Technische Hochschule Ingolstadt

Development of a mobile application for the algorithmic attribution of symptoms to potential diseases

BACHELOR THESIS

Angelina Petzold
Immatriculation Number: 00108359

First Examiner	Prof. Dr. Sebastian Apel
Second Examiner	Prof. Dr. Marc Aubreville
Start Date	October 11, 2022
Submission Date	March 11, 2023

ABSTRACT

ACKNOWLEDGEMENTS

Mein hauptsächlicher Dank gilt meinem Betreuer Professor Doktor Sebastion Apel für seine hilfreichen Anregungen während des gesamten Betreuungszeitraumes. Die Freiheit, die er mir bei der Wahl des Themas gelassen hat, war nicht selbstverständlich.

Auch die Teilnehmerinnen und Teilnehmer meiner Befragung haben durch ihre Auskunftbereitschaft und interessanten Beiträge meine Bachelorarbeit wesentlich mitgeprägt.

Letztlich richte ich auch ein Dankeschön an meine Kommilitonin Frau Jenny Hofbauer für das ausführliche Korrekturlesen und dem Mut machen während der gesamten Arbeit.

My main thanks go to my supervisor Professor Doctor Sebastion Apel for his helpful suggestions during the whole period of supervision. The freedom he gave me in choosing the topic was not self-evident.

The participants in my survey also played a major role in shaping my bachelor thesis through their willingness to provide information and interesting contributions.

Finally, I would also like to thank my fellow student Ms. Jenny Hofbauer for her detailed proofreading and encouragement throughout the thesis.

Contents

Acronyms

BAEK German Medical Association (Bundesärztekammer)

AI Artifical Intelligence

API Application Programming Interface

SDK Software Development Kit

OOP Object Orientated Programming

VM Virtual Machine

PDF Portable Document Format

SRP Single Responsibility Principle

OCP Open-Closed Principle

LSP Liskov Substitution Principle

ISP Interface Segregation Principle

SQL Structured Query Language

REST Representational State Transfer

RPC Remote Procedure Call

DIP Dependeny Inversion Principle

BaaS Backend as a Service

UML Unified Modeling Language

NHS National Health Service

HTTP Hypter Text Transfer Protocol

JSON JavaScript Object Notation

1. Introduction

1.1. The Bachelor's Thesis Problem

People in Germany are becoming more interested in physical and mental health matters. This is most likely attributed to the COVID-19 pandemic that has been circulating in recent years [1]. Along with positive outcomes, such as increased care for fellow citizens [1] and greater awareness of health issues, the consistent growth of interest in health issues also is causing problems. With an increasing number of anxious and concerned patients, medical practices and general practitioners have long since exceeded their capacity limits and have reached their breaking point [2]. Patients also notice this: Overcrowded waiting rooms, long waiting periods, and nerve-racking telephone loops are becoming the norm for doctor visits. The German Medical Association (BAEK), draws attention to a second issue: as society ages, so does the medical industry. Every fifth doctor is about to retire. More than 13% of doctors are between the ages of 60 and 65, while another 8.5% are over the age of 65. Over the following few years, this will exacerbate clinics' and offices' already stressful staffing situation [3]. The bachelor's thesis problem can be traced back to the preceding situation. The population is fearful, mainly caused by the COVID-19 pandemic, and doctors are reaching their limits. The resulting problems are of great importance. General practitioners are forced to order patient stops and issue access bans [2]. This also means that patients needing immediate medical attention may be turned away, and medical care may be denied. In addition to the concerned patients, the number of seriously (COVID-19) ill people has steadily increased. There have been approximately 146,000 deaths in Germany since the start of the pandemic (as of August 19, 2022). [4] As part of this work, a survey was launched to highlight the problem in more detail. The results indicate that around 80% of those questioned have put off a visit to the doctor in recent years, even though they have suffered from symptoms.

