



Data Collection for Automatic Wound Segmentation

Graduation assignment by Mattijs Kuhlmann at Nedap Healthcare for the creation of a data collection application for automatic wound segmentation.

FINAL report version 1.0 (17-06-2019)

Graduation period: 11-02-2019 until 07-07-2019

Student: Mattijs Kuhlmann (424446)

Company: Nedap (Healthcare)

Company supervisor: Michiel Klitsie

School: Saxion University, Enschede

Study: HBO-IT: Software Engineering

School coach: Jan Stroet

Table of Contents

Abstract	3
1: Introduction.....	4
2: The Process.....	5
2.1: Phases.....	5
2.1.1: Planning phase	5
2.1.2: Global research phase	5
2.1.3: Implementation phase	6
2.1.4: Review phase.....	6
2.2: Programming Strategy & Process Documentation.....	7
3: The Product - Design	8
3.1: The Context	8
3.2: Global Design	10
3.3: Specific Requirements and Design	11
3.3.1: Front-end requirements.....	11
3.3.2: Front-end design	12
3.3.3: Back-end requirements	14
4: The Product - Realization	15
4.1: Back-End Application	16
4.1.1: Initial implementation choices.....	16
4.1.2 API functionality	16
4.1.3 Security and Testing	21
4.2: Front-End Application	23
4.2.1: Initial implementation choices.....	23
4.2.2: Android implementation	25
5: Conclusion and future recommendations.....	37
6: References.....	38
7: Appendix	39
7.1: Initial sprint backlog	39
7.2: Sprint progress & documentation example	41

Abstract

This document contains the description of the graduation assignment of Mattijs Kuhlmann, part of the Bachelor HBO IT, Software Engineering of Saxion University of Applied Sciences in Enschede, the Netherlands. The assignment has been carried out at Nedap Healthcare, located in Groenlo, the Netherlands.

This document explains the creation of a data collection application prototype for wound photos, which will support the **automation of the wound care process**. It includes the description of a process, created designs and an explanation of the realization of the application. The design and realization chapters explain the reasoning and workings of the design and technology of the application.

The realized application consists of a front-end Android application that uses a back-end Python web service to enable a user to collect wound photos on which the area of the wound is marked. The initial marking is generated by a wound-segmentation algorithm, which is called Vesalius. A user is then able to give feedback on this initial marking. The back-end is able to securely store the photos and marking of the wound area, for it to be used for future training of the Vesalius algorithm.

Future steps to this project involve the integration of parts of this prototype into a wound care application, which eventually could lead to automation of wound care registration processes.

1: Introduction

Healthcare expenses in the Netherlands made up 10.1% of the total GDP in 2017 (Zorguitgaven stijgen in, 2018). The healthcare sector in the Netherlands is big, because a lot of people get into contact with this sector. The sector is also old, and this makes it almost unavoidable that some processes are outdated and slower than they could be with current technical possibilities. That's why there are many technology companies that aim to update this sector, and one of them is Nedap Healthcare. Nedap Healthcare is a market group within Nedap, that creates software solutions to increase efficiency of healthcare processes and aims to automate as many steps as possible. This results in giving caretakers more time for health care process. These products are made for their clients, which are mainly healthcare organizations in the Netherlands. The market group has deployed multiple products and is working on many new ones as well.

One of these new products is aimed at wound care. Wound care is a big part in healthcare and many clients of Nedap Healthcare currently use their systems to keep track of wounds by putting images and information about them in an electronic health record. This process, however, is tedious and most clients work in different ways. Because of these reasons an application is being developed which aims to make taking care of wounds by caretakers more efficient, more standardized and eventually more automated. The name of this application is "Ons Wondzorg".

Currently, a team within Nedap is working on a Minimum Viable Product (MVP) of this application. The basic functionality of this application focusses on selecting a client, taking pictures of a wound and then filling in information about the wound. Some of the information which is needed is the size of the wound and the types of tissue of the wound. But manually measuring and filling in this information takes time, while this is a job that is a good candidate for automation. With this goal in mind, a first version of a wound segmentation algorithm was created by Nedap and the project has been called "Vesalius", after the Flemish-born anatomist. The algorithm segments the area of a wound on a picture with the help of a neural network.

Automating measurements like extracting the size of the wound and classifying different kind of tissue all depend on finding the actual location of a wound inside a picture. It is important that this process is optimized. Improving a neural network greatly depends on gathering data, as the result of neural networks improve with the amount of data. However, **the current neural network does not have enough data to be able to give back good results**. Besides the lack of data, another problem is that it is hard to understand and to know what this wound segmentation algorithm can do in practice because **there is no practical implementation** of the neural network yet.

With the problems clear, a global description of an assignment was formulated that is able to tackle the problems mentioned above:

*Create a mobile **Android prototype application**, which should be able to take photos, show the **result of the wound segmentation** algorithm and give the user the opportunity to provide **feedback** on it. The images and the feedback should be stored at a secure location for future improvement of the wound-segmentation algorithm.*

This document will explain the realization of the assignment from planning to actual implementation. This document has the following goals:

To explain how the product evolved from design to implementation, to explain what specific choices were made, to make the reader understand some of the complex technical workings behind the product and to make the reader understand and learn from some of the mistakes that were made during the implementation that led to changes.

2: The Process

To get from the problem description towards a software system, a structured process needed to be designed. This resulted in the formulation of phases, which have their own input and output in terms of products.

2.1: Phases

Figure 1 contains a visualization of the phases, their order, the flow and the input and output products.

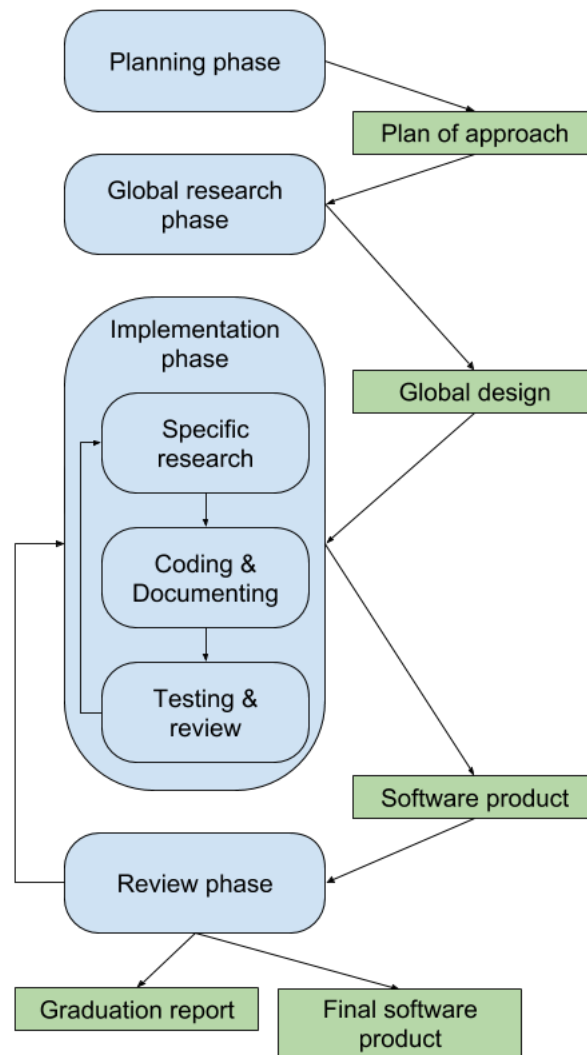


Figure 1: Phase flow diagram, graduation assignment

2.1.1: Planning phase

The start of the graduation assignment consisted of getting to know the company and to start getting familiar with the scope of the assignment. The product of this phase was the **Plan of Approach**, which describes the assignment, the plan of the implementation process and the global planning of the assignment.

2.1.2: Global research phase

After finishing the Plan of Approach, the next step was to work towards a global design of the final product. The goal of this phase was to have a good overview and idea of the end product. For this, the following points needed to be clear:

- The requirements of the product
- The initial backlog of tickets, extracted from the requirements
- Decisions on techniques and different kind of technical choices that will be used for the realization of the product
- Global and more specific visual designs of the separate parts of the product

These points were concluded in a *global design document*.

2.1.3: Implementation phase

The implementation phase will be the longest phase, as this will be the phase in which the final product is realized. The input for this phase will be the global design document.

As can be seen in figure 1, the implementation phase consisted of multiple smaller stages: *specific research*, *coding & documenting* and finally *testing & review*. These stages were traversed multiple times for multiple parts of the product. Note that this visual representation shows a target process and that the order and occurrence of every stage changed for every specific part that was worked on. This was especially the case for testing, because that is a process that came back in every stage of the implementation phase. There was also a lot of room for research and trying things out in this phase, because that was the most realistic way to create software, especially for a student.

In the **planning phase**, a strategy had been created which is based on Scrum and Kanban. The goal of this strategy was to make the **implementation phase** *focused*, *structured* and *reviewable*. The explanation of this strategy can be found in chapter 2.2.

2.1.3.2 Implementation stages

2.1.3.2.1 Research

Each part of the product will have its own specifics that will require implementation choices. In the global design document, as the name says, global design choices are made, but more specific questions arise when working on a specific part. Without enough experience to go in depth in advance, the decision has been made to go in depth as soon as a ticket from the initial backlog has been taken. This going in depth will raise questions, which will be answered in the explanation about the product.

2.1.3.2.2 Coding & Documenting

Based on the ticket, the design, the specific research and with help of the defined strategy, the part will be implemented. This also includes any documentation to make the code understandable, like code comments, diagrams and a clear markdown files in the repositories.

2.1.3.2.3 Testing & Review

After coding has been finished, according to the strategy, the ticket will be moved to Review. This is the moment where there will be time to test functionality and to let the supervisor do this as well. Based on the result of this phase, a story will go to either Done, or it will go back to Doing.

2.1.4: Review phase

The *review phase* will be entered once the *implementation phase* is finished, thus all tickets are in Done. This, however, does not necessarily mean that the product is final. This last phase has the goal to thoroughly review and test the product and to really think about future changes, conclusions and recommendations to be mentioned this graduation report. If, however, some functionality fails this review, there is the option to go back into the *implementation phase* if necessary.

Finally, there will be time to wrap up and complete all documentation and prepare the final presentation, based on the final software product and this report.

2.2: Programming Strategy & Process Documentation

During the **implementation phase**, the following programming strategy has been used to make the programming more *focused, structured* and *reviewable*.

- Once an initial sprint backlog has been created, it will be placed on Trello and it will consist of the following lists:
 - Backlog, Doing, Review, Done (including the sprint in which I finished it)
 - These tickets will consist of the following points:
 - A functional description of the functionality that will be implemented
 - One or more type tags: *Android, API, Server (or others)*
 - A priority tag: *low, medium, high, highest*
 - When Done: A time approximation in working days that it took to finish the story (Cycle time in Kanban)
 - A ticket will be moved from Review to Done once the *definition of done* has been met (see below).
- Every time a ticket is finished, another one will be picked up, like in Kanban. The next ticket will be picked up based on the *timeline, priority* and *dependencies*.
- The work will be reflected by doing “sprints” of *two weeks* for a total of six sprints and doing a (small) retrospective after each sprint, in which the following things will be documented: the stories that were burned, how long those stories took to implement, how the process went and how it could be improved.
- To be able to keep the team, including the company supervisor, up to date on the current progress and blocking issues, the daily standups of the “Ons Wondzorg” team will be joined. This will also help to learn more about the progress and strategies Nedap uses to develop applications.
- At least once per week, a meeting with the company supervisor will be planned to discuss the progress based on the planning, and during the implementation phase, based on the Trello board.
- **Definition of Done**
To be sure that a ticket is really finished, the weekly meeting with the company supervisor will be used to go through the tickets that are in the Review list. Besides this, code will be reviewed by the company supervisor (or other people) by means of pull requests on GitHub. Once the code has been reviewed and OK’d, the user story will be moved to Done, and otherwise, the story will go back into Doing.

