It’s main functionalities are storing user information and keeping track of which images contains which tags, so that it is possible to search for images depending on specific tags. A separate table was also created to easily keep track of all tags currently used in the application.

Since the information needed on a user is exceedingly sparse, there was only a single table needed:

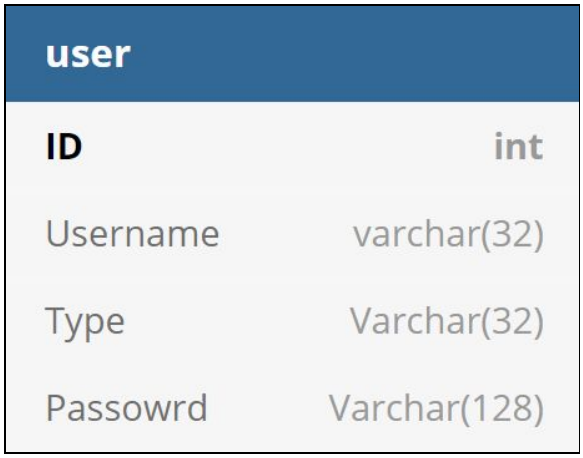


Image text: User table in DB.

The same is true for tags, as the only time this table is used is either a single insertion or a select to get all tags.

Image Text: Tags table, used to store the different types of tags.

The tables used for drawings connected to an image uses a one to many relation through the *ImagePath* column. *ImagePath* is a value combining the filename of the image and the folder the image is stored in. Why this is done is explained in 4.4.2. This way it is possible to add multiple drawings to one image. Which gives the possibility to query for every image id with specific criteria. In this instance only searching for tag name is implemented, but this way there exist a possibility to easily add more ways to search for drawings based on different criterias.

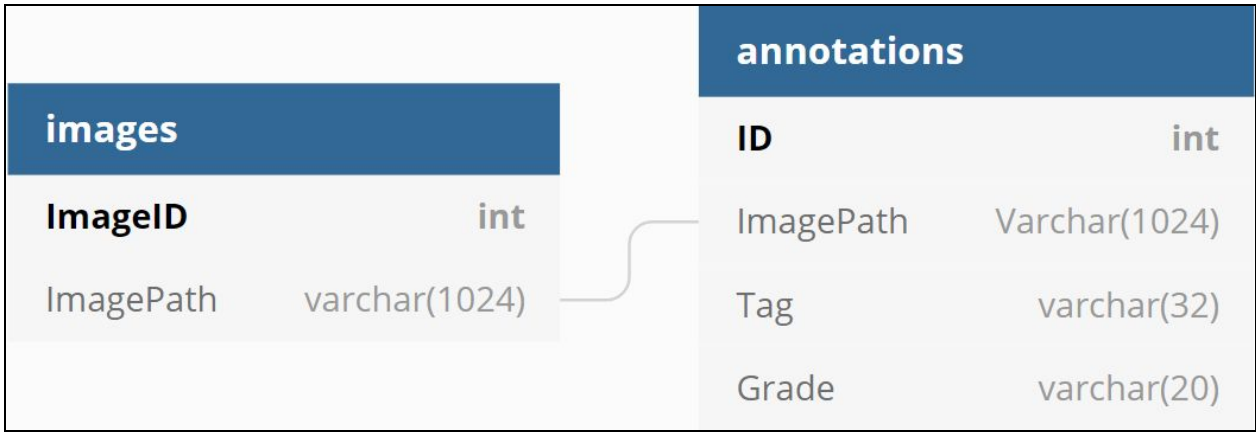


Image Text: A many to one relation between annotations and images.

4.2.1 SQLAlchemy

When connecting python to MySQL with SQLAlchemy it starts by creating a connection between the flask server and the database server. During this configuration we give it the connection string as well as username and password. *(dbUser, dbPassword, dbUrl, dbName)*. These variables are collected from a local file to add another layer of obscurity. The *dialect* was also decided (the SQL language used when doing query search), which is set to the default, MySQL.

```
app.config["SQLALCHEMY_DATABASE_URI"] = "mysql://%s:%s@%s/%s" % (dbUser,
dbPassword, dbUrl, dbName)
```

After the app is configured to connect to the correct database, a db object can be created. This is done with:

```
db = SQLAlchemy(app)
db.create_all()
```

Python code: Declaration of SQLAlchemy object and creation of db tables.

The first line is the default declaration of a SQLAlchemy object that takes the flask application as a parameter. Here the db connection with username and password is already configured, so there is no further configuration needed. The second line of code creates a model of all the tables in the given database to make it easy to interact with the different tables and column of the database. To access the data stored in the database, database models must be created.

4.2.2 Table Classes

Since SQLAlchemy is an ORM, there needs to exist a representation of the tables/models of the database. This is done by creating classes where each class represent one table in the database.

As shown in the code below, each class inherits from db.Model. This is a base class for all models in Flask-SQLAlchemy. This class defines several fields as class variables. Fields are created as instances of the db.Column class, which takes the type and other optional criteria as arguments, like primary key, if they are unique etc.

```
class Images(db.Model):
   ImageID = db.Column("ImageID", db.Integer, primary_key=True)
   ImagePath = db.Column("ImagePath", db.String)

   def __init__(self, ImageID, ImagePath):
       self.ImageID = ImageID
       self.ImagePath = ImagePath
```

Python code: Class declaration that represent annotations in the database.

It is also possible to define methods within the classes to add further functionality. As shown below in the class *User*. Because of the extra security required when storing user information, specifically passwords in this case. Password hashing and checking has been implemented as well. This makes it possible to call these methods directly when interacting with user objects other places in the application. In the login code *check_password* is used to check if the given input hashed through *set_password* match with the hashed password in the db. While in register it's used to hash new passwords.

```python
class User(UserMixin, db.Model):
    id = db.Column("ID", db.Integer, primary_key=True)
    username = db.Column("Username", db.String, )
    password = db.Column("Password", db.String)
    type = db.Column("Type", db.String)

    def __init__(self, username, password, type):
        self.username = username
        self.password = self.set_password(password)
        self.type = type

    def set_password(self, password):
        return generate_password_hash(password)

    def get_password(self):
        return self.password

    def check_password(self, password):
        return check_password_hash(self.password, password)
```

Python code: Class declaration that represent user in the db, with some extra methods for added functionality.

UserMixin is used to make user handling easier as it provides default functionality that a user is required to have. Examples would be checks to see if the user is still active, if they are anonymous and if they are authenticated.

4.3 Config/Startup

4.3.1 Server-side

4.3.1.1 configuration.py

```
def ConfigureApp(app):
   dbUser, dbPassword, dbUrl, dbName = ReadDatabaseCredentialsFromFile()
   app.config["SECRET_KEY"] = ReadSecretKeyFromFile()
   app.config["SQLALCHEMY_DATABASE_URI"] = "mysql://%s:%s@%s/%s" %
                                           (dbUser, dbPassword, dbUrl, dbName)
   app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
   app.permanent_session_lifetime = timedelta(minutes=75)
```

Python code: *Configuration.py contains functions used for initial configuration of the application.*

ReadDatabaseCredentialsFromFile() reads all details needed to connect to the database from a local file called “Login.txt”. Format of the file being read from is shown below, where “|” is used to separate the variables from each other.

```
username|password|databaseURL|databaseName
```

The text file must contain only one line, trailing empty lines will cause an error with the current implementation. This also applies to the text file read from in *ReadSecretKeyFromFile().*

ReadSecretKeyFromFile() is functionally very similar to *ReadDatabaseCredentialsFromFile(),* except it reads and returns only a single variable.

Also created in configuration.py is *LogFormat()***.** It sets the standard format of messages logged throughout the application.

ConfigureApp(app) configures a number of parameters in the referenced flask object “app”.

- *[“SECRET_KEY”]:* Set to a randomly generated key that are read from a local file. It must be set to enable use of sessions in Flask, and is used to cryptographically sign cookies.
- *[“SQLALCHEMY_DATABASE_URI”]*: Set to the connection string used to connect to a database with SQLAlchemy. See 4.2 for further details.
- *[“SQLALCHEMY_TRACK_MODIFICATIONS”]***:** set to False to disable unnecessary overhead towards the connected database.
- Finally, session expiry time is set to 75 minutes. That is, 75 minutes of inactivity. As long as the user is active, the session will be automatically renewed.

4.3.1.2 Global scope

```
nestedImageList = {}
imagePathLookupTable = {}

deepZoomList = SessionDeepzoomStorage()

logger = customLogger.StartLogging()

app = Flask(__name__)
configuration.ConfigureApp(app)

import xmlAndDB
import dbClasses
import userHandling

login_manager = LoginManager()
login_manager.init_app(app)
```

Python code: Global server variables.

All variables referenced over multiple python files are declared at the top of “app.py” as shown in the code snippet above.

nestedImageList will contain a dictionary with filenames and parent folders of all images available. Further details on this at 4.4.2.

imagePathLookupTable will contain a dictionary used to lookup an image’s path on the storage server. Further details on this at 4.4.2.

deepZoomList will contain a custom made data structure that is used to keep track of necessary session state elements that cannot be stored as a session cookie on the client-side. Further details on this at 4.6.1.5.

logger contains a logger object. See 4.9.4 for further details.

app holds the flask web-application object and is further configured by calling the *configureApp()* function with *app* as a reference.

Login_manager holds a LoginManager class object that contains all the settings used for logging in users. See 4.9.1 for more details on this.

The three imports at the bottom are placed there because they make use of the global variable *db*, and therefore have to be imported after db has been declared.

A better way to do this would probably be to instead reference *db* with each external function call that requires access to it. It would decrease the scope of the global variable and generally make the code more maintainable.

4.3.1.3 main

```
if __name__ == '__main__':
   app.run(host="0.0.0.0", port=8082, threaded=True)
```

The python main method is short and simple. All it does is to start the flask web-server by calling the *run()* method on *app*, with parameters specifying how the server should be accessed. In this case the server is reachable on 0.0.0.0 at port 8082. Setting the host address to 0.0.0.0 means that it will be reachable on the ip address(es) the machine has been allocated. This has been deemed not a security risk as all traffic is routed through a seperate apache server. Whose security is managed by the university's it-staff.

During testing, it was discovered that the main bottleneck of this application is the amount of requests the server can handle per second. One way we increased request throughput was to set the application to run multi-threaded. Because Flask has built in support for multi-threading, this is accomplished by simply setting the threaded parameter to true in the app.run method.

When testing the difference, it was discovered that the average rate of requests per second when running it on a single core was 53.5. With multithreading enabled, the average increased to 119 requests per second.

```
@app.route('/')
@login_required
def Main():
   GetAvailableImages()
   ImageListHTML = GenerateImageListHtml()
   return render_template("index.html", imageList=ImageListHTML)
```

The main python web-route is configured with a "login_required" flag. This will trigger a check to see if the client is authenticated. When the check is completed, one of two things will happen.

1: The client is not authenticated and will be redirected to the login page and asked to log in. On successful login, the client will be redirected back to main route.

2: The client is authenticated and allowed access to the function.

The function itself runs GetAvailableImages() to get a list of all the images available to the client, generates HTML code for displaying it graphically and returns the index page along with the image list HTML code.

Deeper explanation on *GetAvailableImages()* and *GenerateImageListHtlm()* can be found here 4.4.2 and here 4.4.3.

4.3.2 Client-side

```
$(document).ready(function () {
   updateAllTags(0);
   initializeCanvas();
   addGuiHandlers();
   addXmlHandlers();
   addDrawingHandlers()
});
```

Whenever a client accesses the application there needs to be added logic and data to different parts of the website. Primarily event handlers and data from the database. To ensure that everything in the website is properly loaded before anything is added, jquery's document.ready is used. This ensures that only once every element is properly loaded, the following functions are called:

updateAllTags(): Fetches all tags from the database and places them in a global variable called *allTags*. These need to be fetched before a user can use the tag-based search feature.

initializeCanvas(): An OpenSeadragon viewer element is used to display images to the user. This function creates the viewer element, adds necessary overlays on top of the viewer and sets event handlers associated with the viewer element.

addGuiHandlers(): Adds event handlers that has to do with user interactions with GUI elements and handlers that not fit into any of the other more defined categories.

addXmlHandlers(): Adds event handlers related to XML features.

addDrawingHandlers(): Adds event handlers related to drawing features.

4.4 Image List

Given the large number of images, users need a way to be able to choose graphically between all these images. The first idea to solve this was to create a seperate window that acts like a file explorer. Much like windows' own solution when asked to pick a file. But this would require quite a bit of work and it was not even sure it would be the most optimal solution. So instead a simpler but equally efficient method in the form of a layered list was created.

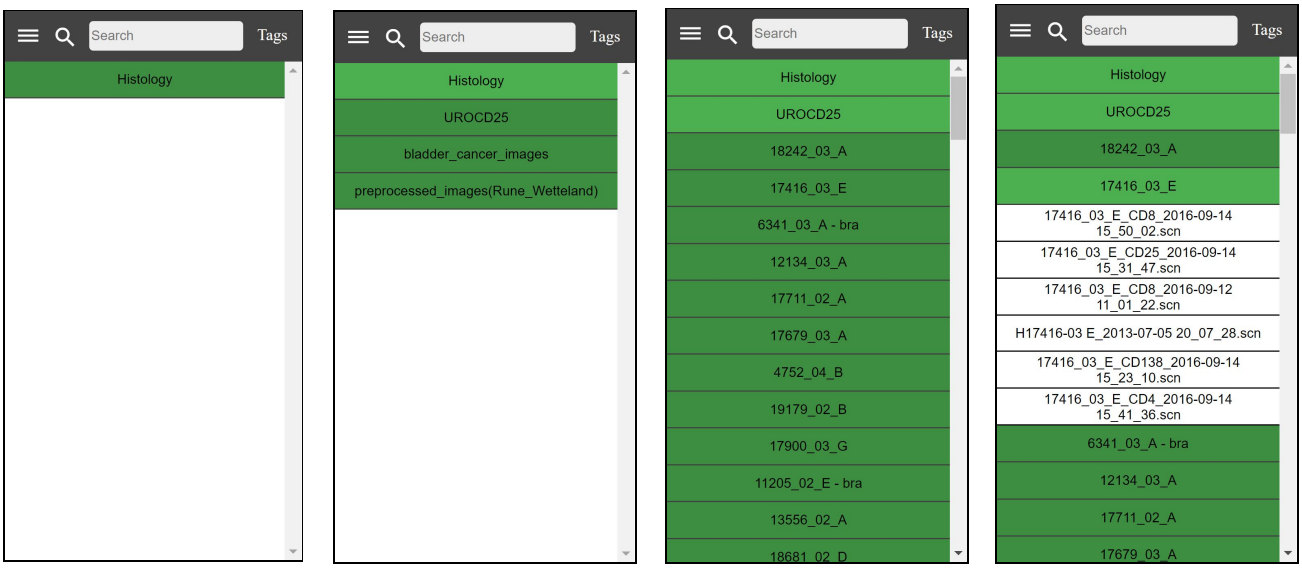


Image text: Progression on how the layered image list works by going one layer further down for each image.

In the images above, one can see that there are two different types of layers, the first type is colored green while the other is white. The green layers represents folders, which means that when clicking on them, they will expand either a new layer of folders, or a layer of images. Because the white layer represents images, clicking on one of the white rows, the image with the same name will be displayed.

Given the large size the image list uses on the site, a button in the top left menu was added to be able to hide the list. This code is rather simplistic as it's only an event handler that toggles the visibility of the image list.

The HTML code is structured the same way the folder list is shown. With different divs representing folders being given classes that represent this. And images being given similar classes for their appearance.

```
<div class="folder" style="display: none;">
   <button class="folderButtons" style="background-color: rgb(62, 142, 65);">
                                                        UROCD25</button>
      <div class="folder" style="display: none;">
         <button class="folderButtons" style="background-color: rgb(62, 142, 65);">
                                                        18242_03_A</button>
         <button class="imageLinks"
                 id="18242_03_A/18242_03_A_CD138_2016-08-11{space}21_07_24.scn"
                 style="display: none;">18242_03_A_CD138_2016-08-11 21_07_24.scn
          </button>
      </div>
</div>
```

HTML Code: Example code of the HTML structure used in the image List.

