ulm university universität ulm

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Bachelorarbeit
im Studiengang Informatik

Konzeption und Realisierung einer Web-Anwendung zur Unterstützung von Ärtzen bei der Notfallbehandlung von Patienten

vorgelegt von

Tobias Müller

Juli 2020

1. Gutachter	Prof. Dr. Manfred Reichert
Betreuer:	Rüdiger Pryss
Betreuer:	Sascha d'Almeida
Betreuer:	Philipp Mohr
Betreuer:	Marco Schweitzer
Matrikelnummer	903845
Arbeit vorgelegt am:	13.07.2020

Abstract

In the past the manpower in German hospitals has risen and therefore the costs. Furthermore, many hospitals use incompatible information management systems making the exchange of data difficult. We look at different approaches to relieve the current healthcare system with the implementation of e-Health. Our focus is the Estonian healthcare system, as one of the most advanced in the European Union. Moreover, we look at systems to help first responders and paramedics as well as systems helping hospital staff with an app. To make our application as accessible as possible we discuss the benefits and disadvantages of QR Codes and Data Matrix codes. Based on the insights gained we propose an application combining the approaches. This application functions as a central database for all medical workers. Our goal was to make the application as transparent as possible and therefore the patient can see all the documents about them. We describe how the application could be used by medical staff and doctors too. Our application provides easy and quick access for medical personnel to the files of patients, reducing time spend for administrative tasks.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sinngemäße Übernahmen aus anderen Werken sind als solche kenntlich gemacht und mit genauer Quellenangabe (auch aus elektronischen Medien) versehen.

Ulm, den 13.07.2020 Tobias Müller

Contents

List of Figures

Listings

1 Introduction

In 1991 a full-time equivalent of roughly 875 000 people worked in hospitals and took care of approximately 14 600 000 patients [per][pat]. These numbers have gone up by 6% and 33% respectively to roughly 930 000 full-time equivalents and approximately 19 400 000 patients in 2018 [per][pat]. The biggest rise in the hospital workers comes from the doctors, because the complexity of the diseases, medication, and devices has increased. In 1991 more than 95 000 doctors worked in hospitals, this has risen by 72% to more than 164 000 in 2018 and at the same time the number of persons working in a hospital, which are not doctors has declined by 2% from 1991 to 2018 [per]. While it is seemingly good that the number of doctors has risen much more than the number of the patients many doctors still must do extra hours every week. In 2018 32% of all doctors in Germany worked more than 48 hours per week on a regular basis [arb]. The costs for hospitals in 1991 was equivalent to 8,7 billion EUR for doctors and 15,6 billion EUR for the nursing service in 2017 [kra], which means both values are adjusted for inflation and the currency exchange was considered. In 2017 doctors costed 20,4 billion EUR and the nursing service costed 19,2 billion EUR [kos]. This corresponds to a 134% and a 23% rise in costs for doctors and the nursing service, respectively. Another problem for the German health care system is that the German society gets older year by year. In 1991 15% of the population was 65 or older and this percentage has risen to 19% in 2005 and 22% in 2018 and is expected to rise to 31% in 2060 [alt] respectively. This is important because 46% of the patients in 2005 where over the age of 65 and in 2018 this group made up 53% of the patients in the hospitals [ope]. Therefore, with a rising number of old people the number of patients is expected to rise too. This means even more doctors are needed in the future, which means that the costs will rise even more. Therefore, we want to investigate a possible way to digitize the health care system to reduce costs and improve the care for patients.

Since life is getting more and more digitized this trend does not stop in the field of health care. This has certain advantages and disadvantages some of them we will discuss and address in this thesis. We worked closely together with doctors who expounded the disadvantages and shortcomings of the current system, which are described in the following lines. Together we worked on a system to tackle some of these issues. While working on this new system we used the feedback from the doctors to create a system that makes the everyday work in a hospital easier and more time efficient, while reducing complexity.

The current health care system in Germany is almost entirely decentralized, which means that every hospital has their own records about a patient. The clinical information systems of different hospitals are often incompatible with each other. This even goes as far as two clinical information systems from the same developer cannot exchange data with each other, because the systems were adjusted for the specific needs of these hospitals and within a hospital different department use incompatible tools or different systems sometimes too. To make the systems

compatible plugins are heavily used. This makes it only a temporary solution since with every software update these plugins could become incompatible. Which leaves two options, either the plugins do get updated too, which costs money, or the system does not get updated, which could become a security threat. Because of these problems the information exchange between hospitals is heavily paper based. For example, if a patient needs to take an x-ray in hospital A and these files are needed in hospital B, but hospital B cannot access these files, since there is no central database. In this case the patients must go to hospital A themselves so that the files can be burned to a CD. The patient then takes this CD to hospital B. As another example when moving over larger distances one must visit a different family doctor. Since there is no centralized system the sick person should bring all their reports, passes, vaccination record, and CDs. Then the doctor must insert all this information in their clinical information system by hand, which can occupy much time depending on how complex and how much information there is. This time is missing for patient care. But there is a silver lining. Some data nowadays can be send digitally from one hospital to the other. Even if it is not much it is a good start to save time and resources. Especially device passes can be a problem of their own. For instance the insertion of cardiac pacemakers are a standard procedure nowadays. People who undergo such a procedure usually get a pass that states that they have metal in their heart (important when you get an MRI or you fly somewhere). That way multiple operations can result in multiple metal devices inside a patient's body. Since there can be multiple pieces of metal every piece has its own pass. This can sum up to quite a lot of passes. Another problem is the pass design. For example, the passes for anticoagulatory agents (blood thinners) can have a different design depending on the exact name of the substance and its manufacturer. This includes especially the color of the pass. If the passes were color coded by body parts or organs and had standardized structure much time could be saved, and this would increase the patient's compliance and understanding. This lack of standardization is one problem we want to address with this thesis together with an approach on how to centralize the German system. With a centralized system the second opinion of a doctor can be made more time efficient, since the doctor can see which tests were performed on the patient and the results of these tests and do not have to run these tests again.

In the next chapter *Related Work* we want to look at things that have been done regarding digitizing the health care system. Furthermore, we want to analyze the reasons why certain systems worked and others failed. At the end of this chapter we want to glance at the systems in other countries and compare some parts to the German system, to discuss which aspects could be adapted and which are not suitable yet.

Another aspect which we want to look closely at are medication plans. If a patient needs medication the doctor prescribes the drugs to the patient and hands the patient this medication plan. This plan states what medication should be taken and the corresponding intake pattern for each drug. These plans are typically printed with a Data Matrix or a QR Code on them, which encode the medication and the intake patterns. What and how they encode the data will be discussed in chapter *Background Information* as well as ICD Codes, which encode diseases and are widely used in the medical environment. Moreover, we want to look at different 2D codes to encode the data, like QR Codes, and compare them to each other. Our focus is usability improvement and the ability to encode more data in one of these codes. After improving the usability of these plans the family doctors could save time for writing prescriptions.

Afterwards we want to present and discuss our implementation in the chapter *Implementation*. First, we want to explain what we did and what problems were addressed doing so. Because the implementation was planned with a mobile application in mind, we will also discuss the characteristics and challenges which come with such a system. We want to provide an approach to solve these issues. Since the software will store highly personal data the requirements for the software security and the privacy of this data are huge. Therefore, we want to discuss what can be done and consider what should be done, without restricting the usability too much.

Some pictures of our application are then shown in chapter *Look and feel*.

In the last chapter *Use Case* we want to discuss our implementation with a use case and determine what works well and what does not. Moreover, we want to discuss what can be improved regarding the insights gained in the previous chapters.

2
Related Work

In this chapter we want to look at different approaches to digitize medical files and how the processes in the health care system can be improved. First, we will investigate the solutions from the corporate sector, then into one solution for hospitals, and at the end of this chapter into a countrywide solution.

2.1 Corporate sector

In case of an emergency time is of the essence. Many smartphone manufacturers have therefore implemented an emergency function. This displays general data about the owner of the smartphone, if the person entered them first. That includes information like name, blood type, and phone numbers of emergency contacts. Some private companies offer services where more important medical data can be accessed quickly. One company of these companies is SOSQR [sos], which we will investigate a little bit further. The data is stored on a website and is entered there by the person them self. Then the person can print out a QR Code, which represents a link to the website with the data. Afterwards the QR Code can be put on a motorcycle helmet, as a lock screen wallpaper, or simply as a card in the wallet. To access the data in case of an emergency the paramedic or any other rescuer must scan the QR Code and enter a four-digit PIN, which is printed near the QR Code. While a paramedic can greatly benefit from more information to much information could potentially slow down the process. One great advantage of this system is automatic translation of the saved report. If the person carried such a QR Code and had an accident while traveling in a foreign country the paramedics could still read and understand what the person entered in their report. Therefore, getting help can be easier in countries, where the verbal communication is not good. Since the vocabulary of tourists is generally not that large, they could benefit the most out of this feature. The functionality of these services can replace bracelets some people wear to signal paramedics that they are allergic to certain things like penicillin or nickel, by attaching a QR Code to a bracelet. One minor disadvantage this system has is the requirement for an internet connection. There are still places in Germany which have no mobile internet connection [bre] depending on the internet service provider and therefore this technology becomes useless. The places with no mobile internet connection are apprehended by the Federal Network Agency (ger: Bundesnetzagentur) and displayed on a map to find dead zones and ultimately eradicate them. One possible example is a hiking trail, especially in forests, where the reception is weak to non-existing and getting help is exceedingly difficult. When help arrives they could read the QR Code but probably could not access the web page with the information. But the information can be read on the way to the hospital when the cell reception is better.

Another company is Mercedes Benz [mera] which offers a rescue sticker. These stickers are put in the fuel filler cap and on the B-pillar of the opposite site. These parts of the car are statistically nearly undamaged in case of an accident. After scanning the QR Code a rescue cart for the vehicle is displayed which shows all important information for the rescue. It also guides the fire department to locate parts of the which could harm the rescue team or the occupants, like airbags. These cards can be accessed offline which is great when the accident happened in a dead zone. Furthermore, the cards are displayed in the language of the mobile device [merb] if this language is available. If the language should not be available English gets automatically displayed. When using the app these cards can be displayed in 3D or with augmented reality directly on the vehicle. This makes the rescue process easier and safer, but the paramedics have to know about these possibilities to search for the QR Code. Why or why not QR Codes should be used in such situations is discussed later in the section about *QR Codes*.

2.2 Hospital solution

One study which offers a solution for a hospital is the study by Mersini et al. [MST13]. They proposed a system based on a smartphone app and QR Codes for in-house use. To implement the proposed system a central database in the hospital is needed, where the patient's medical files can be stored. Therewith the system can unleash its full potential all medical records of the patients must be available electronically. When a new patient with no electronic health records comes to the hospital, then a new entry with the patient's personal information in the database will be created. These files can include allergies, medication, and diagnoses. Consequently, the patient's full history shall be available to make better decisions faster. The system can also "include updates on the progress of the patient" as well as "sending referrals directly to [...] laborator[ies]" and the user gets an instant notification in the app as soon as the laboratory results are available.

They use SQLite in their app to get updates and send updates from or to the server via HTTP requests. This also allows the app to store some data on the device, which enables offline working. The personnel can create new records or edit records which are stored while the device has no connection to the server. When the connection to the server is restored the app sends the updated or created records to the server. One problem that occurs rather sooner than later are merging conflicts. This problem is solved by the server, but how the server accomplishes is not described in their study.

When a patient comes to a doctor the doctor can access the patient's medical records and based on the information the doctor can make the diagnose. This diagnose can be added to the system which then issues a referral with a QR Code, if needed. This referral QR Code can be send to the patient's smartphone or tablet. If the patient prefers a non-electronically referral the QR Code can be printed together with the information it encodes. Therefore, the person receiving the referral only needs to scan the QR Code to get all necessary information to find the person in the system.

To ensure the security and privacy of the data every staff member has a role, which allows them to access certain files and do certain actions. For example, a nurse can update a patient's vital signs and check for allergies and the patient's medication. A doctor can also see the patient's laboratory results to consider this for their medical outcome.

The proposed system can eliminate the need for handwritten notes, which might get misinterpreted. Furthermore, some forms can be filled out automatically and records can be accessed and updated easier as well as vital signs. Altogether the system frees up time from administrative tasks, that can be used for patient care.

Another system proposed by Pryss et. al. [PMLR15] focuses more on digitizing ward rounds in hospitals. In their paper they developed an iPad application which assists medical staff during ward rounds. To develop the app, they first observed the day to day routines in ward rounds in four different departments to get an initial feeling for the needs of the app. They also asked doctors and nurses for their feedback to the app, thus an app was developed that was as easy and fast as the paper-based system used by the medical staff at the time. In contrast Mersini et. al. did not mention anything like this in their paper. The app developed by Pryss et. al. lets the medical staff see all the patients in their ward. In this overview all tasks for the selected patient are displayed with a status indication. Such a task for example could be doing an X-ray.

When navigating to the patient's details a different overview is shown. This shows all the patient's personal data like date of birth. Furthermore, vital signs and the anamnesis is displayed like in the system from Mersini et. al. [MST13]. Additionally, the current medication, the diagnosis, tasks, and the diagnostics are shown.

When examining diagnostics, the medical staff can change the state of the corresponding task to update it. On the other hand, the staff can easily create a new task on the diagnostics page, for example to request a new laboratory test. Text templates make the creation of tasks easier and faster as well as the ability to make a voice recording instead of typing.

After the development they evaluated the staff again with the result that the medical staff was satisfied with the application. One downside of the application might be that the application distracts the users from the patient and therefore the communication between staff and patient gets neglected. Furthermore, the staff wished to access medical reports. But they argue that the evaluation might not be suitable to be generalized, because the questionnaire was not designed by an expert, the sample size was relatively small, the participants were interrupted or distracted by their patients, and persons not directly participating in ward rounds were not questioned, like laboratory professionals or physical therapists.

2.3 Countrywide solution

Sabes-Figuera et al. conducted a survey among hospitals in Europe for the European Commission [SFMA+13]. They created a benchmark for the e-Health services in all countries of the European Union plus Iceland and Norway to compare the countries with each other. But we will focus on the comparison between Germany and Estonia, since Estonia is one of the most advanced countries in the European Union regrading digitalization.

The availability of inputting and viewing electronic health records is the best in Estonia and the usage of these features is the second highest. Germany is around the average for availability and usage. In Estonia, the exchange of health information has the highest availability and the second highest use, while Germany is below average for both availability and use. The use of telehealth is below average while the availability is above average in Estonia. Germany is below

average for both key figures. Sabes-Figuera et al. conclude that "most healthcare professionals do integrate these eHealth functionalities in their clinical care routines". This indicates that the doctors realize how e-Health can benefit their daily routine. All the hospitals from the survey shared their medical information electronically with external family doctors and specialists in Estonia. Whereas in Germany this information is not shared by all hospitals, compared to all countries in the survey the sharing rate with external family doctors is near the average and the sharing with external specialists is above average. This creates new problems of interchangeability between departments, since "54% [of the hospitals using electronic health records] reported interoperability problems at technical, semantic and/or organisational level". As we described in *Introduction* in Germany clinical information systems are often incompatible with each other even in one hospital with different information systems for different departments. This circumstance facilitates the interchangeability problem. The patients in Estonia could access their electronic health records in 75% of all surveyed to a certain degree. In Germany, no patient could access their data online. Relating to security measures Germany and Estonia are closer together with Estonia leading in most of the categories compared.

As we have seen in the survey from Sabes-Figuera et al. Estonia is better prepared for e-Health than Germany. But overall Germany could at least improve its rating from 2010 to 2012 but it was not as much as Estonia and it is still below average of all 30 countries. Based on these insights we first want to compare the structure of the health care systems from Estonia and Germany to discover similarities and differences to be able to compare the technical aspects of e-Health in both countries after that. Põlluste and Lember [PL16] wrote in their paper about the development of the Estonian health care system from 1918 to the early 2010s, which is approximately the time of the conduct of the survey form Sabes-Figuera et al. From 1918 to 1940 the health care system in Estonia was decentralized and the responsibilities for the health care laid upon the local municipalities. Estonia had three types of hospitals back then. First the state-owned hospitals, then the municipal hospitals and lastly private hospitals. Sickness funds were used to finance the health care system, by covering civil servants and employees. After the second world war was over the Estonian health care system was massively influenced by the Soviet Union and implemented the Semashko system. This meant, that the responsibility of the health care system laid on the state. The state was also the employer of the physicians and the system was entirely funded from the state budget. In towns there were polyclinics where the primary health care and the specialized doctors all practiced. Since both types of doctors where available so close together people tented to skip the primary health care doctors and go straight to the specialized doctors. This led to the point that "certain problems almost never reached the [public health care] doctor[s]". In comparison Germany was partitioned into four different zones. These zones became the Federal Republic of Germany and the German Democratic Republic. Both states had a completely different health care systems.

As written by Hurst [Hur91] the health services were provided by independent and public health providers in Western Germany. Most of the population was covered by public health insurance and the remaining parts were covered by private health insurance, which were "mainly higher income earners". If the income of a person passed a threshold, they could change from public to private health insurance. These public insurances were mainly funded by payroll taxes, where half of the amount was paid by the employee and the other half by the employer. However, the amount is capped. For private insurances dues are paid depending on the gender, age, and number of children of the insured person. Unlike the Estonian system in Western Germany small fees were charged for prescriptions, hospital stay, and nonemergency patient

transport. On the other hand, the health care systems of Estonia and the German Democratic Republic were quite similar since both countries were influenced by the Soviet Union. In Eastern Germany the health care system was also centralized and funded by taxes and the doctors were salaried and controlled by the state. Moreover, polyclinics were also standard as well as that all public health services were without charge. But the health care system suffered from a shortage of physicians in the 1980s. As written by Erices and Gumz [EG14] this was based mainly on the inadequate planned economy and the hemorrhage of doctors to Western Germany. The main reasons for doctors to leave were the poor equipment, missing medicine and medical equipment, physical and mental overload because of the under staffing, and the low wages. After the reunification the health care system of the western states was almost completely transferred to the eastern states, for example polyclinics were maintained for five more years [MMB02].

After the fall of the Soviet Union and regaining independence Estonia began to reform the health care system. They established a social health insurance, a new system of sickness funds and divergent from the Semashko system they defined what should be covered by the health insurance instead of covering all treatments. So, the funding did not come from the state anymore. As we have seen this is quite like all the public and private health insurances in Germany. Furthermore, the health care system became decentralized again and now was based on family medicine. This required the introduction of family doctors. Each of these family doctors have a patient list, where the patient can register themselves. This way the patient can choose which family doctor is the first contact for their health-related problems. The patient lists have roughly 1 600 persons on them and if the patient list exceeds 2 000 persons than the doctor "is expected to hire an assistant doctor" to prevent overburdening one family doctor. As Põlluste and Lember describe, the family doctors act as a kind of gatekeeper in the health care system. The family doctors give "advice concerning the prevention of diseases, takes preventive measures and issues health certificates, certificate of incapacity for work and prescriptions". Furthermore, they send the person to specialized medical personal if they can not treat the patient. This is also a big contrast to the Semashko system and ensures that the specialized doctors get exonerated. In Germany, every person can choose their family doctor too. Furthermore, they act also as gatekeepers to specialized doctors. To get an appointment with specialized doctors the patient needs a referral from the family doctor but specialized doctors can be visited without a referral in case of an emergency. Further on dentists, ophthalmologists, and gynaecologists can be visit without a referral too [fre]. All insured persons can access the public health care services without any extra charge. This includes visiting the family doctor, certificates of incapacity of work, and prescriptions. But for other documents like the health care certificate, which is needed for a driver's license, the patient could have to pay a fee. As discussed before in Germany some services like prescribed costs a slight fee. We have seen that the health care systems of Estonia and Germany are similar structured, but the the Estonian healthcare is considerably more advanced in e-Health than the German healthcare as seen at the beginning of this section. Hence, we want to look more deeply at the e-Health system of Estonia to see what is possible and has been proved to work.