As mentioned above, the programming has been divided into sprints in which tickets are finished. This sprint process documentation consists of the following parts per sprint:

- User stories finished and the time spent on these.
- A small retrospective consisting of: what went well, what didn’t go well and what am I going to do different next time?

A visualization of the completed user stories per sprint as well as an example of the sprint documentation can be found in the *Sprint progress & documentation example* in the appendix (chapter 7.2).

3: The Product - Design

Disclaimer: as this project is about wound care, this chapter contains some graphical images.

This chapter aims to explain the design of the final product, the forming and reasoning of requirements and initial designs.

The starting point was the assignment description:

*Create a **mobile prototype application**, which should be able to take photos, show the **result of the wound segmentation** algorithm and give the user the opportunity to provide **feedback** on it. The images and the feedback should be stored at a secure location for future improvement of the wound-segmentation algorithm.*

3.1: The Context

The first thing that had to be done was getting a good overview of the context and starting situation of the assignment.

The application can best be described as a **data collection tool for wound images**, which will be used to train a wound-segmentation algorithm. In the future, the goal of this algorithm will be to automate steps in the process of taking care of wounds in the “Ons Wondzorg” application. The current vision of Nedap for the use of this algorithm can be seen in figure 2. The parts on which this assignment focuses on are shown as light blue arrows, which is mainly about the wound segmentation. Besides this, arrows in figure 2 with a light blue outline depend on the wound segmentation part.

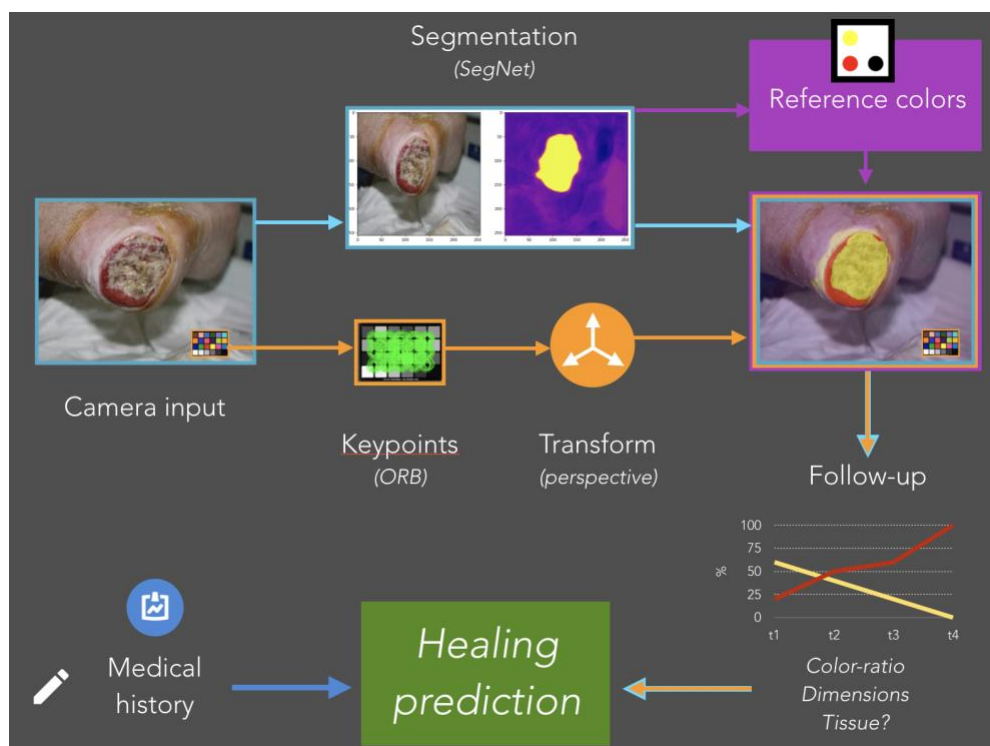


Figure 2: Vision of the use of automatic wound-segmentation by Nedap

The prototype could be used to collect data from patients. In the systems of Nedap Healthcare, these patients are recognized with a Care organization ID and a Patient ID. Currently, the prototype of the “Ons Wondzorg” application is built as a **native Android Kotlin** application, because most clients in this sector are Android phone users. The users of the wound care application are **professional care**

takers in home care and at health care institutions. The data collection application has the **same target group**, as parts of it can be used in the wound care application in the future.

The application will be used to collect data for training a wound-segmentation algorithm, as the application will also utilize this algorithm and show the result, it is of importance to know the inspiration and basic workings of it. The predicting model is generated by a neural network and is based on research by Wang, C., et al (2015). Their neural network consists of different convolutional layers which are visualized in figure 3. This algorithm is recreated at Nedap as a Python program with the TensorFlow and Keras libraries. The version of Nedap has been called **Vesalius** and takes an RGB image of any size, reshapes it to a 256*256 RGB image and returns a 256*256 **probability mask**. This mask consists of a probability value per pixel in a two-dimensional array. The value represents the pixel's probability that the pixel is part of the wound area.

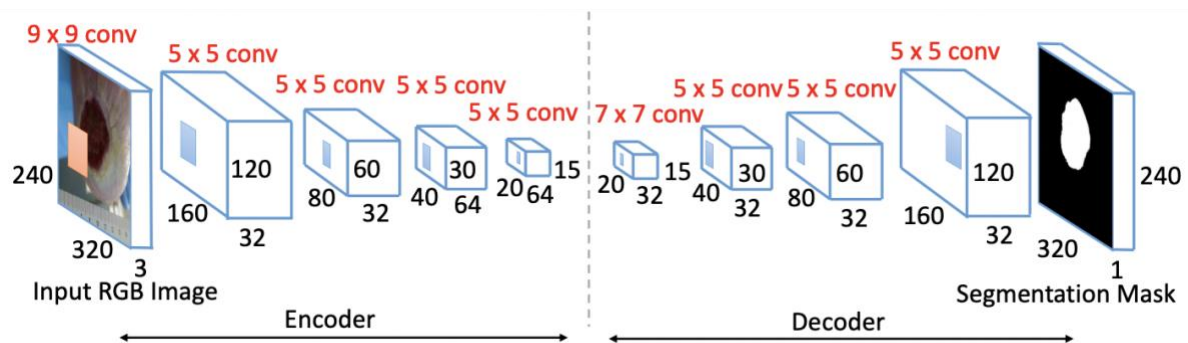


Figure 3: Visualization of Convolutional neural network for wound-segmentation (Wang, C., et al, 2015)

Vesalius can be trained with a test set of pairs of RGB images. These pairs consist of a wound image JPEG file and a pixel annotated PNG image file. Figure 4 shows an example of such a pair.



Figure 4: example of training data images

As can be seen, the pixel annotated image consists of three different colors:

- Wound area: RGB(255,0,0)
- Skin area: RGB(0,255,0)
- Background: RGB(192,192,192)

Note that the current version of the Vesalius algorithm only predicts the wound area and not the skin area. The algorithm therefore only makes a difference between the wound area and the background (which also includes the skin area).

3.2: Global Design

The input and output of the Vesalius algorithm were a very important starting point, as these things were the constants in the assignment. The next step was to start with an initial design of the application which consisted of requirements, implementation choices and basic designs.

To fulfill the assignment goals, the decision was made to create two different applications:

- **A front-end application.**
 - o This application must handle all user interaction of the assignment, mainly consisting of taking photos and giving feedback.
- **A back-end application.**
 - o This application must handle the execution of the wound segmentation algorithm and storing of the feedback.

Next, requirements were formulated for the separate front-end and back-end application. These requirements were all based on the research that was done to come to the assignment description and the context. A visualization of the requirements and their interaction can be seen in figure 5.

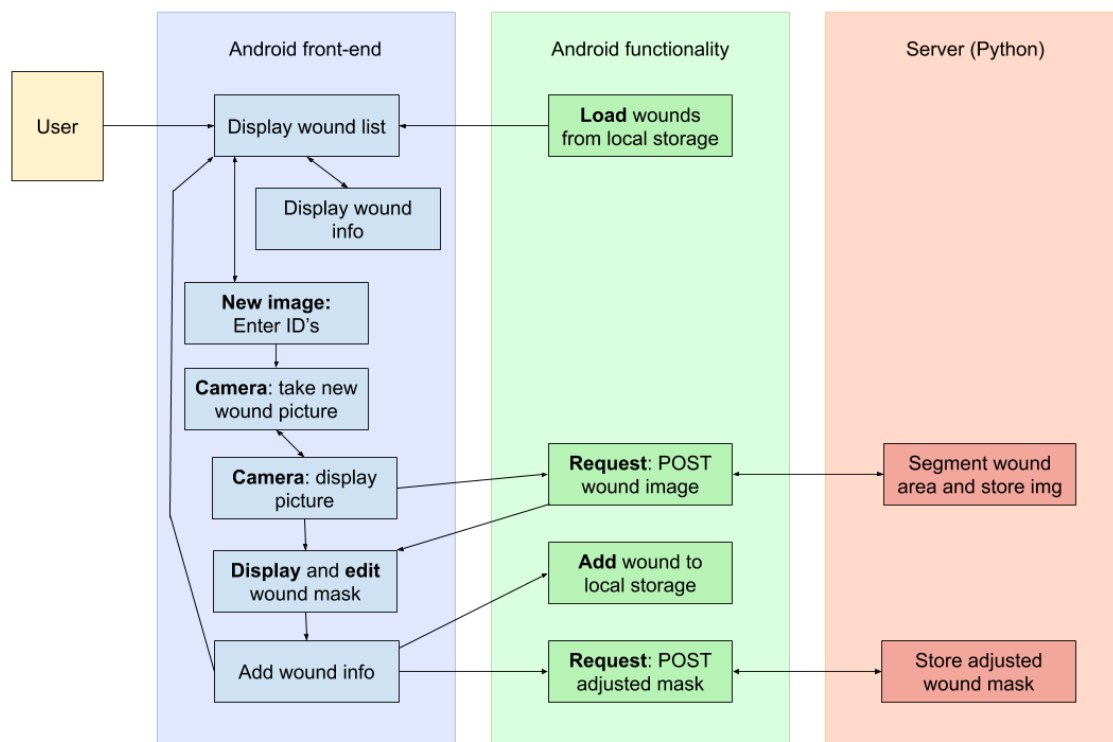


Figure 5: global functionality design diagram

3.3: Specific Requirements and Design

This chapter contains the requirements that were set up for the specific applications, as well as initial designs.

3.3.1: Front-end requirements

For the front-end application, the following list of requirements was formulated.