The folders work by having an event handler appended to every single one. And when they are clicked, their state is toggled, so if they are expanded, they will be minimized, and if they are minimized they will be expanded. That row's color will also be changed, where a lighter green means the row is currently expanded, while the darker green means it's minimized. Which makes it easier for the user to see what folders are expanded and not.

A problem that arose was something called event propagation. This means that if an element contains an event handler, for example, click. If that same element also contains a parent node that has the same event handler, both the clicked object, as well as the parent will be triggered.

For the image list this posed a large problem as there could exist multiple layers of propagation. Luckily there is a built in solution to this in JavaScript. One simply adds *event.stopPropagation()* inside any eventhandler where the previously described problem arise. A happy coincidence that occured when putting *stopPropagation()* in the *.folderButtons* event handler is that *folderButtons* exist in every layer, so there was no need to add the code any other place.

```
$(".folderButtons").click(function () {
   event.stopPropagation();
});
```

JavaScript code: Using event.stopPropagation() in any event handler stops any eventhandler in parent elements to trigger.

Just like the folder buttons, there is added an event handler to every single image, these will similarly be triggered when clicked, but instead of changing anything in the image list, an image is loaded by calling the method *changeImage*. And for safety, a check was also added to ask the user if they want to change image if the number of drawing points is not 0, as that indicates a drawing is in progress and changing image will cancel the drawing.

```
$(".imageLinks").on("click", function () {
    if (tempDrawingPoints.length === 0) {
        changeImage(this);
    } else if (confirm("Changing image will cancel drawing, continue?")) {
        changeImage(this);
    }
});
```

JavaScript code: When an image is clicked, changeImage is called if a drawing is not in progress.

4.4.1 Refreshing the image list

The image list needs to be refreshed manually by running a script provided with the project files.

How to run the script: Run the command “python3 updateImagelist.py” from within the project folder in a command line interface.

```
import imageList
imageList.RefreshImageList()
```

Python code: updateImagelist.py

```
def RefreshImageList():
   try:
       listOfImages = glob.glob("//home/prosjekt/**/*.scn", recursive=True)
       strippedListOfImages = stripBeginningOfPaths(listOfImages)
       with open('ImageList.txt', 'w') as f:
           for item in strippedListOfImages:
               f.write("%s\n" % item)
   except OSError:
       print(“Error writing file”)
```

Python code: Recursively searches for .scn files in the “//home/prosjekt/” folder and writes their paths to a text file.

Glob is a module in python for unix style pathname pattern expansion. In this function it is used to perform a recursive search for files with the “.scn” filename ending. Starting the search from “//home/prosjekt” and returning a list with the full file path for each file matching the search criteria.

While testing, this search took between 20 seconds and 3 minutes to complete, which is a significant amount of time. The large difference in search time is most likely due to caching. As it

took 3 minutes the first time the scan was run and the information was then most likely not cached, while the consecutive searches were much faster. Likely because the information was cached from earlier searches. To minimize the frequency of this having to be done, the result of the search is written to the file ImageList.txt. This makes it possible to efficiently populating the list of images without running the search every time.

Before writing the list of paths to file, the recurring head of each path (“//home/prosjekt”) is stripped away in *stripBeginningOfPaths()*.

```
def stripBeginningOfPaths(list):
   foo = []
   for element in list:
        foo.append(element.replace("//home/prosjekt", ""))
   return foo
```

Python code: Takes a list of paths, and removes the head of each path, “//home/prosjekt”.

4.4.2 Reading from ImageList.txt

As mentioned, ImageList.txt is used to populate the imagelist in the webpage. So every time the server starts, it calls *GetAvailableImages(),* which reads the document and creates two global variables:

imagePathLookupTable: By creating a list from ImageList.txt and then sending it through *ImageListToDict().* Here it goes through each image path, splits it on “/”, rejoins the last two indexes after the split and appends the paths to a dictionary with the joined last indexes as keys.

Example:

```
["Histology/UROC25/2006/test/image.scn"]
                    ↓
{"test/image.scn": "Histology/UROC25/2006/test/image.scn"}
```

This is done to be able to have unique identifiers for the images on the frontend, while revealing as little as possible about the backend file structure. The reason we need to use both a folder and file name as key is for the existence of duplicate images located elsewhere in the project directory.

nestedImageList. By sending *imagePathLookupTable* through the recursive function *BuildNested(). nestedImageList* is created to be structurally the same as the layered image list described in 4.4 as to make it easier to generate HTML code from it.

Example:

```
{"imagefolder/image1.scn": "Histology/UROC/2010/imagefolder/image1.scn",
"imagefolder/image2.scn": "Histology/UROC/2009/imagefolder/image2.scn",
"imagefolder/image3.scn": "Histology/UROC/2010/imagefolder/image3.scn"}
```

Would be converted to

```
{"Histology":
    {"UROC":{
        "2010":{
            "imagefolder":[
                "image1.scn",
                "image3.scn"
            ]
        },
        "2009":{
            "imagefolder":[
                "image1.scn"
            ]
        }
    }
    }
}
```

Conversion to the nested structure is done because this structure is easier to work with when generating HTML code for the image list.

4.4.3 Generating image list HTML code

Once the nested structure is created, the generation of the HTML code can commence. By sending *nestedImageList* into a recursive function for generating the HTML code, similarly to the *BuildNested()* method. Once generated, the HTML page is rendered with the image list through jinja.

```
return render_template("index.html", imageList=ImageListHTML)
```

Python code: Once the HTML code is generated, it's put in the variable ImageListHTML which is then sent to the front end.

```
<div id="imageExplorer">
   {{ imageList|safe }}
</div>
```

HTML code: The HTML code sent from the server is rendered here by Jinja.

4.5 Search

As explained in 4.4, given that there are 1100 images, displaying every single one at once is not feasible. Another problem is actually navigating through all the images. If one knows which folder the wanted image is located in, that is fine. But having to know the location of every single image is not realistic. The best solution was to implement search functionality, both for image name, and for images that contains a drawing which has a specified tag attached to it.

As default when a user uses the search bar, the application will search for image names. If they want to search for tags, all that is needed is press the “Tags” button right next to the search field.

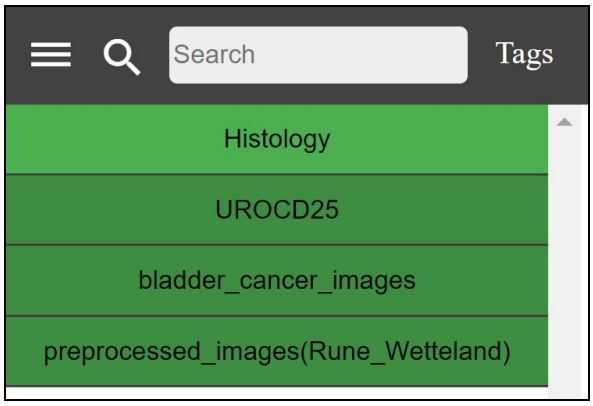


Image text: Default look where tags is not enabled and nothing is searched for.

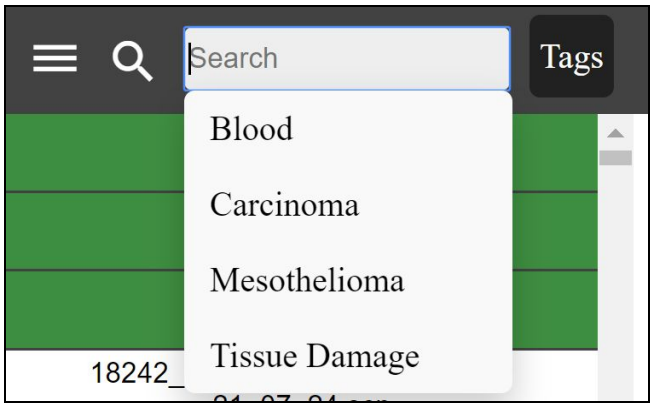


Image text: When tags is enabled, clicking on the search bar displays all tags in the db and can be selected for searching.

The HTML code for the search field consists of a div within the header list and contains the search field as well as the button (*div id=“searchTags”*) that decides whether the search should be by tags, or by image name.

As shown in the last two images, when the tags button is enabled, the general look of the button is changed to simulate it being pressed in. A list containing all tags will also appear when clicking on the search field.

When the search field is clicked, independent of whether the “Tags” button is enabled or not, every row in the image list will be expanded. This is done because it makes it visually easier to see which images are filtered and not. It also seemed logical, since when there is nothing in the search field, nothing is filtered and every image is a match. *populateSearchDropDown()* is also called, but this time only if the user is currently searching by tags.

The general ui changes are done by changing the class and id on the HTML element for “Tags”. Where either the class or the id decides the look of the button. (Being pressed in or not).

```
<div id="searchTags" class="">Tags</div>
```

```
<div id="" class="searchTagsClicked">Tags</div>
```

HTML code: When the tags button is clicked, the HTML code is changed so that the appearance of the button simulates it being pressed in.

While the addition of the drop down menu that appears are done by populating a div with every tag in the database. As shown in the code below, every tag in the global variable *allTags*, explained here, is appended to the HTML element with the class *dropdown-search-content* to act as a separate button. This makes searching for tags much easier as there is no need to know what type of cancer cells have been/are being worked on, since the application tells the user.

There is also added a new eventhandler on each item in the dropdown list which, when clicked automatically enters the search field with the tag clicked. And then calls the *fetchSearchTags* function with the relevant tag value. *fetchSearchTag* is explained in 4.5.2.

```
function populateSearchDropDown(){
   var searchDropdown = $(".dropdown-search-content");
   if ($(".dropdown-search-content a").length === 0) {
       allTags.forEach(function (tag) {
           searchDropdown.append("<a class='classTags'>" + tag + "</a>");
       });
   }
   $(".dropdown-search-content a").on("click", function () {
       $("#searchField").val($(this).html());
       $(".dropdown-search-content").empty();
       fetchSearchTags($(this).html());
   });
}
```

JavaScript code: Populates the dropdown menu under search when tag search is enabled. Then after created the list, adds an event handler on each to trigger search on the clicked word.

A handler for when the “Tags” button is pressed was added. Depending on its current state, two different things will happen. If the button is currently pressed down (*searchTagEnabled* is true), it will empty any information inside the search field, as well as the drop down menu under the searchfield. It will also remove its current class and add a new id to the HTML element. This is to change its appearance from being pressed down, to be in a more “neutral” state. Last but not least, it will set the *searchTagEnabled* to false, which removes the logic that searches for tags, and returns to regular search based on file name. *seachTagEnabled* is a global variable that is used to keep track of whether the “Tags” button is enabled or not.

If the button is not currently pressed down (*searchTagEnabled* is false), it will add the class that would otherwise be removed, and remove the id that would otherwise be added. Additionally it will set the *searchTagEnabled* variable to true, to change the logic to support tag search.

In the search field text box a handler was created that is triggered every time a key is released while the *searchField* is in focus. “On keyup” was the chosen trigger as it gives a continuous feedback while searching for images, as the number of images shown is reduced/increased on every letter added/deleted.

When the event is triggered, the following logic is dependent on whether *searchTagEnabled* is true or false. If searchTagEnabled is true, the search will only be started by pressing the enter button. This is to limit the amount queries done to the database. And thanks to the dropdown list that appears when *searchTagEnabled* is true, the need to manually type the tags is obviated.

```
$("#searchField").on("keyup", function (e) {
        var imageList = $(".imageLinks").toArray();
        var searchValue = $(this).val();
        if (!searchTagEnabled) {
            filterImages(imageList, searchValue);
        } else if (e.key === "Enter") {
            fetchSearchTags(searchValue);
        }
    });
```

JavaScript code: Event handler that triggers on every “keyup” in the field, but only initiate tag search if the enter key is the one pressed.

4.5.1 Images

As shown in the previous code block, if *searchTagEnabled* is not true, *filterImages* is called.

filterImages takes in two parameters, *imageList* and *searchValue*. *imageList* is an array of every single image described in 4.4, created by getting every element with class *.imageLinks, as* shown in the previous code block. And *searchValue* is the current text in the search field.

The function iterates through every image in *imageList*. For each image it checks whether the current *searchValue* exists in the element. If it doesn't exist, hide the element, if it does, show it. For example: we have an image with name "H18242-03". If we type H182 that image will be shown. If we type 8242 the image will still be shown, this is because 8242 exists within H18242.

The reason *.show* is used even though every element is shown when we begin to search is that if the user makes an error while typing, for example if they type "H1242" instead of "H18242". The wanted image will be hidden. But as soon as the user starts deleting, images that was hidden because they did not contain H1242 will now be shown again.

```
function filterImages(imageList, searchValue) {
  imageList.forEach(function (element) {
        if (!element.innerHTML.includes(searchValue)) {
            $(element).hide();
        }
        if (element.innerHTML.includes(searchValue)) {
            $(element).show();
        }
   });
}
```

JavaScript code: Code that filters while searching for image names.

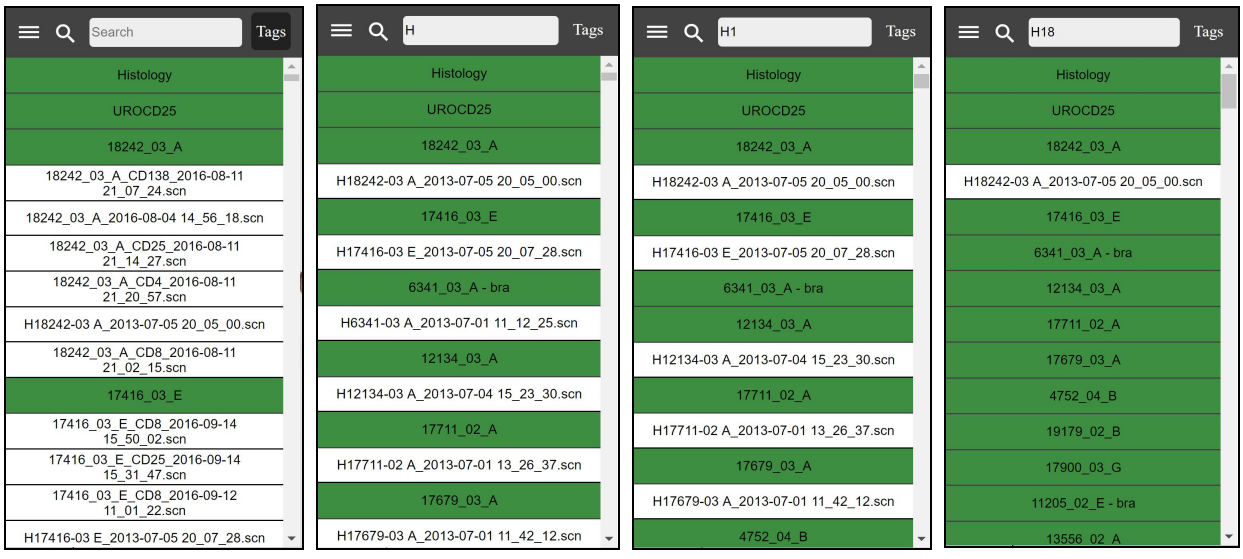


Image text: Example of image list being continuously updated as the user keeps typing into the search field.

4.5.2 Tags

There are a fundamental difference when it comes to searching after images depending on if one is searching by file name, or if one is searching by tags. This is because how different the access path of the data being searched for is. As shown, when searching for image names every single image is already loaded into the client in a list that can be iterated through.

When it comes to images with markings containing specific tags, there is no information already loaded to the client. This means that the client needs to request the necessary information from the server.

When *searchTagEnabled* is true, the function *fetchSearchTags* is called and a request is sent to the server with url ending in *searchTags,* to perform a query against the database. The tag that are being searched for is sent with the request so that the query is customized for the specified tag.

```
@app.route("/searchTags", methods=["POST"])
```

The server responds with a dictionary of all the images found, which is sent forward to *filterImagesByTag* where the filtering logic is.