Under the name e-estonia [e-e] Estonia has launched a program to digitize among other things visits to the authorities and health care. The latter is the one we want to examine and consists of three components: e-health records, e-ambulance, and e-prescription. The e-health records are health records that integrate "data from Estonia's different healthcare providers to create

a common record every patient can access online". This acts as a central database that can display the data from different systems in a standard format. Therefore, the e-health records allows the doctors to access a patient's full history in just one file. Like the application proposed by Mersini et al. [MST13] these files can include basic medical results like allergies, medication, and diagnoses. The system is even capable of providing the doctors with new records or test results as they are entered, like blood tests or X-ray images. Consequently the waiting time for test results is reduced to a minimum and more time can be devoted to patient care. To ensure the integrity of the records, which means that the data stored in such records cannot be modified unauthorized without noticing it, blockchain technology is used. The blockchain also logs who accessed the files to prevent misuse of all the highly personal data stored in the system. Therefore, the logs cannot be manipulated by anybody to cover up their misdemeanor. All persons can access their own records all the time and this extends to their underage children and people who gave them access to access them. These records include the doctors visits which can be reviewed afterwards, the current prescriptions, and a log of all doctors who had accessed their files. This can all be accessed with their ID-card, which is like the German government-issued identification card, which gives access to all the e-Estonia features. They also add, that "in an emergency situation, a doctor can use a patient's ID code to read time-critical information, such as blood type, allergies, recent treatments, on-going medication or pregnancy.", which can benefit the care of the patient greatly. For example if a patient has a penicillin allergy and the doctors sees this early in the treatment process and can react accordingly. "The system also compiles data for national statistics, so the ministry can measure health trends, track epidemics", which can be really helpful to combat the ongoing covid-19 pandemic. This might be one reason why the number of infected people is much lower with 1 800 [cova] in Estonia than in Germany with 176 752 [covb] as of 2020-05-21. This results in 0.14% [popb] and 0.21% [popa] of the population infected in Estonia and Germany respectively. But it is still to early to tell for sure if the lower numbers are due to e-estonia, since many factors matter like the distribution of population in cities and rural areas. The next component is the e-Ambulance. This component helps the dispatcher of the emergency service to get time critical information for the paramedics and within 30 seconds the position of the phone can be detected to send an ambulance as fast as possible. The dispatcher can use the ID of the patient to get the patient's health records. These records are automatically filled in a form which provide basic information about the patient like age, allergies, and the blood type. This data can then be accessed by the ambulance on their way to the patient, which ensures that the paramedics can make the right decisions when arriving at the location. On the way to the hospital the paramedics can assess the condition of the patient to input this into the health information system. Afterwards the doctors can automatically access the patient's medical records, most recent test results, and the consultations with the family doctor or medical specialists. Therefore, the hospital can prepare for the patient before the patient arrives at the hospital and doctors can quickly aid them. But in comparison to the emergency function of smartphones and the QR Code based solution the data in this system can be accessed on the way to the patient. The eAmbulance system lets the paramedics read this data on the way to the patient, which saves the paramedics time and the patient care can begin sooner. The last component is the e-Prescription component. These are a "centralized paperless system for issuing and handling medical prescriptions", which means that the doctor can prescribe medicine with an online form. All the patient needs then is to show the pharmacist the ID-card. Thereby the pharmacist can access the information from the patient and can issue the medicine. Since the e-Prescription system works with the data from

the national health insurance fund the medicine is discounted according the subsides appertain to the patient. One major benefit is the issuing for repeated prescriptions. Since everything necessary is available online the patient can contact the doctor digital and the doctor can issue a repeated prescription quickly in order that the patient can directly go to the pharmacy.

One major problem remaining is the information exchange rate between European countries. As of 2013 the amount of hospitals that share information with other EU countries is less than 8% [SFMA+13]. To ensure that the system can work across borders in Europe a standardized format for the data or even an information system is needed. Currently the European Commission is working on a system that allows for such a sharing of health data across countries in Europe [Ano18]. Depending on the implementation such a system might be the best possible solution for the EU.

3 Background Information

In this chapter we want to dive into codes that are used in the medical field and look at why and what they are used. Thereafter we want to discuss some codes to encode data and compare them to each other.

3.1 ICD Codes

In 1853 a resolution was passed for a consistent international nomenclature of causes of death as stated by the DIMDI (German Institute for Medical Documentation and Information) [icdb]. In 1893 the International List of Causes of Death (ILCD) was implemented in North America and in 1899 it was recommended in Europe to implement the ILCD too. After that in 1900 the first international overhaul conference about the ILCDs were held and the list was declared mandatory. Thereupon followed four more such conferences. In 1948 the 6. revision conference were held and the WHO was entrusted with the creation of the International Classification of Diseases (ICD). Contrary to ILCD the ICD codes encoded not only the causes of death. As stated by the World Health Organization [whoa][whob] ICD codes encode "diseases, disorders, injuries and other related health conditions". Since the data format of the codes is standardized this enables hospitals and countries to share these data with one another. Furthermore, the codes are consistent over the years, which makes data analysis possible. Therewith health and death trends can be found, and appropriate actions can be made. Because all member states of the WHO and the few missing regions probably also use the ICD codes, it is quite safe to say that ICD codes are used all over the world. For example a doctor in Tokyo can read the diagnoses done by a German doctor.

In Germany, all doctors and psychotherapists are required to encode their diagnoses with an ICD-10-GM code [icda]. The GM stands for German Modification, which means that the international codes are translated to German. Hence the international compatibility is still in place.

3.2 PZN

In Germany, the PZN (pharmaceutical central number) encodes medication, assistive technology, and pharmacy products. The numbers are published by the IFA (information agency for proprietary medicinal products) [ifa]. The PZN is an eight-digit number, which encodes medication unambiguously based on the denotation, the administration form, and the packaging size. This number is only valid for Germany, but the German PZN can easily be transformed

Figure 3.1: Code 39 of "HELLO WORLD"

to the PPN (Pharmacy Product Number), which is unique worldwide. Therefore, it is sufficient to only store the PZN instead of the PPN. On the package of medication, the PZN is either encoded as a Code 39 or in a Data Matrix. Since 2019-02-09 the PZN does not have to be encoded in Code 39 anymore, instead the Data Matrix is used. The Data Matrix encodes more than just the PZN, it also encodes the serial number, the batch number, and the expiration date.

3.3 Code 39

The Code 39 is a linear barcode, which means that the code has only one dimension. As stated, by Fang et al. [FWL+06] the Code 39 can encode uppercase letters, all digits, and some special characters like "-" or "." and was invented in 1974. Each character consists of 9 bars with 5 of them black and 4 white. Three of the black bars are wider than the other black bars, Thus the name Code 39. The Code 39 has a start, information, and end part. Start and end encode an asterisk while the information part encodes the representation of the information plus the check sum. Therefore, the Code 39 has no build in error correction but rather error detection. One example can be seen in figure 3.1. With the proposed procedure by Fang et al. a rotated Code 39 or a Code 39 with noise can still be read if the asterisks can be found and the noise is not too high. An extension to the Code 39 is the Full ASCII Code 39 where all ASCII symbols, which are not part of the Code 39 already, are represented by two Code 39 characters. Furthermore, the size of a Code 39 grows with the size of the stored information. Therefore, the code can be as long as you want, but a code which is too long can cause problems when reading it.

3.4 Data Matrix

The Data Matrix is the first two-dimensional code we want to look at. As written by Albrecht [Alb12] the Data Matrix was developed in the late 1980s in the USA by Acuity Corporation and first used in 1994 in Germany. Unlike the Code 39 the Data Matrix has a maximum number of characters it can encode. While only using numeric characters 3 116 digits can be encoded. The Data Matrix can encode up to 2 335 alphanumeric characters and 1 556 bytes. Contrary to Code 39 the Data Matrix has built in error correction. Data Matrix uses the Reed-Solomon error correction. The smallest Data Matrix has 3 bytes for data and 5 bytes for error correction. Therefore, up to 2 errors can be corrected. The largest possible Data Matrix has 1 558 bytes for data and 620 bytes for error correction and can correct up to 310 errors. One example can be seen in figure 3.2. Besides the square pattern a rectangular pattern is also

Figure 3.2: Data Matrix of "HELLO WORLD"

Figure 3.3: QR Code of "HELLO WORLD"

possible with the same features. One big advantage of the Data Matrix is the possibility to emboss, laser, or engrave the code into products. Therefore, the Data Matrix becomes permanently associated with the object, which would not be possible by printing. As we learned in the section about *PZN* that the Data Matrix is replacing the Code 39 on medication, assistive technology, and pharmacy products. One reason might be the build in error correction and another may be the compactness of the Data Matrix versus the Code 39. This is illustrated in figure 3.4. It is also used to encode the medication plan. In this use case it encodes some personal information like name and birthday and the PZN of the medication with the intake pattern.

3.5 QR Codes

The second two-dimensional code we want to look at is the QR Code. It was developed in 1994 by DENSO WAVE [qrc]. The goal was to develop a code that can be read as fast as possible. This problem was solved with the three big square marks, which enables position detection. Besides the fast reading the QR Code can also be read from any angle. This was a big advantage versus codes like Data Matrix or Code 39. Although these codes could be read from different angles as shown by Lin and Lin [LL13]. Since this technique must be implemented in the readers of these codes the QR Code is still predominant regarding this feature, because it is built in and every QR Code reader can use it. After the introduction the code was first used by the Japanese automotive industry, which could improve their efficiency. Compared to the Data Matrix the QR Code can encode more data. It can encode up to 7 089 numeric, 4 296 alphanumeric, 2 953 bytes, or 1 817 Kanji characters, which is the name of the Japanese characters. Moreover, a mixture of these types is possible. One example can be seen in figure 3.3. Contrary to the Data Matrix the QR Code has a variable error correction. The error correction capability is divided in four level (L, M, Q, H) which can restore 7%, 15%, 25%, or 30% of the codewords, respectively. With more error correction the amount of data which can be encoded drops. So, the peak numbers presented early can only be reached with an error correction level L. Furthermore, DENSO WAVE developed new enhancements. The

(a) QR Code **(b)** Data Matrix

(c) Code 39

Figure 3.4: "The quick brown fox jumps over the lazy dog" encoded with different codes

first is the Micro QR Code, which as the name suggest is a smaller version of the QR Code. The maximum amount of numerical characters a Micro QR Code can encode is 35 while the smallest QR Code can encode 41. It still can encode alphanumeric, byte, and Kanji characters. The second one is the iQR Code, which can hold 80% more information than the normal QR Code. This allows roughly 40 000 characters. By implication this means that the iQR Code needs less space than the normal QR code and even than the Data Matrix. Like the Data Matrix the iQR Code can also be presented as a rectangular. Beyond this the iQR Code has an error correction level S which can restore a code where 50% of the data part is damaged. The third code is the SQRC which is a QR Code with reading restrictions. It can encode public and private information. While the public information can be read with any device the private information can only be read with a device that has the cryptographic key to decode the hidden information. A user without the key does not see that private information is hidden. The last code is the Frame QR which is a QR Code with a canvas area, where images or letters can be inserted, while preserving the features of the QR Code. While these are great additions only the QR Code and the public SQRC information can be read with a smartphone out of the box. This is also an advantage versus the Data Matrix and the Code 39 which both can not be read out of the box by smartphones. In figure 3.4 you can see that the size of the squares in the Data Matrix are larger than the QR Codes. As previously mentioned, the Code 39 needs the most space out of these three codes. While encoding the same data the QR Codes need more space to be readable, but they can encode more data. If everybody is to read the code or a massive amount of data is to be encoded QR Codes should be chosen. If only little space is available a Data Matrix should be used.

4 Implementation

We decided to use the razor pages form the .NET framework for our implementation, because the razor pages come with different templates that facilitates building a website. First the template offered the functionality for account creation. Therefore, we had not to deal with password handling, which can be quite tricky. Furthermore, it enabled us to publish our website quickly to test it as we will show in the next chapter where we discuss a *Use Case*. As of 2018 the General Data Protection Regulation must be implemented, which the templates also support.

Our goal was to implement an application that can be used to relive the workload of paramedics, hospital staff, and doctors in Germany. We wanted to create a centralized database were the patient's information could be stored and easily accessed by all medical workers as it is possible in the Estonian healthcare system. Our idea is to use QR Codes to access this information, since nearly everybody already has a smartphone capable of scanning QR Codes. After scanning the QR Code at first an emergency message should be displayed which gives the emergency responder all critical information about the patient. Quite like the techniques discussed in the section about the *Corporate sector* solutions. Razor pages use the Bootstrap Framework and therefore, the website is quite usable with smaller screen devices. Nonetheless the application can greatly benefit from a smartphone application. We will discuss the *Mobile* aspects in greater detail in the following section. Our application is divided in two different parts. The first part is the identity part where the user accounts with all their information is managed. The second part is the data part where all medical files are managed.

4.1 User Management

First, we want to talk about the identity part, which was mostly provided by the ASP.NET Core and adjusted for our needs. The class representing the user is called $MedicUser$ and inherits from the ASP.NET Core $IdentityUser$ class, which provides a basic user with an Id, a username, email address, and phone number. Furthermore, it provides a field to store the password hash and a counter for the failed login attempts. Both will be explained more in detail in the section about *Security*. Our model adds more personal information about the user like the name of the user, the users date of birth and place of birth, the sex of the user, and the profession of the user, which can help in some cases, because some profession benefits certain diseases. For example, miners have a higher chance for lung diseases, doctors have a higher risk of infections, and athletes have a higher risk of injuries. Additionally, the health insurance company and the insurance number are stored in the model to help the doctors with the accounting process with the health insurances. As we learned in the section about *Countrywide*

solution in Germany every person has a family doctor and therefore this information is also stored in the model. To identify the family doctor distinctively the name, street, city, and postal code are stored. Furthermore, the model holds the information about what medical personal status of the user. This determines the role of the user in the system, like the system proposed by Mersini et al. [MST13]. The roles are *user*, *noos*, *oos*, and *doctor*. Medical personal without the obligation to secrecy get the role *noos* and persons with the obligation of secrecy therefore, get the role *oos*. In case the user is a doctor the model has also the possibility to add the lifetime medical number, which every doctor has. Another important field is the Boolean $medicalPersonalConfirmed$ field, which indicates if the value in the *medicalPersonal* field is confirmed by the system and therefore also the role of the user. We will see why this field is important soon. How the roles work is discussed in greater detail in the section about Privacy. The user also can store a private and public emergency notice like the SOSQR [sos] application discussed in the Corporate sector section. Finally, the user has an URL that leads to the user's data. Hereafter we will call this user URL to avoid confusion with the URL entered in the browser. With this URL the emergency notices and the medical files can be accessed via the QR Code. Both the private and the public notice entries are what you see is what you get editors to enable the user to structure the text with bullet points, use different colors and font sizes and insert tables. Therefore, the text can be better readable and highlight important points. On the other hand, to colorful and unstructured texts created by the user can make the text more difficult to read, which costs more time. We used the free open source editor TinyMCE [tin].

```
<div class="row">
  <div class="form-group col-md-6">
      <label asp-for="Input.MedicalPersonal"></label>
      <select asp-for="Input.MedicalPersonal" class="form-control">
          @for (int x = 0; x < Model.roles.GetLength(0); x++)
          {
              <option value="@Model.roles[x, 0]">@Model.roles[x, 1]</option>
          }
      </select>
      <span asp-validation-for="Input.MedicalPersonal" class="text-danger"></
    span>
  </div>
  ...
</div>
```

Listing 4.1: Register front-end example

Displayed in Listing 4.1 is one input field of the form from the register page as an example, because all other fields are structured similarly. All fields that are semantically similar are surrounded by a *row* class, which is a wrapper for columns. Each row can have up to twelve columns. As seen in the second line our input field is furthermore enclosed by a *col-md-6* class. This class uses six of the twelve available columns, which corresponds with 50% of the available space. The parameter *md* ensures that this rules only applies for medium or larger sized screens with equal to or greater than 768 pixels. For a semantic group of four entries we used the classes *col-lg-3* *col-md-6* for each entry like our example here. This class causes on larger screens with equal to or greater than 992 pixels that every entry takes up one-fourth of the available space since every entry needs three of the twelve columns. When the screen is smaller than 992 pixels but equal to or greater than 768 pixels each of these four entries only

takes six of the twelve columns and therefore half of the space. If the screen is smaller than 768 pixels every entry in the register form takes up a full row. This ensures that the form uses the space that is available on larger screens optimal and maintains readability on smaller screens especially smartphone screens.

```
public readonly string[] sexes = Util.Util.PossibleSex;
public readonly string[,] roles = Util.Util.AllRoles;

[BindProperty]
public InputModel Input { get; set; }

public class InputModel
{
    [Required]
    [DataType(DataType.Text)]
    [Display(Name = "Are you medical personal?")]
    public string MedicalPersonal { get; set; }
    ...
}

public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    Util.Util.ValidateRegister(Input, ModelState);

    if (ModelState.IsValid)
    {
        var user = new MedicUser {
            Name = Input.Name,
            UserName = Input.Email,
            Email = Input.Email,
            URL = Util.Util.GetUniqueRandomUserUrl(_userManager),
            MedicalPersonal = Input.MedicalPersonal,
            MedicalPersonalConfirmed = false,
        };
        var result = await _userManager.CreateAsync(user, Input.Password);
        var roleResult = await _userManager.AddToRoleAsync(user, user.
    MedicalPersonal);
        if (result.Succeeded && roleResult.Succeeded)
        {
            if (_userManager.Options.SignIn.RequireConfirmedAccount)
            {
                return RedirectToPage("RegisterConfirmation", new { email =
    Input.Email });
            }
        }
    }
    return Page();
}
```