#	Requirement	Explanation
1	The front-end application must be an Android application.	The front-end application needed to be a mobile application that can be used by caretakers while taking care of wounds. It also needs the option to be implemented in the wound care application that is currently being developed. That application is made in Android, which is the mobile operating system that the users also mainly use in practice.
2	The application should be able to load wounds that were previously taken by the user. These wounds should be displayed in a list and clicking on a list item should show the details of the wound photo.	To see the photos that the user has taken before and to check the wound area on these images, opening the application should show a list of all the previously taken images. Clicking on one of these images would show the image details. This requirement is not necessary for the data collection process; therefore, it is formulated as should .
3	A user should be able to see the details of a previously taken wound photo, along with the information.	When clicking a photo in the list, a user should see the details of the photo and the photo itself. This requirement is also formulated as should because it is not necessary for the data collection process.
4	A user must start the data collection process by filling in a Care Organization ID and Patient ID.	To start the data collection process, some data needs to be filled in by the user for structured storing of the results. The photos that will be taken can be stored using the same ID's that Nedap uses in their system to structure patients (<i>Care Organization ID</i> and <i>Patient ID</i>). For the prototype, this information needs to be filled in manually. After integration into Nedap applications, this information is available within the app.
5	A user must be able to take a photo of a wound.	The next step in the process is taking a photo of the wound, as this is one of the inputs in the Vesalius algorithm. Users must be able to take this within the application.
6	A user must be able to view the photo and then accept or otherwise retake it.	A user must be able to review the photo, then accept or be able to retake it.
7	The application must be able to send the taken photo to a server and receive the segmentation result.	With the patient information ready and a photo taken, the data is available to segment the wound from the image. As the wound segmentation algorithm will be hosted on a separate server application, the photo must be sent there and retrieve the result.
8	A user must be able to see the result of the wound-recognition algorithm on top of the taken wound photo.	After the segmentation result has been retrieved, the result needs to be drawn on top of the taken picture, so the result can be displayed and understood.

9	A user must be able to adjust the result of the wound-recognition algorithm.	The next step in the application flow is for the user to give feedback on the result of the algorithm. The user must be able to do this by changing the wound area on top of the taken photo.
10	The application must be able to send the wound area result to a server.	After the user adjusted the wound area, the final result must be stored. As this is handled by the back-end, the result must be sent there.

3.3.2: Front-end design

The front-end requirements were used to make an initial design of the application with the following screens:

1. A **wound list** screen with a list of cards that contains already photographed and segmented wounds. And a floating action button to add a new wound.
2. A **wound detail** screen, which displays the taken picture and shows patient information
3. An **enter patient info** screen, in which the user enters patient information before taking the picture.
4. A **camera screen** where a user can take a picture. After taking a picture, it will be shown, and the user can decide to either take a new picture or continue to the next screen in which the result of the wound-segmentation algorithm is shown and can be adjusted.
5. A **wound area adjustment** screen which displays the picture and draws the segmented wound area on top of it. The user can adjust the layer here. It will contain buttons to change the draw or erasing size, an undo button, an eraser button and a continue button. Optimally, it would also be able to zoom in on specific areas of the photo to increase precision.

An initial visualization consisting of mockups of the above-mentioned screens can be seen in figure 6. This figure consists of mockups that show the initial design of the five screens that are mentioned above. The most interesting design is the adjust layer screen mockup. This will be the most complex screen of the application because a user must be able to give feedback on the wound area. The plan was that a user can do this by touching the screen. The following buttons were added in this design:

- A hand button which let the user zoom in and move the photo to increase specific drawing.
- A button that let the user change the draw size.
- A button that let that user switch between drawing and erasing.
- An undo button that let a user undo its last action.

The main goal of these mockups was to have a direction to work towards. As specific research is done during the implementation phase, the realized screens are expected to be different.

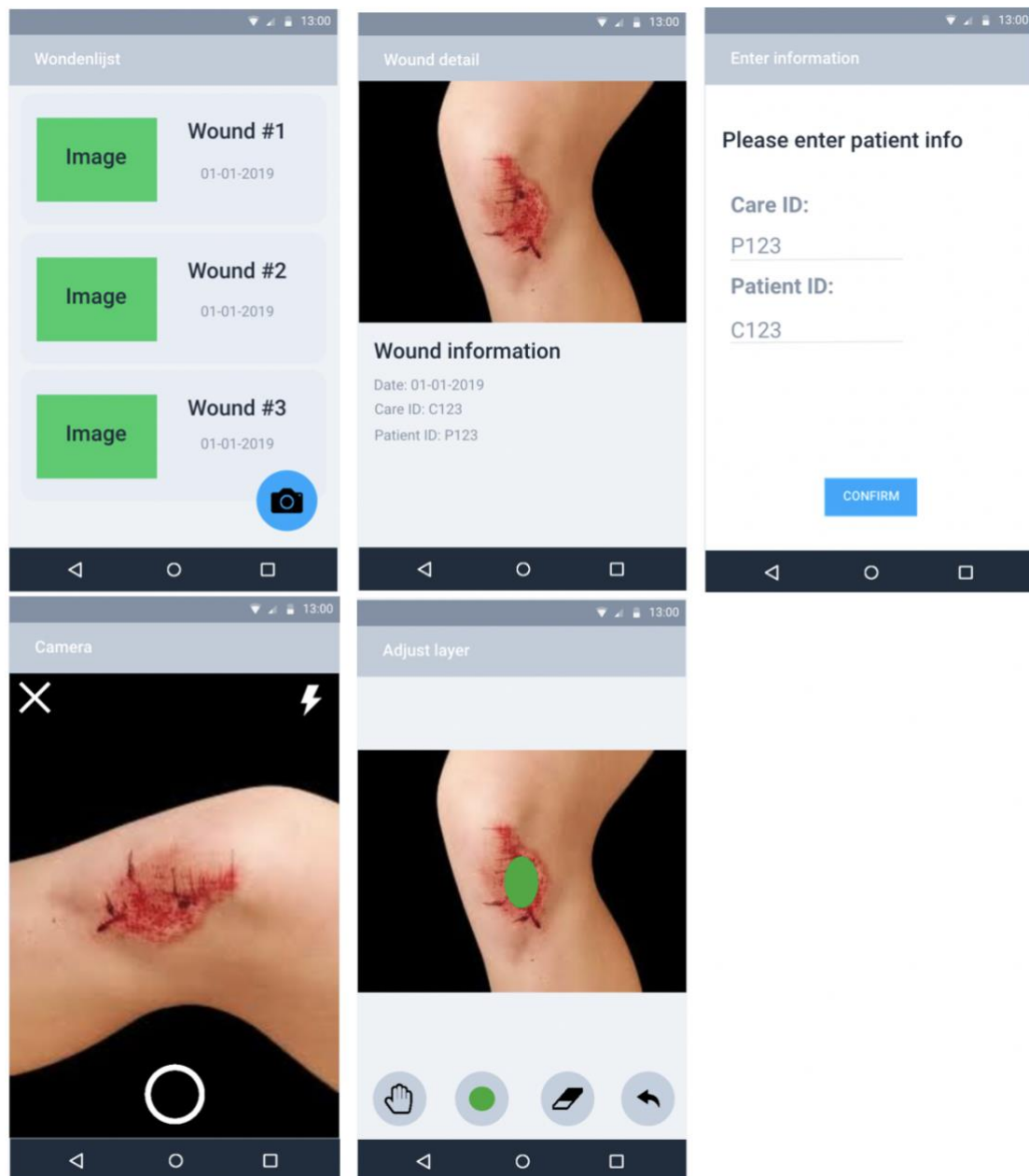


Figure 6: Initial front-end screen designs

3.3.3: Back-end requirements

The functioning of the front-end application depends on the back-end application. The main functionality of the back-end consists of segmenting wound images with the Vesalius algorithm and storing the result in the format which the Vesalius algorithm uses for training.

The following list of requirements for the back-end application was set up:

#	Requirement	Explanation
1	The back-end API must be available to the front-end application via internet.	The front-end application must be able to communicate with the API from all locations, so it needs to be connected to internet.
2	The back-end application must be able to receive photo data.	The API needs to have an endpoint that receives photos from the front-end application.
3	The back-end application must be able to load a TensorFlow model and apply it to images to generate a wound-segmentation result.	The back-end must be able to apply the Vesalius algorithm to the received photo and generate the wound-segmentation result.
4	The back-end application must be able to return the wound-segmentation result.	Once the Vesalius algorithm has been applied to the photo, the API call has to send the wound-segmentation result back to the front-end application.
5	The back-end application must be able to store the image JPEG and wound-segmentation result as an annotated pixel PNG.	To be able to use the data for future training, the server must store the original photo in a structured way. The server must also store the result of the wound-segmentation, so that the initial result of the algorithm can be compared with the feedback from the user.
6	The back-end application must be able to receive an adjusted wound-segmentation result.	The API must have an endpoint which receives the feedback of the user. This feedback consists of the pixels on the photo that are part of the wound.
7	The back-end application must be able to store the adjusted wound-segmentation result as an annotated pixel PNG.	The final feedback result, received from the previous API call, must be stored on the server for it to be used for future training.

*Note that this application is used for **data collection**, so it will not be possible for the stored data to be retrieved in a later stadium **by the application**. This choice was made because of the use case of this application (data collection) and the fact that this application handles sensitive data and requires a lot of certification and airtight authorization to be able to do this.*

4: The Product - Realization

Based on the requirements, the product was realized. This chapter explains the realization of the product by explaining the **implementation choices**, and the **final implementation** along with **specific challenges** that have been tackled. This chapter is divided in an explanation of the back-end and front-end. The implementation was given structure by creating an initial backlog backed on the requirements and implementation choices, this initial backlog can be found in the appendix (chapter 7.1)

Where *figure 5* in *chapter 3* shows a visualization of the design of the product, *figure 7* shows a visualization of the actual realization of the product. The main differences consist of not having the implementation of a wound list and wound detail screen in the application. Besides that, an extra step has been added in changing the wound-area: changing of the wound threshold. On the back-end a more specific plan was made for storing the collected data in a separate data storage location.