On the backend, the tag sent with the POST request is saved to the variable *tag* and then added to the *queryString.* The *query* variable is the result returned from the database after executing *queryString*. But this is returned as a SQLAlchemy matrix object that is not a supported format to send over the web. To solve this, the object is iterated through, where only the first element in each sublist is retrieved, as that is the relevant information and the rest is metadata. The retrieved data is appended to a list, which is then again added to a JSON object. The information is now serializable and can be returned.

```
tag = json.loads(request.data)["tag"]
query = db.engine.execute("select i.ImagePath from images as i inner join
annotations a on (i.ImagePath = a.ImagePath) where a.tag = '{}';".format(tag))
```

Python Code: The query executed to get every image with the relevant tag.

In *filterImagesByTag() (shown below),* a double for loop is used, and each element in *imageLinks* are compared to each element in *dbResult*. Where *imageLinks* is every image in the image list just like in *filterImages*. And *dbResult* is the list of every image found in the database which have a drawing with the tag in it.

But before the comparison can be done, each element in *dbResult* needs to be formatted to match the format of *imageLinks*. These are different because of limitations on character usage in web, and linux pathing. The first change done is by changing every space (“ “) into a custom string “{string}”, this is because the encoding used by HTML does not support regular space and would automatically change spaces to “%20”. To have total control over the text, any spaces used in the image list on the webpage is converted to “{space}” instead.

As explained in 4.4.2 there are a some duplicate images that posed as a problem when the images were originally displayed on the webpage. As such, the structure of the XML files was changed from only containing the name of the file, to both the file and the folder it was contained in. This made the names unique even if the same image existed in another folder. Here, another limitation arose, this time the allowed naming convention in linux. The most natural name to give the XML files was “folder/filename.xml”, but linux does not allow for some special characters, like “/” to be used in the file name. A similar solution as for spaces on the web application was used. “/” was changed to “[slash]” and once this was done, the format was the same and the comparisons would yield accurate results.

The decision to hide or show an image is done by comparing every image in the list of results from the database, with every image in the *imageList*. If a match is found, set the match variable to true. Then after every image in the database list is checked, see if there was a match found. If match is true, show the current image, and else, hide it. Then move to the next image in the *imageList*, and compare every image in the database list with this new image.

```
function filterImagesByTag(dbResult) {
   var imageList = $(".imageLinks").toArray();

   imageList.forEach(function (imageListElement) {
       var match = false;
       dbResult.forEach(function (dbElement) {
           var foo = dbElement.replace(new RegExp(" ", "g"), "{space}");
           var element = foo.replace("[slash]", "/");
           if(imageListElement.id === element + ".scn"){
               match = true;
           }
       });
       if(match){
           $(imageListElement).show();
       }else{
           $(imageListElement).hide();
       }
       match = false;
   })
}
```

JavaScript code: Iterates through every image and shows/hides the image depending on the result from the database.

4.6 OpenSeadragon

Image text: Visual representation of an OpenSeadragon viewer with navigator in the upper right corner, scale bar in the lower left corner and an overlay to display drawings on top of pictures.

4.6.1 Viewer

```
function initializeCanvas() {

   if (viewer) {
       viewer.close();
       $("#display").text("");
   }

   viewer = OpenSeadragon({
       id: "display",
       zoomPerScroll: 1.10,
       animationTime: 0.5,
       showNavigationControl: false,

       navigatorId: "",
       showNavigator: true,
   });

   addOverlays();
   addViewerHandlers();
}
```

JavaScript Code: Initialize canvas function.

The *initializeCanvas()* function takes care of all the initial setup related to the OpenSeadragon viewer element that is used to display images to the user.

Prior to constructing the viewer, it checks if a viewer already exists. If it does, the viewer is closed before constructing a new viewer with the *OpenSeadragon({options...})* function.

Id: The HTML element id where the viewer should be placed. In this case it is placed inside an empty tag with id “display”.

zoomPerScroll: Sets the distance to zoom per mouse scroll. Default is 1.20 and setting it to 1.00 will effectively disable the scroll to zoom feature. It was set to 1.10 to slow it down, as the default value made it quicker than what seemed practical.

animationTime: Sets the duration of animations that occur when zooming and dragging the image. Set lower than the default time of 1.2 to make the animations quicker.

showNavigationControl: Sets whether or not to show navigation buttons that can be used to navigate the image.

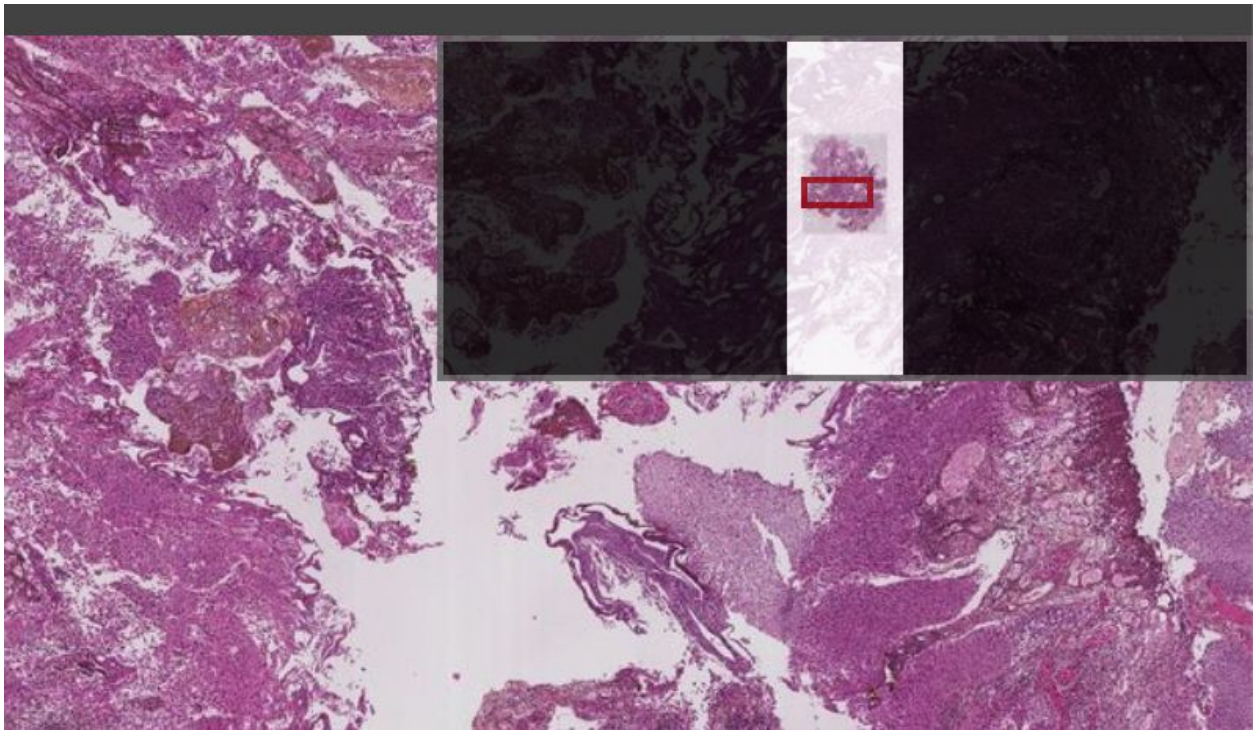


Image text: Image shows navigator overlaying the main image. Upper right corner of the image.

Navigator functionality is included with the OpenSeadragon library. It is activated by setting the *showNavigator* option to true when constructing the viewer object. Setting *navigatorID* to empty string forces default settings on the navigator.

The *addOverlays()* function constructs and attaches a scale bar and an HTML canvas overlay to the viewer object.

The *addViewerHandlers()* function adds event handlers related to the viewer object.

4.6.1.1 OpenSeadragon-scalebar

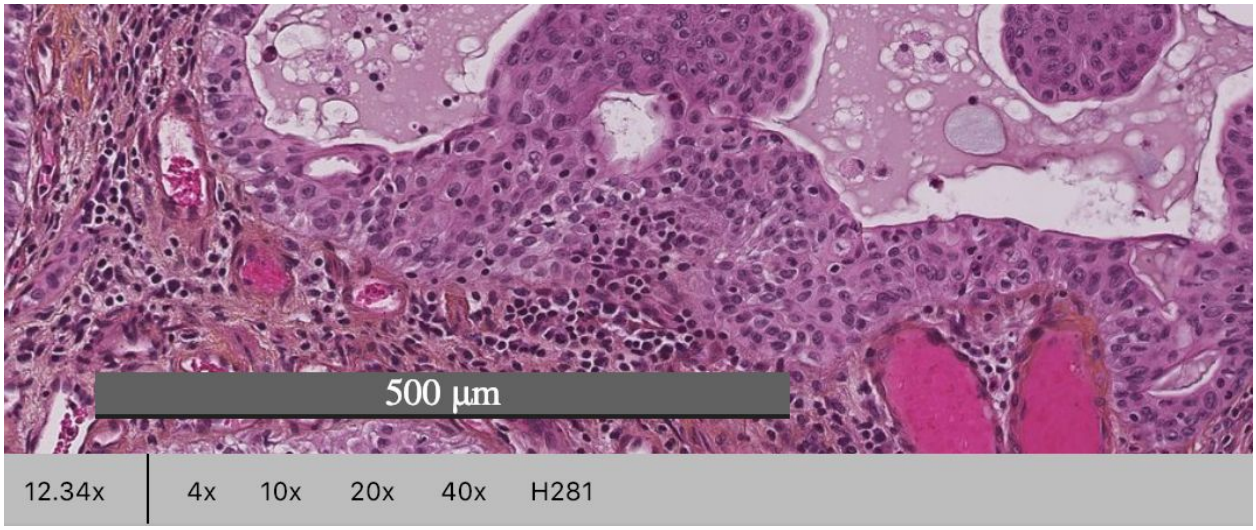


Image text: Visual representation of the scale bar on top of the main image.

```
viewer.scalebar({
   stayInsideImage: false,
   backgroundColor: "#616161",
   fontColor: "white",
   color: "#212121",

   xOffset: 45,
   yOffset: 15,
   maxWidth: 0.18,
   pixelsPerMeter: 4000000
});
```

JavaScript code: Scalebar constructor.

With the scale bar plugin for Openseadragon installed, it is pretty simple to configure and attach a scale bar to the viewer window. After the viewer has been created, a scale bar can be added with the viewer.scalebar({ ..Options}) function.

stayInsideImage: Set to false to make its position within the viewer constant. If set to true, the position will be relative to the loaded image.

backgroundColor: Sets the background color of the scale bar.

fontColor: Sets the color of text in the scale bar.

Color: Sets the color of the bottom solid line of the scale bar.

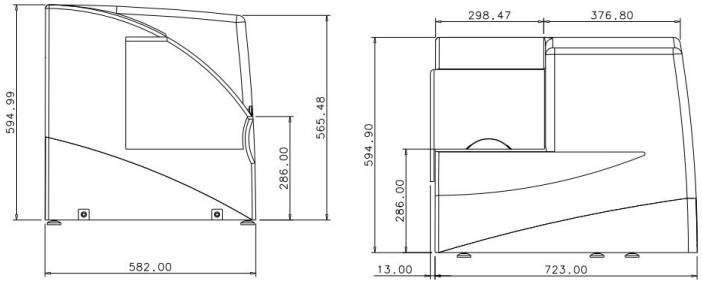
xOffset: Offsets the scale bar's position along the x-axis(horizontal).

yOffset: Offsets the scale bars position along the y-axis(vertical).

maxWidth: As the zoom-level changes, the scale bar will scale with it and wary in width. This parameter limits the maximum length the bar can have.

pixelsPerMeter: Number of pixels per meter on the image must be set for the scale bar to be able to show accurate measurements. Correct value for this parameter has been derived from the Leica scanners specifications(Shown below).

Leica SCN400
Specifications



Slide loading	4 slides in a carrier, scanned automatically	
Scanning speed	– 20x magnification, 15 x 15 mm – 40x magnification, 15 x 15 mm	– 100 seconds – 220 seconds
Glass slide dimensions	Standard size	26 x 76 mm (thickness 0.9 to 1.2 mm including coverglass)
Scanning range (for high resolution)	Standard size	24 x 62 mm
Objective lens	Optical system with Leica lenses and optical components specifically designed for digital sensor scanning	
Available magnifications	– 5x – 10x – 20x – 40x	
Spatial resolution	0.25 µm/pixel (40x magnification)	
Scanning method	– High speed, high sensitivity linear CCD device, – User selectable (automatic, semi-automatic, manual scanning)	

Image text: Image of the leica scanner specs. It provides the spatial resolution of 0,25µm/pixel(@40x magnification) used to correctly scale the scale bar in accordance with the real measurements of the slide.[42]

$$\frac{1}{\frac{0.25\mu m}{pixel}} = 4\,000\,000\,\frac{pixels}{m}$$

Image text: Formula converting µm per pixel to pixels per meter.

4.6.1.2 OpenSeadragon-canvas overlay

```
canvasOverlay = viewer.canvasOverlay({
   onRedraw: function () {updateDrawings();},
   clearBeforeRedraw: true
});
```

Javascript code: HTML canvas overlay constructor

The canvas overlay may seem simple on first glance, but there is a considerable and delicate process happening behind the scenes. The constructor needs two options to be set. A *onRedraw* callback function must be defined and whether or not the entire canvas should be cleared before every redraw callback.

The *onRedraw* callback function runs *updateDrawings()* and is called whenever the user changes their view of the image and is basically what makes the content of the canvas scale with the image.

```
function updateDrawings() {
   if (currentImageLoaded) {
       canvasOverlay.context2d().strokeStyle = "rgba(255,0,0,1)";
       canvasOverlay.context2d().fillStyle = "rgba(255,0,0,1)";
       canvasOverlay.context2d().lineWidth = 200 / viewer.viewport.getZoom(true);

       if (tempDrawingPoints.length > 0) {
           if (tempDrawingPoints.length === 1) {
               canvasOverlay.context2d().beginPath();
               canvasOverlay.context2d().arc(tempDrawingPoints[0].x,
                                   tempDrawingPoints[0].y, 350 /
                                   viewer.viewport.getZoom(true), 0, 2 * Math.PI);
               canvasOverlay.context2d().fill();
               canvasOverlay.context2d().closePath();
           } else {
               drawDrawings(tempDrawingPoints);
           }
       }
       drawings.forEach(function (element) {
           var points = element.points;
           drawDrawings(points);
       });
   }
}

function drawDrawings(points) {
   canvasOverlay.context2d().beginPath();
   for (var i = 0; i < points.length; i++) {
       if (i === 0) {
           canvasOverlay.context2d().moveTo(points[i].x, points[i].y);
       } else if (i === points.length - 1) {
           canvasOverlay.context2d().lineTo(points[i].x, points[i].y);
           canvasOverlay.context2d().stroke();
           canvasOverlay.context2d().closePath();
       } else {
           canvasOverlay.context2d().lineTo(points[i].x, points[i].y);
       }
   }
}
```

JavaScript code: These two functions together takes care of drawing all drawings onto the canvas overlay.

All finished drawings that belongs to the currently displayed image are stored in a global variable, *drawings*. There can also be an unfinished drawing, one that is currently being drawn and therefore still temporary. The points related to the drawing currently being drawn are stored in a second global variable, *tempDrawingPoints*.

updateDrawings() draws each drawing from *drawings* onto the canvas overlay, as well as the temporary unsaved drawing.this is done by first setting *strokestyle*, *fillstyle* and *linewidth* of the lines being drawn. *Strokestyle* holds the color of lines, *fillstyle* holds the color of fillable objects(e.g. squares, triangles and circles), and *linewidth* holds the width of drawn lines in image pixels(not screen pixels).