Listing 4.2: Register back-end excerpt

In the back end we need an *InputModel* to store the inserted information from the user, which can be by the front-end. An excerpt from the model can be seen in listing 4.2 in the lines four to 14. The *Required* tag tells the system that this entry cannot be null and therefore must have a value, which was inserted by the user. The *DataType* tag signals the system which data type is expected. Besides the text data type date, postal code, HTML, password, phone number,

and email are also possible. The password data type hides the input in the password fields, so that the inserted cannot be read. The *Display* tag is used to give the variable a different name when displayed to the user. Every input entry has the same structure throughout our entire application. Each entry consists of a label, an input field, and a span, therefore we want to explain the structure of our example in Listing 4.1. The *asp-for* tag is a Tag Helper provided by the ASP.NET Core and lets us access variables from the back end. The label displays the name of the field for the user and to get the name we use the Tag Helper. With the Tag Helper we can access the *Input* model from the back end and therefore also the variables, which is *MedicalPersonal* in our case. As we saw in Listing 4.2 the *MedicalPersonal* variable has a *Display* tag with the value "Are you medical personal?". The label is now set to this value. If we did not specify a name with the *Display* tag the variable name would be displayed, which would be *MedicalPersonal* in this case. Following the label are either input or select tags to get the user's information. In line five to eight of the Listing 4.1 we can see one feature of the razor pages, which let us use C# code in the HTML document. In these lines we fill the drop-down list (<select>) from which the user can choose their employment status in the medical field. The *Model* objects always point to the corresponding back-end class and therewith we can get all options we want to display to the user. In line two of the back-end code from Listing 4.2 we get the values form our *Util* class. This ensures we always use the same options on different pages and increases the maintainability. The *roles* array is a two-dimensional array and stores the names of the roles with their respective display name. As you can see, we used the $asp-for$ tag again. This time it tells the application where to store the information the user provided. In the back end the inserted value is available with the *Input* class and the variable *MedicalPersonal*. After sending the form with the HTML POST method the *OnPostAsync* in the back end gets called. This method is the handler for all POST requests and an excerpt can be seen in line 16 in Listing 4.2. Before we can save all the information, we must check if all inserted values are valid. Most of the inputs can be checked by the types of the variables and is done automatically by the system. For example, the variable which stores the birth date of the user must be a valid date or a variable with the *Required* tag must not be *null*. Because the system can only check for syntactical errors we must still check for semantic errors. In line 18 we call the *ValidateRegister* function with the *Input* object, that stores all inserted user values, and the *ModelState* object, which is a dictionary of all variables of the *Input* object and their validation status. This function can be seen in Listing 4.3.

```
public static void ValidateRegister(RegisterModel.InputModel inputModel,
    ModelStateDictionary modelState)
{
    bool validRole = false;

    for (int i = 0; i < AllRoles.GetLength(0); i++)
    {
        if (inputModel.MedicalPersonal == AllRoles[i, 0])
        {
            validRole = true;
            break;
        }
    }

    if (!validRole)
    {
```

```
16            modelState.AddModelError("Input.MedicalPersonal", "Please choose one
    entry from the list");
        }
18  }
```

Listing 4.3: Validation of the registration inputs in the Util class

The function is straightforward because we only iterate over the *AllRoles* array and check if the insert role is in this array. If the inserted role is not in the array, we add a new error to the *ModelState*. In line 20 in Listing 4.2 we test if all model values are valid. If this is not the case we return the page again in line 40. Now the *span* tags display the error message in line 10 in Listing 4.1, if the key specified with *asp-validation-for* is not valid. If the model is valid we can create a new user. We must generate a unique user URL so that we can access the user's files later. The code can be seen in Listing 4.4.

```
public static string GetUniqueRandomUserUrl(UserManager<MedicUser> userManager)
2   {
        string random = GetRandomString();
4
        while (userManager.Users.Where(user => user.URL.Equals(random)).Count() !=
    0)
6       {
            random = GetRandomString();
8       }
        return random;
10  }
```

Listing 4.4: Create a random URL for the user

As you can see in line 28 in Listing 4.2, we set the *MedicalPersonalConfirmed* variable to false, since everybody can choose the doctor option. Nevertheless, we assign the user the role they chose. How we want to handle and check this and how the *GetRandomString* function works is described in greater detail in the section about *Privacy*. After the user and the user's role is created successfully we want to send the user the registration confirmation email. This can be seen in lines 34 to 37, where we call the *RegisterConfirmation* page to handle this. The page can be seen in Listing 4.5. We first check if an email was provided and then if we can find a user with this email in our database. If both variables contain valid entries, we begin to build the message of the email. After we built the message, we call the Task *Execute*, which uses SendGrid to assemble and send the email to the provided address.

```
public async Task<IActionResult> OnGetAsync(string email)
2   {
        if (email == null)
4       {
            return RedirectToPage("/Index");
6       }

8       var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
10      {
            return NotFound($"Unable to load user with email '{email}'.");
12      }

14      var code = await _userManager.GenerateEmailConfirmationTokenAsync(user);
```

```
    code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
    EmailConfirmationUrl = Url.Page(
        "/Account/ConfirmEmail",
        pageHandler: null,
        values: new { area = "Identity", email, code },
        protocol: Request.Scheme);

    Execute(user).Wait();

    return Page();
}
```

Listing 4.5: Register confirmation page

After the user registered and confirmed their email address they can log in and change their profile any time. This includes everything the user could enter on the register page, except the lifetime medical number, since this number should not change, and the username. If the medical personal status changes the $MedicalPersonalConfirmed$ is set to false again. Now the user can also add two-factor authentication to their account. To be compliant with the GDPR the user can download or delete their personal data, but this only includes the information about the user and not their other files they probably have. Before the application can be released this must be fixed that all data from the user can be downloaded or deleted. Additionally the user can display their QR Code which leads to their emergency notice and later to their files. We decided to use a QR Code because every person and not only medical personal should be able to access the emergency notice. This is achieved with QR Codes since all modern smartphone cameras can read QR Codes without any extra applications. They also offer great error correction which can be helpful in some cases where the code got damaged in an accident.

4.2 Medical Files Management

The medical file part consists of the four main pages allergies, medication, reports, diseases, and vaccinations and the two pages ICD-Codes and abbreviations. All main pages plus the abbreviations page supports all CRUD operations and can display all entries in a list view, which can be viewed on their index page. All index pages are paginated so that not all entries are loaded at once. The ICD page only allows for reading access.

First, we want to start with the allergies page, because it is the easiest of the main pages and all pages are similarly structured. The allergy model can be seen in Listing 4.6.

```
public class Allergy
{
    public int ID { get; set; }
    [Required]
    public string UserID { get; set; }

    [Required]
    public int PersonalID { get; set; }
    public string Name { get; set; }

    public string Reaction { get; set; }
```

```
12
        [DataType(DataType.Html)]
14      public string Description { get; set; }

16      [Timestamp]
        public byte[] Version { get; set; }
18  }
```

Listing 4.6: Allergy model

An Allergy consist of a *Name*, the severity of the *Reaction*, and a *Descpription* for further information and are provided by the user. The name and the severity are required, while the description is optional. For the description field we use the same what you see is what you get editor as on the register page and therefore we must set the data type of the description to HTML. The severity can be severe, mild, or moderate and we store this information in the *Util* class, so that we have a central point to access it. Each severity has a different shape and color combination on the index page. A red triangle is used for severe, a yellow square is used for moderate, and a green circle is used for mild cases. This lets the users quicker understand what severity an allergy has because the user must not read the text. In the case the user is colorblind or has dyschromatopsia the shapes help these users instead of the colors. The remaining parameters are only for internal use and can therefore not be entered by the user. The value *ID* is used internally to store the allergy in the database. The *UserID* is the id of the user the allergy entry belongs to and the *PersonalID* is the id of the allergy for the user. We will discuss the need of a separate *PersonaID* in the section about *Privacy*. The last value *Version* is used later to detect concurrency errors.

All medical file pages inherit from a base class *MedicPageModel*, which inherits form the *PageModel* class provided by the .NET CORE. The *MedicPageModel* has only three properties. The first is a Boolean variable *OwnPage* indicating if the current page is the page of the user or the page of another user. The second variable *UserOfPage* is a representation of the user whom the page belongs to. The last variable *MedicUrl* is the string of the user URL. In Listing 4.7 we can see the back end implementation of the index page of the allergy files.

```
    public string NameSort { get; set; }
2   public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }
4   public bool IsDoctor { get; set; }
    public PaginatedList<Allergy> Allergy { get;set; }
6
    public async Task OnGetAsync(string url, string sortOrder, string filter, string
        search, int? pageIndex)
8   {
        if (User.IsInRole("doctor"))
10      {
            IsDoctor = true;
12      }

14      if (url == null)
        {
16          OwnPage = true;
        }
18      else
        {
```

```
20          string userHasNoPermission = Util.Util.GetHandler(User, _userManager, "
    Allergies", url, "doctor", "oos");

22          if (userHasNoPermission != null)
            {
24              url = null;
                OwnPage = true;
26          }
            else
28          {
                OwnPage = false;
30          }
        }
32
        MedicUrl = url;
34
        CurrentSort = sortOrder;
36      NameSort = string.IsNullOrEmpty(sortOrder) ? "name_desc" : "";

38      if (search != null)
        {
40          pageIndex = 1;
        }
42      else
        {
44          search = filter;
        }
46
        CurrentFilter = search;
48
        UserOfPage = Util.Util.GetUserOfFile(_userManager, url, User);
50
        IQueryable<Allergy> allergies = from a in _context.Allergies
52                                      where a.UserID == UserOfPage.Id
                                        select a;
54
        if (!string.IsNullOrEmpty(search))
56      {
            allergies = allergies.Where(a => a.Name.Contains(search));
58      }

60      allergies = sortOrder switch
        {
62          "name_desc" => allergies.OrderByDescending(a => a.Name),
            _ => allergies.OrderBy(a => a.Name)
64      };

66      allergies = allergies.OrderByDescending(a => a.Reaction);

68      Allergy = await PaginatedList<Allergy>.CreateAsync(allergies.AsNoTracking(),
        pageIndex ?? 1);
}
```

Listing 4.7: Back-end index page of allergies

The index page has only one GET handler in line seven. All the parameters in handler function take the value from the query part of a URL. When calling the page with the query

*/Allergies?url=*12345 the variable *url* in the back end has the value 12345. After calling the function we first set all page parameter. Since a doctor can create, edit, view, and delete an entry and a user with obligation of secrecy can only view the entry we have check if the user is a doctor. This is done in lines nine to twelve. After that we want to check if the user wants to see their own page or the page of another user. We first check if a user URL is provided in lines 14 to 17 and if this is not the case, we know that the user wants to call their own page. If a user URL was provided by the user, we first must check if the user is authorized to see the page. This procedure was outsourced to the *GetHandler* function because we must do this on every page. The code can be seen in Listing 4.8.

```
public static string GetHandler(ClaimsPrincipal user, UserManager<MedicUser>
    userManager, string pageName, string url, params string[] roles)
{
    if (user == null)
    {
        return "/Index";
    }

    MedicUser medicUser = GetUserOfFile(userManager, null, user);

    if (url != null)
    {
        if (!HasUserOneOfTheseRoles(medicUser , user, roles))
        {
            return $"/{pageName}/Index";
        }

        Util.Url = url;
    }

    return null;
}
```

Listing 4.8: Method to redirect not authorized accesses

The *GetHandler* function returns either a string of the page where we will redirect a user without the necessary authorization or null if the user is authorized to see the page. If there is no logged in user we want to redirect the user to the main index page. After that we want to get the user of the accessed file. The implementation of the function can be seen in Listing 4.9.

```
public static MedicUser GetUserOfFile(UserManager<MedicUser> userManager, string
     url, ClaimsPrincipal claimedUser)
{
    if (url != null)
    {
        IQueryable<MedicUser> user = userManager.Users.Where(user => user.URL ==
     url);

        if (user.Count() == 1)
        {
            return user.FirstOrDefault();
        }
        else
        {
            return GetUserById(userManager, claimedUser);
```

```
14          }
        }
16      else
        {
18          return GetUserById(userManager, claimedUser);
        }
20  }
```

Listing 4.9: Method to get the user of a file

We first check if a user URL is provided. If none is provided, we use the *GetUserById* function to get the user calling the page. If a user URL is provided, we first check if a user with this user URL exists and if we find one user with this user URL, we can return the user. There should always be only one user if the provided user URL is valid, since we check that the assigned user URL is unique in the registration process (see Listing 4.4). If we find no user with the user URL we return the user calling the page. We handle both cases that no or a wrong user URL is provided the same way. Then we can continue in Listing 4.8. We again check if a user URL was provided. If a user URL was provided, we check if the user has a role that authorizes them to access the page and that the role is confirmed by checking if *MedicalPersonalConfirmed* is true. Should this not be the case we want to redirect the user to their medical file page. When the user has a role that authorizes them to access the page, we save the user URL and return *null* to indicate that the user is authorized. We do the same if the user wants to access their own files. When for example a doctor is working on a patient's files, we must save the user URL therewith the doctor can navigate through the patient's files. This can be seen in Listing 4.10.

```
@{
2       string url = Util.Util.Url;
}
4   <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-page="/Allergies/Index" asp-
        route-url="@url">Allergies/Intolerances</a>
6   </li>
```

Listing 4.10: Example of the navigation

When the doctor clicks on the link the *asp-rout-url* sets the URL query *url* to the value stored in *@url*. If we would not do this the doctor would always access their own files.

Now we can continue in Listing 4.7. We store the user URL, the current sort order, and the sort order for the allergy name in page parameters. If the *search* parameter is empty, we set it to the value of the *filter* value. Otherwise we set the *pageIndex* to 1, to start at page one of the allergy list. After that we save the *search* value in the page parameter *CurrentFitler*. Next, we want to know the user of this page. This is done by the *GetUserOfFile* explained earlier in Listing 4.9. With this information we can retrieve all the allergies from the user. The query can be seen in lines 51 to 53. Then we want to apply the search string to filter the allergies, if a search string is provided. See lines 55 to 58 for that. Then we sort the remaining allergies in lines 60 to 64. Depending on the value stored in *sortOrder* either ascending or descending by name. Finally, we sort all allergies by their severity, thus all allergies are always presented from severe to mild. Depending on *sortOrder* all severe allergies are sorted ascending or descending. This also applies to moderate and mild allergies. Finally, we created the paginated allergy list with the code provided by Microsoft [RAa].

Now we can use all this to build the index page. Depending on the *OwnPage* variable we display a slightly different heading as seen in Listing 4.11.

```
@if (Model.OwnPage)
{
    <h1>Your Allergies/Intolerances</h1>
}
else
{
    <h1>Allergies/Intolerances of @Model.UserOfPage.Name</h1>
}
```

Listing 4.11: Front-end index page heading of allergies

This makes it obvious if a doctor is accessing their own files or the files of a patient. Next, we see the search function in Listing 4.12.

```
<form asp-page="./Index" method="get">
    <p>
        Allergy: <input type="text" name="search" value="@Model.CurrentFilter"
    />
        <input type="submit" value="Search" class="btn btn-primary" /> |
        <a asp-page="./Index">Back to full list</a>
    </p>
</form>
```

Listing 4.12: Search index page of allergies

This form uses the GET method for the *asp-page* "*./Index*" and therefore points to the back end GET handler function in line seven in Listing 4.7. If the user already searched for an allergy it is displayed in the input field. We use the value that is stored in the *CurrentFilter* variable, that we set in the back end. If the user clicks on the link to go back to the full list the page is called without any query parameters and this also resets the sort order as we will see later.

The Listing 4.13 only shows the column with the name of the allergy. Normally the index page also shows the severity, but since it has a similar structure as the name, we do not show it here. The remark is not shown on the index page, since it is not as important as the other two values and can require much space.

```
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort" asp-
    route-filter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model => model.Allergy[0].Name)
                </a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Allergy)
        {
            <tr>
                <td>
```

```
16                      @Html.DisplayFor(modelItem => item.Name)
                    </td>
18                  <td>
                        @if (Model.MedicUrl != null)
20                      {
                            @if (Model.IsDoctor)
22                          {
                                <a asp-page="./Edit" asp-route-url="@Model.MedicUrl"
     asp-route-id="@item.PersonalID">Edit</a> @("|")
24                              <a asp-page="./Details" asp-route-url="@Model.
    MedicUrl" asp-route-id="@item.PersonalID"> Details</a> @("|")
                                <a asp-page="./Delete" asp-route-url="@Model.
    MedicUrl" asp-route-id="@item.PersonalID">Delete</a>
26                          }
                            else
28                          {
                                <a asp-page="./Details" asp-route-url="@Model.
    MedicUrl" asp-route-id="@item.PersonalID"> Details</a>
30                          }
                        }
32                      else
                        {
34                          <a asp-page="./Edit" asp-route-id="@item.PersonalID">
    Edit</a> @("|")
                            <a asp-page="./Details" asp-route-id="@item.PersonalID">
    Details</a> @("|")
36                          <a asp-page="./Delete" asp-route-id="@item.PersonalID">
    Delete</a>
                        }
38                  </td>
                </tr>
40          }
        </tbody>
42  </table>
```

Listing 4.13: Front-end index page of allergies

As we can see in lines five to seven when clicking on the header of the name field the page is called again with the value of *NameSort* and the value of *CurrentFilter* as query parameters for *sortOrder* and *filter*, respectively. In the body of the table we display the values for all allergies provided by the back end. In lines 19 to 37 we print what the user can do with these files depending if it is their file or what role they have. A doctor and the user of the file both can edit, delete, and view the details of the file. A user with the *oos* role can only view the details of the file. All other users get redirected to their own files as we saw in lines 20 to 30 in Listing 4.7. The values *asp-route-id* and *asp-route-url* set the URL query of *id* and *url*, which are needed to retrieve the users file. How this works will be described later.

Listing 4.14 shows how the pagination works. The Listing shows only the button for the next page, since the button for the previous page works similarly.

```
@{
2       var nextDisabled = !Model.Allergy.HasNextPage ? "disabled" : "";
}
4
<a asp-page="./Index"
6       asp-route-sortOrder="@Model.CurrentSort"
```

```
    asp-route-pageIndex="@(Model.Allergy.PageIndex + 1)"
    asp-route-filter="@Model.CurrentFilter"
    class="btn btn-primary @nextDisabled">
    Next
</a>
```

Listing 4.14: Pagination of the allergies index page

First, we must check if we can display a next page. By clicking the button, we increment the *PageIndex* by one to display the next page. We also must forward the current sort order and filter, so that it is still the same on the next page. With the class we can control if the button can be clicked or not. If *nextDisabled=disabled* the button gets disabled and can not be clicked anymore.

But before we can list any entry, we must create them first. On the index page is a create new button, which when pressed calls the create page of an allergy. The code of the GET handler is displayed in Listing 4.15.

```
public IActionResult OnGet(string url, string name, string reaction, string
    description)
{
    string redirect = Util.Util.GetHandler(User, _userManager, "Allergies", url,
    "doctor");
    if (redirect != null)
    {
        return RedirectToPage(redirect);
    }

    Util.Util.SetPageParameter(redirect, url, _userManager, User, this);

    Allergy = new Allergy();
    Util.Util.SetAllergyCreateParameter(Allergy, name, reaction, description);

    return Page();
}
```

Listing 4.15: GET handler create allergy

The GET handler can be called with a user URL and all the possible parameters of an allergy entry. We first check if the user calling the page is authorized to do so in line three with the *GetHandler* function as seen before in Listing 4.8. If we should redirect the user, we redirect them in line six to the specified page. Otherwise we can continue to build the page. Then we call the *SetPageParameter* function, which can be seen in Listing 4.16.

```
public static void SetPageParameter(string redirect, string url, UserManager<
    MedicUser> userManager, ClaimsPrincipal user, MedicPageModel pageModel)
{
    if (redirect == null && url == null)
    {
        pageModel.OwnPage = true;
    }
    else
    {
        pageModel.MedicUrl = url;
        pageModel.OwnPage = false;
    }
```

```
12
        pageModel.UserOfPage = GetUserOfFile(userManager, url, user);
14  }
```

Listing 4.16: Method to set all page parameters

This function sets all the page parameter inherited from the *MedicModelPage* class. Next, we create a new *Allergy* object to store the user's inputs and if some values are already provided with the GET request we set them in line twelve. The used function only sets all allergy properties with the provided values. Then we can deliver the page to the user.