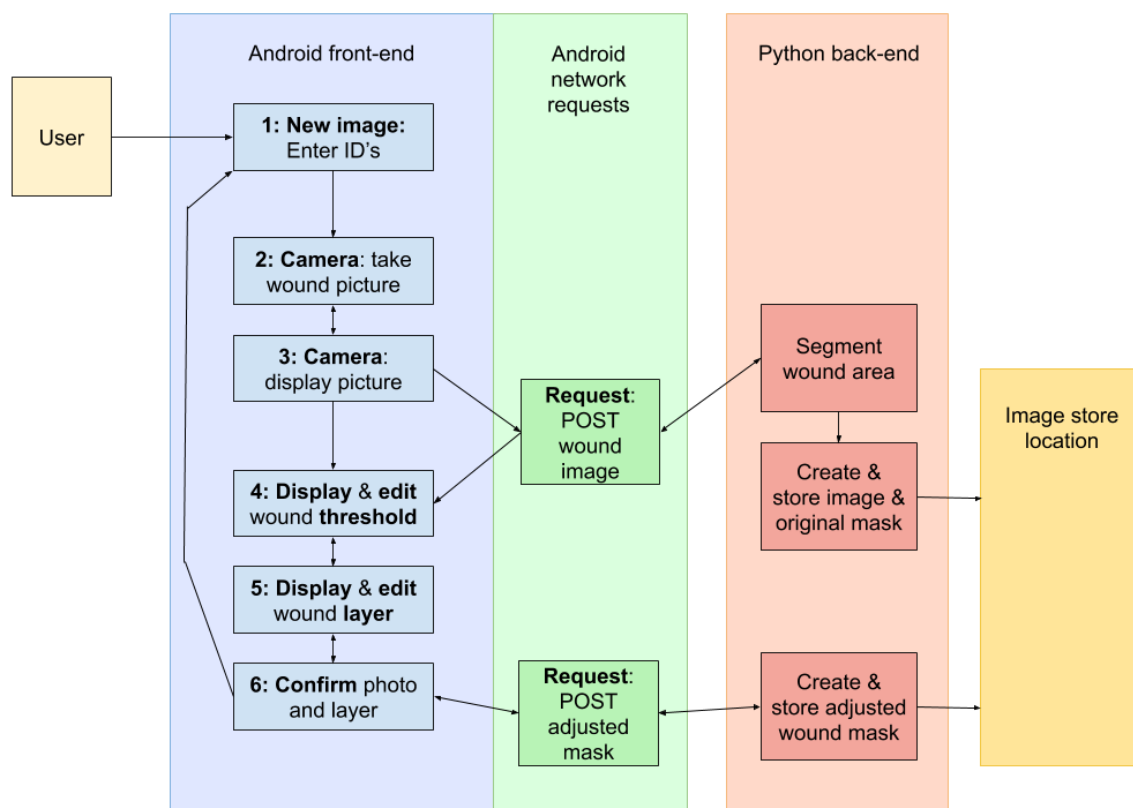


Figure 7: Realized functionality realization diagram

In this chapter; the choices, challenges and implementation are explained underneath in the order: back-end -> front-end. As this was the order in which the application was implemented. Starting with the back-end made it easier to imagine and design the specific data flow from front-end to back-end. And it would result in being able to start implementing the front-end without having to use mock-data.

The back-end implementation exists in a GitHub repository with the name: vesalius-backend.

4.1: Back-End Application

This part consists of the description of the realization of the back-end. First, implementation choices will be explained. What follows is the actual implementation of the product and the explanation of important challenges and choices that were made.

4.1.1: Initial implementation choices

After initial research, the following **implementation choices** were made to implement the back-end.

The first thing that was decided was the programming language in which the API and Server were going to be implemented. The choice was made to write the whole back-end application in **Python 3.6** because of the following reasons:

- It is possible for a Python application, with help of libraries, to **fulfill all back-end requirements**. Of which the main requirements are being able to **host API endpoints** and to **apply the Vesalius algorithm**.
- It is possible to keep the **complexity low** by keeping the API and Server in the same repository.
- There are **code examples** and **people with expertise** available within the company.

The choice to use Python depended on library support for the main requirements **host API endpoints** and **applying the Vesalius algorithm**. For hosting API endpoints, the Connexion library ([Connexion](#)) has been used. This choice has been made because it supports the creation of a REST API service based on an OpenApi specification ([OpenApi](#)). Connexion uses a design-first approach which makes sure that any API is properly designed and documented before it works. As the OpenApi specification also contains examples values, specific calls can easily be simulated. This can be used for testing the application with specific inputs. Connexion also supports authorization with API keys and has HTTPS support.

For implementation, a good IDE was needed. Visual Studio Code ([Visual Studio Code](#)) has been chosen because it is intuitive and has plugin support, like Python autocompletion. These plugins also support the use of linters and formatting. Linting was done with pep8 ([pep8](#)) and formatting with autopep8 ([autopep8](#)).

The Vesalius back-end works with the Tensorflow library ([Tensorflow](#)) and uses Keras as a high-level API ([Keras](#)) to export and import a machine-learning model. That's why these libraries have been used within the application for the image wound-segmentation.

Testing of the back-end will be done manually with help of the OpenApi specification. Automated testing of the API is possible with pytest fixtures ([Pytest Fixtures](#)).

4.1.2 API functionality

The functionality and implementation of the back-end can best be explained by looking at the different API calls separately. Every API call has its own functionality and challenges, which will be explained here.

The Python back-end can be reached via an API, the full specification is documented in an OpenApi specification, which is required by the Connexion library. This specification is specified in a YAML file and is located inside the vesalius-backend repository at vesalius/vesalius_api.yml. However, the Connexion library also gives the option to view a parsed version of this specification by navigating to the API endpoint: `<{BASE URL}/ui/>`

After realizing the back-end, a development API has been deployed and can be reached via the following base URL: <https://development.ons-vesalius.nl/> and as a result; the OpenApi specification can be viewed and used for testing by navigating to <https://development.ons-vesalius.nl/ui/>.

4.1.2.1 Request GET /status

The first request is a small request which is outside the scope of the assignment, and only used for a status check of the API.

Request and endpoint	GET /status
Functionality	Simple request to check the status of the API. This request is required for all Nedap API's to see if the API can be reached.
Specification request data	No specific header data required.
Specification return data	Returns 200 and no specific body data.

4.1.2.2 Request POST /wound_image

The second request is a POST request which handles back-end requirements 2, 3, 4 and 5, which are about the functionality to extract the wound-area from a photo, return this area (in the format of a probability mask) and store the image.

The specification below contains the realization of the API call, but this realization contains some implementation choices that have to be explained further. These implementation choices were made based on problems and questions that came up during the realization of this API call.

- Because this API call is based on the requirement that a photo has to be sent to the server from the front-end application, the format of this data needed to be determined. The options that were considered were:
 - o Uploading the image to an external database and only sending the URL of this image to the server. This option has a high complexity due to adding another data store into the equation, because the image has to be uploaded first to this store. This option would have positive impact on server load and security, due to more optimized data handling by the external data store. Besides this, the data doesn't need to be stored separately by the server. And it makes fetching the images in a later stadium easier.
 - o Sending a Base64-encoded photo to the server. This option has a low complexity due to the data being a string value that can be handled with one request to the server. The downsides consist of a large data size and an unencrypted image string.

The choice was made to keep complexity low and send Base64 encoded images. As this option has the lowest complexity and the application is a development prototype. In a production application, the preference would be the first option. The main downsides of sending raw Base64 are the increase of data size of the requests and the fact that *unencrypted* Base64 can be intercepted and seen by unauthorized parties. However, these downsides will be tackled in the prototype by setting a maximum image size on the application and the data could be encrypted by using an HTTPS connection.

- The parameters of this request were chosen as follows:

- **Store_image** was chosen to add the option to either store the images on the server or not, because this would give the option to reduce the data created during development of the application.
 - **Get_scaled_mask** was added to tackle an interesting challenge. The Vesalius algorithm takes a 256*256 rescaled image and also returns a 256*256 probability map. However, the original image which is sent from the front-end application will most often be larger than this. This parameter gives the option to let the API return the probability mask of the size of the original image. This reduces the calculation load and the complexity of the front-end application by removing the need to rescale. The downside is that it increases the size of the return body in most cases. Optimally, scaling could be done on the front-end application to decrease the size of this return body.
- As can be seen in the second argument, **get_scaled_mask** was added because of resizing challenges. Because the Vesalius algorithm takes and returns only 256*256 data objects and because the size of the photo that is taken on the front-end application is almost certainly larger than 256*256, resizing was an important factor in the application. The choice was made to handle this part on the back-end in this prototype. Python has a lot of libraries that handle this very easily compared to Kotlin (the language of the Android application). One of the main downsides to this solution consist of an increase in bandwidth use of network requests. Another downside is less flexibility within the front-end application to display the image and the wound area. This choice has mainly been made to decrease complexity and to be able to get to a working prototype. However, this is a part that has to be looked at again in the future.
 - Another important choice that was made was the format of the probability mask. The Vesalius algorithm returns a floating-point value for every pixel in the image. When resizing this on the back-end side, this potentially results in really large data objects that are returned in the body and that have to be loaded into memory in the front-end application. To reduce the size of these objects, the choice was made to send a byte value per pixel instead of a floating-point value per pixel. This was done by multiplying the floating-point values by 100. A floating-point value of 0.55 would become byte-value 55, which saves bandwidth when sent in a request.
 - To make sure that pairs of images can be found in the data store. The images are stored in a specific structure, which is based on the care organization ID, the patient ID and the image ID. This format of structuring makes sure that images from the same patient are stored together and that each image will end up having a unique store location. More information about this structured storing of images can be found in chapter 4.1.3.2.

Request and endpoint	POST <code>/wound_image?store_image={boolean}&get_scaled_mask={boolean}</code>
Functionality	Request with body which includes a wound image and information about the image, stores the image on the server and returns a wound-segmentation result. The image is stored on the server as well an initial result of the wound-segmentation algorithm as an annotated pixel image. This pixel image is generated based on an optional specified probability threshold. The call returns the complete probability map, consisting of a probability value per pixel. The size of this map depends on the parameter: <code>get_scaled_mask</code> .
Specification request data	Header: <ul style="list-style-type: none"> - Requires the key X-API-KEY in header for authorization, with the API authorization key as value. Parameters:

	<ul style="list-style-type: none"> - Store_image is a Boolean if the images should also be stored on the server. This query is mainly for development purposes. As a lot of test-images will be sent during development, it is not necessary to store these. - Get_scaled_mask is a Boolean if the returned probability map should be scaled back to the dimensions of the sent image. <p>Body:</p> <p>Contains an image with information about the image (care organization ID, patient ID, image ID), which is used for structured storing of the images. The body also contains an optional threshold for the initial stored annotated pixel image (default: 0.5).</p>
Specification return data	<p>Success: Body with a two-dimensional array of probability values of the mask, which represents the image matrix. It also returns information about the dimensions of the image and mask and the names of the files in which the images were stored on the server.</p> <p>Error: The specific return codes are specified in the OpenApi specification. Any errors during the request are currently handled by sending the Python exception message back.</p>

4.1.2.3 Request POST /user_adjusted_mask

The third request is a POST request which handles back-end requirements 6 and 7, which entail the functionality to store the adjusted wound-area.

The specification below contains the realization of the API call, but also this realization contains some implementation choices that have to be explained further.

- An important choice was the format in which the adjusted wound area would be returned. First, the choice was made to return an array of X and Y pairs of every pixel which was part of the wound was sent. This meant that the maximum size of this array could be two Integer values for every pixel in the image. Realizing this, two things were changed to decrease this maximum size:
 - o Instead of an array of X and Y value pairs, an array of indices was sent. These indices are part of a one-dimensional array that represents the matrix. Where index=0 equals the pixels at x=0 and y=0, and index=Max (width * height) equals the pixels at x=Width and y=Height. This divided the maximum size of the array by a factor of two.
 - o Another change was made that involves the parameter **send_wound_indices**. This parameter is set to True if the array of indices contains all the pixels that are part of the wound or False if it contains all indices that are **not** part of the wound. If the number of wound-pixels is larger than half of the total number of pixels, the pixels that are **not** part of the wound can be sent instead. This again, has a maximum reduction of the array size by a factor of two.