The value of *linewidth* is determined by a statically set number divided by the current zoom level. This way of scaling the size works fine when the resolution of all images are around the same size. Big variations in image resolution will cause line widths to appear unnaturally large or small on some images. Images displayed with this application today are mostly similar in resolution, though there are some exceptions. A better way to do this scaling would be to divide the current zoom level on a value based on the displayed image's resolution.

If *tempDrawingPoints* contains only one point, that point will be drawn as a circle. Only under this specific circumstance will circles be drawn onto the canvas. Every other circumstance will have lines drawn between each drawings points.

The canvas drawing procedure must be done in a specific order to avoid unwanted graphical issues.

```
canvasOverlay.context2d().beginPath();
canvasOverlay.context2d().arc(tempDrawingPoints[0].x, tempDrawingPoints[0].y,
                              350 / viewer.viewport.getZoom(true), 0, 2 * Math.PI);
canvasOverlay.context2d().fill();
canvasOverlay.context2d().closePath();
```

JavaScript code: snippet from updateDrawings(). Showing the procedure of drawing a circle.

beginPath() and *closePath()* must always be called on the context of *canvasOverlay* to specify when a drawing starts and when it ends. If these are not called, all drawing done will keep piling up in memory and be redrawn on every redraw callback, effectively causing the canvas to not be cleared before calling redraw.

In between *beginpath()* and *closePath()*, creation of elements to be drawn and the actual drawing happens. In this example, a circle is created using *.arc()*, *.fill()* is then called to, well, fill the canvas with the element(s) that were just created.

The procedure with lines are a bit different:

```
canvasOverlay.context2d().beginPath();
for (var i = 0; i < points.length; i++) {
   if (i === 0) {
       canvasOverlay.context2d().moveTo(points[i].x, points[i].y);
   } else if (i === points.length - 1) {
       canvasOverlay.context2d().lineTo(points[i].x, points[i].y);
       canvasOverlay.context2d().stroke();
       canvasOverlay.context2d().closePath();
   } else {
       canvasOverlay.context2d().lineTo(points[i].x, points[i].y);
   }
}
```

JavaScript code: snippet from updateDrawings(). Code for creating a line-based drawing.

The procedure starts again with a *beginPath()* call, then uses a for loop to iterate through each point of a drawing. For the first and last point of a drawing, there are special cases.

First point *(i === 0): The .moveTo(x, y*) function is called, to specify at which coordinates the drawing starts.

Points in between the first and last: Creates a new line with *.lineTo(x, y)*.

Last point*(i === points.length - 1):* The last line is created with *.lineTo(x, y)*, all of the lines created for the current drawing are put onto the canvas by calling *.stroke()*, and lastly the current drawing is marked as done with a *.closePath()* function call.

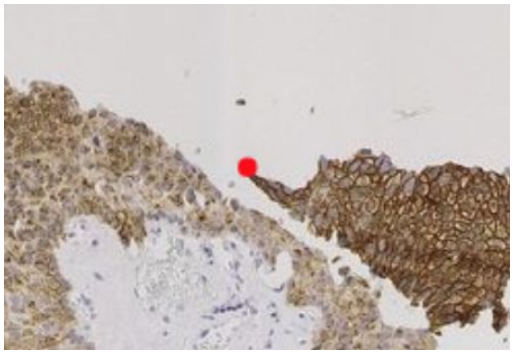


Image text: Shows a temporary drawing containing only one point. A circle is drawn when only one point is present to show the user where the first point was placed.

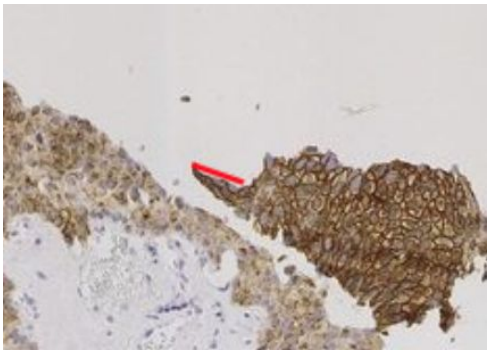


Image text: Shows the same drawing again with a second point placed.

Examples of what will happen when paths are not properly begun and closed as mentioned above:

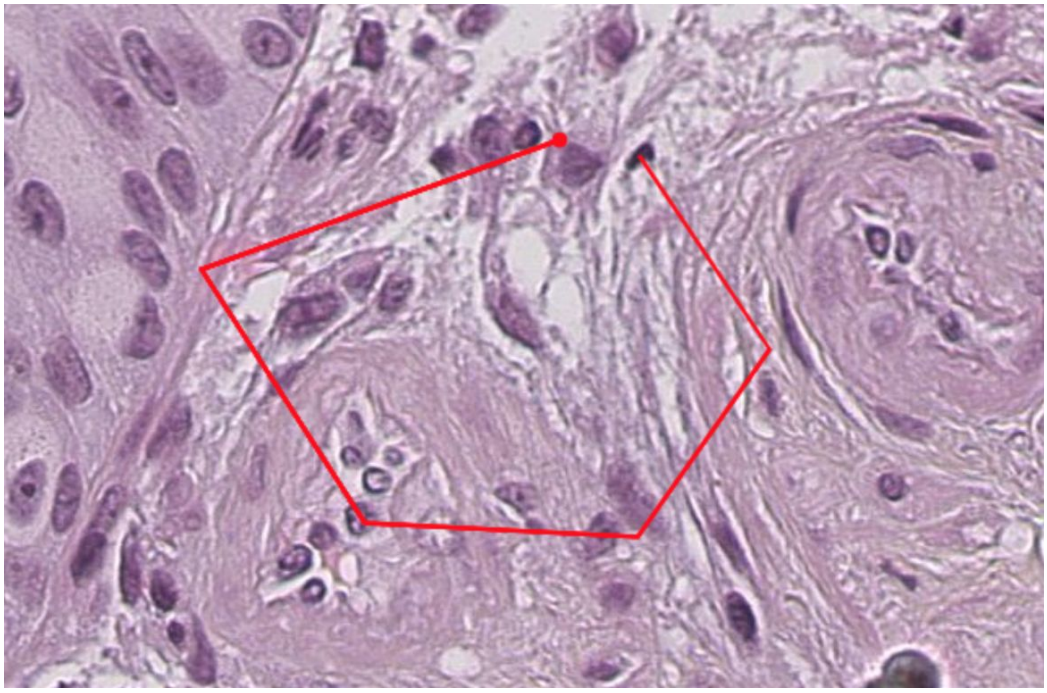


Image text: Shows a drawing on top of an image where it has not been specified in code where the path begins and ends.

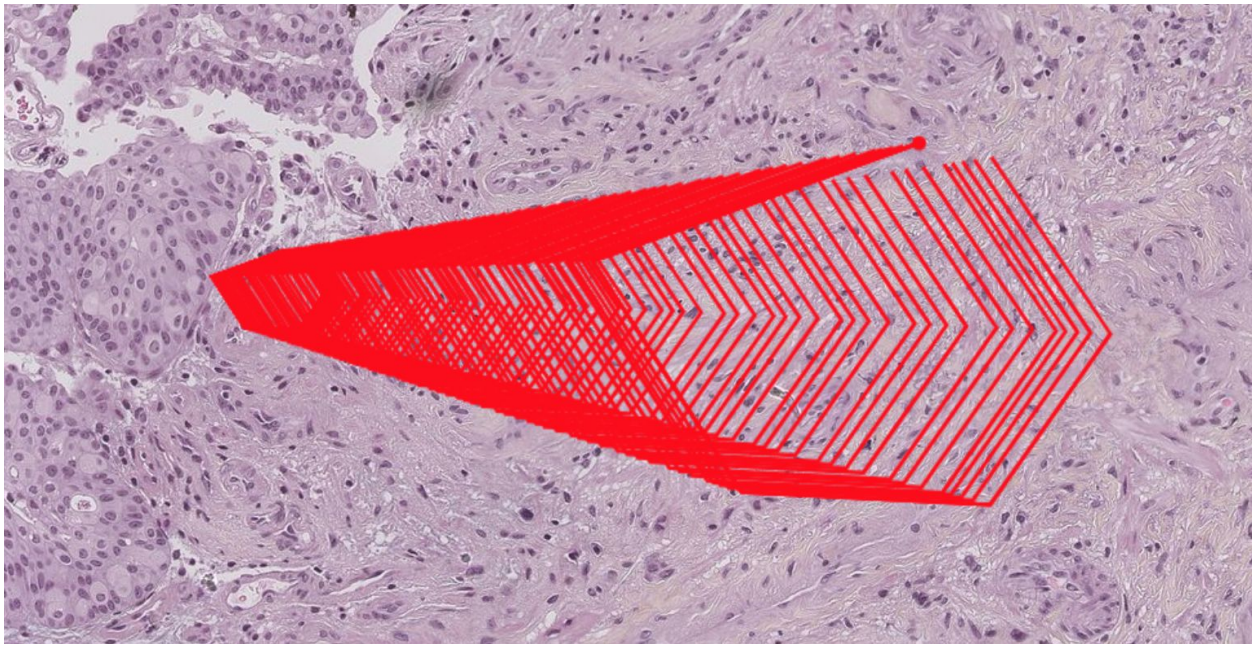


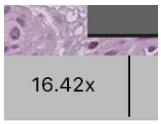
Image text: Shows the same drawing again after zooming and dragging the image a bit. The same drawing will pile up in memory again and again, and when .stroke() is called, all the drawing data that has piled up in memory will be drawn like we see in this image.

4.6.1.3 Viewer event handlers

```
viewer.addHandler("animation-finish", function () {
   $("#currentZoomLevel").html(Math.round(viewer.viewport.getZoom(true) * 100) /
100 + "x");
});
```

JavaScript code: This handler is triggered whenever an animation happening on the viewer finishes.

On animation-finish the current zoom level is extracted from the viewer object and displayed in the HTML footer for the user to see.




```
viewer.addHandler("open", function () {
   viewer.viewport.zoomTo(0.6);
});
```

JavaScript code: This handler is triggered whenever a new image is loaded into the viewer. It sets the initial zoom level to 0.6x as it gives a good overview of the image at that level.

When an image or tile fails to load, it's often because the user's session has expired. And no longer has access to functions in the backend. *fetchAuthenticated()* makes a request to the server at */authenticated*, asking if the requesting client is authenticated. If the response is a 401 status code, the user will be alerted that they have been logged out for inactivity and the page will be reloaded, taking the user back to the login page.

A global variable *aborts* is used. An alert box will only be created if *aborts* is 0, and displaying an alert box will increment *aborts*. As there usually are multiple tiles failing to load at once, this logic has been put in place to prevent more than one alert box popping up to alert the user.

```
@app.route('/authenticated')
def IsAuthenticated():
   if not current_user.is_authenticated:
       return "", 401
   return "", 200
```

Python code: The authenticated *route checks if the requesting user is authenticated, and returns a 401 unauthorized status code if it's not. Returns a 200 OK status code if the user is authenticated.*

4.6.1.4 Selecting an image

```
function changeImage(image) {
   if (currentImageLoaded) {
       cancelDrawing();
   }
   var id = image.id;
   currentImageLoaded = id.replace(new RegExp("{space}", "g"), " ");
   currentImageUrl = serverUrl + "/app/" + currentImageLoaded;

   open_slide(currentImageUrl);
   getXMLfromServer();
}

function open_slide(url) {
   drawings = [];
   viewer.open(url);
}
```

JavaScript code: Frontend functions for displaying an image chosen by the user.

When a user chooses an image from the image list, an event handler is triggered that makes a call to the *changeImage()* function shown above.

If an image is already opened when changing image, *cancelDrawing()* is called to make sure drawing is disabled when a new image opens.

changeImage() takes in an object *image*. This object represents the HTML element that the user clicked on when choosing which image to display. From *image*, the HTML id is extracted, with all instances of "{space}" being replaced with actual spaces(" "). The reason we do this string replace is explained at 4.5.2. The reformated id is then stored in a global variable *currentImageLoaded* and then used to build the full URL OpenSeadragon needs to be able to open the image. The URL is structured like this, *https://histology.ux.uis.no/app/<parent folder>/<image filename>*, and is stored in a global variable "*currentImageUrl*".

The *open_slide()* function clears the drawings from the previous loaded picture and opens the new picture.

Finally *getXMLfromServer()* is called. This function fetches an XML file from the server containing drawings to be drawn onto the newly loaded image.

```
@app.route('/app/<folder>/<filename>')
@login_required
def ChangeImage(folder, filename):
   global imagePathLookupTable
   session["ID"] = binascii.hexlify(os.urandom(20))
   path = "//home/prosjekt"+imagePathLookupTable[folder+"/"+filename]
   image = openslide.OpenSlide(path)
   deepZoomGen = DeepZoomGenerator(image, tile_size=254, overlap=1,
limit_bounds=False)
   deepZoomList.append(session["ID"], deepZoomGen)
   return deepZoomGen.get_dzi("jpeg")
```

Python code: Route for changing image.

When *viewer.open()* is called from the frontend, OpenSeadragon will send a request to this route asking for the Deep Zoom info about the image with a path matching the given folder and filename.

Using the OpenSlide library the requested image is loaded into an OpenSlide object and then a DeepZoomGenerator object is created from it. The DeepZoomGenerator makes it possible to extract individual tiles from the image.

For this to work for multiple users simultaneously, the DeepZoomGenerators had to be stored and accessible on a per session basis. As the generators are not serializable, this problem was solved by storing a randomly generated hex string in client *session["ID"]* and storing the generator in a custom data structure using the *session["ID"]* value as key. This way, each client has a way to access their own generator when needed.

Further details on the custom data structure used can be found at 4.6.1.5.

```
@app.route('/app/<dummy>/<dummy2>/<level>/<tile>')
@login_required
def GetTile(dummy, dummy2, level, tile):
   col, row = GetNumericTileCoordinatesFromString(tile)
   deepZoomGen = deepZoomList.get(session["ID"])
   img = deepZoomGen.get_tile(int(level), (int(col), int(row)))
   return ServePilImage(img)
```

Python code: Flask route that serves image tiles to the OpenSeadragon viewer.

After an image has been chosen and a DeepZoomGenerator created, OpenSeadragon can start populating the viewer with image tiles. It does this sending a request to this route, with a level and which tile number on that level it wants. OpenSeadragon sends the requests to the same route as in ChangeImage with a level and tile parameter at the end. As we do not need the

folder name or filename of the image to serve tiles, these are replaced with “dummy” names and not used.

The requested tile is extracted from the DeepZoomGenerator that was created in the *ChangeImage* route, and is returned as a PIL image object. PIL is a library for python used for opening, saving and manipulating images.

OpenSeadragon does not support images served as PIL image object, so the image must be converted before being served. This happens in *ServePilImage()* (shown below).

One request serves one tile, and with images as large as the ones handled by this application, it’s going to be a large number of requests going through per second when traversing an image. This function is therefore considered as one of the main bottlenecks and has to be as efficient as possible.

```
def GetNumericTileCoordinatesFromString(tile):
   col, row = str.split(tile, "_")
   row = str.replace(row, ".jpeg", "")
   return col, row
```

Python code: Simple function that takes a string with tile coordinates as parameter formatted like “col_row.jpeg”, and returns only the “col” and “row” part of the string.

```
def ServePilImage(pil_img):
   img_io = BytesIO()
   pil_img.save(img_io, 'JPEG', quality=80)
   img_io.seek(0)
   return send_file(img_io, mimetype='image/jpeg')
```

Python code: Converts a PIL image object to a JPEG image.[43]

This function takes a PIL Image object as parameter and converts it to a standard jpeg image. The only way to do this on a PIL image is with some form of IO operations, which are generally quite slow. As this function is part of a very heavily trafficked web-route, it is crucial that it’s as efficient as possible.

Using *BytesIO* and the *pil_img.save()* function, the PIL image is saved to memory as a JPEG image and immediately read back from memory and returned with the *send_file()* function. When sending a file with *send_file()*, the *mimetype* parameter must be set to specify which type of file is being sent.

Using memory instead of disc for the conversion made this function a lot faster.