The user can now fill out the form. For the severity we use a drop-down list, which is filled by the severity values stored in the *Util* class. After the user pressed submit the POST handler in Listing 4.17 is called.

```
public async Task<IActionResult> OnPostAsync(string url)
2   {
        string redirect = Util.Util.GetHandler(User, _userManager, "Allergies", url,
        "doctor");
4       if (redirect != null)
        {
6           return RedirectToPage(redirect);
        }
8
        Util.Util.SetPageParameter(redirect, url, _userManager, User, this);
10
        Allergy.UserID = UserOfPage.Id;
12      ModelState.Remove("Allergy.UserID");

14      IQueryable<Allergy> allergies = from a in _context.Allergies
                                        where a.UserID == UserOfPage.Id
16                                      orderby a.PersonalID descending
                                        select a;
18      Allergy.PersonalID = allergies.Count() == 0 ? 1 : allergies.FirstOrDefault()
        .PersonalID + 1;

20      Util.Util.ValidateAllergy(Allergy, ModelState);
        if (!ModelState.IsValid)
22      {
            return Page();
24      }

26      _context.Allergies.Add(Allergy);
        await _context.SaveChangesAsync();
28
        return RedirectToPage("./Index", new { url });
30  }
```

Listing 4.17: Create new allergy

The beginning is the same as the GET handler except the POST handler only takes the user URL from the call. We first check if the user sending the POST request is authorized to do so. If the user should not be authorized, we redirect them to the page provided by the *GetHandler* function. Then we can set all page parameters and set the *UserID* of the allergy because this could not be provided by the user, because there is no field for the *UserID* in the form Next

we must remove the *UserID* entry from the *ModelState*, which is false since the *UserID* was not provided in the form send by the user although it is required for a valid allergy. Then we fetch all the allergies the user has and sort them by the *PersonalID* descending. We now can give the newly created allergy a *PersonalID*, which is unique for the user. Depending on the quantity of allergies the user has it is either one or the largest $PersonalID + 1$. The only value we must check is the severity value and is done by checking if the value is in the array with the reactions stored in the *Util* class. The other two values only must be checked if they are syntactically correct, which is done by the system. Should the model not be valid we return the page again for the user to fix the errors. Now we can save a new allergy in line 26 and 27. After we successfully stored the new allergy the user is redirected to the index page of the allergies. To stay on the same user, we forward the user URL, otherwise a doctor would need to enter the user URL of the patient again. Now we can call the edit, delete, and display page displayed in Listing 4.13 on the index page. As you can see, we always forward the *PersonalID* of the entry and in case the user is working on the files of another user we must forward the user URL.

We will first talk about the detail page, which is shown in Listing 4.18.

```
public async Task<IActionResult> OnGetAsync(int? id, string url)
{
    if (id == null)
    {
        return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
    = "Please select an allergy with an id." });
    }

    Allergy = await _context.Allergies.FirstOrDefaultAsync(m => m.PersonalID ==
    id && m.UserID == UserOfPage.Id);

    if (Allergy == null)
    {
        return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
    = $"There was no allergy found with the id: {id}" });
    }

    return Page();
}
```

Listing 4.18: Show details of an allergy back-end

The GET handler has both the personal *id* and the user URL as parameters. If no personal *id* was provided we redirect the user calling the page we redirect the user to an error page. Otherwise we again use the *GetHandler* and *SetPageParameter* functions to check if the user is authorized and set all the page parameters if this is true. Both functions are not displayed in the Listing because they have the same structure as in Listing 4.15. Then we can retrieve the allergy from the database with the aid of the *PersonalID* of the allergy and the *UserID*. If we cannot find an allergy we also redirect the user to an error page. This most likely happens when the *PersonalID* is wrong because the user entered it manually. If everything is correct we can return the page which displays the allergy and because we declared the description as HTML, the description text is formatted correctly as HTML text. The user then can either go back to the full list or edit this entry as seen in Listing 4.19.

```
<div>
    <a asp-page="./Edit" asp-route-url="@Model.MedicUrl" asp-route-id="@Model.
    Allergy.PersonalID">Edit</a> |
    <a asp-page="./Index" asp-route-url="@Model.MedicUrl">Back to List</a>
</div>
```

Listing 4.19: Show details of an allergy front-end

After clicking the edit button, the GET handler of the edit page is called. The structure of the handler is the same as the GET handler of the details page as seen before in Listing 4.18 including the authorization check and the call of the *SetPageParameter* function. The only difference in this handler are the two lines displayed in Listing 4.20.

```
HttpContext.Session.SetInt32("Allergy.ID", Allergy.ID);
HttpContext.Session.Set("Allergy.Version", Allergy.Version);
```

Listing 4.20: GET handler edit allergy excerpt

These two lines are inserted right above the return of the page. Here we store the *ID* and the *Version* of the allergy the user wants to edit in the session and therefore in the server. The details are discussed in the section about *Security*. The front-end looks like the front-end of the create page. After the user saves the edited entry the POST handler is called, which is shown in Listing 4.21. The first lines are excluded since they are the same as the lines three to twelve in the POST handler of the create page in Listing 4.17.

```
public async Task<IActionResult> OnPostAsync(string url)
{
    int? id = HttpContext.Session.GetInt32("Allergy.ID");

    Util.Util.ValidateAllergy(Allergy, ModelState);

    if (!ModelState.IsValid)
    {
        return Page();
    }

    if (!id.HasValue)
    {
        return RedirectToPage("./Index", new { url });
    }

    var AllergyToUpdate = _context.Allergies.FirstOrDefault(x => x.ID == id.
    Value);

    if (AllergyToUpdate == null)
    {
        return HandleDeletedAllergy();
    }

    _context.Entry(AllergyToUpdate).Property("Version").OriginalValue =
    HttpContext.Session.Get("Allergy.Version");

    if (await TryUpdateModelAsync<Allergy>(AllergyToUpdate, "Allergy", x => x.ID
    , x => x.UserID, x => x.PersonalID, x => x.Name, x => x.Reaction, x => x.
    Description))
    {
```

```
28          try
            {
30              await _context.SaveChangesAsync();

32              HttpContext.Session.Remove("Allergy.ID");
                HttpContext.Session.Remove("Allergy.Version");
34
                return RedirectToPage("./Index", new { url });
36          }
            catch (DbUpdateConcurrencyException ex)
38          {
                var exceptionEntry = ex.Entries.Single();
40              var clientValues = (Allergy)exceptionEntry.Entity;
                var databaseEntry = exceptionEntry.GetDatabaseValues();
42
                if (databaseEntry == null)
44              {
                    ModelState.AddModelError(string.Empty, "Unable to save. The
    allergy was deleted by another user.");
46              }

48              var dbValues = (Allergy)databaseEntry.ToObject();
                SetDbErrorMessage(dbValues, clientValues, _context);
50
                ModelState.Remove("Allergy.Version");
52              HttpContext.Session.Set("Allergy.Version", (byte[])dbValues.Version)
    ;
            }
54      }

56      return Page();
}
```

Listing 4.21: POST handler edit allergy excerpt part 1

In the third line retrieve the *id* of the allergy stored in the session. Should the session be expired the edited values get discarded and the user gets redirected to the index page as shown in lines 12 to 15. The user has currently ten minutes time to enter all changes before the session expires. This should be plenty of time to edit an entry. Nevertheless, a system which does not discard all changes would be more user friendly. We know the user URL and the *PersonalID* and could call the edit page again and forward the entries of the edited allergy, so that the user can continue editing. Additionally, a timer could be added showing the user how much time is left. In the best cast the timer and therefore the session time reset every time the user clicks something or presses a key. After retrieving the *id* we check if the edited allergy model is valid in lines five to ten. This is the same as in the POST handler of the create page in Listing 4.17. Now we try to retrieve the allergy from the database by its *id* in line 17. Afterwards we check if the allergy were deleted before we could save the changes in lines 19 to 22. If the system could not retrieve an allergy it was deleted by another user and we call the *HandleDeletedAllergy* function. The function adds an error to the *ModelState* telling the user the edited allergy was deleted, sets the *IsDeleted* page parameter to true, and returns the page with all the changes the user made. Now that the *IsDeleted* page parameter is true the page does show a create as new button instead of the save button. This can be seen in Listing 4.22.

```
<div class="form-group">
```

```
    <input type="submit" value="Create as new" class="btn btn-primary" asp-page-
    handler="New" asp-route-url="@Model.MedicUrl" />
</div>
```

Listing 4.22: Front-end edit allergy excerpt

The *asp-page-handler* attribute specifies the handler in the back end. This button now calls the *new* POST handler as seen in Listing 4.23 instead of the normal POST handler.

```
public async Task<IActionResult> OnPostNewAsync(string url)
{
    return RedirectToPage("./Create", new { url, Allergy.Name, Allergy.Reaction,
     Allergy.Description });
}
```

Listing 4.23: POST handler to create a new allergy

In this POST handler we check if the user is authorized the same way as in all the other functions. If the user is authorized, we redirect them to the create page and forward all allergy values and the user URL to stay on the same user. Now we can use the forwarded allergy values to fill the fields in the create page, so that the user can save the allergy again. The remaining part of the normal POST handler after the retrieval of the allergy from the database is displayed in Listing 4.24.

```
public async Task<IActionResult> OnPostAsync(string url)
{
    _context.Entry(AllergyToUpdate).Property("Version").OriginalValue =
    HttpContext.Session.Get("Allergy.Version");

    if (await TryUpdateModelAsync<Allergy>(AllergyToUpdate, "Allergy", x => x.ID
    , x => x.UserID, x => x.PersonalID, x => x.Name, x => x.Reaction, x => x.
    Description))
    {
        try
        {
            await _context.SaveChangesAsync();

            HttpContext.Session.Remove("Allergy.ID");
            HttpContext.Session.Remove("Allergy.Version");

            return RedirectToPage("./Index", new { url });
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Allergy)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();

            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty, "Unable to save. The
    allergy was deleted by another user.");
            }

            var dbValues = (Allergy)databaseEntry.ToObject();
            SetDbErrorMessage(dbValues, clientValues, _context);
```

```
30                  ModelState.Remove("Allergy.Version");
                    HttpContext.Session.Set("Allergy.Version", (byte[])dbValues.Version)
    ;
32              }
    }
34
    return Page();
36 }
```

Listing 4.24: POST handler edit allergy excerpt part 2

After we checked if the allergy was deleted from the database we now must check if the entry was altered by another user before the current user saved the changes. Before we can do that we must set the original value of the *Version* property and therefore we retrieve the version information from the current session. In line five we try to update the allergy from the database with the new values edited by the user. If this succeeds, we run into the try-catch block. Here we try to save the changes of the allergy and if everything goes well, we can remove the session entries for the *ID* and the *Version* and redirect the user to the index page. It can happen that another user has changed and saved the entry before the current user could save their changes and then the *SaveChangesAsync* function throws a concurrency exception. In this case we retrieve the allergy object the current user wants to save and the allergy that is stored in the database. Now we check again if the entry was deleted and display the error message to the user and afterwards we can cast the allergy from the database to an allergy object. Then we use the *SetDbErrorMessage* function in line 28 to set all error messages. This function checks which entries of the allergy are changed and displays an error message with the changed values, if this is the case. Next, we must remove the old *Version* value in the session and replace it with the *Version* value from the database. This works like a GET call without discarding all the changes made by the user. Finally, we return the page to the user. Now the user can see which values have changed and what the new values are. The user can now decide what values to use and save the edited entry again.

Another page which the concurrency must be checked is the delete page. The GET handler can be seen in Listing 4.25.

```
public async Task<IActionResult> OnGetAsync(int? id, string url, bool?
    concurrencyError)
2 {
    if (id == null)
4   {
        return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
    = "Please select an allergy with an id." });
6   }

8   string redirect = Util.Util.GetHandler(User, _userManager, "Allergies", url,
    "doctor");
    if (redirect != null)
10  {
        return RedirectToPage(redirect);
12  }

14  Util.Util.SetPageParameter(redirect, url, _userManager, User, this);
```

```
16      Allergy = await _context.Allergies.FirstOrDefaultAsync(m => m.PersonalID ==
        id && m.UserID == UserOfPage.Id);
        if (Allergy == null)
18      {
            return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
        = $"There was no allergy found with the id: {id}" });
20      }

22      if (concurrencyError.GetValueOrDefault())
        {
24          ConcurrencyErrorMessage = "The record you attempted to delete "
                + "was modified by another user after you selected delete. "
26              + "The delete operation was canceled and the current values in the "
                + "database have been displayed. If you still want to delete this "
28              + "record, click the Delete button again.";
        }
30
        HttpContext.Session.Set("Allergy.Version", Allergy.Version);
32
        return Page();
34  }
```

Listing 4.25: GET handler to delete an allergy excerpt

The GET handler is like the GET handler of the detail page as shown in Listing 4.18. We first check if an *id* was provided and redirect the user to an error page if no *id* was provided. Then we check if the user is authorized and redirect them if this is not the case. Differently from the details page we only call the *GetUserOfFile* function displayed in Listing 4.9 instead of the *SetPageParameter* function. With this information we can retrieve the allergy from the database to display it to the user. We will explain the lines 22 to 29 later. Then we save the *Version* for the concurrency check later and return the page. After the user pressed the delete button the POST handler in Listing 4.26 is called.

```
public async Task<IActionResult> OnPostAsync(int? id, string url)
2   {
        string redirect = Util.Util.GetHandler(User, _userManager, "Allergies", url,
        "doctor");
4
        if (redirect != null)
6       {
            return RedirectToPage(redirect);
8       }

10      if (id == null)
        {
12          return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
        = "Please select an allergy with an id." });
        }
14
        MedicUser user = Util.Util.GetUserOfFile(_userManager, url, User);
16
        try
18      {
            Allergy = await _context.Allergies.AsNoTracking().FirstOrDefaultAsync(m
        => m.PersonalID == id && m.UserID == user.Id);
20
```

```
        if (await _context.Allergies.AnyAsync(m => m.PersonalID == id && m.
    UserID == user.Id))
22      {
            Allergy.Version = HttpContext.Session.Get("Allergy.Version");
24          _context.Allergies.Remove(Allergy);
            await _context.SaveChangesAsync();
26          HttpContext.Session.Remove("Allergy.Version");
        }
28
        return RedirectToPage("./Index", new { url });
30  }
    catch (DbUpdateConcurrencyException)
32  {
        return RedirectToPage($"./Delete", new { url, concurrencyError = true,
    id });
34  }
}
```

Listing 4.26: POST handler to delete an allergy

As always, we first check if the user is authorized to access the page and then we check if an *id* was provided. Like the GET handler we only need to call the *GetUserOfFile* function, because we only need the user of this file. Next, we try to delete the allergy entry and to do this we first need to retrieve the allergy entry from the database by the *PersonalID* and the *Id* of the user. If we could find the allergy in the database, we retrieve the *Version* information from the session and pass this value to the allergy. Then we try to delete the allergy from the database and save the changes. If everything works, we can remove the *Version* information from the session. Finally, we can redirect the user to the allergy index page. This is also done when no allergy was found in the database. If we encounter a concurrency error while saving the changes we redirect the user to the same delete page while forwarding the user URL, the *id*, and the *concurrencyError* set to true. Now the lines 22 to 29 in Listing 4.25 come into effect. The user now sees the error message and the new values of the allergy entry. If the user decides to delete the file nonetheless the user can press the delete button again.

The explanation for all the other pages is shorter since they are like the allergy pages. The vaccination page is like the allergy page. Vaccinations also have an *ID*, *UserID*, *PersonalID*, *Description*, and *Version* field. These are used to store and retrieve the vaccination to and from the database as well as handling concurrency errors. Furthermore they have a *name*, *agent*, a Boolean *refresh* field indicating if the vaccination must be refreshed, a *RefreshUntil* date when the vaccination must be refreshed, and four fields to identify the family doctor giving the vaccine, which are *name*, *street*, *city*, and *postalcode*. The edit and create pages check that either both the refresh checkbox is checked and a *RefreshUntil* date are provided or neither of these two are selected. Otherwise the back end would return an error message, so that the user can decide if the vaccination needs to be refreshed or not. When creating a new vaccination entry, the fields for the family doctor are filled automatically with the values stored by the user. On the index page the vaccinations are displayed on a paginated list and can either be sorted by name or by the *RefreshUntil* date. The user can also search the vaccinations for a specific agent or vaccine name.

The ICD Code page only has an index and a details page. The index page shows all ICD with the corresponding name in a paginated list. Both the codes and the names can be sorted

ascending and descending. When changing the sort order the old gets overridden. It is also possible to search for a specific ICD Code either by the code itself or the name it encodes.

The next page are the abbreviations. These let the doctors store abbreviations for an ICD Code so that they can use the abbreviation instead of the ICD Code when entering diseases. For storing to and retrieving from the database we need an *ID*, *UserID*, *PersonalID*, and a *Version* to check for concurrency errors. We also need two string fields *Abbr* and *Code* to store the abbreviation with the corresponding ICD Code. On the index page a paginated list is shown, where the doctor can sort the entries either by the code of the abbreviation. They also, can search for entries by their code or name. When creating or editing an abbreviation the doctor can choose the code from a drop-down list and after posting the new or edited entry the system checks if the entered code is a valid ICD Code.