Request and endpoint	POST /user_adjusted_mask?store_image={boolean}&send_wound_indices={boolean}
-----------------------------	---

Functional ity	Request with body which contains the user-adjusted wound area and information about the image, stores the user-adjusted annotated pixel image on the server.
Specificati on request data	<p>Header:</p> <ul style="list-style-type: none"> - Requires the key X-API-KEY in header for authorization, with the API authorization key as value. <p>Parameters:</p> <ul style="list-style-type: none"> - Store_image is true if the annotated pixel image should be stored on the server. This query is mainly for development purposes. As a lot of test-images will be sent during development, it is not always necessary to store these. - Send_wound_indices is true if the user-adjusted wound area contains the wound pixels or false if it contains the pixels that are not part of the wound. <p>Body:</p> <p>Contains an array of pixels of the image that are part of the wound area. It also contains the information about the image used for structured storing (care organization ID, patient ID and image ID). Along with the dimensions of the wound-image, used for creating the annotated pixel image.</p>
Specificati on return data	<p>Success: Returns a body with the name of the file in which the image was stored on the server, along with the dimensions of the stored image.</p> <p>Error: The specific return codes are specified in the OpenApi specification. Any errors during the request are currently handled by sending the Python exception message back.</p>

4.1.3 Security and Testing

Especially for back-end applications, it's important to make good decisions when it comes to security and testing, these choices are explained here.

4.1.3.1: API security

Although it will not be possible to fetch stored sensitive medical data (images) via the API, it is still important to add an authorization layer so that **not everyone can use the functionality of the API**. This is done with an Authorization Token, which is added in the header as an **X-API-KEY**.

Another important aspect is the fact that the application will send, possibly sensitive, images via the internet. These images are not encrypted and simply base64 encoded. To make sure that the sent images won't be able to be read by third parties, the hosted development API is secured with an HTTPS connection. This means that all data in the request is encrypted.

4.1.3.2: Storing of images

The storing of the images also involves an important security factor. The images will be used to further train the wound-segmentation algorithm, but as the images consist of sensitive medical data, they have to be stored in a secure location. This secure location is available on an internally hosted server at Nedap, but as this location isn't directly connected to the internet, the following plan was made:

1. Store the images on the virtual machine where the server is running. The images are stored according to the file structure shown in figure 8.

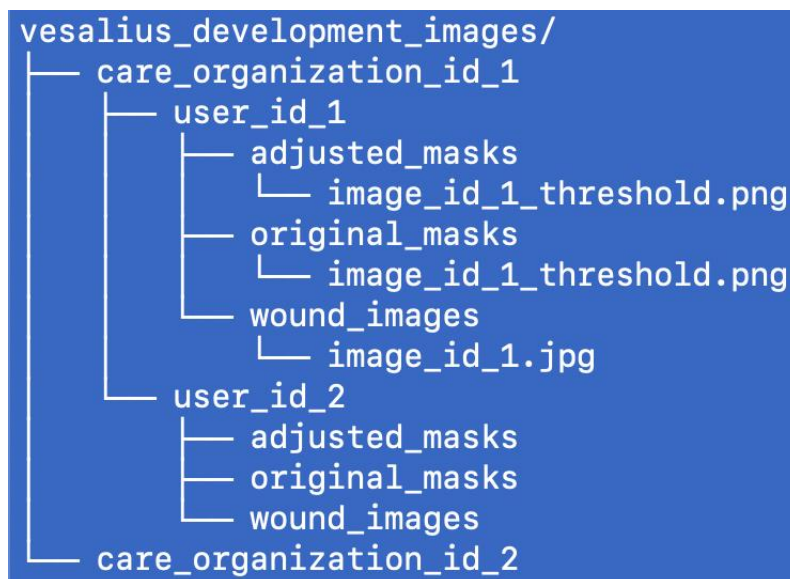


Figure 8: file structure of stored images

2. A function that automatically copies over the contents of the *vesalius_development_images* folder to the available secure location. After the copying, the contents of the folder on the virtual machine are removed. This function will be executed once per timeframe (for example once per day).

4.1.3.3: Testing

One of the most important parts of software development is testing. During the development, this has been done manually with the help of the OpenApi /ui page. Because the OpenApi specification

contains examples of calls including the body data, these calls have been used to manually test the API.

Besides this, automated tests have been created in Python with the pytest package. These tests can be executed by executing the 'pytest' command while having the repository open in a terminal window. Tests that have been created consist of API response tests and tests that check some parts of the return data. The API response tests check if certain body data leads to the expected return code. For example: if the image base64 string is invalid, the expected return value is 400. Other tests check some parts of the return data. The precise result of the wound-segmentation algorithm depends on the trained model, which changes over time. This makes it hard to check if a specific image returns specific values. However, what could be tested are the dimensions of the return data, as they should not change depending on the trained model.

An example of running the tests successfully is shown in figure 9 and having a test fail is shown in figure 10.

```
(vesalius-env) nvc3926:vesalius-backend mattijs.kuhlmann$ pytest -W ignore::DeprecationWarning
===== test session starts =====
platform darwin -- Python 3.6.7, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: /Users/mattijs.kuhlmann/Documents/GitHub/vesalius-backend
collected 4 items

tests/test_api.py .... [100%]

===== 4 passed in 4.89 seconds =====
(vesalius-env) nvc3926:vesalius-backend mattijs.kuhlmann$
```

Figure 9: example of successfully passing all tests

```
(vesalius-env) nvc3926:vesalius-backend mattijs.kuhlmann$ pytest -W ignore::DeprecationWarning
===== test session starts =====
platform darwin -- Python 3.6.7, pytest-4.4.1, py-1.8.0, pluggy-0.9.0
rootdir: /Users/mattijs.kuhlmann/Documents/GitHub/vesalius-backend
collected 4 items

tests/test_api.py ...F [100%]

===== FAILURES =====
test_wound_image
-----
client = <FlaskClient <Flask 'vesalius.run'>>

def test_wound_image(client):
    response = client.post(wound_image_test_url,
                           headers=correct_header_data,
                           json=wound_image_correct_body_data)
> assert response.status_code == 200
E assert 400 == 200
E + where 400 = <Response streamed [400 BAD REQUEST]>.status_code

tests/test_api.py:61: AssertionError
===== 1 failed, 3 passed in 4.61 seconds =====
(vesalius-env) nvc3926:vesalius-backend mattijs.kuhlmann$
```

Figure 10: Example of failing test

4.2: Front-End Application

This part will consist of the description of the realization of the front-end. First, implementation choices will be explained. What follows is the actual implementation of the product and the explanation of all interesting challenges and choices that were made. The actual implementation of the front-end will be explained per screen of the Android Application.

4.2.1: Initial implementation choices

Based on the requirements, initial research was done that led to the following initial implementation choices.

For the front-end Application, the choice was made to create an Android application. The main reasons consist of the fact that currently an Android MVP is being made for regulating wound care and that most users of this application will be Android users. As the Vesalius data collection prototype app will initially have the same target group, the choice for Android was made.

Initial research focused on finding technological possibilities within Android to be able to fulfill the front-end requirements. The following choices were made:

- Native Android applications are originally written in Java code and work on the Java Virtual Machine. But the support for Kotlin ([Kotlin Documentation](#)) has been added since. The decision has been made to write the application in Kotlin, because this is a more optimized and modern language.
- As the application's data source are photos of wounds, camera functionality within Android needed to be researched. The choice was made to use an external library called Fotoapparat ([Fotoapparat](#)). This library greatly simplifies the use of a camera within Android and can fulfill the camera requirement of the application without any problems. One of the main benefits is that this library handles a lot of exceptions that come with different Android devices and cameras. Another reason for using this library is the fact that the, currently in development, wound care application also uses this library. This makes examples available for almost the same use case and makes integrating the two applications easier in a later stadium.
- The application needs to be able to communicate with an API via internet, so network functionality was researched. The decision was made to use the Retrofit library for this ([Retrofit](#)). This library is able to handle all API calls and thus fulfill all the front-end requirements on this part. This library is also used within Nedap, so the expertise and examples are available.
- The most complex part of this application is the displaying and adjusting of the wound area on top of a photo. This required research in how it is possible to draw on top of a view within Android and how the view keeps track of the pixels that are drawn upon. For this, a small and simple application was made. This application, shown in figure 11, shows the possibility for drawing on top of a photo in Android and undoing the last action, by keeping track of the points that are drawn on the screen. Creating this application showed that it was possible to fulfill the requirements.



Figure 11: Test application for custom view drawing

- Besides the choices that focus on fulfilling requirements, another implementation choice was made that focused on architecture. Creating an application without any architecture or structure is most often the easiest, but it has downsides. For Android, a main problem is that the Activity class (the base class for any screen) can get extremely full ([Guide to App Architecture](#), n.d.). This is because all functionality, domain logic and UI changes consist in this one file. Having all functionality and domain logic inside this Activity class, also makes it hard to test the application. Most Android architecture frameworks aim to separate the functionality and domain logic from the Activity class and thus the User Interface (UI). The main goals of having an architecture with this separation of concerns are creating robust, testable and maintainable applications. Android provides its developers with some architecture components that can be used to achieve this ([Android Architecture Components](#), n.d.).

In this application, the choice was made to use some of these components ([LiveData](#) and [ViewModels](#)) to achieve a more professional, robust and maintainable application. The architecture principle that is most conform this is the [MVVM-principle](#) (Model-view-viewmodel).

4.2.2: Android implementation

This part explains the actual implementation of the Android front-end application per feature. During the development of the application, quite some choices have been made that led to change in the implementations of components of this application. These changes will be elaborated on.

The realized Android application consist of three features: **Enter info**, **Take Wound Photo** and **Adjust Wound Area**. These features consist of one or more screens or UI components with different functionalities. Together with the back-end application, these features fulfill all requirements that are needed to achieve a complete data collection flow. To give an overview, figure 12 shows the front-end functionality visualization.

Note that this chapter goes quite in depth into the implementation of this part. The goal of this is to explain a principle of modern Application development and how this was applied to this application.