4.6.1.5 Storing DeepZoomGenerators

```
class SessionDeepzoomStorage:
   dictionary = {}
   doubleSidedQueue = deque()
   counter = 0

   def append(self, sessionID, deepZoomGen):
       try:
           if self.counter >= 1000:
               oldestKey = self.doubleSidedQueue.pop()
               self.doubleSidedQueue.appendleft(sessionID)

               self.dictionary[sessionID] = deepZoomGen
               self.dictionary.pop(oldestKey, None)
           else:
               self.doubleSidedQueue.appendleft(sessionID)
               self.dictionary[sessionID] = deepZoomGen
               self.counter += 1
           return True
       except:
           return False

   def get(self, sessionID):
       try:
           return self.dictionary[sessionID]
       except:
           return None
```

Python code: Custom data structure for keeping track of client DeepZoomGenerator state.

This custom data structure is a combination of a dictionary and a doubly linked list(deque) and is used for storing DeepZoomGenerators and link each one to a unique session id.

The DeepZoomGenerators are stored in the dictionary with sessionID as key. The same key is appended to a doubly linked list that acts as a queue and both are set to reach a maximum length of 1000 elements. This limit is implemented as a way to stop objects of this class to grow infinitely large.

When the maximum length is reached, each new append to the structure will discard the oldest entry. The oldest entry is found by popping the linked list structure. All interaction with this data structure has a O(1) run time. Although it sacrifices on memory usage to make the application more responsive. Preserving memory usage was not a priority, as the server the application run on is equipped with 400GB of memory.

4.7 Drawing

One of the most important functions requested from the university's biomedical laboratory was the ability to create custom drawings on top of the images. Being able to draw on, and annotate images were requested because the images are being used for research in collaboration with the local hospital. Different types of deceased cells will be used to help train neural networks to discover the different types of marked cells. So the application needed a way to mark relevant areas that are going to be used in the algorithm.

```
function Drawing(name, points, tags, creator, grade){
   this.name = name;
   this.points = points;
   this.tags = tags;
   this.creator = creator;
   this.grade = grade;
}
```

JavaScript code: Constructor for drawing objects.

A drawing object contains several different attributes:

Name: Name of the drawing.
Points: A collection of image coordinates that make up the drawing.
Tags: Tags associated with the drawing. E.g. blood, tissue damage, other damage etc.
Creator: Username of the user that created the drawing.
Grade: Grade can be used to specify severity.

4.7.1 New drawing

To start a new drawing, the user clicks on the button called *New Drawing,* located on the left side of the header.

Image text: Left side of website header.

When the button is clicked, drawing mode is enabled by setting the global variable *drawingEnabled* to true, the button's label is changed from "New Drawing" to "Save Drawing" and some new drawing tools will appear in the bottom right corner. The user can now start

drawing on the canvas by clicking on the image to place points that will have lines drawn between them.

By default, clicking on the canvas will zoom in on the image, and dragging the image will move the image. When in drawing mode, these event actions must be suppressed to prevent unwanted movement of the image while making a drawing. An example of how this is done is provided in the code block below.

```
viewer.addHandler('canvas-drag', function (e) {
   if (drawingEnabled) {
       e.preventDefaultAction = true;
   } else {
       e.preventDefaultAction = false;
   }
})
```

JavaScript code: Handler for a "canvas-drag" event that prevents default action when drawing is enabled.

Image text: Drawing tool buttons

While drawing, the user has three different tools available to help manage the drawing.

Left button(Enable dragging): This button is a toggle button. When activated, drawing will be disabled and the user will be able to drag the image around.

Center button(Undo): This button will remove the last point/line that was added to the drawing. Can be repeatedly clicked to remove more and more of the drawing until there are no more points/lines left to undo.

Right button(Cancel drawing): This button will discard the current drawing and take the user out of drawing mode.

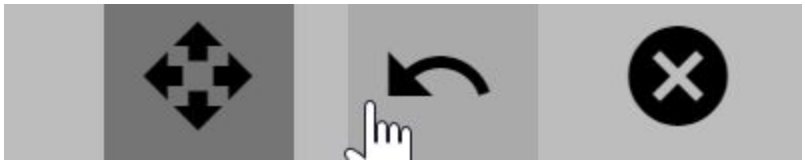


Image text: Enable dragging button shown in activated state. The undo button shows what happens when one of the buttons are hovered over.

Image text: Pressing the “Cancel” button will trigger an alert and ask the user to confirm cancelation.

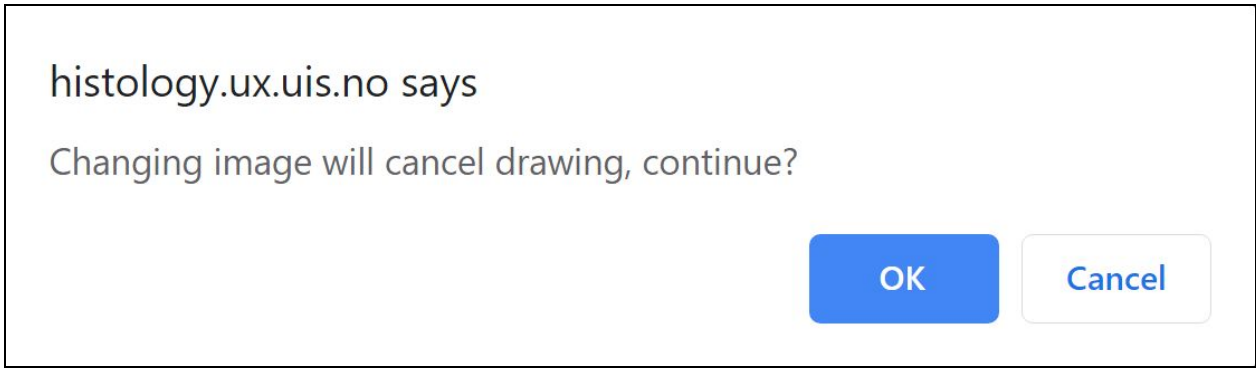


Image text: Changing image while in drawing mode will trigger an alert and ask the user to confirm the action, as it will cancel the drawing.

4.7.2 Save drawing

When a user is finished with their drawing, they can click the previously called *New Drawing* button, which has now changed to *Save Drawing.*

Image Text: New Drawing has been changed to Save Drawing, this combined with the new tools on the bottom right corner helps tell the use that drawing mode is active.

Once this button is pressed, a new window will be opened on the left, on top of the image list. And *Save Drawing* will be made unclickable. This window is generated anew whenever the *Save Drawing* button is pressed. This is necessary since there is a possibility for new tags to be added during the time a user takes to create a drawing.

This window gives the user the opportunity to add information such as a name for the drawing, specify it’s grade and add multiple tags if wanted. It also gives the user the ability to add a new tag to the database in case they have found new types of cells which has not been tagged before.

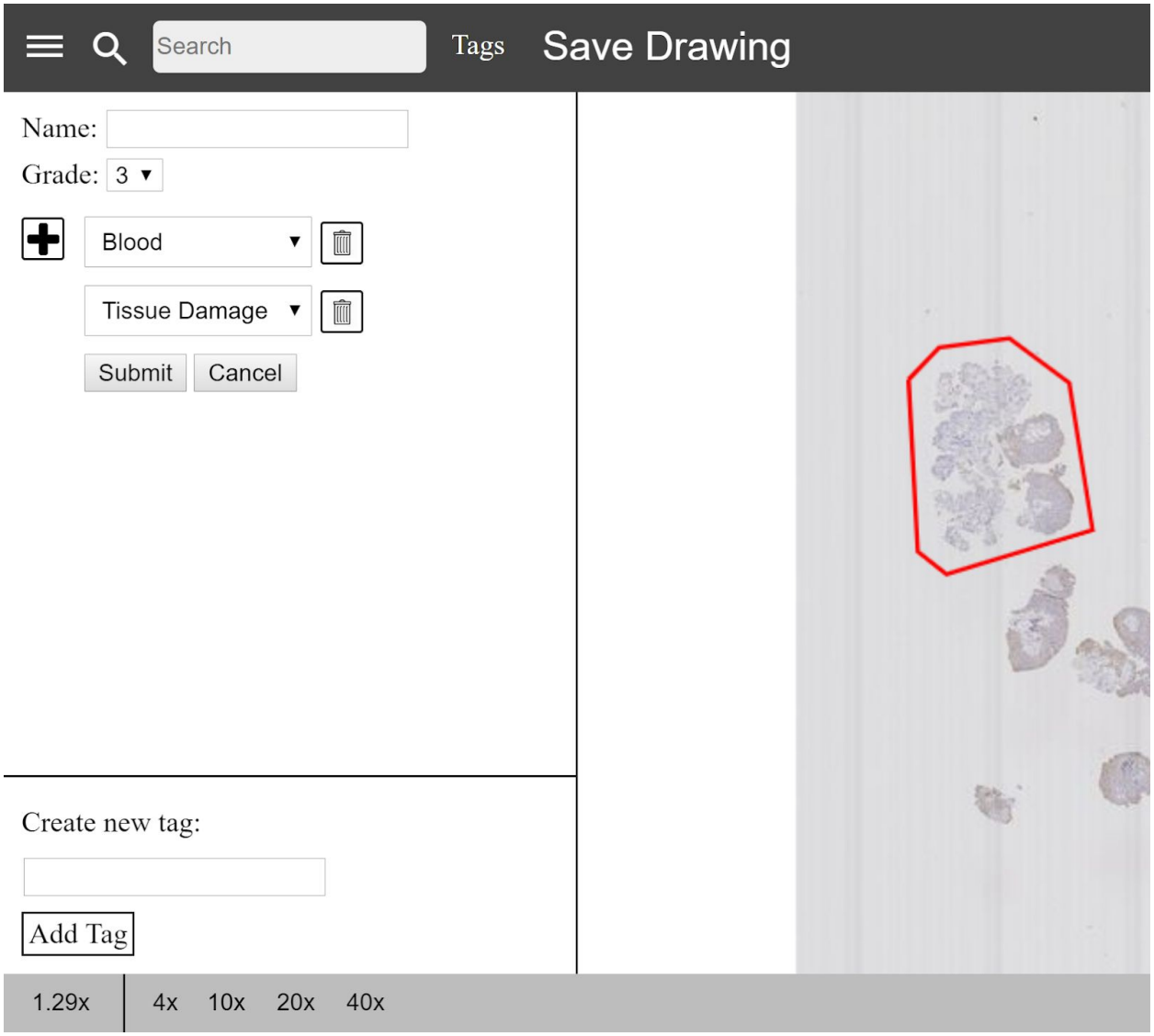


Image text: A new window is opened when the Save Drawing button is pressed. Pressing the + button adds a new drop down menu to add more tags. And pressing the trashcan icon deletes the menu in case the user realises its not needed after all.

Once the user have entered the relevant information they can click the *Submit* button. This sends the information to the server by calling the function *sendXMLtoServer.*

sendXMLtoServer takes two arguments, the first being an XML representation of the drawing's coordinates on the image, as well as secondary information like tags, name and grade. While the other variable is a modifier which is needed because *sendXMLtoServer* is called from two places, and some logic is dependent on the different places. The other location it's used, besides here, are when a user is uploading a brand new XML file and the drawing is not already rendered. For this circumstance, the drawing is already rendered, so the action is set to 0.

```
function sendXMLtoServer(xml, action) {
    if (action === 1)
        XMLtoDrawing(xml);
        canvasOverlay._updateCanvas();
    }
```

JavaScript code: The action variable decides if the current XML needs to be redrawn by calling XMLtoDrawing.

Using XHR the XML is sent to the server to be stored. Converting the drawing from a list of coordinates to an XML is done the same way as explained in 4.8.1.

```
xmlHttp.open("POST", "postxml/" + currentImageLoaded.substring(0,
                                  currentImageLoaded.length - 4));
xmlHttp.send(jQuery.parseXML(xml));
```

JavaScript code: Calls the backend through XHR with the XML generated earlier.

```
@app.route('/postxml/<foldername>/<filename>', methods=["POST"])
@login_required
def PostXML(foldername, filename):
```

Python code: Route to which the XML is sent.

Since the information needs to be accessible by other users in the future, the information needs to be stored more permanently. The first thing to do is check if the image has a pre existing file it can save the XML to. If not, it needs to create that file with the base XML structure.

As shown in the code below, the check is for a global variable called *xmlStoragePath* combined with the image name. *xmlStoragePath* contains the static folder location where all XML files are stored, and which can be easily changed if the program's location is changed internally in the server. As explained in search the complete file names consist of both the folder in which the image is stored and the file name for the image, with a custom string "*[slash]*" instead of a regular "/" as linux have character restrictions on filenames.

```
try:
   file = foldername + "[slash]" + filename + ".xml"
   if not os.path.isfile(xmlStoragePath + file):
       createXmlFileIfNotExist(xmlStoragePath + file)
```

Python code: Checks if the file exists. And if it does not, call a function with complete name to create file before continuing.

Insertions to the database is also required since the search functionality uses what exists in the database. But just like with the file, if there already exists an entry with the same imagepath in the images table, the insert is not executed.

```
db.engine.execute("insert into images(ImagePath) values('{}');".format(imagePath))
```

Python code: Inserts the ImagePath into the database.

Since the information have been transferred to a new programming language (python), it needs to be converted into an XML object again so it can be properly stored on the server. When this is done the drawings are also added to the database so it can be used by search in the future.

4.8 XML

In previous chapters, the use of XML has been discussed in relation to drawing, how drawings are stored in XML files and loaded from the same files. One of the main criteria of the task given by the Biomedical data Analysis laboratory was the ability to upload and download XML to, and from the server to make it easier to upload old data, as well as download new data to be used in other software.

The XML structure is copied to look like the structure created by Aperio Imagescope, as such, one is able to upload images from Aperio and use them on the web, as well as downloading from the web application and import to Aperio.

```
<Annotations>
  <Annotation>
    <Regions>
      <Region tags="Blood|Tissue Damage" name="" creator="admin" grade="4">
        <Vertices>
          <Vertex X="42353.910264087615" Y="61131.9385708072" Z="0"></Vertex>
          <Vertex X="41779.25951606253" Y="61857.50769710151" Z="0"></Vertex>
          <Vertex X="42081.09627260096" Y="62350.894702981626" Z="0"></Vertex>
          <Vertex X="42812.469951905616" Y="62374.11291502305" Z="0"></Vertex>
          <Vertex X="43009.824754257665" Y="62054.86249945356" Z="0"></Vertex>
          <Vertex X="43450.970783044606" Y="61909.7486741947" Z="0"></Vertex>
          <Vertex X="43398.729805951414" Y="61427.970774335285" Z="0"></Vertex>
          <Vertex X="42649.94246761569" Y="61085.50214672437" Z="0"></Vertex>
          <Vertex X="42353.910264087615" Y="61131.9385708072" Z="0"></Vertex>
          <Vertex X="42353.910264087615" Y="61131.9385708072" Z="0"></Vertex>
        </Vertices>
      </Region>
    </Regions>
  </Annotation>
</Annotations>
```

Image text: Example of how the structure of the XML files look. The region and its sub elements represents a drawing. So if there were more drawings in this image, there would be more "Region" elements.

Normally, when looking at an image and its coordinates, the coordinates are going to be represented as integers. However, given of the nature of how coordinates are handled in this application in relation to zooming and scaling and how OpenSeadragon handles coordinates, image coordinates are represented as floating numbers.

OpenSeadragon uses its Point object class as its way to return coordinates from a canvas-click event. A Point contains a 2 dimensional vector that represents a position on the OpenSeadragon viewer element(4.6.1).

As it is possible to zoom further in on the image after the lowest layer(full resolution) has been reached, as well as zooming ranges on each individual image layer, there are room for OpenSeadragon to provide better accuracy than integers here. Though it is not a lot, it will make a difference on the placement of drawings that scale with the image.