The next page is the report page, where health reports can be saved. Each report has an *ID*, *PersonalID*, *UserID*, and *Version* as all the other files. Furthermore, a report has a *Title*, *Subject*, *Hospital*, *Department*, *DoctorName*, *Date*, and *Description* attributes. The *Date* attribute is filled out automatically with the current date the user creates the report. The user can also add and upload multiple files to the report. The code can be seen in Listing 4.27.

```
foreach (var formFile in FileUpload.FormFiles)
{
    var formFileContent =
        await FileHelpers.ProcessFormFile<BufferedMultipleFileUploadDb>(
            formFile, ModelState, _permittedExtensions,
            _fileSizeLimit);

    if (!ModelState.IsValid)
    {
        Result = "Please correct the form.";
        return Page();
    }

    var file = new MedicFile()
    {
        UserID = UserOfPage.Id,
        PersonalID = personalIdFiles,
        ReportID = Report.PersonalID,
        Content = formFileContent,
        UntrustedName = formFile.FileName,
        Note = FileUpload.Note,
        Size = formFile.Length,
        UploadDT = DateTime.UtcNow
    };

    personalIdFiles++;
    _context.MedicFiles.Add(file);
}
```

Listing 4.27: Upload files

If the user uploads one or more files we iterate over each of the files and process its content. Should some inputs be invalid we return the page again so that the user can fix the errors. If everything is correct we create a new file. We need the *UserID*, a *PersonalID*, *ReportID* to link the file to the report it was uploaded to, *Content*, *UntrustedName*, *Note*, *Size*, and the

upload date. The *ID* is created by the system. Some more details are discussed in the section *Security*. Then we increment the personal id, which is used for the next file. Finally, we can save the file and after all uploaded files are correctly saved, we can save the report itself. When editing a report new files can be upload and the stored files can be downloaded or deleted. The download handler can be seen in Listing 4.28.

```
public async Task<IActionResult> OnGetDownloadAsync(int? id, string url)
{
    if (id == null)
    {
        return NotFound();
    }

    string redirect = Util.Util.GetHandler(User, _userManager, "Reports", url, "
    doctor", "oos");

    if (redirect != null)
    {
        return RedirectToPage(redirect);
    }

    Util.Util.SetPageParameter(redirect, url, _userManager, User, this);

    var requestedFile = _context.MedicFiles.SingleOrDefault(x => x.PersonalID ==
     id && x.UserID == UserOfPage.Id);

    if (requestedFile == null)
    {
        return Page();
    }

    var stream = new MemoryStream(requestedFile.Content);

    return File(stream, MediaTypeNames.Application.Octet, WebUtility.HtmlEncode(
    requestedFile.UntrustedName));
}
```

Listing 4.28: Download handler

In this handler we do the same things as in the other edit GET handler. We first check if the forwarded *id* is valid, then if the user is authorized to download the file and finally, we set the page parameter. Thus, we can retrieve the file from the database and if we find a file, we can open a stream and send the file to the user. The user can also delete the file. After pressing the delete button the user gets redirected to a page where the user must confirm the deletion. Both deleting and downloading a file can be done on the details and delete page of a report too. When deleting a report all files affiliated to the report get deleted too, to leave no files without a report. If this would not be done it could happen that a report that has the same *PersonlID* as the deleted shows the files that were left in the database. To ensure that all files are deleted after the report is deleted, we first delete all files before we remove the report from the database. The reports can be sorted by the date and searched by the subject.

With this report page we give the medical staff access to medical reports as they wanted it from the application by Pryss et. al. [PMLR15].

The next page is the diseases page. A disease has an *ID*, *UserID*, *PersonalID*, and a *Version* attribute as all the other pages. Furthermore, it has a *Code* (the ICD Code), *Name*, *SinceDate*, *UntilDate*, *EndedThrough*, *EndedAt*, *ExpectedDuration*, *IDReport*, *TreatedBy*, *Relapse*, and *Description* field. When creating or editing a disease the *ExpectedDuration* can have the values unknown, lifetime, and until date. If the user chooses the until date, then the *UntilDate* field must have a value. In the other two cases the *UntilDate* field must be empty. If this disease is a relapse the doctor can check the *Relapse* checkbox to indicate this and saving time writing this down. If something is inserted in the *EndedThrough* field the disease must have a *EndedAt* date and if only one of these fields has a value the back end returns an error to the user. It is also possible that both fields are empty. The *IDReport* field is filled by a drop-down list, which displays all reports from the user from newer to older. From this list the user has to choose one report to link the disease with this report. A doctor can enter the name, an abbreviation, or the ICD Code of the disease. They can choose from a *datalist* for the codes and the names. Both lists are sorted alphabetically, but the list for the names additionally shows the abbreviations of the doctor at the top of the list. If the doctor enters a name from the list and an ICD Code that do not match an error gets returned, but if the doctor enters a name not from the list and a valid ICD Code, this name and code combination is stored as a new abbreviation. After the user pressed the create button the POST handler gets called. The beginning is the same as in every other page. We first check if the user is authorized and then set the page parameters. The Listing 4.29 shows the new part which handles the ICD Code and name.

```
if (Disease.Code != null)
{
    Disease.Code = Disease.Code.ToUpper();
}

Util.Util.ValidateDisease(Disease, ModelState, reports, codes, names,
    abbreviations);

if (!ModelState.IsValid)
{
    return Page();
}

MedicUser currentUser = Util.Util.GetUserOfFile(_userManager, null, User);

if (Disease.Code == null)
{
    Disease.Name = Disease.Name.ToLower();
    Disease.Code = Util.Util.CheckIfNameHasACode(names, abbreviations, Disease.
    Name);
    Disease.Name = codes.Find(c => c.ICDCode == Disease.Code).Name;
}
else
{
    if (Disease.Name != null && User.IsInRole("doctor") && currentUser.
    MedicalPersonalConfirmed && !names.Any(x => Disease.Name == x.Name) && !
    abbreviations.Any(x => Disease.Name == x.Abbr))
    {
        IQueryable<Abbreviation> abbreviations = from a in _context.
    Abbreviations
```

```
26                                                  where a.UserID == currentUser.
    Id
                                                    orderby a.PersonalID descending
28                                                  select a;

30          int abbreviationPersonalID = abbreviations.Count() == 0 ? 1 :
    abbreviations.FirstOrDefault().PersonalID + 1;

32          Abbreviation abbreviation = new Abbreviation
            {
34              UserID = medicUser.Id,
                Abbr = Disease.Name,
36              Code = Disease.Code,
                PersonalID = abbreviationPersonalID
38          };

40          _context.Abbreviations.Add(abbreviation);
        }
42      Disease.Name = codes.Find(c => c.ICDCode == Disease.Code).Name;
    }
```

Listing 4.29: Creating new disease entry

First, we check if a code was provided therewith, we can cast the code to an uppercase string. We do this to make the code comparable in the *ValidateDisease* function. Otherwise the user would get an error message if the code contained lowercase characters, since all the stored ICD Codes only have uppercase characters. The *ValidateDisease* function also checks if the user entered a valid name/abbreviation without a code, a valid code without a name, a valid code and a valid name/abbreviation that encode the same disease, or a valid code with a name not in the stored in the database as abbreviation or a name from the ICD Codes. If the entered disease is valid we can continue and retrieve the user currently creating or editing the disease. We then check if a code was provided and if this is not the case, we change the name to all lowercase characters to make it comparable. At this point we can be sure that the user entered a name, since we tested that with the *ValidateDisease* function in line 6. The *CheckIfNameHasACode* function in line 18 returns null if the there is no ICD Code for the entered name and if there is an ICD Code the code gets returned. We can be sure, that the *CheckIfNameHasACode* function returns a valid code, because otherwise the *ValidateDisease* function would return an error and we would not reach line 18. Then we use this code to set the name to the correct name of the ICD Code. If a code was provided, we check if we must create a new abbreviation. We can only create a new abbreviation if the user entered something in the name field, that is not already a valid name or an abbreviation of the user. Furthermore, we must check if the user is a confirmed doctor and if all these requirements are fulfilled we can create a new abbreviation and add it to the database. At the end we get the correct name from the ICD Code and write it to the *Name* attribute. Afterwards we can add the new disease in the database and save all changes. On the index page the user can sort and search the entries by name and code like on the code page. The stored id value in *IDReport* is used to create a link to the report like in Listing 4.30.

```
@if (Model.MedicUrl == null)
2 {
    <a href="/Reports/Details?id=@Html.DisplayFor(modelItem => item.IDReport)">
    Report</a>
```

```
4 }
  else
6 {
      <a href="/Reports/Details/@Model.MedicUrl?id=@Html.DisplayFor(modelItem =>
      item.IDReport)">Report</a>
8 }
```

Listing 4.30: Report link on the index page

We always set the *MedicUrl* to the forwarded user URL in the GET handler, therefore we can first check if the user is working on other user's files by checking if the *MedicUrl* has a value. If the field has a value we must include it in the link, otherwise the user would call their own report with an id that might not exist.

The last page is the medication page. A medication also has an *ID*, *UserID*, *PersonalID*, and a *Version* attribute. The remaining attributes are: *Name*, *Agent*, *PZN*, *SinceDate*, *UntilDate*, *ExpectedDuration*, *Dose*, *DeliverySystem*, *Morning*, *Noon*, *Evening*, *Night*, *Importance*, and *Description*. The *PZN* encodes both the name and the agent of the medication as well as some other values as we learned in the section about the *PZN*. It would therefore be sufficient to only provide the PZN while creating a new medication entry, but since we found no publicly available database for PZNs the user must enter a name, agent, and the PZN every time. The *ExpectedDuration* can have the values unknown, lifetime, and until date like on the disease page when creating or editing a medication. Furthermore, the *ExpectedDuration* and the *UntilDate* must match the same way as on the disease page. This means an until date must be selected if the "untilDate" option is selected. In the other cases no until date may be selected. The *Dose* is a string value, so that inputs like "10mg" or "10ml" are possible. The *DeliverySystem* is by default filled with "oral" since it is the most common method. If the user needs to enter something different they can choose from a drop-down list. The user can select the options "normal", "slightly important", "important", "quite important", and "very important" in the importance field. In the system these values are stored as numbers from 1 to 5, where 1 represents "normal". *Morning*, *Noon*, *Evening*, and *Night* represent the intake pattern. On the index page all entries can be sorted by the agent and the since date. When sorting by agent the medication entries are sorted descending by their importance after sorting them by the agent, so that more important medication is always above less important medication. One unique feature of the medication page is the ability to copy an already existing entry. When the user clicks the copy button the copy page gets called, which is basically an edit page, where the *Name*, *Agent*, *PZN*, and *Description* are emptied. After pressing the save button the edited medication entry gets saved as a new one. Another unique feature is the QR Code that encodes the medication to act like a medication plan. Currently the PZN, the until date, and the intake pattern separated by a comma is encoded. If the until date is null or an entry for an intake pattern is 0 an empty string is displayed instead of null or 0, respectively. We encode all medication that have a since date in the past and with an until date that is either null or in the present or future. The date is represented as "YYYYMMDD". We decided to use a QR Code instead of a Data Matrix, which is currently widely used on medication plans, because the QR Code allows us to store more information and space is not an issue. If no personal information like the name should be encoded it is sufficient to only use numerical characters to encode the medication plan. We know that the PZN is an eight-digit number, the date can also represented as an eight-digit number in the pattern "YYYYMMDD" and each intake is a number between 0 and 9. If no date is provided, we could use for example eight zeros to

indicate that, without changing the length of the string. Therefore, we only need 20 digits to encode a medication. This would us allow to encode over 300 medication with a QR Code with the weakest error correction. The average patient does not need to take so much medication and therefore, we can add personal information at the beginning of the encoded text. Even if 1 000 characters are used for the personal information, we could encode 42 medication with the highest error correction [qrc], which should be enough. Compared to the current Data Matrix encoding the medication plan our solution offers more information, since in the current solution no until date is provided.

4.3 Mobile

A mobile application enables the hospital staff to access the patient's files at the patient rather than a stationary computer. This solution minimizes travel distances, since the staff can go directly from patient to patient without needing to go to a stationary computer. Our application provides this mobility. With an ordinary smartphone every staff member can quickly scan the QR Code to obtain the patients files and because we are using the responsive design framework Bootstrap for our website, the web pages are rendered in a small screen friendly way. For example when the header row has not enough space all entries collapse into a burger menu to avoid overflows or line breaks. As seen in the section about *User Management* we use a different number of columns depending on the screen size. On smaller screens each input field gets the full row, so that the presentation of the data is as clear as on larger screens. One problem the Bootstrap framework could not solve were the index pages of the medical files. Most of the index pages already display many information from each entry in a table, which needs space that is not available on smaller screens, because the table cannot be wrapped to the next line as input fields can be. We fixed this problem as shown in Listing 4.31.

```
<th class="d-sm-none d-lg-table-cell">
    @Html.DisplayNameFor(model => model.Medication[0].UntilDate)
</th>
<th class="d-lg-none">
    active
</th>

<td class="d-sm-none d-lg-table-cell">
    @Html.DisplayFor(modelItem => item.UntilDate)
</td>
@if (item.UntilDate.HasValue && item.UntilDate < DateTime.Now)
{
    <td class="d-lg-none">Yes</td>
}
else
{
    <td class="d-lg-none">No</td>
}
<td>
```

Listing 4.31: Mobile index medication page excerpt

The Listing shows an excerpt from the medication index page with only two values. The top part shows the table header and the bottom part shows the corresponding data part. The

column for the *UntilDate* is only displayed when the screen is larger and when the screen is smaller it is not displayed to save space. In comparison the new *active* column is hidden on larger screens and only shown on smaller screens. This *active* column replaces the three columns *UntilDate*, *SinceDate*, and *ExpectedDuration*, which saves a great amount of space. Some other columns are hidden without a replacement. Thus, we can shrink the width of the table to a minimum, so that it does not overflow in the landscape mode of a smartphone. The web page should therefore be used in landscape mode, since the table still overflows in the portrait mode.

One drawback here is that the data always gets sent regardless of whether the user uses a desktop computer or a smartphone. With an app solution this overhead of data can be omitted. An app does not need the CSS and JavaScript files, which would save additional bandwidth. This would benefit especially users on the go like paramedics. The app should feature a build-in QR Code scanner to quickly open the patient's files. Furthermore, it would be great if the user could adjust where the app redirects the user to. For example, instead of showing the user the emergency message every time the QR Code gets scanned the app shows an overview of all medical files or goes directly to a specific file like diseases. Furthermore, the proposal of Mersini et al. [MST13] can be used to extend the app with a database. This database could store the files of the last patients a doctor looked up, so that if the connection to the central database is lost the doctor can still work. It would also be possible to edit or create an entry and later upload it to the central database when the connection is restored.

4.4 Security

Our application stores highly private information and these should be protected as effectively as possible because a central database is a more lucrative target than multiple smaller decentralized databases. The Open Web Application Security Project [owa] (short: OWASP) is a project dedicated to improve the security of software. OWASP issues the OWASP Top Ten every year, which are the top 10 security risks for web application like our application. We want to show some of these security risks and how we handled them in our application. The first security risk are injections. The most common example are SQL injections, where the attacker sends a manipulated value to the server and executed by the SQL server. With this attack the attacker could download the whole database or manipulate/delete entries. In our application we only use LINQ to access the database. LINQ prevents a SQL injection by turning the user input into parameter values and therefore preventing the malicious commands from running [ste]. Furthermore, the URL query parameters are escaped. A call to */Allergies/Details?id='or'1'='1* is transformed to */Allergies/Details?id=%E2%80%99or%E2%80%981%E2%80%99=%E2%80%991*. The GET handler shown in Listing 4.32 expects an integer value and in this call *id* would be *null* and no database access is executed.

```
public async Task<IActionResult> OnGetAsync(int? id, string url)
{
    if (id == null)
    {
        return RedirectToPage("/NotFound", new { model = "allergy", errorMessage
    = "Please select an allergy with an id." });
    }
```

```
}
```

Listing 4.32: GET handler allergy detail excerpt

The second security risk is a broken authentication, where the attacker gets control over an account in the system. Accounts of doctors would be the main target of an attacker, because these accounts have the most access. One way of getting the access to the accounts of users are brute force attacks, where attacker tries many passwords and hoping one guess is correct. Brute force attacks are prevented by the system by locking the account after 5 false password attempts for 5 minutes, which is the standard setting from the .NET CORE. This system could be abused to deny the service to a user, if the attacker knows the email address of the user. The .NET CORE takes care of salting and hashing the passwords too. Furthermore, the user can add a two-factor authentication and is enforced to use a strong password. This is done by forcing the user to use at least one uppercase and one lowercase character as well as one digit, one special character, and at least six characters. This is the default set by the .NET CORE.

Next is the broken access control risk, where an attacker can access files, they are not authorized to do so. To prevent this, we only let user access files in the system if they are logged in, which can be seen in Listing 4.33.

```
services.AddRazorPages().AddRazorPagesOptions(options =>
  {
    options.Conventions.AuthorizeFolder("/Abbreviations");
    options.Conventions.AuthorizeFolder("/Allergies");
    options.Conventions.AuthorizeFolder("/Codes");
    options.Conventions.AuthorizeFolder("/Diseases");
    options.Conventions.AuthorizeFolder("/Medications");
    options.Conventions.AuthorizeFolder("/Reports");
    options.Conventions.AuthorizeFolder("/Vaccinations");
  });
```

Listing 4.33: Authorization for all folders

This enables us to log every access and attribute this to a user in the system. Furthermore we always check if a user is authorized to see a file, when the user calls a page, and return the appropriate page. This is done with the function *GetHandler* (Listing 4.8). Every handler first calls the *GetHandler* to determine if the user is authorized and if this is not the case, they get redirected to their index page.

Cross-site scripting is another security risk. Here the attacker uploads malicious code to a website and when a user visits this website the code gets executed in the browser of the user. Razor automatically encodes all the output from variables [RAb] and escapes the information when the variable is accessed by the "@" symbol in an HTML context. This prevents the browser to interpret user inputs as scripts. The used method can be seen in Listing 4.34 from the details page of the allergy page.

```
@Html.DisplayFor(model => model.Allergy.Description)
```

Listing 4.34: Razor encoding example

The last security risk we want to discuss is insufficient logging and monitoring, to quickly detect suspicious changes on the website. Currently only account creations are logged, which is inadequate for the data we want to store. The application should log every access to files

to find suspicious behavior early, which lets us act earlier and may prevent damage to the system. Furthermore, a blockchain solution as discussed in *Countrywide solution* should be implemented to make tampering the log nearly impossible.

Currently all the files uploaded to the system are only scanned for their file ending. In the release version the files should be scanned by an anti-virus or anti-malware API to protect the users downloading these files from malware.

4.5 Privacy

Another aspect besides security is privacy. In our application every user can see their own files, user with an obligation of secrecy can view the files of other users, and doctors can view, edit, delete, and create new files of other users. As mentioned in the previous section we always check just after the page is called in the HTML handlers if the user is authorized to do the action they want to do. This ensures that only persons which are authorized can see the files of another user. Depending on the role of the user different pages are delivered. For example, if an user with the *oos* role calls another user's files the application returns the index pages without the edit, create, and delete button. Nonetheless we still must check if the user is authorized to prevent an attacker from accessing the files, by directly calling the HTML handler.

The biggest issue here is how we can ensure that the user is eligible to the role they chose in the registration process. This is the reason we added the $medicalPersonalConfirmed$ field and set it to false in the registration process. To solve this problem, we need another role called $verifier$. The users with these roles would be able to set the $medicalPersonalConfirmed$ of other users to true. This could be done in two possible ways. The first one would be a video identification where the user must present the verifier documents that can prove the user is eligible for the claimed role. This procedure would be like the video identification when applying for a new bank account or mobile phone contract. Another possible solution could be a verifier in the hospital. This would be a selected group of people which would act similar to the personnel in a post office. The administrators of the system then must verify the users with the $verifier$ role. The first solution can easily be outsourced and would save the hospitals time and money.