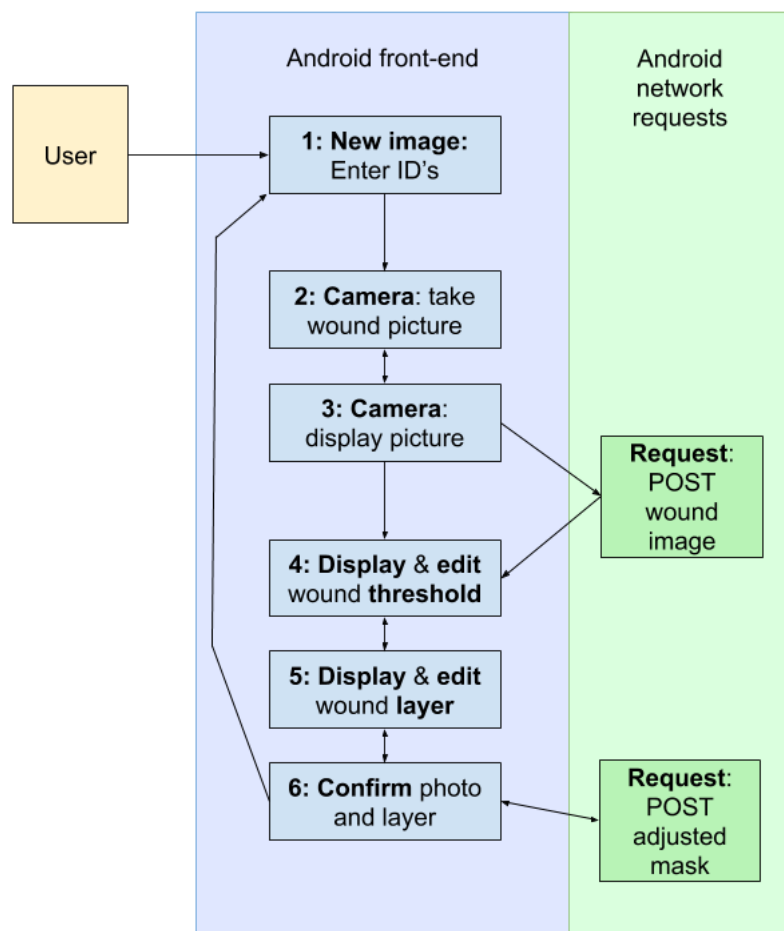


Figure 12: Android functionality visualization

Note that the realized Android application does not implement all requirements from the design. Requirement 2 (displaying an array of previously added wounds) and 3 (showing the details of a previously added wound) were not implemented because they are not required to complete the data collection flow. Besides this, the usefulness and priority were low because the images that would be shown, would only consist of local images.

The Android implementation exists in a GitHub repository with the name: Vesalius-Android. The source code is also available under the same name in the same directory as this report.

4.2.2.1: Enter info feature

When a user first opens the application, the first feature will start. This feature has a low complexity and fulfills front-end requirement **4** (A user must start the data collection process by filling in a Care Organization ID and Patient ID).

This feature only consists of one screen, visualized as number 1 in figure 12. The implementation of this screen is shown in figure 13. The functionality of this feature is to start the data collection process by filling in the Care organization ID and Patient ID, this information is used for storing the images in a structured way.

Note: This information is filled in manually for this prototype, but after possible integration in official Nedap applications, this data is available in the application itself.

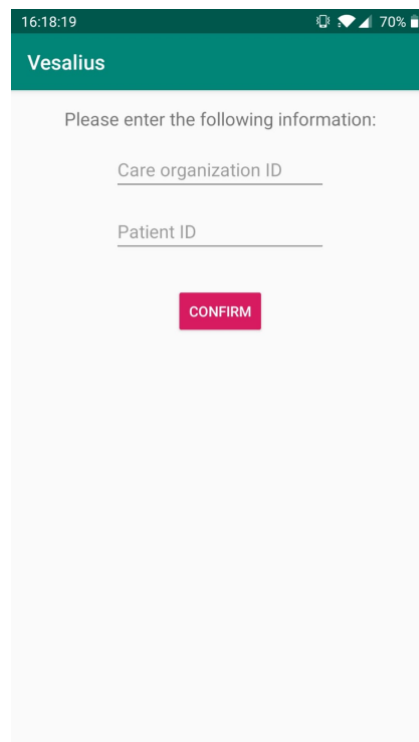


Figure 13: Enter info screen implementation

This screen consists of two text input fields and a confirm button. In the background the following happens: clicking the confirm button checks if both fields are filled in and otherwise asks the user to fill in all the fields. If all fields are filled in, a Wound object is created containing this information and a timestamp as Wound ID. This object is then passed onto the next screen. This feature only consists of one class file: AddInfoActivity. Because the functionality of this feature is minimal, all functionality is handled in this activity class, without the use of a ViewModel class. Note that future implementations of this application won't need this screen, because all data exists in the current user's authentication data.

4.2.2.2: Take wound photo feature

After filling in wound information the next feature is started, which consists of taking a wound photo, reviewing the photo and then either retaking or confirming it.

This feature fulfills front-end requirements **5** (A user must be able to take a photo of a wound) and **6** (A user must be able to view the photo and then accept or otherwise retake it). This feature consists of two screens, visualized as number 2 and 3 in figure 12. The implementation of these two screens are shown in figures 13 and 14.



Figure 14: Take wound picture screen implementation



Figure 15: Preview picture screen implementation

While the functionality of this feature is quite simple, the implementation within Android is more complex. To explain this complexity, the file structure within Android of this feature can be seen in figure 16. The main class of this screen is defined in *TakeWoundPhotoActivity*, this activity class contains and displays *TakePhotoFragment* (this fragment extends the *CameraControl* interface, which enforces the use of some specific camera-related functions) and *PreviewFragment*. These fragment classes contain the UI logic of both screens. While the activity class and the fragment classes handle all UI logic, some functionality has been moved to a [ViewModel](#) class: *TakeWoundPhotoViewModel*. This ViewModel class works independent from the activity and fragment classes and its main functionality is handling the result data of a taken photo by storing the photo in a file on the device. The file location of this device is then set in a [MutableLiveData](#) object called *cameraResult*. The reason this is done is because this makes it possible for the *PreviewFragment* to observe this *MutableLiveData* object and act on any changes of this object. In this case, the *PreviewFragment* will display the image which is stored on the specified file location when a change happens. And with the help of an interface callback function, the *TakeWoundPhotoActivity* is notified of a 'click' on the "take picture"-button so that it can display the *PreviewFragment*. The final step of this feature consists of previewing the photo and either clicking the back button. Clicking the back button automatically removes the *PreviewFragment* from the navigation stack and shows the *TakePhotoFragment* again. Clicking the

confirm button will add the current image file path to the wound object that had been created in the previous feature and passes this on to the next feature.

For further explanation and understanding of the workings of this feature, see the [vesalius-android repository](#), which contains all (commented) code.

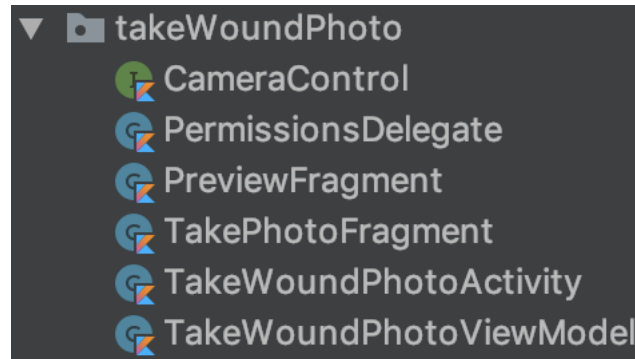


Figure 16: Android file structure of take wound photo feature

This feature creates the initial data object (the photo), that is part of the data collection process. This means that this feature is responsible for the format and size of the photo. The choice has been made to restrict the size of a photo to a size of maximum 1280*960 pixels. This is done to limit the size of the API request, which sends the image, as some newer phones are able to create very large images. This choice has no negative influence on the quality of the training of the Vesalius algorithm because the algorithm decreases the size of all images to 256*256 before training. Keeping the images to a limited size also lowers the complexity and improves the performance of the **Adjust Wound Area** feature, which will be explained in that part.

4.2.2.3: Adjust wound layer feature

When this feature is started, it means that a wound object exists which contains all ID's and the file location of a photo. This makes it possible for the most complex part of the application to be handled: the displaying and adjusting of the wound area on top of the photo.

This feature fulfills front-end requirements **7** (The application must be able to send the taken photo to a server and receive the segmentation result), **8** (A user must be able to see the result of the wound-recognition algorithm on top of the taken wound photo), **9** (A user must be able to adjust the result of the wound-recognition algorithm) and **10** (The application must be able to send the wound area result to a server). This feature consists of one screen with multiple UI compositions, which are visualized in figure 12 as number 4, 5, 6 and both the requests. The final implementation of the two main different UI compositions can be seen in figure 17, these compositions show the implementation of requirements **8** and **9**.

The compositions in figure 17 show the implementation of the adjust wound layer feature. The choice was made to split this feature up into two parts: changing the threshold and drawing/erasing the wound area. Because the Vesalius algorithm returns a probability value per pixel the first screen is used to change the wound area based on the direct result of the Vesalius algorithm. Based on the slider, a threshold value is changed and only the pixels with a value above this threshold are drawn as part of the wound area. By also returning the value of this threshold to the back-end, information is given that can lead to a more accurate threshold value. After changing the threshold, the user is also given the option to draw or erase parts of the wound area. This is done with the visible buttons and by touching the photo on the device. The buttons are used to switch between drawing and erasing, changing the drawing/erasing size and to reset the area back to the set threshold value.



Figure 17: Implementation of two main UI compositions of Adjust Wound Area feature

The workings behind the UI compositions of this feature and especially the displaying of the wound area are quite complex. These are explained below, starting with figure 18, which shows the Android file structure of this feature.

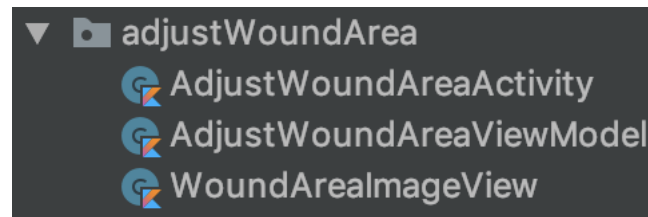


Figure 18: Android file structure of Adjust Wound Area feature

As can be seen in figure 18, the current feature consists of three classes and to explain the workings behind this feature, the explanation is split up into two parts:

1. The **first part** will explain the **UI component flow** in this feature. This is done by explaining the functionality of this feature, the implementation of the *AdjustWoundAreaActivity* and *AdjustWoundAreaViewModel* classes and how they interact. This part mainly goes in depth into the Android architecture.
2. The **second part** explains the workings and reasoning of the **Wound Area Image View**, which is a custom View that allows for drawing and editing a wound area on an image, while remembering all pixel coordinates which are selected to be part of the wound area. This part tells more about the **data structures** and **challenges** that were faced during the realization of this, as well as the way in which this data is converted to what is drawn on the screen.

4.2.2.3.1 UI component flow

The user of the application has the option to change the wound area in two different ways: **changing the probability-mask-threshold** and **drawing and erasing** the wound area on the screen. Besides the user input, this feature also handles some network requests which can have a loading state, a success state or failing state. All this means that this feature consists of multiple **states**, which all change the UI composition of the screen.

Because of the functionality and complexity of this screen, the decision was made to separate the UI and the functionality as much as possible. This was done by creating the *AdjustWoundAreaViewModel* class, which has a couple of main tasks:

- Keeping track of the **feature state** of the feature in a LiveData object -> the Activity observes this object and updates the UI composition when a change occurs.
- Keeping track of **wound area data** in the form of data structures and LiveData objects -> the Activity requests these data structures to change the *WoundAreaImageView*. The LiveData objects are observed by the Activity and updates other parts of the *WoundAreaImageView* when a change occurs.
- Performing operations such as API calls and using the result of the API calls to create the **wound area data**, after which the **state** can be changed.

Figure 19 shows a visualization of the *principle* of the interaction between the **Activity** and the **ViewModel** within this feature. A user gives some sort of input via the visible UI components, which is received by the Activity. This input either directly changes the *WoundAreaImageView*, or the *WoundAreaImageView* is changed based on operations done by the ViewModel. The user input and ViewModel operations also change LiveData objects within the ViewModel, such as the feature **state**. The Activity observes these changes and updates the UI composition accordingly.