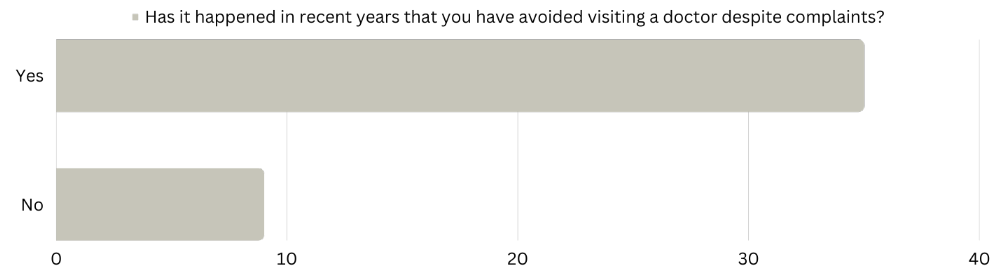


Figure 1.1.: Survey Question - Avoiding to visit the doctor

Another question in this survey asked respondents to list the justifications for delaying these doctor visits or the reasons they might consider forgoing a visit to the doctor. Those reasons range from long waiting times to difficulties scheduling an appointment. Figure 1.2 shows the mentioned distribution of the answers. All questions of the survey, together with the answers, can be found in appendix A.

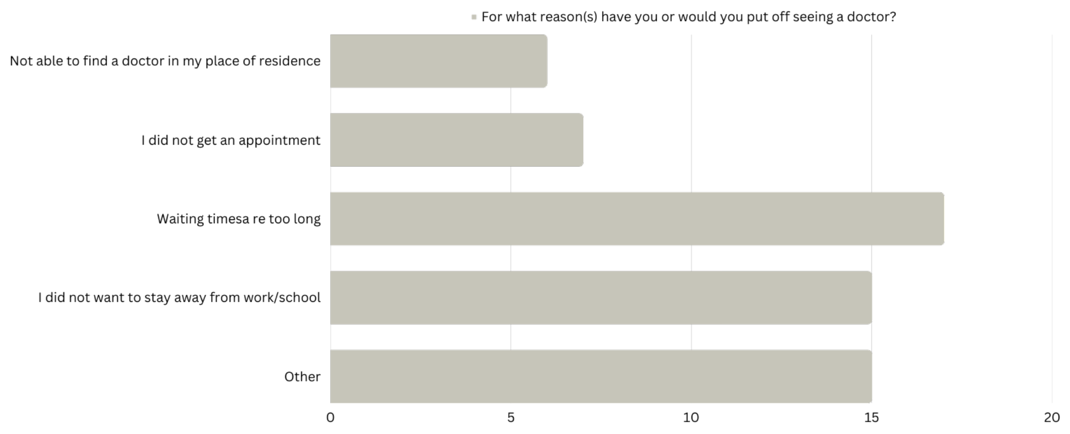


Figure 1.2.: Survey Question - Reasons of avoiding to visit the doctor

1.2. Motivation

The mentioned problem imposes the question of addressing patients' concerns while relieving the burden on doctors. Digitalization provides a solution to this problem. Online consultation hours and appointment scheduling have recently helped relieve medical practices. Smartphones, in particular, are becoming an increasingly important part of our daily lives. The goal of this bachelor thesis is to provide a method for efficiently minimizing the problems mentioned above through the use of mobile applications. Such an application can advise a worried user and help alleviate their fears.

1.3. The Bachelor Thesis Goal

The goals of the bachelor thesis can be defined as follows:

- Requirements analysis for the application
- Design and conception of the application
- Creation of a firestore database
- Comparing different algorithmic disease assignment methods

The application should be able to generate a diagnosis based on a user's specified symptoms. This diagnosis is made after successful data gathering regarding the user's symptoms and a subsequent determination of the possible diseases. Another goal of the application is that doctors can expand the database, whereby a verification possibility must be provided to ensure that the person trying to log in is, in fact, a doctor. They should be provided with the possibility to add disease-related data or advice regarding diseases and illnesses for users. In general, this is to create a place for users to get information regarding their health status and, at the same time, get advice on how to improve it. The following research question can be defined here:

How can a mobile application for the algorithmic attribution of symptoms to potential diseases be designed and implemented?