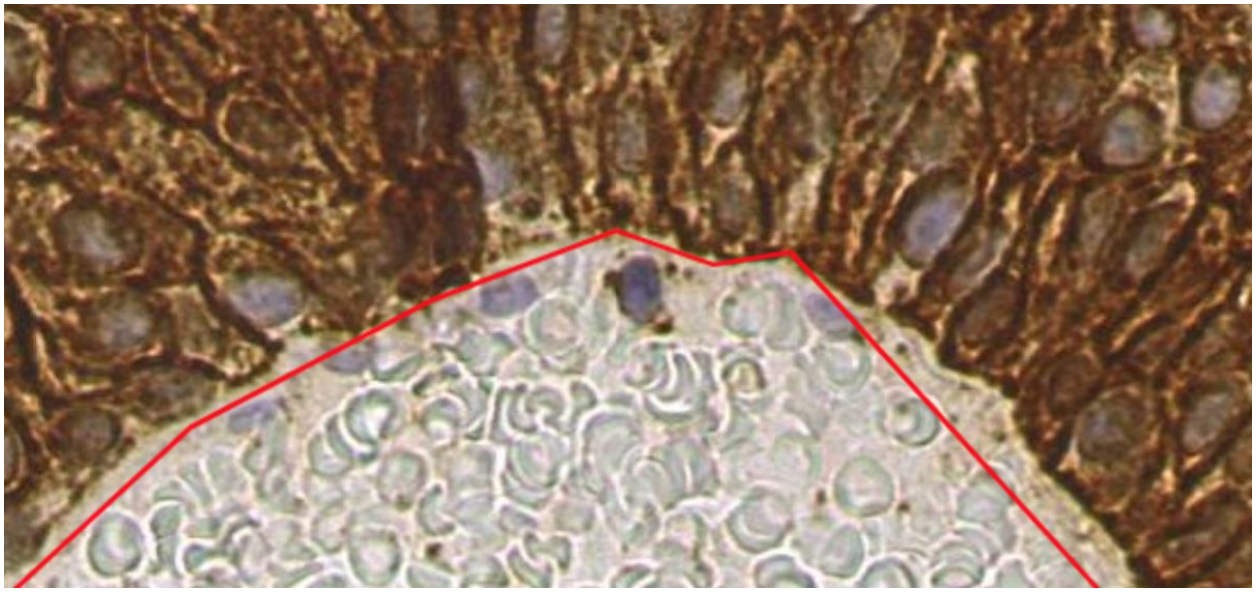


Image text: Example drawing with floating point coordinates.

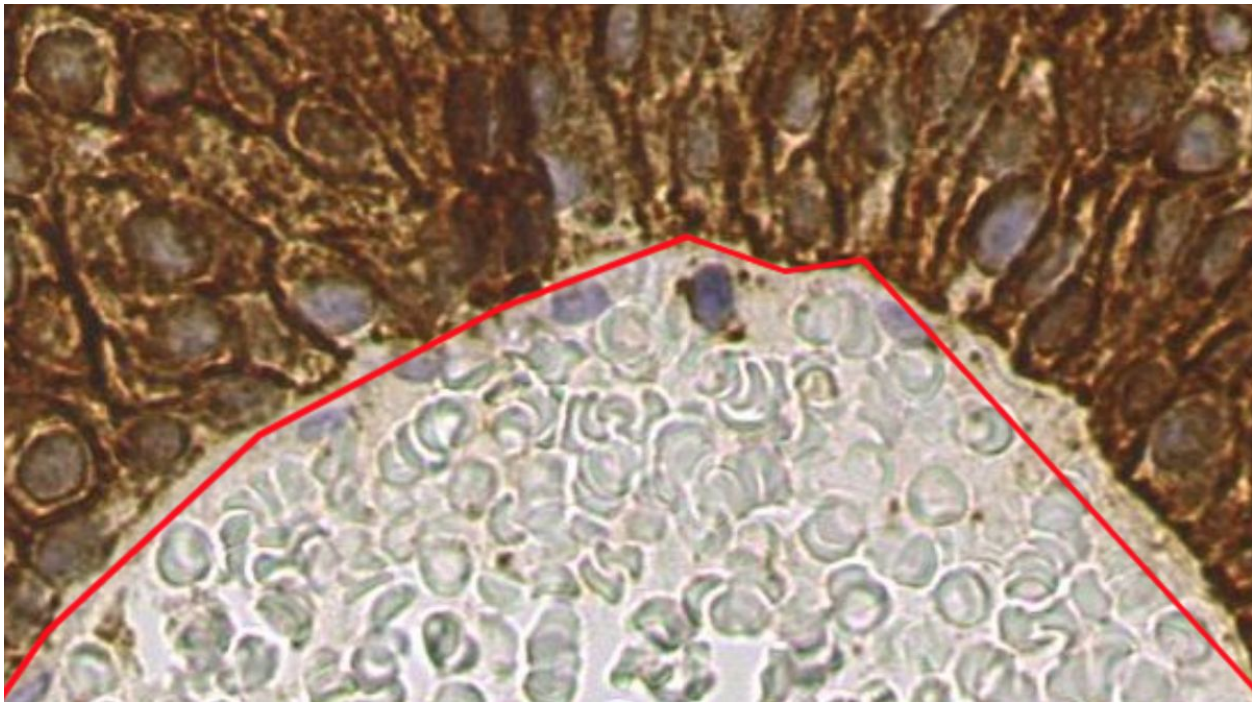


Image text: Same drawing again, overlapped with a copy of itself where the coordinates have been rounded to integers. The lines are now thicker, which proves they do not perfectly overlap each other. The positional difference is negligible, but it is there.

4.8.1 Download

In the dropdown menu discussed earlier in 4.1, one of the elements are named “Download xml”, which lets the user download an XML file containing all the drawings in the currently opened image. If an image is not currently selected, the user will be prompted with an alert telling them that no image is selected.

```
<a id="DownloadXML">Download xml</a>
```

HTML code: HTML line in drop down menu for the download button

An event handler is added to the download button which initiates the logic for creating and downloading the XML file to the users computer.

First, *generateXML* is called with the global variable *drawings* that contains all drawings in the current open image. (talked more about at 4.6.1.2)

GenerateXML starts by creating a new XML objects that will be the resulting file being downloaded. When the object is created, the elements needed to form the correct structure (shown in 4.9) will be added.

Once the base structure is added, the drawing data is generated. It iterates through every drawing attached to this image and gather the needed information from the drawing object.

```
listOfDrawings.forEach(function (drawing) {
	var points = drawing.points;
	var tags = drawing.tags;
	var name = drawing.name;
	var creator = drawing.creator;
	var grade = drawing.grade;
	var tagsAsString = "";
```

JavaScript code: GenerateXML Iterates through all drawings and adds the info from the current drawing to variables.

Given the possibility to add multiple tags to one drawing, there needed to be a way to show every tag in a drawing and make it easy to understand which tags are individual tags and not part of a multi word tag. Which was accomplished by creating a string where each tag is appended but separated by a special character “|”. As shown in the XML structure example this could look like:

```
Blood|Tissue Damage
```

Which should make it easy to see that “Tissue Damage” is one tag, and not two separate tags “Tissue” and “Damage”

Everything is now ready to create the drawing element, named region. Instead of appending multiple element for each piece of information, all the relevant info is added as its own attribute, making it easier to distinguish each region.

```
var region = xml.createElement("Region");
```

JavaScript code: Creates the “region” element representing a single drawing and where additional data is added as attributes.

When all other information is added, all that remains is the coordinates representing each point in the drawing. Which is stored as a list that can be iterated through. Here, each point is represented as the element named *vertex,* which is given the attributes x, y and z for the coordinates. Since the drawings are in a two dimensional plane, the z coordinate is always 0, but is needed to be compatible with Aperio ImageScope.

```
points.forEach(function (point) {
            var vertex = xml.createElement("Vertex");
            vertex.setAttribute("X", "" + point.x);
            vertex.setAttribute("Y", "" + point.y);
            vertex.setAttribute("Z", "0");
            vertex.textContent = "\n";
            vertices.appendChild(vertex);
      });
```

JavaScript code: Iterates through every point saved in a drawing and adds them as an attribute to vertex. Before moving to the next point the vertex element is appended to the vertices.

Once every *vertex* element is created and appended to the parent node *vertices,* the XML structure needs to be completed. This is done by going through every subsequent node and appending the child. *Vertices* is appended to *Region, Region* to *Regions*, and so on.

Once the root element *annotations* is appended to the XML document, the XML is fully generated and is ready to be returned. The XML is serialized to string to make it easier to handle later on, and then returned. The string returned from *generateXML* is immediately sent into a new function called *downloadXML*.

```
var xml = generateXML(drawings);
downloadXML(currentImageLoaded.substring(0,
            currentImageLoaded.length - 4) + ".xml", xml);
```

JavaScript code: Eventhandler to show the further usage of xml.

To make the code more maintainable, the logic to initiate the download was separated into its own function. Which takes the filename and XML strings as parameters. To keep consistency, the name given to the file being downloaded is the same as the name of the image. But the filename extension is switched by removing the last 4 characters of the image name, which corresponds to ".scn" and appends ".xml" in its stead. And it takes the string which represents the XML file.

downloadXML works by creating an invisible download link that is automatically clicked, and then removed again. The download element is a standard hyperlink *<a>* tag with the download attribute added. The download attribute is what tells browser to download whatever is in the url. In this case, the url is defined as a data url, where the media type is set to plain text, character set to be utf-8, and the actual data is the XML string put through the *encodeURIComponent* function which escapes the characters for further security.

```
function downloadXML(filename, text) {
	var element = document.createElement('a');
	element.setAttribute('href', 'data:text/plain;charset=utf-8,' +
				encodeURIComponent(text));
	element.setAttribute('download', filename);

	element.style.display = 'none';
	document.body.appendChild(element);

	element.click();
	document.body.removeChild(element);
}
```

JavaScript code: the downloadXML creates a invisible link that automatically clicked to download the xml content.[44]

There was also the possibility of getting the XML file from the server, instead of generating the file again client-side. But given the small size of XML files, the performance difference would be negligible, and creating it client side removed an attack vector for a man in the middle attack.

4.8.2 Upload

Like the download button, one of the elements in the dropdown menu is "Upload xml", clicking this while having an image selected will open a separate window where the user can select a XML file to upload, if the XML is in a compatible format, new drawings will instantly be uploaded to the application, and visible when selecting this image in the future.

```
<a id="UploadXML">Upload xml</a>
```

HTML code: HTML line in drop down menu for the upload button

Once a user clicks on the Upload button, the following event handler will trigger. This code triggers another hidden HTML element which is the actual code for uploading, the visible button is just the instantiator.

```
$("#UploadXML").click(function () {
    $("#FileInput").trigger("click");
});
```

JavaScript code: When the “Upload xml” button is clicked another click is simulated in the background on a hidden element.

```
<input id="FileInput" type="file" style="display: none;">
```

HTML code: An HTML input element with type=”file” will open a file explorer that lets the user chose a file to upload.

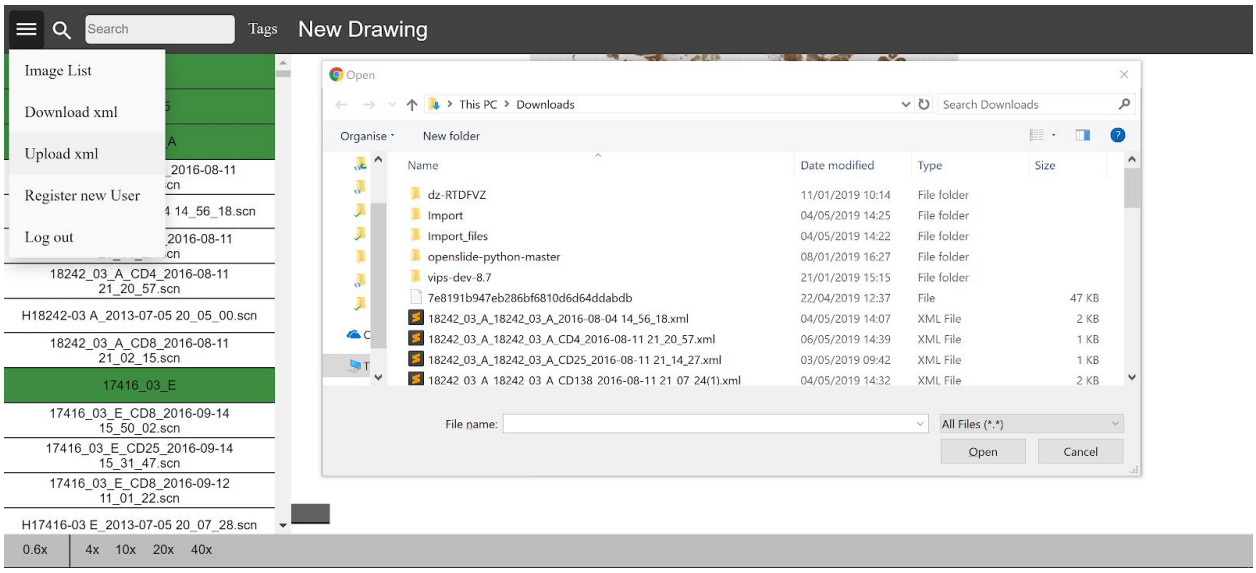


Image text: Unce the “Upload xml” button is pressed a separate file explorer will be opened

Once the file is selected, the following code will be triggered with the variable *e* representing the chosen file. Since XML is really just a specifically formatted string, much like JSON, the file can easily be read as a string, which is natively supported in javascript.

```
$("#FileInput").on("change", function (e) {
        var file = e.target.files[0];
        var reader = new FileReader();
        reader.readAsText(file, "UTF-8");

        reader.onload = readerEvent => {
            var content = readerEvent.target.result;

            sendXMLtoServer(content, 1)
        }
    });
```

JavaScript code: Once a file is selected, it is read as a string and the string is then sent onward to be formatted for the server.

Once the XML is stored in a easily read variable, it is to be rendered over the image, as well as stored on the server. *sendXMLtoServer* is talked about before, during drawing, but with a small difference. As explained in drawing, the function takes in two variables. The first being the XML sent to the server.

While the other is a modifier variable, and this is needed because the different calls to this function have some small variances. That is, whether the current XML being handled should be drawn on the image or not. And for the case of uploading a new drawing/XML, we want to also update the current loaded image with a drawing. In this scenario we call the function with the *action* variable set to 1, to make sure this happens.

```
if (action === 1) {
    XMLtoDrawing(xml);
    canvasOverlay._updateCanvas();
}
```

JavaScript code: Code snippet from sendXMLtoServer *that is run during upload but not during drawing.*

Also in *sendXMLtoServer*, is a call to the backend with the current XML. This, as well as the logic happening in the backend is all explained in drawing with the exact same result.

```
xmlHttp.open("POST", "postxml/" + currentImageLoaded.substring(0,
                                  currentImageLoaded.length - 4));
xmlHttp.send(jQuery.parseXML(xml));
```

JavaScript code: Calls the backend with a url equal to the name of the file for the image.

4.9 Security

4.9.1 Login

As mentioned in 4.3.1.3, to ensure that only users that have entered correct credentials can access the rest of the site, most of the url routes are configured with a "*login_required*" flag. This will act as a check to validate that the user has newly been logged in. And if the check fails, the user will be redirected to the login screen.