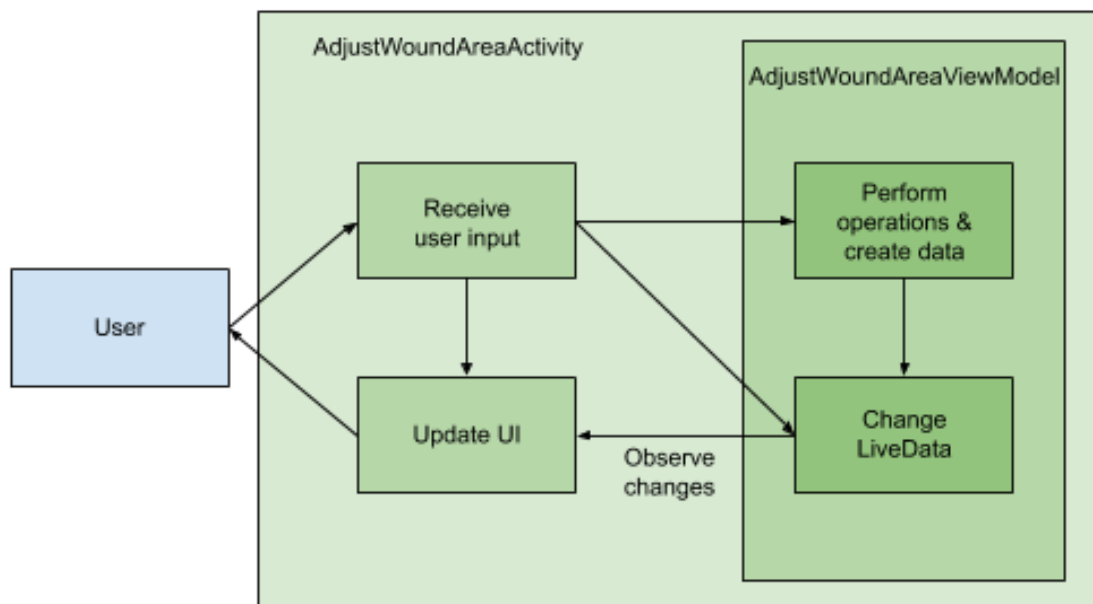


Figure 19: Visualization of interaction principle between components of Adjust Wound Area feature

To explain the main flow and usage of this feature, the relationships between the most important feature states and their corresponding UI compositions are explained underneath. Figure 20 shows the UI compositions and figure 21 shows these relationships in a state flow diagram.



Figure 20: Display of different UI compositions, with different feature states.

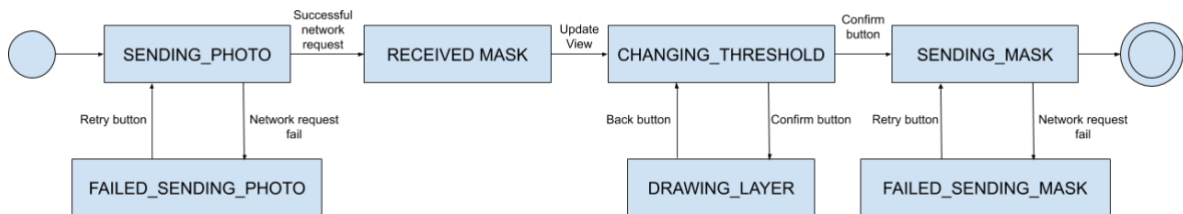


Figure 21: State transition diagram of adjust wound area feature

When this feature starts, the wound photo is set in the *WoundAreaImageView* and the *ViewModel* sends a request to the server, containing the photo. The initial corresponding state is *SENDING_PHOTO*, which is seen in the most left UI composition in figure 20.

After the photo has been sent to the server and the result of the Vesalius algorithm has been returned, the *ViewModel* uses this result to generate the necessary data objects and changes the state to *RECEIVED_MASK*. The Activity then updates the *WoundAreaImageView* with the generated data objects. The wound area becomes visible and the state is changed to *CHANGING_THRESHOLD*, which is seen in the second UI composition in figure 20.

The user can now use the slider to change the threshold, which is stored in a *LiveData* object in the *ViewModel*. Changes of this object are observed by the Activity and change the *WoundAreaImageView*.

After a user clicks the confirm button, the state changes to *DRAWING_LAYER*, and the activity changes the UI to the third composition of figure 20. With the help of the UI components, the user can draw or erase parts of the wound area by touching the *WoundAreaImageView*. Once a user is satisfied with the result, the confirm button can once again be clicked. This will raise an alert screen and after the user accepts this, the state changes to *SENDING_MASK* and the result is sent to the server. The UI composition of this screen is comparable to the first composition of figure 20.

If this network request is successful, the data collection process is finished, and the application navigates back to the first feature: **enter info**. However, requests that depend on internet connectivity can also fail. This can happen during the states *SENDING_PHOTO* and *SENDING_MASK* and will result in changing the states to respectively *FAILED_SENDING_PHOTO* and *FAILED_SENDING_MASK*. The UI composition of these two states can be seen in the fourth composition of figure 20. Clicking the retry button will change the state back to respectively *SENDING_PHOTO* and *SENDING_MASK*.

4.2.2.3.2 Wound Area Image View

The *AdjustWoundAreaActivity* and *AdjustWoundAreaViewModel* handle the UI compositions and the network requests. This is done to prepare and handle the data, which is needed for the user to see the wound photo and to view and adjust its wound area. This is possible because of the *WoundAreaImageView* class. This class extends the [ImageView](#) class, which is the standard Android UI component in which images can be shown. The custom version makes it possible to control what is drawn on top of this UI component. The realization of this custom class is complex and changed multiple times during the implementation, due to challenges. The main challenges had to do with the **data types and data structures**, which influence the memory use of the application and finding a balance between performance and the **design and way of drawing of the wound area** on top of the image.

4.2.2.3.2.1 Data types and data structures

The goal of the adjust wound area feature is to keep track of all pixels that a user indicates as part of the wound area, to be able to finally send the coordinates of these pixels to the server. Whether a pixel should be drawn or not can be influenced by two things: the *result of the Vesalius algorithm in combination with the threshold* and the *manual drawing and erasing by a user*. This means that the class should keep two things in memory:

1. The result of the Vesalius algorithm, which is a probability value per pixel.
2. The pixels which are part of the wound area.

The **first version** of the *WoundAreaImageView* kept this information in memory by creating an array of instances of a data class per pixel, which contained:

- Wound area pixel: Boolean value if the pixel is part of the wound area.
- Probability: Double value of the Vesalius algorithm result of this pixel.
- X: Integer value for the X coordinate of this pixel.
- Y: Integer value for the Y coordinate of this pixel.

To be able to find if a specific pixel of the view needed to be drawn, a pixel instance with a specified X and Y value needed to be found in this list. Then the wound area pixel value needed to be changed based on its probability value and specified threshold.

However, based on observed memory use by using the built-in profiler in Android Studio, and the number of times this array needed to be traversed, it became clear that this was not a good solution and a couple of changes have been made since:

- Instead of storing the probability values as Double values (memory size of 8 bytes), the choice was made to change this to Byte values (memory size of 1 byte).
- The creation of an instance of a data class increased the memory use by a large amount. This is why it was decided to remove this and to instead keep the wound area pixels and probability values outside of a data class instance. This data is currently kept in two separate, one-dimensional arrays.
- The representation of the data of an image normally consists of a matrix, with a height-number of rows and a width-number of columns. The Vesalius algorithm result is also represented in a matrix, containing probability values. Originally, the rows and columns of this matrix were traversed, and the row and column indices were used to define the X and Y values of the pixel data instance. The pixel data was then stored in the shape of a one-dimensional array without separate row and column indices. Because of missing the separate row and column indices, the X and Y values were stored. However, this changed with the realization that the X and Y values can simply be calculated by taking the index of the pixel in a one-

dimensional array together with the width of the image. The other way around, the index can be calculated by using the X and Y values of a pixel together with the width of the image:

- $\text{Index} = (\text{Y} * \text{width}) + \text{X}$
- $\text{X} = \text{index} \% \text{width}$
- $\text{Y} = \text{index} / \text{width}$

Because of this it was possible to directly look up any pixel of the image in the one-dimensional arrays.

The change of the data and data types resulted in a big reduction of memory use and the calculation of indices resulted in removing the need to traverse the arrays more than necessary.

4.2.2.3.2.2 Design & drawing of the wound area

The necessary data is kept in memory and can be changed in an efficient way. But the next step was to convert this data into an efficient way to draw the wound area. Finding a clear design and efficient way of drawing this area was a challenge.

The way in which Android allows for drawing on a [View](#) is done by overriding the [onDraw](#) function, which is called every time a change happens within the activity. Within this onDraw function, it is possible to draw points and other shapes within the view by providing an X and Y coordinate together with a Paint object. This Paint object contains visual aspects like size, color and alpha. Because of the existence of an array of Boolean values per pixel and the ability to calculate the X and Y coordinates with an index, it was possible to traverse the array and draw a point on the View for every True value. The result of this can be seen in figure 22.



Figure 22: Wound area by drawing every separate pixel.

After testing this implementation, it became clear that another solution needed to be found. The testing showed that the performance of this solution depended heavily on the size of the wound area. With an image size of 960 * 1280, the maximum number of points that could be drawn was more than 1.2 million. Because of this, the application would slow down a lot. So, it was clear that another

solution was necessary. Either the drawing needed to be done more efficient by drawing larger shapes instead of separate points, or another solution needed to be thought of. After talking to a UI/UX designer, it was pointed out that coloring the whole wound area makes it harder to see the image underneath. This was partly solved by lowering the alpha value of the Paint, as can be seen in figure 22. However, since this doesn't resolve the performance issues, the suggestion was made to look into the possibility to draw the border of the wound area. Since this would lower the number of drawn pixels by a large amount.

To achieve this, the border of the wound area needs to be found. This is done by comparing any pixel that is part of the wound area to its neighboring pixels. If a neighboring pixel is not part of the wound area, this means that the pixel is a border pixel. Every time the wound area changes, this comparison is done and only the border pixels are drawn. Because all pixels are represented in a one-dimensional array, a function was written that calculates all existing neighbor indices for any index. The closest neighbors for all indices are calculated once, during the initialization of the feature. These are kept in memory to be able to quickly find and compare the pixels with its neighbors.

The final design of the wound layer for this prototype can be seen in figures 16 and 19. However, the optimal design of this feature is not determined yet, but other designs that have been experimented with are for example a combination of drawing the separate pixels and the border. This can be seen in figure 23. Future user testing would be required to find the most optimal design and implementation.



Figure 23: Other implemented design possibility or drawing wound area.

5: Conclusion and future recommendations

The goal of this graduation assignment was to create a **prototype application** which can collect data for a wound-segmentation algorithm. Requirements were set up, a process was defined, designs were created, and the applications were implemented. Using the created Android application, it is possible for a user to finish the complete data collection process consisting of a wound photo and the marked wound area. The realized Android application uses a created Python back-end service that segments the wound area and stores the result of the data collection process in the exact format that the wound-segmentation algorithm uses for training.