As we have seen we created a unique user URL in the registration process in Listing 4.4. The code to generate the random string can be seen in Listing 4.35. After creating the GUID we must replace the "/" and "+" characters. The "/" must be replaces otherwise it would be interpreted as a part of the path of a user URL and the "+" must be replaced else it would be interpreted as a space and the user URL could not be interpreted correctly. The string created by the GUID structure always ends with two "=", therefore we remove these characters, since they do not increase the number of different user URLs. Altogether the *GetRandomString* function returns a string with 22 characters. With all upper and lower-case characters, all ten digits, and the two special characters "-" and "_" we get $(26 + 26 + 10 + 2)^{22} = 5.44 \cdot 10^{39}$ different user URLs. Even if all the roughly 8 billion people currently on earth would have an account on this application the chance that a new user URL would collide with one of the others is extremely small with a probability of just $1.46 \cdot 10^{-30}$. But it still could happen and therefore, we check if the new generated user URL is already in our system with the *GetUniqueRandomUserUrl* function. An attacker could always guess all user URLs just by

try and error. To slow the attacker, we could count the accesses from an IP address and block the address when the accesses exceed a threshold in a certain amount of time. If the attacker knows this mechanism the attacker could still use a botnet to test all possible user URLs. This would result in higher costs for our server since the server must store all the blocked IP addresses and could result in users not be able to access the service, because they were unwittingly part of the botnet. The costs outweigh the possible privacy benefits and because of that we do not want to implement such a feature. Nonetheless it is still like searching for the needle in a haystack, since the number of possible user URLs is so huge. An easy method to make this more difficult we deliver for every possible user URL, the same empty page except if the user explicitly entered a public emergency message. This is the only indicator that a user URL is used by a person. We do not expect that every user uses this feature and therefore some empty pages belong to real users. Then again if the attacker could get access to an account, which can read the private notices, the attacker could probably find more valid user URLs, since the user is more likely to enter some private note that can only be seen by authorized personal rather than the public note, which can be read by everybody. Furthermore, we plan to implement that the user can change the user URL and therefore the QR Code every time. This would be helpful in case the user lost their QR Code and want to prevent that anybody not authorized could read their data.

```
public static string GetRandomString()
    {
      Guid guid = Guid.NewGuid();
        return Convert.ToBase64String(guid.ToByteArray()).Replace("/", "-").
    Replace("+", "_").Replace("=", "");
    }
```

Listing 4.35: Create a random string

An additional feature could be to limit the accesses of users with roles that can see other users files. This would it make harder for an attacker with access to an account that can read other user's files to guess user URLs. It would slow down an attacker making it easier to detect them and react with counter measurements.

Beierle et. al. [BTA+20] developed a privacy model for mobile data applications to protect the user's privacy as much as possible. They developed an android app to collect "smartphone sensor and usage data as well as applies standardized psychological questionnaires to the user." for their survey about personality traits and what permissions the user give the application. With their app they also tested their privacy model which consists of nine measures. One measure is asking the user for consent and explaining what data is collected and how the data is used. This is typically done by agreeing to a privacy policy, which is missing in our application and should be added. But they also found that the users in average only spend 10 seconds "reading" the privacy policy. The second measure is to let the users view their own data which is collected by the system. This is already implemented in our system since everybody can see all files assigned to their profile. If a doctor creates a new entry for the user, the user can see the entry. The next measure is an opt-out option for the user from the application. In our application the user can easily delete their account. When deleting the account all files stored for the user are also deleted which leaves no personal information about the user in our database. Another measure is the utilization of the permission system of the smartphone operating system. The only permission a smartphone app implementing our

solution would need is access to the camera to scan the QR Code of a patient. The last of their proposed measures that can be implemented in our system is secured transfer of data. Our current application uses HTTP/2 and TLS, therefore the traffic between the client and server is encrypted and the highly private information of users are protected in transfer. In a smartphone app the traffic should also be encrypted. The remaining three measures can not be implemented in a smartphone app or in our current application because they would make our application unusable. One of these three measures are random identifiers which should prevent linking the data with personal details. But linking the entries of a user to the user's personal details is exactly what we must do. Otherwise the medical personal could never find the patients files in the system. The second measure we cannot implement is data anonymization, where the compiled data is anonymized and then send to the server. This cannot be done for the same reason as the previous measure. The last of these measures is the ability to identify individual users without linking their collected data. Since in our application the user is linked with their data this measure cannot be implemented too.

One way to increase the trust in the software would be to make it open source. Then everybody could see what the application does and what not with the data stored. Furthermore privacy and security experts could check the software to find vulnerabilities, which otherwise would not be found and could be exploited by an attacker. As a result, the application would become more secure and private.

5
Look and feel

Following the implementation, we now want to look how the website is presented to the user.

Figure 5.1 shows the index page if the user is not logged in.

The log in page is shown in figure 5.2. Besides the email and password fields the user can choose that the browser remembers the log in so that the user does not have to log back in when visiting the page again. In case the user forgot their password, they can reset it here to as well as resending their email confirmation since a confirmed email is required to log in. If the user does not have an account yet and want to register themselves, they get redirected to the register page shown in figure 5.3 and figure 5.4.

After the user logged in the index pages looks like figure 5.5.

In case the user is logged in as a doctor they can see the toolbar shown in figure 5.6a. Otherwise they see the toolbar shown in figure 5.6b.

Only doctors can access the abbreviation page shown in figure 5.7. The figure shows the paginated list of abbreviation entries of the doctor with the ability to search for abbreviations or ICD Codes and sort for either the abbreviation or the code.

Figure 5.8 shows the allergy index page when called by the patient in figure 5.8a and by the doctor in figure 5.8b. Both parts show the color and shape coding for the severity of the reaction to an allergy.

Figure 5.9 shows all the CRUD operation pages for allergies. The create page shows a drop-down list with the three possible reactions as well as the what you see is what you get editor used in our application. The edit page shows the abilities to format text with this editor and both the details and the delete page show how this is displayed.

The figure 5.10 shows both datalists for diseases. Figure 5.10a a shows the list for the ICD Codes. By including both the name of the disease and the ICD Code the user can search for the right code by entering the code or the name. Figure 5.10b shows the list for the names and the abbreviations. Both the abbreviations and the names are sorted alphabetically. The abbreviations are displayed before the names.

Figure 5.11 shows the index page for medication with the QR Code encoding all current medication.

Figure 5.12 shows the same index page for smaller screens which displays far less information to prevent overflowing. The figure also shows that the toolbar collapsed to a burger menu to save space.