2. Related Works

This chapter provides an overview of a selection of mobile applications comparable to the one in this work.

2.1. Mobile Applications

Ada

Ada is the most prevalent symptom-detection smartphone application. According to their statements, their users are currently around 12 million, while 28 million symptom analyses have already been completed [5]. The application is free in the PlayStore and the Apple Store. Disease detection in the Ada application is based on artificial intelligence (AI) developed by medical professionals of the Ada team. The user can provide personal data in their user profile, such as allergies and medication intake. A symptom analysis starts by asking the user what their worst symptom is. Based on the user's answer, the AI searches its medical lexicon for this symptom and asks a symptom-specific question. An example of this would be to ask the user if he had enough water today for the symptom headache. Ada uses a specially developed reasoning technology to assign symptoms. For this purpose, each symptom and each disease was assigned a joint probability, which makes it possible to calculate the overall probability of the disease for the specific symptom analysis [6]. After the successful diagnosis, the user can download the diagnosis in portable document format (PDF) format, while they are also saved internally. The Ada application also makes its collection of knowledge available to users in the form of a disease dictionary. Ada also offers an application programming interface (API), which enables healthcare organizations to integrate the AI chatbot into their application with the help of platform-specific SDKs (Software Development Kits)[7] [8].

Symptom Checker

The **Symptom Checker** is another currently available application. It was created using the computer language C# and the Unity platform [9]. Here, the user can select from a list of disease specifications, including those for diabetes tests, thrombosis, depression, hair loss, headache, and gastrointestinal infection. During the survey procedure, a doctor's interview is simulated, where the user is prompted with questions after choosing one of these categories, similar to the Ada application. The most likely condition is then diagnosed based on the patient's responses, and various treatment choices are provided. The application was made specifically for people without medical experience and was created by German doctors and medical experts. According to the developers, an algorithm that accesses a medical database produces the diagnosis [9]. Sadly, nothing more can be found about the algorithm and how it works.

Other Solutions

In addition to Ada and the Symptom Checker, other applications for symptom detection are available in the Play Store. However, these are less accurate than those mentioned. Some of these applications are only intended as a reference book for symptoms and diseases without a disease determination. In addition to mobile applications, some web applications and websites are available. However, this will not be discussed further in this work since only mobile applications are dealt with here.

2.2. Differentiation from other Systems

In contrast to both of the applications mentioned, the system's knowledge base should be expanded by doctors who are not part of the development team. Similar to the Diagnosis App, the application presented in this work will not request any personally identifiable information from users. An exception is a case when users wish to identify themselves as doctors. Unlike Ada, the diagnoses are not generated using AI but are calculated by an algorithm. As already mentioned, the calculation process of the Diagnosis app cannot be determined, and therefore no differentiation can be made from this application. With a realistic view, this application cannot work as accurately as artificial intelligence, which has been trained since 2011. This work should not replace Ada as the market leader, not the Diagnosis App, but describe the possible conception and implementation of such an application while comparing different calculation possibilities of the possible diseases based on the user symptoms.

3. Fundamentals

This section explains the basics needed to understand the rest of the work. Since the Flutter framework and the Dart programming language with which this application is to be developed are not topics that are dealt with in a computer science degree, they will also be discussed.