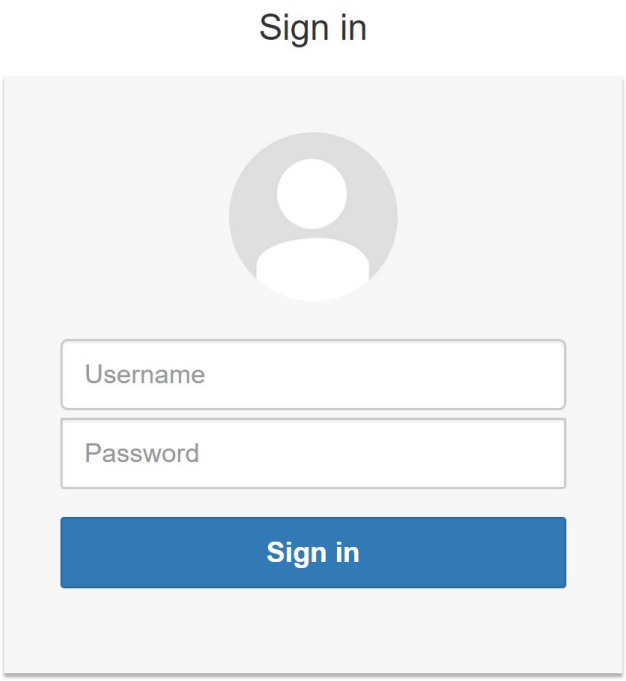
```
@login_manager.unauthorized_handler
def CatchNotLoggedIn():
	return redirect("/login")
```

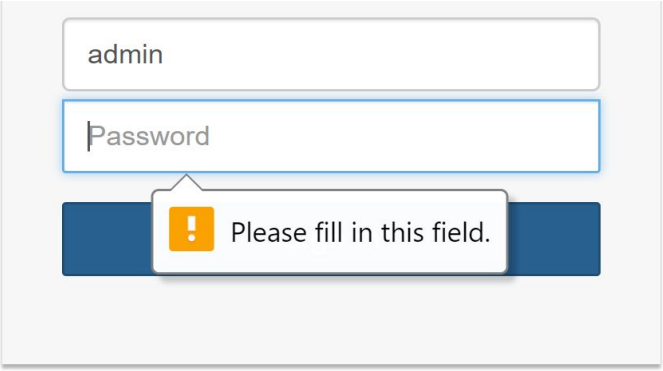
Python code: When a user does not pass the required login check, this method will be triggered and the user will be redirected to the login page.

The login page requires a username and password to be entered before the application will even attempt to login the user.

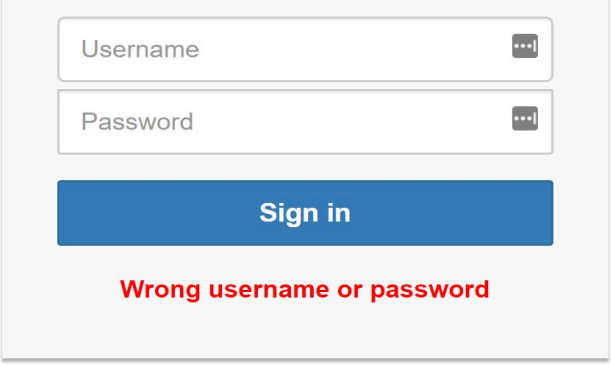
When the user pushes the "sign in" button, the username is queried for in the database, if the username exist, the password is hashed, and compared to the hashed password linked with that username in the database. If everything checks out, the user is logged in, and a session is created for that user to make sure they stay logged in.

If either the password or username is wrong, the fields will clear and an error message will display saying "Wrong username or password"





Both Fields needs to be filled in.



Error message displayed when wrong username or password is entered.

```
@app.route("/login", methods=["GET", "POST"])
def Login():
   # request.method == "POST" when the sign in button is pressed.
   if request.method == "POST":
       username = request.form["username"].lower()
       password = request.form["password"]
       user = User.query.filter_by(username=username).first()
       if user is not None and username == user.username
               and user.check_password(password):
           login_user(user)

           # Send the user to the main page once they are logged in
           return redirect("/")
       Else:
           # Keeps the user on login site and informs that the password/username
             was wrong
           return render_template("login.html", className = "warning",
                                  message="Wrong username or password")
   else:
       return render_template("login.html")
```

Python code: Login method.

As well as managing users access to the web application, keeping the user logged in for an extended period of time is also required. To be able to manage user sessions, Flask-login requires a method to keep checking a user's activity to the users currently stored in session. This is called before the user performs any action which requires them being logged in.

```
@login_manager.user_loader
def RefreshLoginToken(Username):
       return dbClasses.User.query.get(Username)
```

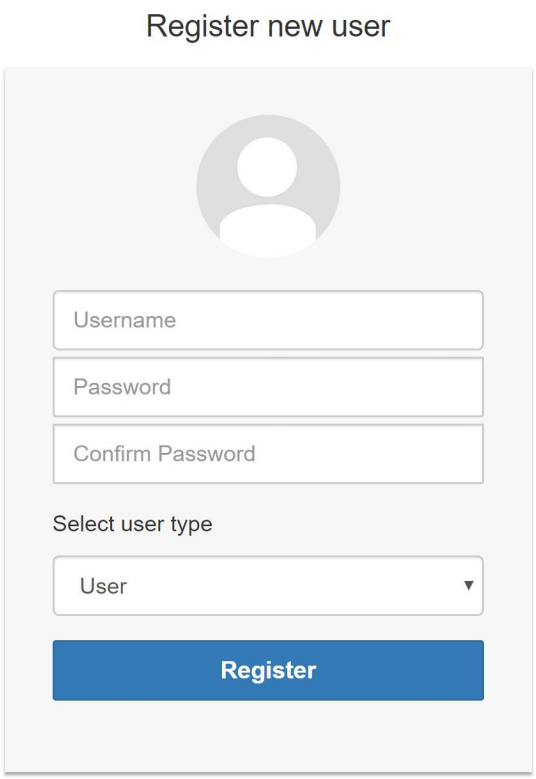
Python code: Callback function used by Flask-login to reload the user from the user id stored in the session.

When a user is done using the application, they can either enter the top left menu and log out, or if they leave the page and remain inactive for 75 minutes, they will automatically be logged out. The user's session will be deleted and the user will be required to log in again next time they want to access the website.

```
@app.route("/logout")
@login_required
def Logout
   logout_user()
   return render_template("login.html", className="info", message="Logged out")
```

Python code: When a user manually logs out, or if they remain inactive for 75 minutes but does not leave the website, their session will be removed then redirected to the login screen.

4.9.2 Register



Registering new users seemed like a necessary functionality needed, as it gives more control over who has access to the webpage than having a single user that everyone working on this project can use to log in and out.

When registering a new user, there is also the possibility to select the type of user, where the current choices are User and Admin.

With the ability of creating new admin accounts, the register site is only accessible if one is logged in as an admin user.

If a regular user attempts to access the register page, they will be redirected to an error page with error code 401(unauthorized).

```
@app.route("/register", methods=["GET", "POST"])
@login_required
def Register():
   if str(current_user.type) != "Admin":
       abort(401)
```

Python code: The beginning of the register method, login is required to access it in the first place, then another check is done for whether the user is of the type admin. If not, load page for error message 401 instead.

When all the forms are filled in and the register form is sent. The information will be sent to the flask application. Here there will be a couple of checks:

```
if User.query.filter_by(username=registerUsername).first() is not None:
   return render_template("register.html", className = "warning",
                                       message = "Username already exists.")
```

Check 1: See if a username already exist.

```
if request.form["secondPassField"] != firstPassField:
   return render_template("register.html", className = "warning",
                                       message = "Passwords does not match")
```

Check 2: Check if both passwords match

If the checks pass, the user will be registered in the database, the password hashed, and the type of user will be set. The admin will then be redirected to the main page.

```
newUser = User(registerUsername, firstPassField, userType)
db.session.add(newUser)
db.session.commit()
return redirect("/")
```

Python code: If the checks passes, a user object representing the new user will be created, and subsequently added to the database.

4.9.3 Flask hashing

Hashing was implemented using werkzeug's security utility. It comes with the ability to automatically salt the passwords and create strong hashes.

There are two methods called, one to generate a hash from the given password, and one to check if a password entered matches an already existing hash.

The first is *generate_password_hash(arg)*, which takes a new password in plaintext as argument, it generates a salt and hash of the password and return a string in the format "method$salt$hash". How this is done in code can be seen in section 4.2.2.

And the other is *check_password_hash(),* which checks if the entered plaintext, when hashed, gives the same hash as the one that already exists. And returns a true or false depending on this.

```
pbkdf2:sha256:50000$6SAC47Lr$c86f7f539f2780c1af59e22a08466aafbaea6fe9a8669aa998e3c0
91a073d1c3
```

Output given from generate_password_hash.

This string should be split up, and the salt/hash should be stored in different locations, the hash is often stored in a database, while the salt is stored in a shadow file only accessible to the admin user.

This way, the password is not known by anyone except the actual user. And can only be accessed by hashing the same password with the same salt and see if the output is the same.

And this is exactly what is done during the login logic, the entered password is put into the *check_password_hash* method, and the output string is compared to what is found in the database connected to the given username.

4.9.4 Flask logging

Logging gives an admin good overview of the current state and history of the server. What has been done by who and if something has gone wrong.

Given the nature of the images used, and how a get request is sent every time a user moves to a new tile, and the server logging every get requests, a lot of information is logged:

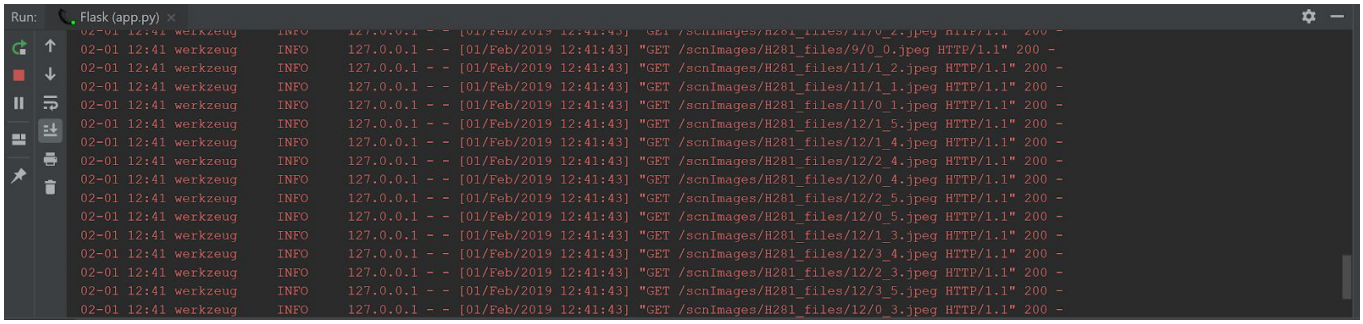


Image text: Get request for every single tile being viewed

Python has its own logging object that can be accessed by "*import logging*".
The basic logging functionality gives the ability to display logging messages at different levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) and the ability to filter by a specific level. An example could be only logging warning messages and up.

Other functionalities are the ability to create one's own logging level, and only save specific warnings to files. With this a custom logger was implemented to only save important information, specifically user activity such as who logged in/out, images requested by which user, drawing information etc.

```
def StartLogging():
   # configure logging format to be more readable.
   logging.basicConfig(level=logging.INFO,
                       format='%(asctime)s %(name)-12s %(levelname)-8s'
                              '%(message)s',
                       datefmt='%m-%d %H:%M',
                       filemode='w')

   # add custom logger that writes to file
   logging.addLevelName(25, "Server")
   UserLog = logging.getLogger("Server")
   UserLog.addHandler(logging.FileHandler("logging/" + DateTime() + '.log', 'a'))
   UserLog.setLevel(25)

   return UserLog
```

Python code: Method for configuring logging format, and what is logged to file.

The custom logger's level was put between INFO and WARNING, and given the name "Server" and everything from Server and up was logged to a .log file in the logging folder. This way a new log file is created every time the server restarts, which makes it easier to go through the history of the server, compared to one ginormous file.

```
2019-02-20 11:09:37 | 127.0.0.1 | admin logged in
2019-02-20 11:10:03 | 127.0.0.1 | admin registered a new user: thomas
2019-02-20 11:10:20 | 127.0.0.1 | thomas logged in
2019-02-20 11:10:24 | 127.0.0.1 | thomas requested image H281.scn
2019-02-20 11:10:45 | 127.0.0.1 | admin logged out
```

Example of information logged.

4.10 Server hosting

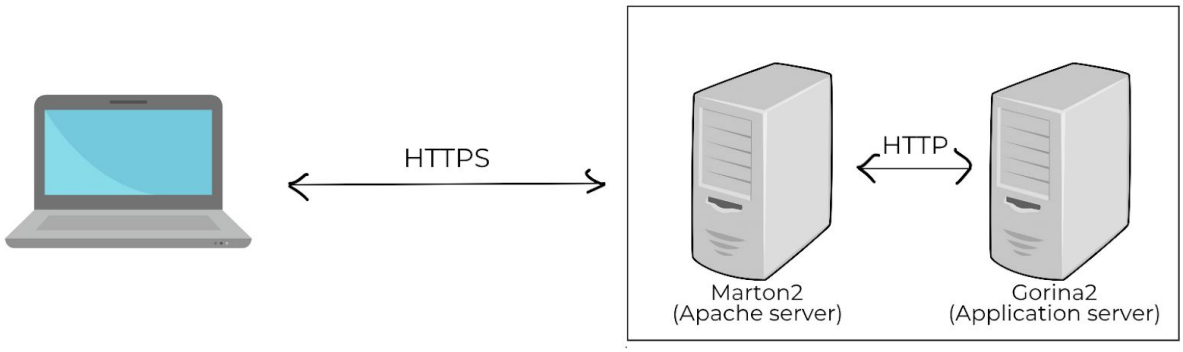


Image text: Illustration of the hosting architecture.[45,46]

External communication with the application server happens through an intermediate apache server. Communication between client and apache server is done over HTTPS. Internal communication between apache server and application server is done over HTTP. Both servers are connected to the same switch and physical access is limited. It is therefore very difficult for unauthorized third parties to intercept the unencrypted HTTP packets.

4.11 Inconsistent image resolution

As discussed earlier, leica images are structured as a collection of images with multiple layers built like a pyramid. Here the resolution for each layer down is increased a power of 4 e.g, one tile of 256x256 is swapped with 4 tiles, all with 256x256 in resolution. But a problem arises if this is not the case, if there is inconsistency between the tiles' increase in resolution, or if the image was created with a lense that makes the resolution change only to be a factor of two for example. This would require different parts of the image to be scaled differently, or for the total image to use a different scaling factor, which open slide does not support.[47] Instead openslide throws an exception:

```
openslide.lowlevel.OpenSlideError: Inconsistent main image resolutions
```

Error message received by server when a user is attempting to open a image with inconsistent main image resolutions.

If such an image is the first image a user attempts to open after having entered the site, they will receive an error message instead of the intended image.

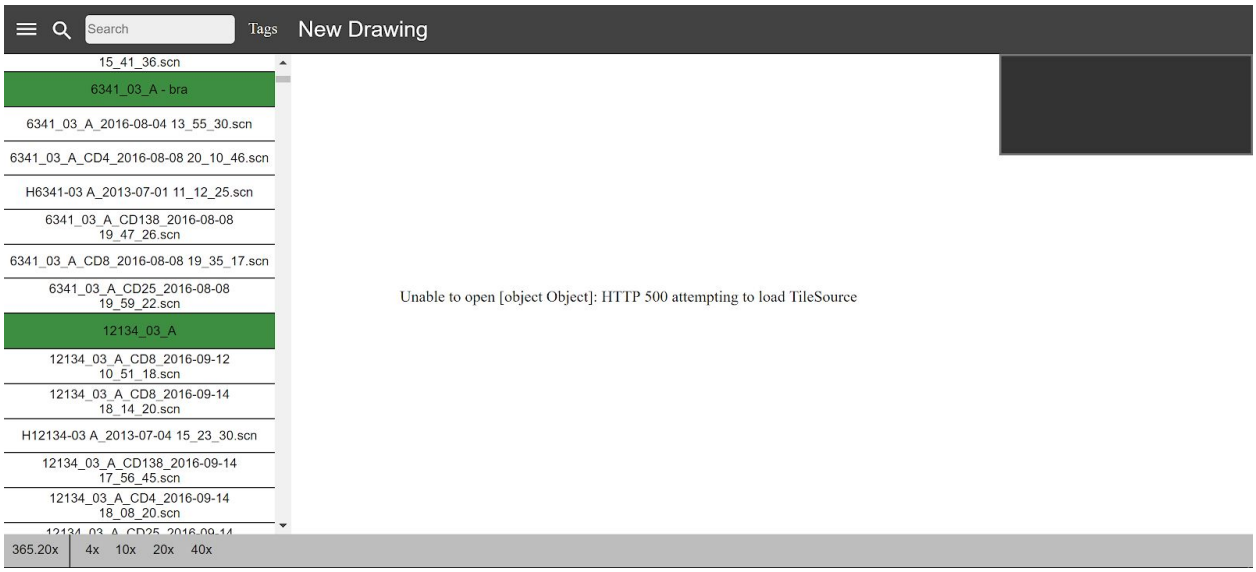


Image text: Error message received when trying to open an image with inconsistent resolution/different scaling factor.