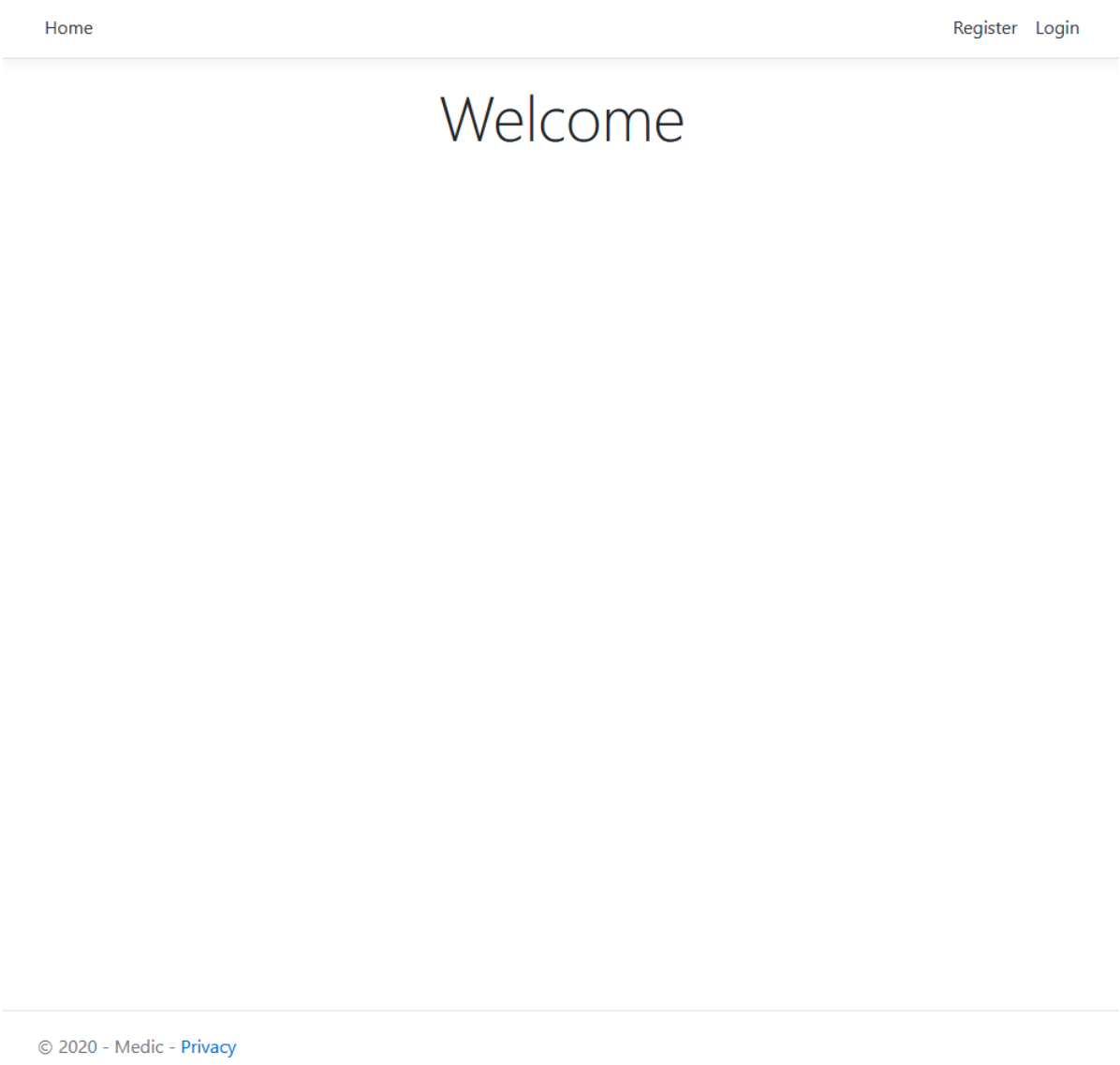


Figure 5.1: Welcome screen

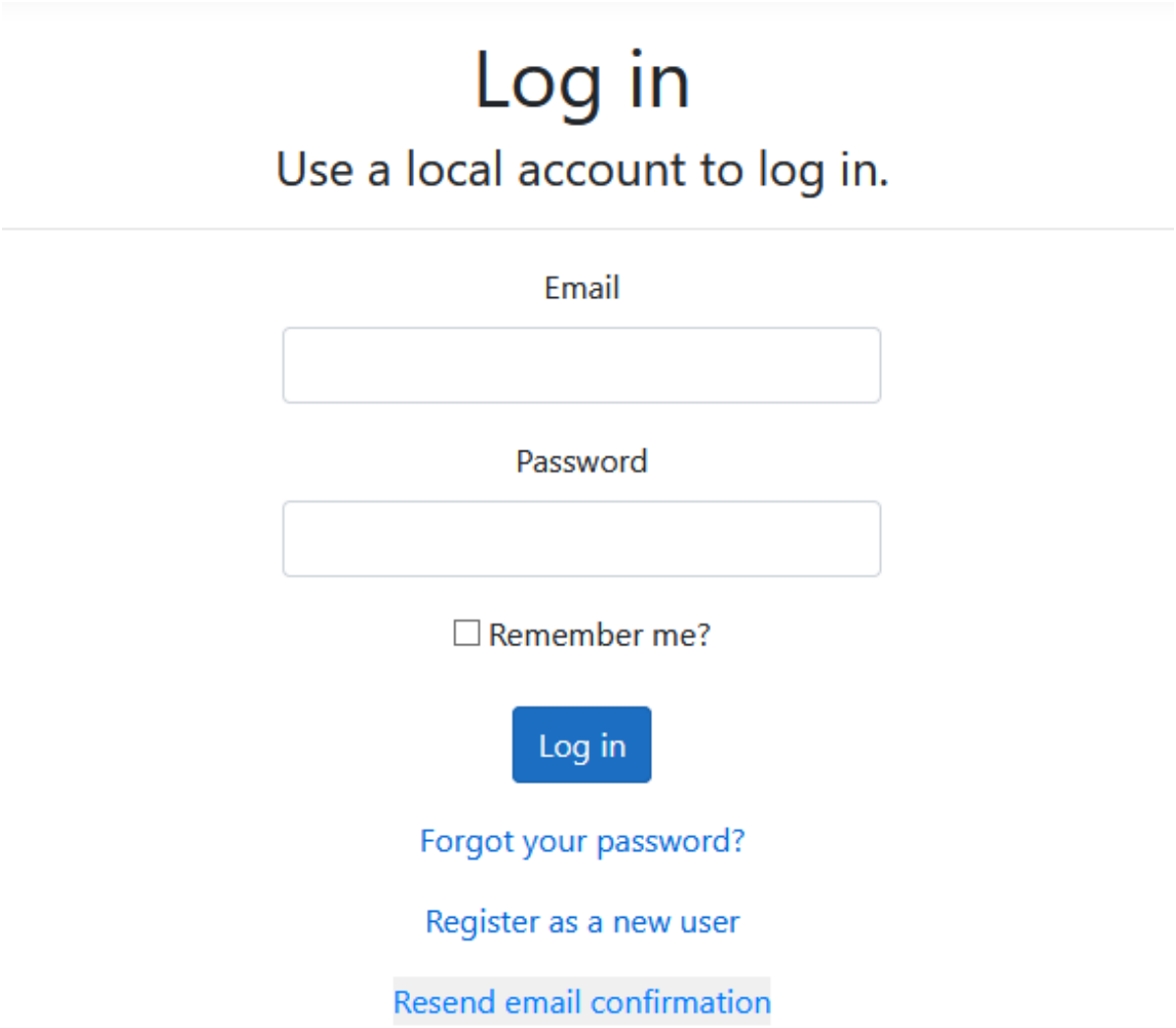


Figure 5.2: Log in screen

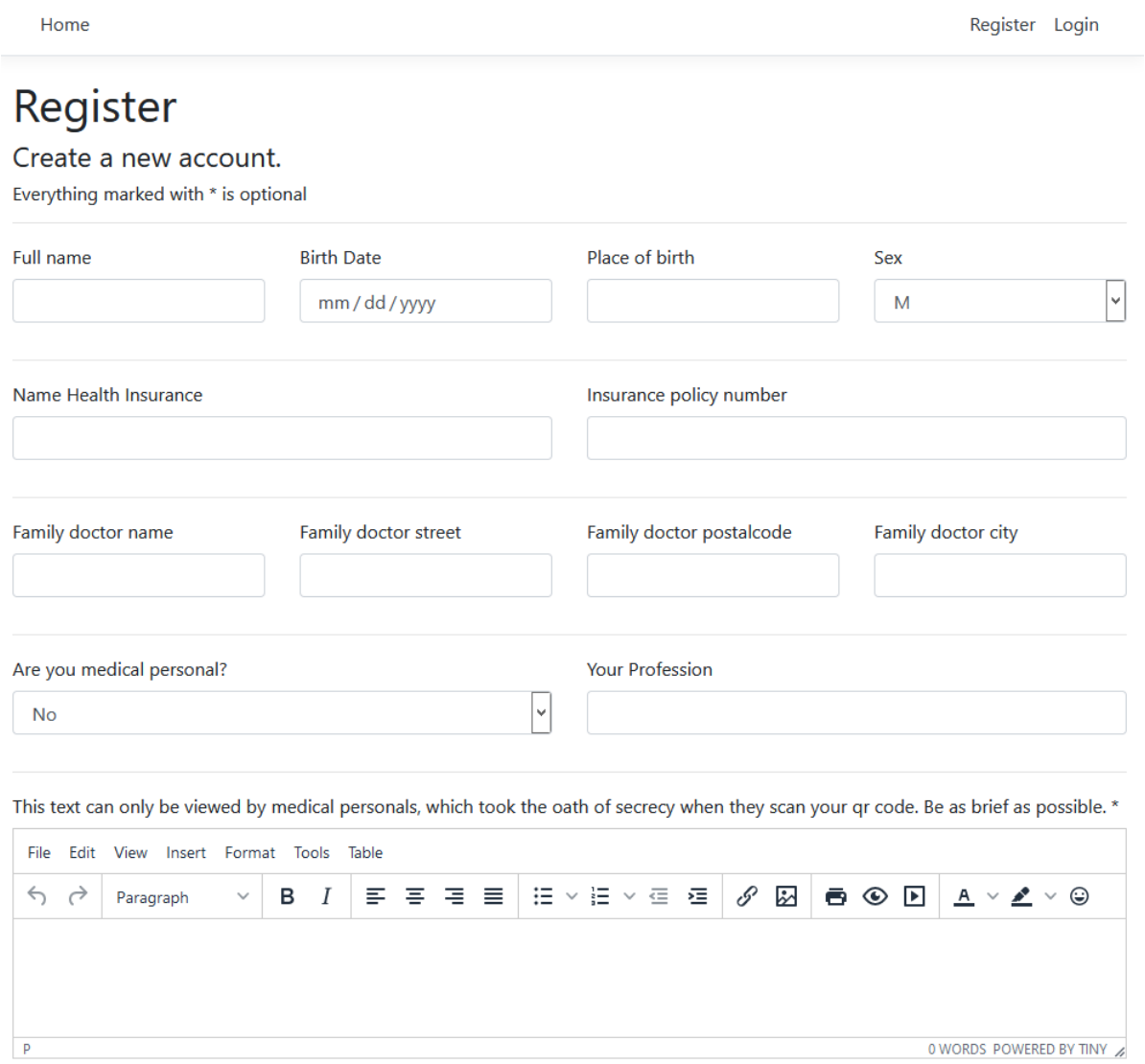


Figure 5.3: Top of the register page

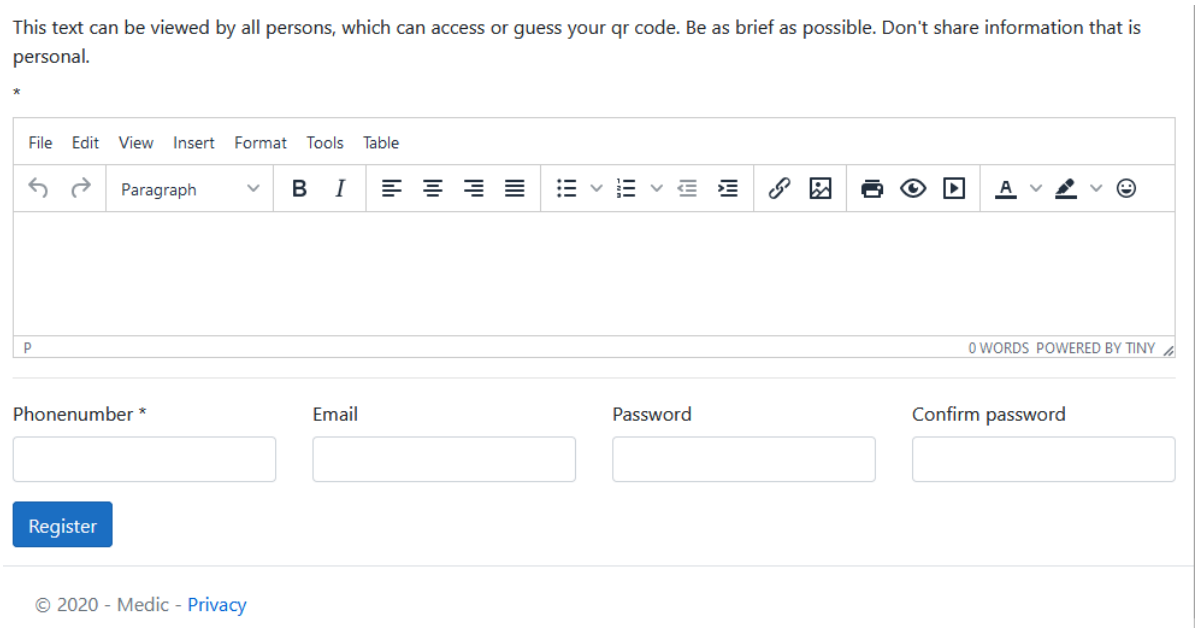


Figure 5.4: Bottom of the register page

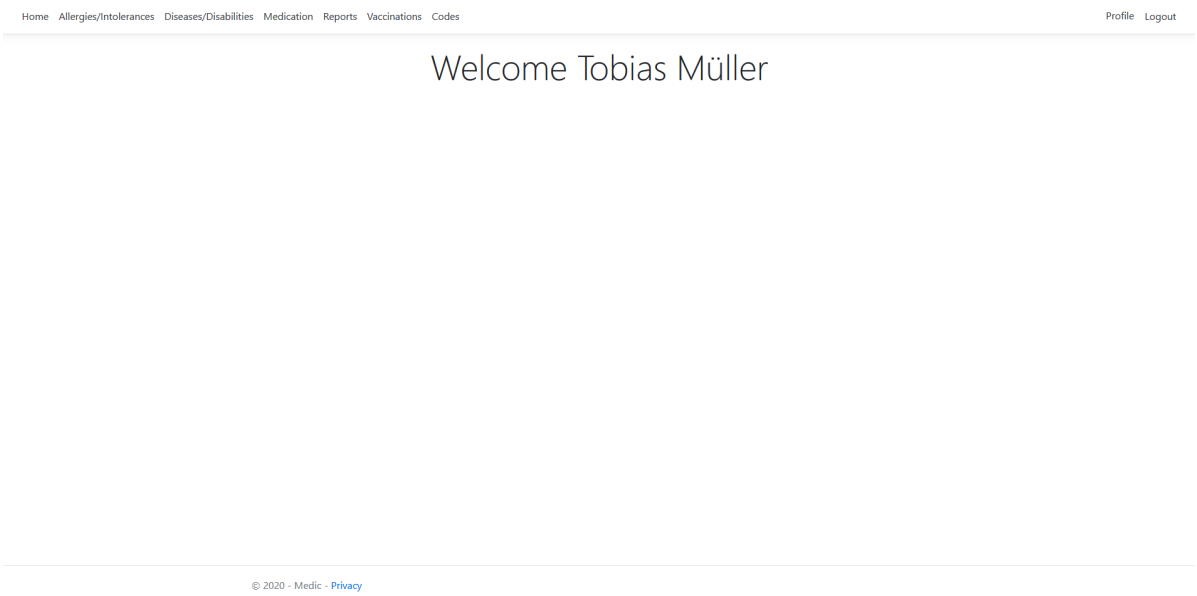
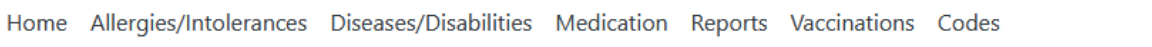
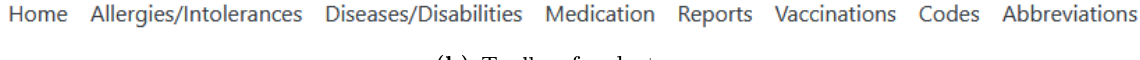


Figure 5.5: Welcome screen when logged in



(a) Toolbar for normal user



(b) Toolbar for doctors

Figure 5.6: The toolbar for different roles

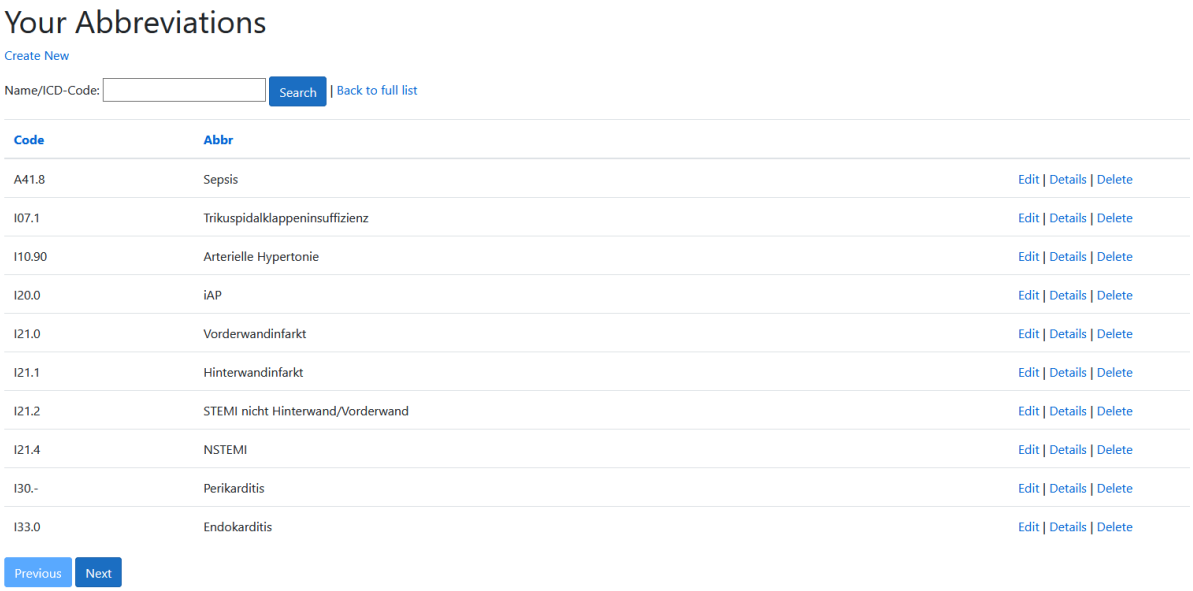
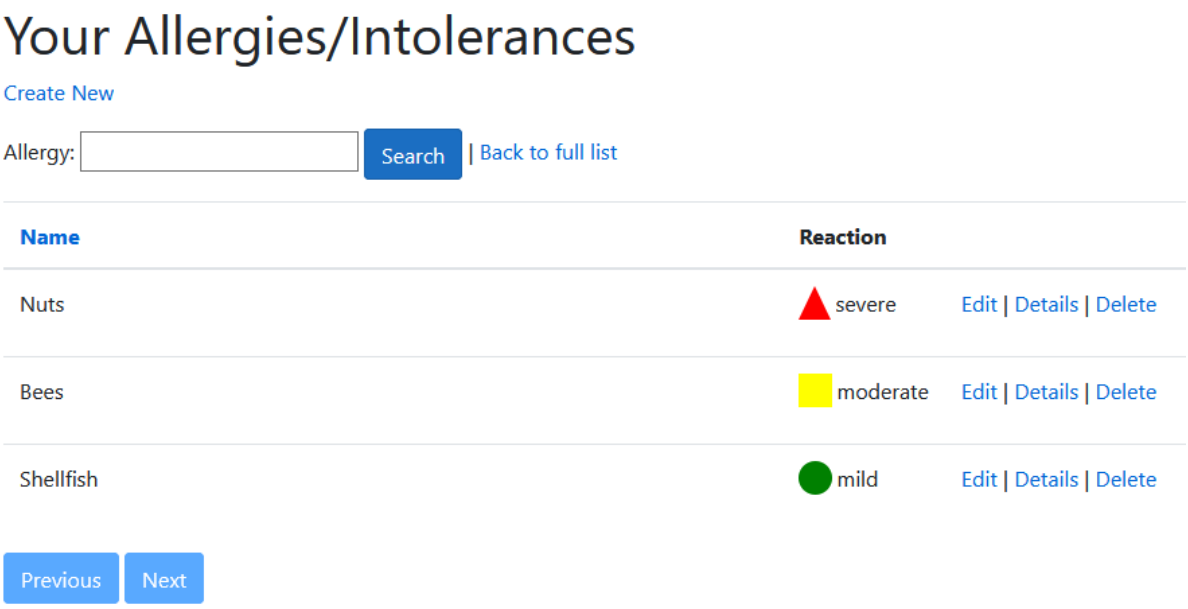


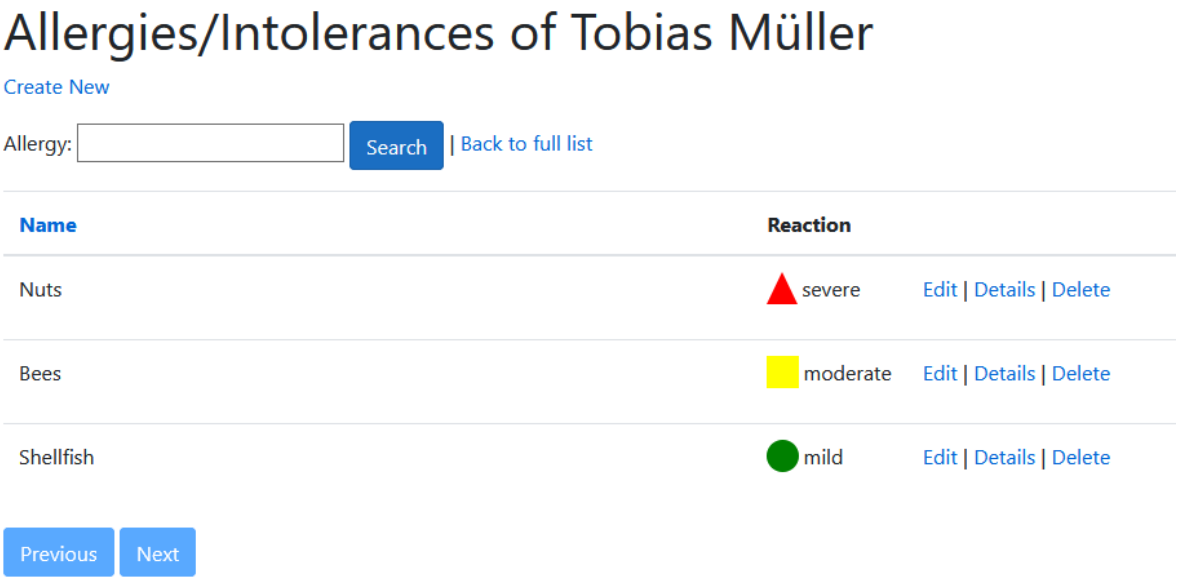
Figure 5.7: Abbreviations page for doctors

Figure 5.13 shows the edit page for medication on a large screen. Here the fields have enough space that four of them can fit in one row.

Figure 5.14 shows the same edit page but for smaller screens. Here every field has their own row to increase usability on smaller screens.