When comparing the specified requirements with the final product, it can be seen that almost all requirements have been implemented. Due to time restrictions, the decision was made to not implement two front-end requirements with a lower priority. Also, when comparing the front-end mockups with the implemented screens, differences can be seen. The wound-area-adjustment feature has been split up in two parts and the design of this feature shows differences with the realization. Multiple factors led to these differences; among these factors there were implementation challenges, specifics that were overlooked during the design phase and suggestions of other people involved during the development.

This all leading to the conclusion that despite the differences, the **main goal of the assignment has been achieved**.

However, this assignment required the creation of a prototype. And a lot needs to be done before this application would be ready for production. If the future goal was to **create a separate data collection application** for wound images, things that would need more work are the following:

- Currently, any rescaling of the result of the Vesalius algorithm is done on the back-end. The front-end application doesn't support this resizing and scaling. This is why the front-end application has some scaling problems based on different screen sizes, as well as not supporting the use of the landscape orientation. Moving scaling functionality to the front-end application could offer more solutions for these problems. Besides this, offering support for resizing on the front-end application brings possibilities for reducing the bandwidth consumption of network requests.
- While the application performs smoothly on newer and higher end smartphones and emulators, the performance on some older Android phones is less good. Drawing of the wound area on top of the image still requires a lot of calculations to be done. Optimization in the future is necessary to improve this.
- The adjust wound area feature is complex when it comes to design. The optimal design and user experience can only be achieved by doing more research with the help of **user testing**.

The use of a standalone data collection application is useful for training the Vesalius algorithm, but the utilization of the Vesalius algorithm within Nedap applications is something that has real future perspective: If trained sufficiently, the Vesalius algorithm can be used to automate processes within the currently developed wound care application. Things that could be automated are the determination of the size of a wound as well as classifying different kinds of tissue within the wound area. Combined with patient information, this all could theoretically lead towards an automatically generated care plan of a wound treatment.

This is why integration of parts of this prototype within the wound care application could be a next step to this future perspective. The parts of this prototype can be used to collect more data for the algorithm, or these parts can be a first step towards the mentioned automation.

All by all, automatic wound-segmentation has a lot of future perspective, but a lot needs to be improved, tested and implemented to get to this. This prototype application, however, is a physical example of what the Vesalius algorithm can do. And this makes the future of healthcare come a little bit closer.

6: References

- Android Architecture Components (n.d.). Retrieved from <https://developer.android.com/topic/libraries/architecture>
- Autopep8 (04-2019). Retrieved from <https://github.com/hhatto/autopep8>
- Connexion (04-2019). Retrieved from <https://github.com/zalando/connexion>
- Fotoapparat (02-2019). Retrieved from <https://github.com/RedApparat/Fotoapparat>
- Guide to App Architecture (n.d.). Retrieved from <https://developer.android.com/jetpack/docs/guide>
- Keras (n.d.). Retrieved from <https://keras.io/>
- Kotlin Documentation (n.d.). Retrieved from <https://kotlinlang.org/docs/reference/>
- OpenApi (n.d.). Retrieved from <https://swagger.io/docs/specification/about/>
- Pep8 (n.d.). Retrieved from <https://pep8.readthedocs.io/en/release-1.7.x/>
- Pytest Fixtures (n.d.). Retrieved from <https://docs.pytest.org/en/latest/fixture.html>
- Retrofit (n.d.). Retrieved from <https://square.github.io/retrofit/>
- Tensorflow (n.d.). Retrieved from <https://www.tensorflow.org/>
- Visual Studio Code (n.d.). Retrieved from <https://code.visualstudio.com/>
- Wang, C., Yan, X., Smith, M., Kochhar, K., Rubin, M., Warren, S. M., ... & Lee, H. (2015). A unified framework for automatic wound segmentation and analysis with deep convolutional neural networks. In Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE (pp. 2415-2418). IEEE.
- Zorguitgaven stijgen in 2017 met 2,1 procent (05-2018). Retrieved from <https://www.cbs.nl/nl-nl/nieuws/2018/22/zorguitgaven-stijgen-in-2017-met-2-1-procent>

7: Appendix

7.1: Initial sprint backlog

Back-end backlog

#	Description	Type	Priority	Implementation remarks
BE1.0	Set up back-end repository and environment	Server & API	Highest	
BE2.0	Set up VM to host the API and Server and SSL certificate for HTTPS support (from Nedap)	Server, API & Process	Medium	
API1.0	Create token-based authorization, saved in environment variable, to be sent in header	API	Highest	
API2.0	Create GET request for wound segmentation	API	Highest	
API3.0	Create POST request for adjusted wound area (validation)	API	Highest	
S1.0	Create functionality to return wound area for a given wound photo (#API2.0)	Server	Highest	
S2.0	Create functionality to receive adjusted wound area segment and save it on the server (#API3.0)	Server	Highest	
T1.0	Create test file for API and security with pytest fixtures	Testing	High	
S3.0	Research HTTPS implementation in connexion/python back-end and need for extra encryption and how-to implement this	Server & API	High	Research a way to encrypt
S10.0	Deploy back-end on production environment with HTTPS support	Server & API	High	Final requirement, Dependency: #BE2.0 & #S3.0

Front-end backlog

#	Description	Type	Priority	Implementation remarks
D1.0	Create specific front-end design for application	Design & Process	Highest	Process over time: design based on trying out and talking to people in work field and UI/UX designers.
A1.0	Set up front-end repository and environment	Android	Highest	Includes technical documentation about packages

A2.0	Research and set up Android project with MVVM architecture	Android	Highest	
A2.1	Insert Retrofit support	Android	Highest	
A2.2	Insert Room support	Android	Highest	
A3.0	Create a general UI theme, visible on multiple screens	Android & Design	Medium	Dependency: #D1.0
A4.0	Create wound-list functionality	Android	High	Based on wound list from MVVM Repository
A4.1	Create final wound-list UI	Android & Design	Medium	Dependency: #D1.0
A5.0	Create enter patient info screen functionality	Android	High	
A5.1	Create final enter patient info screen UI	Android & Design	Medium	
A6.0	Create Camera screen functionality	Android	High	Fotoapparat Library camera component + preview
A6.1	Create final Camera screen UI	Android & Design	Medium	Dependency: #D1.0
A7.0	Create layer adjustment screen functionality	Android	High	Research needed for: Scaling problems & most efficient drawing
A7.1	Create final layer adjustment screen UI	Android & Design	Medium	Dependency: #D1.0
A8.0	Create wound detail screen functionality	Android	High	
A8.1	Create final wound detail screen UI	Android & Design	Medium	Dependency: #D1.0
A10.0	Deploy application on testing/production environment for sharing purposes	Android	Medium	Final requirement

7.2: Sprint progress & documentation example

A visualization of the weight of the stories burned is given in figure 24. As this project didn't involve story points that were planned beforehand, the visualization only shows the distribution of the weight of the completed stories per sprint. This gives an overview of the distribution of work during the project. This distribution was input for the retrospective after each sprint.

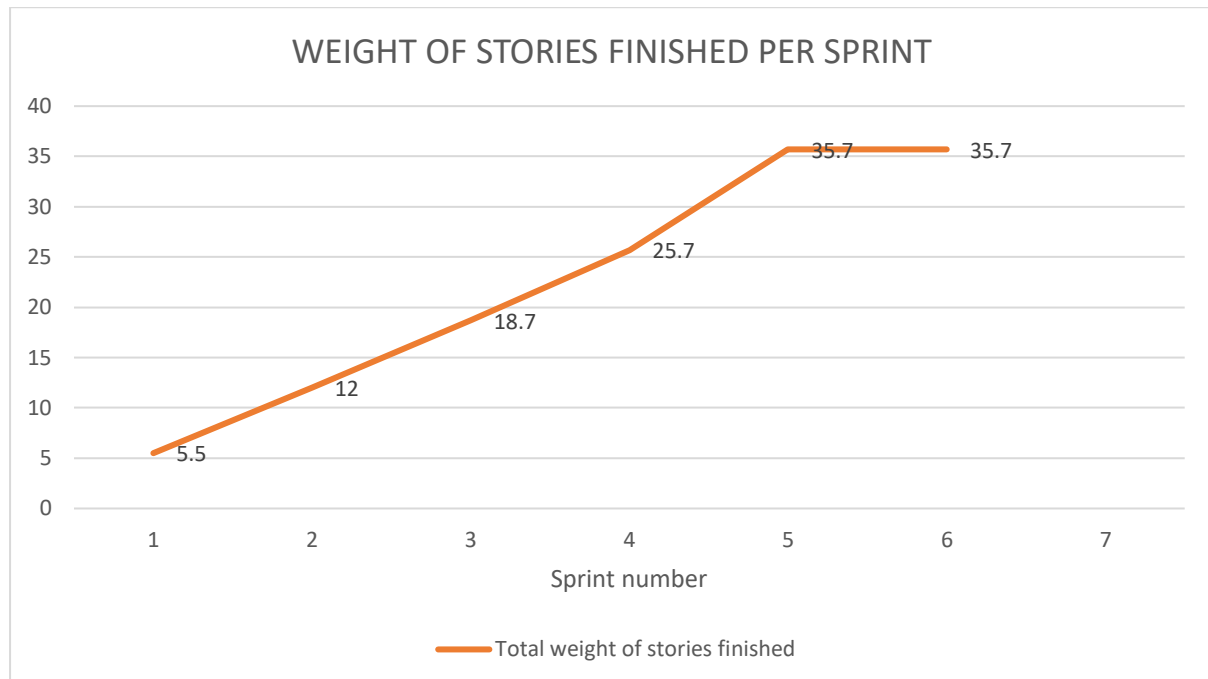


Figure 24: Visualization of total weight of stories per sprint

After each sprint, the finished stories were documented, and a small retrospective was done. All retrospectives exist in the sprint documentation and an example for one sprint is given underneath.

Sprint 1 (25-03 -> 07-04)

Stories finished:

#	Description	Time spent (days)
BE1.0	Set up back-end repository and Python environment	1
API2.0	Create POST request for wound segmentation	2
API3.0	Create POST request for adjusted wound area (validation)	2
API1.0	Create token-based authorization, saved in environment variable, to be sent in header	0.5

What went well?

- I started off well and managed to create a local version of an API with a first version of the requests that I wanted to create.
- The progress went well and structured.
- I learned a lot about the principles of an API by working with the connexion library.
- I could ask questions to specific people and they were able to help me really well.

What didn't go well?

- I noticed that it was hard to imagine the full product, and how the API was going to communicate with the application. This made me realize that I missed having an exact and concrete design, which resulted in a lot of doubting and changing things.
- I planned to ask for help in realizing my product, but I didn't ask for help as much as I would have liked.

What am I going to do different?

- The lack of a concrete design originated from the fact that it was not yet 100% clear what the functionality and data contents of the API requests should be. Next time, it is important to start off by designing and planning these specific things first.
- Missing an exact and concrete design is difficult. But with my lack of experience it's also not that strange. What I could do different is accepting that not everything can go according to a certain way and that I should just try things out sometimes, also if it's not planned.
- Although I asked some questions, I had the feeling that the progress could be quicker if I would have asked more. Asking more questions is a simple solution to the problem, and it could also help me to ask help in creating a more exact and concrete design.