3.1. Flutter and Dart

The framework used to develop the disease-detection application will be Flutter, an open-source-UI-Kit developed by Google [10]. Flutter uses the open-source programming language Dart, which was designed for building Google Chrome browser applications and later benefited immensely from various improvements since it was released in 2011 [11]. The programming language consequently evolved from having much in common with JavaScript to sharing many features with C# and Java [12]. The Flutter and Dart ecosystem, brimming with open-source packages created by other developers worldwide, is one of the framework's best features. It enables programmers to quickly create visually stunning applications by including developers' packages worldwide. Also, Dart is a client-optimized general-purpose programming language that supports cross-platform development. This implies that this application will be created with a single code base yet run on Android- and iOS smartphones [13]. Furthermore, the program may be launched as a web application and utilized on embedded devices. However, as part of the bachelor thesis, the development and testing process will focus entirely on Android development. Dart is also a statically-typed language, meaning that each variable type must be explicitly declared, making it easier to catch bugs and other issues early on in the development process. With null safety, Dart ensures that variables cannot be assigned a null value unless they are explicitly declared nullable. This means that if a variable is expected to have a non-null value, it must be initialized with a non-null value, and any attempts to assign a null value to it will result in a compile-time error. This helps prevent null reference errors and makes it easier to write safe and predictable code [13].

3.1.1. Flutter: Everything is a Widget

When researching how Flutter functions, it is common to come across the phrase, "In Flutter, everything is a widget." The difference between Flutter's widgets and those in other Frameworks' components is that Flutter's widgets can specify how the application's user interface should appear. Eric Windmill was able to divide the widgets into various groups in his book Flutter in Action [14, p. 58]:

- **Layout:** Widgets of this category are able to store children-widgets, an example for such a widget would be a row, column or even a stack.
- **Structures:** As their name implies, structures aid in organizing the application. For instance, MenuDrawer produces a sidedrawer for the application, toasts display a message to the user, and buttons can respond to various click patterns.
- **Styles:** The developer can style widgets in almost any way using Flutter. With a tool like ButtonStyle, a button's background and foreground colors as well as its shape can all be changed.

- **Animations:** Flutter enables its users to breathe life into their applications with a rich palette of animation options. For instance, Flutter developers can use well-known animation features like curves, which are also used in CSS.
- **Positioning and Alignment:** Widgets such as Padding and Center allow it to position its child widget. There are also additional widgets, such as Positioned and Alignment, that allow the developer to position elements in a Stack.

The categorization created by Eric Windmill provides a decent overview of the possibilities in Flutter. There are undoubtedly a lot more widgets and a lot more usage categories that might be defined.

Widgets can be composed, meaning nested inside of one another [14, p. 61], so rather than simply returning the widget it describes, a widget's build method returns a tree of widgets. The Domain Object Model (DOM) in any web browser is comparable to this widget tree. A sample widget tree returned by a build method is shown in figure 2.1. The corresponding code snippet can be found in appendix B.

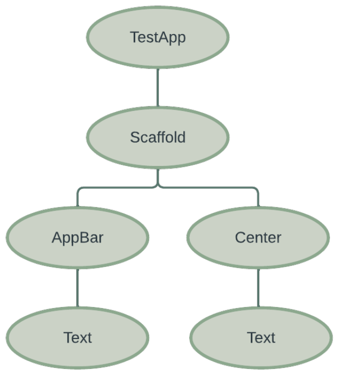


Figure 3.1.: Widget Tree of Appendix B

3.1.2. Flutter: Architectural Layers

Many different application architectural patterns can be used, including layered architecture. The Flutter framework is structured similarly to the familiar approach to project structuring. In software development, layered architecture is a typical design pattern in which the application is divided into different layers. Each layer plays a specific role in the overall functionality of the application. The Flutter architecture includes several critical components, including the Flutter engine, the Dart platform, and the Flutter framework. The Flutter engine renders widgets and manages their interactions with the underlying platform, such as the operating system and device hardware. The Dart platform provides the runtime environment for the Flutter app, including the Dart virtual machine (VM) and the core libraries. Application architecture refers to how the various components of a mobile or web application are organized and how they interact with each other. No layer has privileged access to the layers below, and every part of the framework layer is designed to be optional and interchangeable [15]. The architecture of the framework helps to understand decisions regarding project structuring later. A well-designed application architecture helps improve the system's performance, maintainability, scalability and makes it more modular [16].