And if there is a different image open when trying to switch to an unsupported image, the image will simply not load and the user will remain that the previous image. The number of images with this problem is unknown, but during the development of the application there has not been a large amount of these images found (10-20) which is not a lot when the total amount of images is 1100.

5 Reflection

In this chapter the final result will be discussed, as well as potential improvements that could be implemented in the future

5.1 Result

The initial goal of this application to have a website with the capabilities of viewing histological images stored on unix servers, as well as being able to create drawings on top of the images was accomplished, with some caveats.

As explained earlier there is a problem with a small portion of the images where the resolution scaling is either inconsistent or uses a different scaling factor. As this is an innate limitation with openslide, not much could be done when it came to rendering these images.

Not having to convert every single image to the .dzi format explained in 3.3.1 was a huge bonus. As shown, converting one image gave us over 600 000 smaller images, applying this to 1100 images would yield over half a billion files. And with more images potentially being added in the future the total amount could easily have reached over a billion images. The indexing alone could require tens of Gigabytes in memory use.[48]

The total size of the MySQL database ended up being surprisingly small. This is much due to the use of XML to store the drawing information, which worked to our benefit as this made the downloading/uploading aspect of the application easier.

5.2 Development process

Throughout the project, there was steady progress as the group was highly motivated to get a working application that could be used by the university and hospital. The biggest challenge was getting a clear understanding about the images, the pyramid like structure, tiles, as well as the different file formats and how to handle them. This was not made easier by the general lack of information and documentation on the different technologies surrounding the file formats. With the only exception being OpenSeadragon, which had a well documented API.

5.3 Future development/improvements

In the end, time was the main limitation, and when the end of the project arrived, there were still functionality that the group wished they had implemented.

Search could easily be improved upon, as shown, only the images are hidden. With the large amount of folders they should also have been hidden once all images in the folder was filtered away.

As drawing was one of the core features of the app, making it possible to not only create drawings, but also manage these drawings would be a nice future improvement. For example making each drawing selectable to show stored metadata about it(*Name, creator, tags etc.*) and also being able to edit or delete the drawing.

When it comes to the problem of inconsistent image resolution there could have been done a better job telling the user that the current selected image is not supported, for example by either setting the current image just clicked in the image list to red. Or by using a popup that informs the user of the problem. By making the clicked image's color permanently red, one would eliminate future situations where a user attempts to open the image. As the color would indicate that there have been earlier attempts to open, and that it is not supported.

It was discussed with the biomedical laboratory about giving the users access to metadata about the image they are currently viewing. OpenSeadragon has built in functionality to extract information like the size of each layer in the pyramid, total image size and compression ratio. The relevance if this information was concluded to be useful in certain situations but given a low priority, and in the end it was not implemented, but could be a future improvement.

The custom data structure we made for keeping track of Deep Zoom generators are flawed and should be redone in a better way. With the current implementation, all interaction with it has a O(1) runtime, but memory usage keeps piling up with more and more images being loaded. 30 minutes of extensive testing revealed a current memory usage on the server of 5.5GB. This memory usage theoretically has a hard ceiling since the structure stores a maximum of 1000 generators. Many of these can belong to the same user, although only one is in active use. An improvement would be to come up with a way to actively discard inactive generators. So that for each user, only the active generator is stored in memory on the server. For example by adding logic to discard the previous generator before creating the new one in *ChangeImage()* server side.

Even with some functionality not fully implemented, the group is very happy with the end result. The main functionality of displaying images and annotating them was achieved, as well as implementing security to protect the sensitive information. With only minor features not completed.

6 Sources

1. Contributors to Wikimedia projects. Flask (web framework) - Wikipedia. *Wikimedia Foundation, Inc.* (2010). Available at: https://en.wikipedia.org/wiki/Flask_(web_framework). (Accessed: 16th January 2019)
2. Welcome | Flask (A Python Microframework). Available at: http://flask.pocoo.org/. (Accessed: 16th January 2019)
3. Introduction to Flask — Python for you and me 0.3.alpha1 documentation. Available at: https://pymbook.readthedocs.io/en/latest/flask.html. (Accessed: 16th January 2019)
4. OpenSeadragon. Available at: https://openseadragon.github.io. (Accessed: 16th January 2019)
5. OpenSlide. Available at: https://openslide.org/. (Accessed: 18th January 2019)
6. SQLAlchemy - The Database Toolkit for Python. Available at: https://www.sqlalchemy.org/.
7. Website. Available at: https://ars.els-cdn.com/content/image/1-s2.0-S1046202314002370-gr3.jpg. (Accessed: 28th April 2019)
8. Website. Available at: https://www.ajax-zoom.com/pic/layout/tiles_pyramid_600.png. (Accessed: 28th April 2019)
9. Contributors to Wikimedia projects. Deep Zoom - Wikipedia. *Wikimedia Foundation, Inc.* (2008). Available at: https://en.wikipedia.org/wiki/Deep_Zoom. (Accessed: 16th January 2019)
10. Llc), T. M. (aquent. Deep Zoom File Format Overview. Available at: https://docs.microsoft.com/en-us/previous-versions/windows/silverlight/dotnet-windows-silverlight/cc645077(v%3dvs.95). (Accessed: 16th January 2019)

11. [No title]. Available at: https://support.microsoft.com/en-gb/help/140365/default-cluster-size-for-ntfs-fat-and-exfat. (Accessed: 27th April 2019)
12. [No title]. Available at: https://www.itu.int/itudoc/itu-t/com16/tiff-fx/docs/tiff6.pdf.
13. Leica format. Available at: https://openslide.org/formats/leica/. (Accessed: 18th January 2019)
14. Contributors to Wikimedia projects. Microscope slide - Wikipedia. *Wikimedia Foundation, Inc.* (2003). Available at: https://en.wikipedia.org/wiki/Microscope_slide. (Accessed: 1st February 2019)
15. Farahani, N., Parwani, A. V. & Pantanowitz, L. Whole slide imaging in pathology: advantages, limitations, and emerging perspectives. *Pathol. Lab. Med. Int.* **7**, 23–33 (2015).
16. libvips. libvips/libvips. *GitHub* Available at: https://github.com/libvips/libvips/wiki/Speed-and-memory-use. (Accessed: 16th January 2019)
17. Contributors to Wikimedia projects. VIPS (software) - Wikipedia. *Wikimedia Foundation, Inc.* (2012). Available at: https://en.wikipedia.org/wiki/VIPS_(software).
18. Security Considerations — Flask 1.0.2 documentation. Available at: http://flask.pocoo.org/docs/1.0/security/. (Accessed: 15th February 2019)
19. Security Considerations — Flask 1.0.2 documentation. Available at: http://flask.pocoo.org/docs/1.0/security/. (Accessed: 15th February 2019)
20. Boyd, C. Hacker destroys VFEmail service, wipes backups - Malwarebytes Labs. *Malwarebytes Labs* (2019). Available at: https://blog.malwarebytes.com/cybercrime/2019/02/hacker-destroys-vfemail-service-wipes-backups/. (Accessed: 18th February 2019)

21. Column Elements and Expressions — SQLAlchemy 1.3 Documentation. Available at: https://docs.sqlalchemy.org/en/latest/core/sqlelement.html. (Accessed: 18th February 2019)
22. What is perfect forward secrecy (PFS)? - Definition from WhatIs.com. *WhatIs.com* Available at: https://whatis.techtarget.com/definition/perfect-forward-secrecy. (Accessed: 30th January 2019)
23. Secure your site with HTTPS - Search Console Help. Available at: https://support.google.com/webmasters/answer/6073543?hl=en. (Accessed: 1st February 2019)
24. Do SSL Certificates Affect Search Rankings? (A Data Driven Answer...). *Neil Patel* (2016). Available at: https://neilpatel.com/blog/does-a-ssl-certificate-affect-your-seo-a-data-driven-answer/. (Accessed: 1st February 2019)
25. Chris Bailey (General Manager, T. M. S. Extended Validation Certificates: Warning Against MITM Attacks - TrendLabs Security Intelligence Blog. (2015). Available at: https://blog.trendmicro.com/trendlabs-security-intelligence/extended-validation-certificates-warning-against-mitm-attacks/. (Accessed: 1st February 2019)
26. Cryptographic hash function - Wikipedia. Available at: https://en.wikipedia.org/wiki/Cryptographic_hash_function. (Accessed: 30th January 2019)
27. Cryptographic hash function - Wikipedia. Available at: https://en.wikipedia.org/wiki/Cryptographic_hash_function. (Accessed: 30th January 2019)
28. Properties of a Cryptographic Hash Function - Hash Functions | Coursera. *Coursera* Available at: https://www.coursera.org/lecture/classical-cryptosystems/properties-of-a-cryptographic-hash-function-lBx5e. (Accessed: 30th January 2019)

29. [No title]. Available at: ftp://ftp.arnes.si/packages/crypto-tools/rsa.com/cryptobytes/crypto2n2.pdf.gz. (Accessed: 30th January 2019)
30. More MD5 Collisions - Schneier on Security. Available at: https://www.schneier.com/blog/archives/2005/06/more_md5_collis.html. (Accessed: 30th January 2019)
31. Colliding X.509 Certificates. Available at: https://www.win.tue.nl/~bdeweger/CollidingCertificates/. (Accessed: 30th January 2019)
32. MD5 conversion and MD5 reverse lookup. Available at: https://md5.gromweb.com. (Accessed: 30th January 2019)
33. List of Rainbow Tables. Available at: https://project-rainbowcrack.com/table.htm. (Accessed: 30th January 2019)
34. Decrypt MD5 & SHA1 Password Hashes. Available at: https://hashtoolkit.com/. (Accessed: 30th January 2019)
35. Google. Announcing the first SHA1 collision. *Google Online Security Blog* Available at: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html. (Accessed: 1st February 2019)
36. Cryptanalysis of SHA-1 - Schneier on Security. Available at: https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html. (Accessed: 1st February 2019)

37. Computer Security Division, Information Technology Laboratory, National Institute of Standards, Technology & U.S. Department of Commerce. NIST Policy on Hash Functions - Hash Functions | CSRC. *CSRC | NIST* (2017). Available at: https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions. (Accessed: 30th January 2019)
38. [No title]. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. (Accessed: 1st February 2019)
39. Computer Security Division, Information Technology Laboratory, National Institute of Standards, Technology & U.S. Department of Commerce. Research Results on SHA-1 Collisions | CSRC. *CSRC | NIST* (2017). Available at: https://csrc.nist.gov/News/2017/Research-Results-on-SHA-1-Collisions. (Accessed: 30th January 2019)
40. Hernandez, P. NIST Releases SHA-3 Cryptographic Hash Standard. *NIST* (2015). Available at: https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard. (Accessed: 1st February 2019)
41. Keccak Team. Available at: https://keccak.team/keccak_strengths.html. (Accessed: 1st February 2019)
42. [No title]. Available at: https://www.um.es/documents/1765772/1843567/saimagen_info_Leica_SCN400_2010.pdf/a7c4d67d-250f-47b5-b5a8-1e636c152bcb. (Accessed: 1st February 2019)
43. Website. Available at: https://stackoverflow.com/questions/7877282/how-to-send-image-generated-by-pil-to-browser. (Accessed: 9th January 2019)

44. Website. Available at: https://ourcodeworld.com/articles/read/189/how-to-create-a-file-and-generate-a-download-with-javascript-in-the-browser-without-a-server. (Accessed: 9th March 2019)
45. Website. Available at: https://pixabay.com/no/moderne-desktop-web-flatskjerm-3792395/. (Accessed: 9th February 2019)
46. Website. Available at: https://purepng.com/photo/25167/clipart-dedicated-server. (Accessed: 9th February 2019)
47. Leica format. Available at: https://openslide.org/formats/leica/. (Accessed: 6th May 2019)
48. One billion files on Linux. Available at: https://lwn.net/Articles/400629/. (Accessed: 12th March 2019)
49. Website. Available at: https://openslide.org/api/python/.
50. Website. Available at: https://material.io/design/color/the-color-system.html.
51. JS Foundation-js. foundation. jQuery API Documentation. Available at: http://api.jquery.com/.
52. OpenSeadragon API. Available at: https://openseadragon.github.io/docs/.
53. W3Schools Online Web Tutorials. Available at: https://www.w3schools.com.
54. LibTIFF - TIFF Library and Utilities. Available at: http://www.simplesystems.org/libtiff/.
55. Flask-Login — Flask-Login 0.4.1 documentation. Available at: https://flask-login.readthedocs.io/en/latest/.
56. Logging Cookbook — Python 3.7.3 documentation. Available at: https://docs.python.org/3/howto/logging-cookbook.html.

Appendix

Video demonstration link:
https://drive.google.com/file/d/1YOOuZL8yHl_8YqY_-WqaXDBNBJzV1G-x/view?usp=sharing

Source code: *Attached document.*
https://github.com/Ridalor/Bachelor

Installation instructions

Dependencies

- Python libraries
  - Flask
  - Flask-SQLAlchemy
  - SQLAlchemy
  - Flask-Login
  - openslide-python
  - mysqlclient
  - mysql-connector
- Javascript libraries
  - OpenSeadragon
  - OpenSeadragon-scalebar
  - OpenSeadragon-canvas-overlay
- Other
  - Openslide distribution package for unix based operating systems / Visual C++ for Windows | https://openslide.org/download/
  - libmysqlclient20

Clone github repository to wanted location on the unix network

```
git clone https://github.com/Ridalor/Bachelor.git
```

Flask requirement:
Flask needs a secret key to be set for use of sessions. This key is read from a seperate text file that must be created.

Name it “SecretKey.txt” and put in a randomly generated string on line 1. Make sure the text file is only one line in length.

Database requirement:
Similarly to “SecretKey.txt” there needs to exist a file named “Login.txt” that contains the username, password, databaseURL and databaseName for the database running on the unix net, each variable needs to be separated by a |, and with no spaces or newlines. This is then read during server startup to connect to the database.

After the database structure is created as shown in the document, there needs to be added a admin user. The easiest way to do this is simply commenting out 3 lines of code.

In app.py:
Comment out *@login_required* for the register route.

```
@app.route("/register", methods=["GET", "POST"])
@login_required #Comment out this line of code
```

In userHandlingpy:
Comment out the two lines that checks if the user type is Admin, and *abort(401)*

```
if str(current_user.type) != "Admin": # Comment out this line of code
       abort(401) # Comment out this line of code
```

Once these are commented out, the server can be started and histology.ux.uis/register can be accessed without login in.

URLs:

In main.js, a global variable *serverUrl* is used to set the url where the frontend communicates with the server. Change this to the external access URL of your server.

Example:

```
var serverUrl = "https://histology.ux.uis.no";
```

Other Paths that might need to be changed:

XmlAndDB.py line 7 set where to store XML files created by the server.

```
xmlStoragePath = "//home/prosjekt/Histology/thomaso/"
```

imageList.py line 6 and 32 + app.py line 58.

```
6       listOfImages = glob.glob("//home/prosjekt/**/*.scn", recursive=True) #imageList.py
```

```
32      foo.append(element.replace("//home/prosjekt", "")) #imageList.py
```

```
58      path = "//home/prosjekt"+imagePathLookupTable[folder+"/"+filename] app.py
```

The root path, in this case “//home/prosjekt” should be the same in each of these locations.

Running the application:

Starting the application is simply done by running

```
python3 app.py
```

while located in the same folder as the repository.