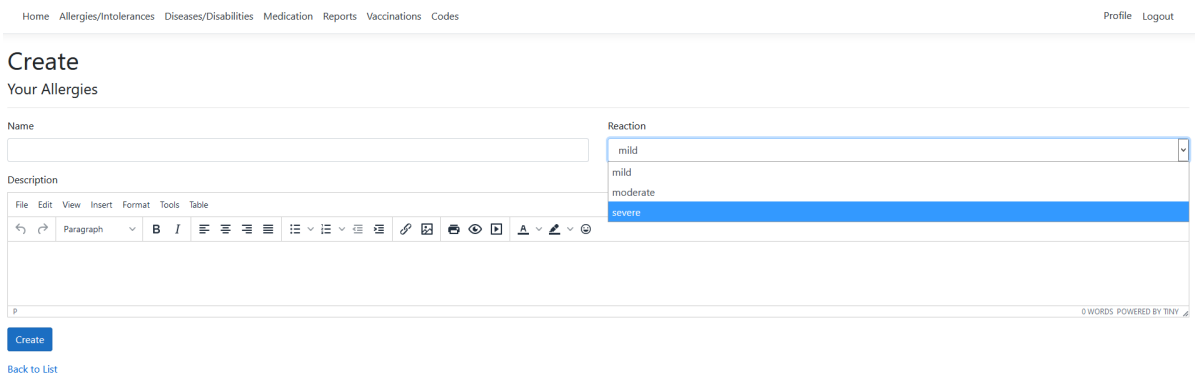


(a) Allergy index page

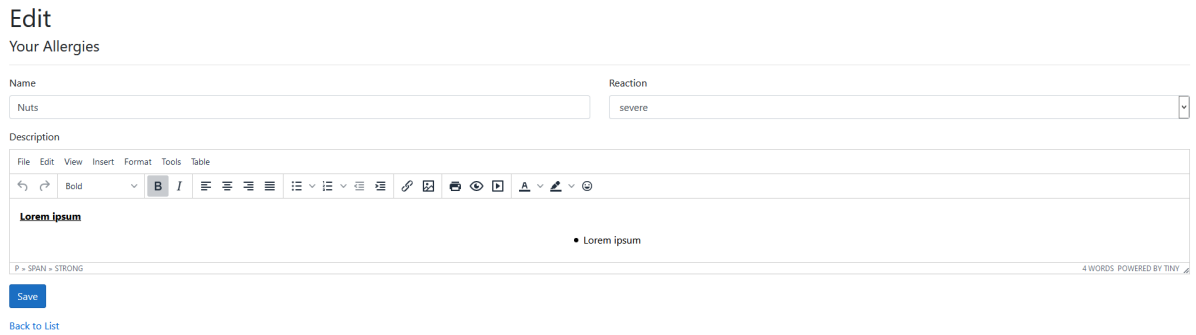


(b) Allergy index page of a patient viewed by a doctor

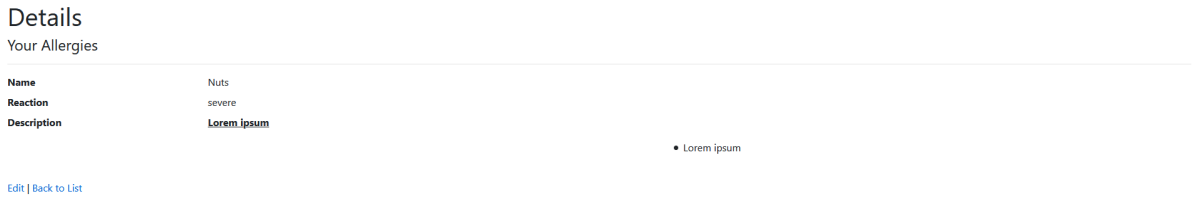
Figure 5.8: Allergy index pages



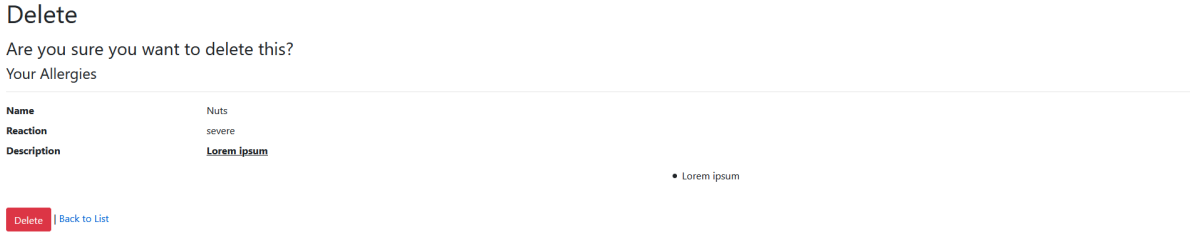
(a) Allergy create page



(b) Allergy edit page

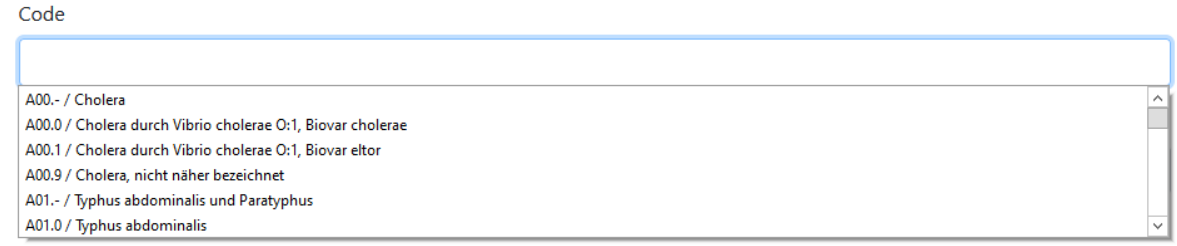


(c) Allergy details page

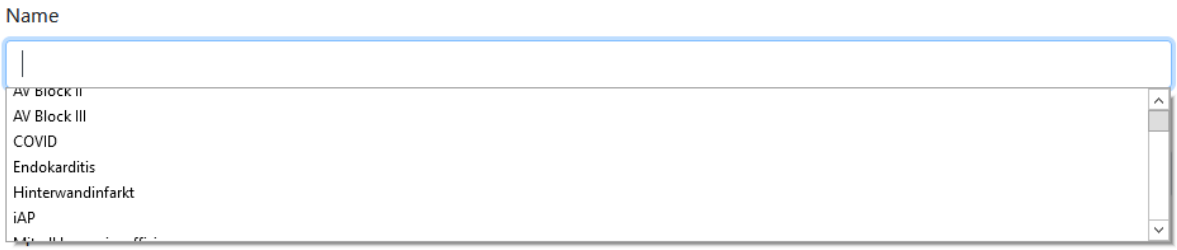


(d) Allergy delete page

Figure 5.9: All allergy CRUD pages



(a) Datalist for the ICD Codes



(b) Datalist for abbreviations and disease names

Figure 5.10: Datalists for diseases

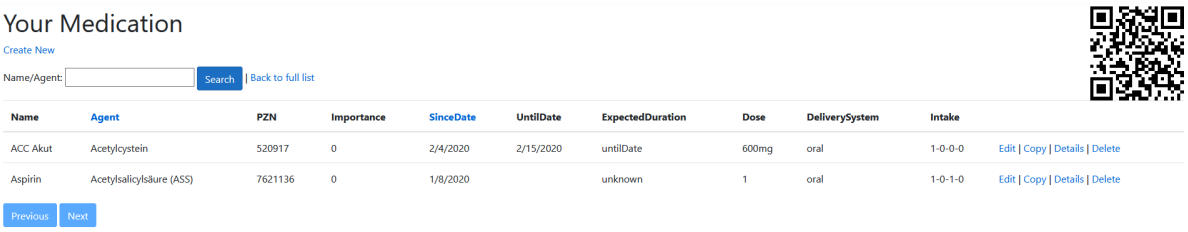


Figure 5.11: Medication index page

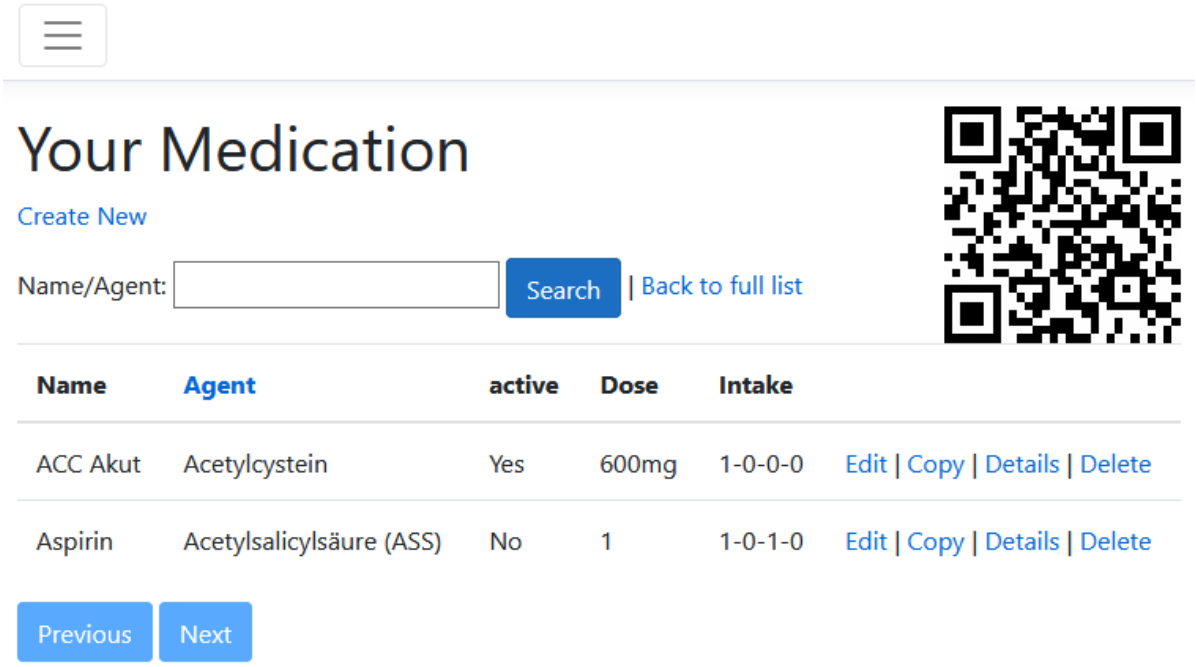


Figure 5.12: Medication index page for smaller screens

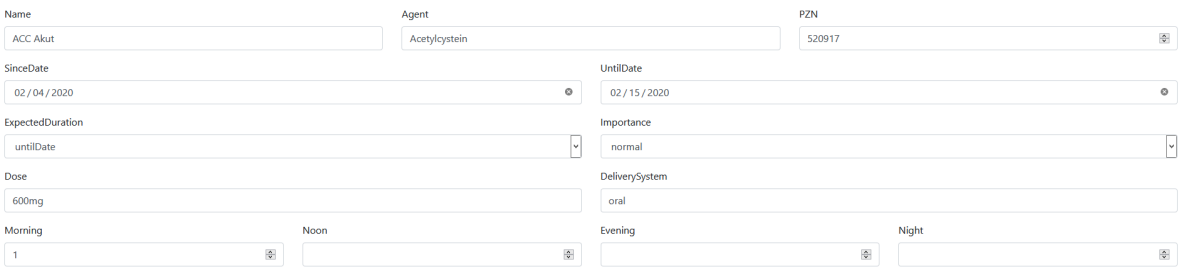


Figure 5.13: Medication edit page for computer screens

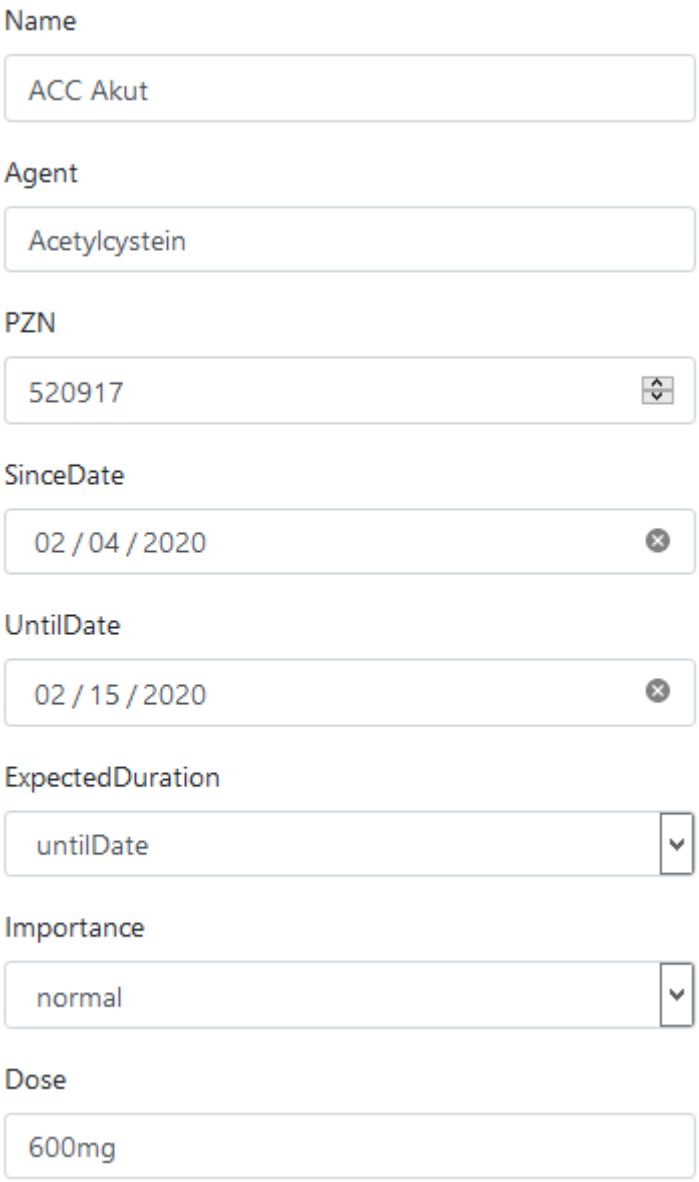


Figure 5.14: Medication edit page for smaller screens

6
Use Case

In this chapter we want to discuss two different use cases of our application and evaluate changes that could improve the application. All cases require that the persons involved use the application.

The first use case is an emergency. The injured person could carry their QR Code on a bracelet, as a wallpaper on their mobile phone, or printed on a card. After the first responder arrived at the injured person and calling for help, the first responder can search for the QR Code and scan it. After scanning the code, the website opens and shows the emergency message of the injured person, if the person entered something. In case the injured person entered something the first responder can use the information provided to take better care of the injured. After the paramedics arrived, they also scan the QR Code of the injured person, now patient. They also view the emergency message the first responder has seen and the private emergency message. This provides the paramedics more and more detailed information about the patient. Then the paramedics can go to the allergy tab to check if the patient has some allergies that could interfere the treatment. Furthermore, the paramedic could check the patients reports tab to see if the patient had an operation, which could be one for the emergency. In the medication tab they can check for medication the patient is taking or did take recently. With this information the paramedics can avoid giving drugs that would result in a negative interdependencies with the medication taken by the patient. The paramedics can also check the disease tab to find diseases the patient might have. The patient could have haemophilia, a disease that prevents the body of a person to make blood clots to stop bleeding. With this information the paramedics could quickly tend the patients bleeding wounds to prevent worse. The more information the paramedics have, the better they can tend the patient. If the patient needs to be transported to the hospital, the hospital staff there can scan the QR Code again to view the full medical history of the patient. The doctor treating the patient can easily access all files of the patient and can add new entries to the patient's files. Therefore, the next doctor can see this entry in the patient's files.

The second use case is a patient which has to see a different doctor, because they are referred to a specialized doctor by their family doctor, moved to a different city, or had to go to the emergency room. With our application the patient must not remember all their diseases, medications, or accidents. They can show the doctor their QR Code, which the doctor scans to access all files of the patient. In this case the patient cannot forget to mention a disease they have, a medication they take, or reports from doctors. Furthermore, the doctor can decide what is important for them to know about the patient. In case the patient was referred by another doctor the current doctor can see what the last doctor did and therefore not do it again. This saves the doctor time and resources.

While testing the application we ran into some inconveniences that should be fixed before the software is released. The first problem are the emergency messages. For paramedics, the emergency page should display way more information, so that the paramedic does not have to cycle through all files to search for important files. Therefore, the emergency page should display allergies, current medication, and important diseases on top of the page. This can then be followed by the private and the public notice. Furthermore, every time the emergency page gets loaded a unique random id should be generated. This id could then be provided by the first responder to the person in the emergency call center. With this id the call center agent could look up the person having an accident and send the user URL to the paramedic team. This enables the paramedic team to access the patient's files before they arrive at the patient. It would work like the e-Ambulance system in Estonia discussed in the section about *Countrywide solution*. In addition, the user should decide which page opens when they scan the QR Code with the user URL. Therewith a doctor who is more interested in the reports does not see the emergency page every time they scan a new code. To increase the usability more languages should be supported. On the create page for vaccination the system should not fill out the fields for the doctor with the patient's family doctor credentials. The user could get the vaccine by a doctor which is not the user's family doctor. For this case it would be better if the system takes the information from the doctor entering a new vaccine. But before we can do this, we must add the possibility for a doctor to add this information to their profile. In the reports the subject should be renamed to organ, to allow better sorting and filtering for doctors. A cardiologist for example is mostly interested in reports regarding the heart. To make accessing the files of a patient easier the system should store the last accessed patients of a doctor. Hence the doctor must not scan the QR Code of the patient every time they want to access the files. This could be provided in the future with the missing logging feature. To create more transparency for the user a list with all the persons who accessed the user's files should be provided. Furthermore, on all edit pages should be a timer indicating how much time is left before the session ends and the changes must be saved. To make the creating of the abbreviations easier for doctors there should be a preset with common diseases often used by the doctor of a medical specialty. This could either integrated in the registration process or the authorization process. We should also implement text templates for easier and faster creation or editing a file, like in the solution by Pryss et. al. [PMLR15]. The last feature that should be reconsidered are the roles and their authorization. A user should not be able to edit or delete their own files that were created by a doctor, which would sophisticate the patients medical history. It could be possible to append information to an entry, to provide additional information like a medical journal. Furthermore nurses and paramedics should get the authorization to add health parameters to the patient's files like in the proposed application by Mersini et al. [MST13].

7
Conclusion

With these improvements our application can solve the problems introduced in the *Introduction*. Our application provides an easy and fast access to a patient's medical files for the entire hospital staff. Because we use QR Codes to access these files every staff member can access these files conveniently with their smartphone. Furthermore, all files are digital, therefore misreading because of bad handwriting is a thing of the past. But in software solution, like ours, typing errors and transposed digits could be entered. While the former is rather uncritical the latter mistake can have serious consequences for instance with the medication dosage. With easing the administrative tasks doctors and the medical staff must do the personnel saves time. The saved time can be used for tending patients and / or to reduce the extra hours many currently must do. Furthermore, the personnel expenses of the hospital and therefore for the healthcare system can be lowered or stopped from rising. As Pryss et. al. [PMLR15] have shown medical staff wants to use such an application if and only if the application is easier and faster to use than the current paper-based system. With our application patients can view their files every time they want, making the healthcare system more transparent for patients. For patients it would be easier to change doctor too, either because of moving or by referral to a specialized doctor. To complete the application the e-Prescription from Estonia could be integrated as well. Gaining the trust of the users is crucial as we have seen with the corona tracing app of the RKI. If users trust the system and see the added value, they will use the application. If nobody wants to use the application, because of privacy concerns, the benefits of the proposed application would be minimally. Therefore, the application should be open source.

Bibliography

[Alb12] Richard Albrecht. *DataMatrix - Mein Produkt bekommt eine Identität.* Unglaube Identech, 2012. accessed 2020-05-31.

[alt] Altersaufbau der Bevölkerung Deutschlands. https://www.destatis.de/DE/Service/Statistik-Visualisiert/bevoelkerungspyramide-d.html. accessed 2020-05-16.

[Ano18] Anonymous. Communication on enabling the digital transformation of health and care in the Digital Single Market; empowering citizens and building a healthier society. https://ec.europa.eu/digital-single-market/en/news/communication-enabling-digital-transformation-health-and-care-digital-single-market-empowering, April 2018. accessed 2020-05-28.

[arb] Ein Drittel der Ärztinnen und Ärzte arbeitete 2018 mehr als 48 Stunden pro Woche. https://www.destatis.de/DE/Presse/Pressemitteilungen/2020/04/PD20_N019_231.html. accessed 2020-05-17.

[bre] Machen Sie jetzt Ihre Breitbandmessung! https://breitbandmessung.de/. accessed 2020-05-23.

[BTA+20] Felix Beierle, Vinh Thuy Tran, Mathias Allemand, Patrick Neff, Winfried Schlee, Thomas Probst, Johannes Zimmermann, and Rüdiger Pryss. What data are smartphone users willing to share with researchers? *Journal of Ambient Intelligence and Humanized Computing*, 11(6):2277–2289, June 2020.

[cova] Information about Coronavirus disease COVID-19 | Government installation profile. https://www.terviseamet.ee/en/covid19. accessed 2020-05-21.

[covb] RKI - Coronavirus SARS-CoV-2 - COVID-19: Fallzahlen in Deutschland und weltweit. https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Fallzahlen.html. 2020-05-21.

[e-e] Healthcare. https://e-estonia.com/solutions/healthcare/. accessed 2020-07-13.

[EG14] Rainer Erices and Antje Gumz. Ddr-gesundheitswesen: Die versorgungslage war überaus kritisch. *Deutsches Ärzteblatt*, 111:A–348 / B, 03 2014.

[fre] Freie Arztwahl. https://www.bundesgesundheitsministerium.de/themen/krankenversicherung/grundprinzipien/freie-arztwahl.html. accessed 2020-05-27.

[FWL+06] X. Fang, F. Wu, B. Luo, H. Zhao, and P. Wang. Automatic recognition of noisy code-39 barcode. In *16th International Conference on Artificial Reality and Telexistence–Workshops (ICAT'06)*, pages 79–82, 2006.

[Hur91] J. W. Hurst. Reform of health care in Germany. *Health Care Financing Review*, 12(3):73–86, 1991.

[icda] ICD-10-GM. https://www.dimdi.de/dynamic/de/klassifikationen/icd/icd-10-gm/. accessed 2020-05-29.

[icdb] ILCD bis ICD-10. https://www.dimdi.de/dynamic/de/klassifikationen/icd/icd-10-who/historie/ilcd-bis-icd-10/. accessed 2020-05-29.

[ifa] IFA GmbH, Informationsstelle für Arzneispezialitäten. https://www.ifaffm.de/de/ifa-codingsystem.html. accessed 2020-05-29.

[kos] Kostennachweis der Krankenhäuser - Fachserie 12 Reihe 6.3 - 2017. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Krankenhaeuser/Publikationen/Downloads-Krankenhaeuser/kostennachweis-krankenhaeuser-2120630177004.html. accessed 2020-05-17.

[kra] Krankenhäuser. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Krankenhaeuser/_inhalt.html. accessed 2020-05-17.

[LL13] Daw-Tung Lin and Chin-Lin Lin. Automatic location for multi-symbology and multiple 1d and 2d barcodes. *Journal of Marine Science and Technology*, 21(6):663–668, 2013.

[mera] Mercedes-Benz Unfall- & Pannenhilfe. https://www.mercedes-benz.de/passengercars/service-finance/roadside-and-accident-assistance.html. accessed 2020-05-23.

[merb] The rescue sticker from Mercedes-Benz. https://www.mercedes-benz.com/en/vehicles/service-parts/the-rescue-sticker-from-mercedes-benz/. accessed 2020-05-23.

[MMB02] Elias Mossialos, Martin McKee, and Rita Baeten, editors. *The impact of EU law on health care systems*. Number no. 39 in Work & society. P.I.E.-Peter Lang, Bruxelles ; New York, 2002.

[MST13] P. Mersini, E. Sakkopoulos, and A. Tsakalidis. Appification of hospital healthcare and data management using qrcodes. In *IISA 2013*, pages 1–6, 2013.

[ope] Operationen und Prozeduren der vollstationären Patientinnen und Patienten in Krankenhäusern (4-Steller) - 2018. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Krankenhaeuser/Publikationen/Downloads-Krankenhaeuser/operationen-prozeduren-5231401187014.html. accessed 2020-05-16.

[owa] OWASP Top Ten Web Application Security Risks | OWASP. https://owasp.org/www-project-top-ten/. accessed 2020-06-20.

[pat] Einrichtungen, Betten und Patientenbewegung. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Krankenhaeuser/Tabellen/gd-krankenhaeuser-jahre.html. accessed 2020-05-16.

[per] Ärztliches und nichtärztliches Personal in Krankenhäusern. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Krankenhaeuser/Tabellen/personal-krankenhaeuser-jahre.html. accessed 2020-05-16.

[PL16] Kaja Polluste and Margus Lember. Primary health care in estonia. *Family Medicine & Primary Care Review*, (1):74–77, 2016.

[PMLR15] Rüdiger Pryss, Nicolas Mundbrod, David Langer, and Manfred Reichert. Supporting medical ward rounds through mobile task and process management. *Information Systems and e-Business Management*, 13(1):107–146, February 2015.

[popa] Bevölkerungsstand. https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Bevoelkerungsstand/Tabellen/zensus-geschlecht-staatsangehoerigkeit-2019.html. accessed 2020-05-21.

[popb] Population at beginning of year - Statistics Estonia. https://www.stat.ee/stat-population-at-beginning-of-year. accessed 2020-05-21.

[qrc] QRcode.com|DENSO WAVE. https://www.qrcode.com/en/. accessed 2020-05-31.

[RAa] Rick-Anderson. Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging. https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/sort-filter-page. accessed 2020-07-13.

[RAb] Rick-Anderson. Prevent Cross-Site Scripting (XSS) in ASP.NET Core. https://docs.microsoft.com/en-us/aspnet/core/security/cross-site-scripting. accessed 2020-06-20.

[SFMA+13] Ramon Sabes-Figuera, Ioannis Maghiros, Fabienne Abadie, Europäische Kommission, and Gemeinsame Forschungsstelle. *European hospital survey. ... ...* Publ. Off. of the Europ. Union, Luxembourg, 2013. OCLC: 931535822, accessed 2020-05-23.

[sos] SOS App | Get Help and Be Safe | Emergency App. http://www.sos-qr.com/. accessed 2020-05-23.

[ste] stevestein. Frequently Asked Questions - ADO.NET. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/frequently-asked-questions. accessed 2020-06-20.

[tin] The Most Advanced WYSIWYG HTML Editor | Trusted Rich Text Editor. https://www.tiny.cloud/. accessed 2020-07-13.

[whoa] WHO | International Classification of Diseases, 11th Revision (ICD-11). http://www.who.int/classifications/icd/en/. Publisher: World Health Organization, accessed 2020-05-29.

[whob] WHO | International Classification of Diseases (ICD) Revision. http://www.who.int/classifications/icd/revision/icd11faq/en/. Publisher: World Health Organization, accessed 2020-05-29.