3.1.3. Programming Paradigm

The programming paradigm of the Dart programming language is object-oriented programming (OOP). It uses obj ects, classes, and inheritance to organize and structure code. Dart also incorporates some functional programming concepts, such as immutable data and first-class functions, which allow for more concise and elegant code. Additionally, Dart supports asynchronous programming, enabling developers to write code that can run concurrently and handle multiple tasks simultaneously. Overall, Dart's combination of OOP and functional programming paradigms makes it a versatile and powerful language for building modern web and mobile applications. The SOLID principles are a set of guidelines for designing object-oriented software. Robert C. Martin introduced them in his book "Agile Software Development, Principles, Patterns, and Practices" as a way to improve the maintainability, extensibility, and flexibility of object-oriented code and develop software prone to fewer bugs and cleaner source code [17].

- **Single Responsibility Principle (SRP):** The SRP nstructs the developer to develop classes and software components so that they take on a maximum of one responsibility. In other words, a class should focus on a single task or piece of functionality and should not be responsible for multiple unrelated things. This helps reduce complexity and improve a software system's maintainability, testability, and extensibility. Another positive side-effect of this principle is that the written code is easier to understand, and error testing can be done more efficiently [17].

- **Open/Closed Principle (OCP):** According to the open-closed principle, software classes should be open for extension but closed for modification, which means a class should be designed to be easily extended or customized without changing its existing code. This allows developers to add new features or behaviors to a class without breaking its functionality. These classes or software components ought to be developed to allow other system entities to use their essential features without requiring access to the original entity's source code.

- **Liskov Substitution Principle (LSP):** The Liskov Substitution Principle (LSP) asserts, in essence, that whenever a function uses a pointer or reference to a base object, it must also use a pointer or reference to any of its derived objects [18]. It is also an extension of the OCP. A subclass should be able to be used wherever its superclass is expected without breaking the program's functionality [19]. The Liskov Substitution Principle helps improve a software system's flexibility and reusability.

- **Interface Segregation Principle (ISP):** The Interface Segregation Principle ensures that clients of a class should not implement an interface containing methods that are irrelevant to its functionality. This helps to avoid creating large and complex interfaces that are difficult to implement and maintain. The Interface Segregation Principle promotes the creation of small, focused, and easy-to-use interfaces [18] [17].

- **Dependency Inversion Principle (DIP):** The fundamental essence of the DIP is that a class should not depend on the specific implementation details of another class. Instead, it should depend on an abstract interface or a set of contracts that define how the two classes should interact [18].

These principles should be considered when developing the application.

3.2. NoSQL Databases

The generic term NoSQL describes database systems that, unlike structured query language (SQL) databases, are not subject to the relational database model. The abbreviation NoSQL stands for "Not only SQL". The reasons why NoSQL databases have gained interest in recent years can be explained based on two aspects: In contrast to relational databases, which present their data storage in a table format, NoSQL databases benefit from different database models: document-oriented, key Value, graph, and column databases. This wide range of different data models allows developers to choose the model that best suits their application design. The result is a minimization of the code to be developed for an application. In addition, NoSQL databases allow administrators to scale their data on one machine and hardware clusters so that data volumes can be expanded without an expensive investment in new servers. NoSQL databases support the use of CRUD operations.

- **C:** create data
- **R:** read data
- **U:** update data
- **D:** delete data

The performance of some NoSQL databases even surpasses that of relational databases, especially with create and read operations [20].

3.2.1. Cloud Firestore

Cloud Firestore, more often called Google Firestore or simply Firestore, is a cloud-hosted NoSQL database option that enables developers to store and synchronize their data in real-time, meaning that data that just got added to the database and changes made on already existing data is instantly shown to the application users. It is a part of Google's Backend-as-a-Service (BaaS) Firebase [21, p. 458]. BaaS is a concept where developers can use a platform to run their applications without managing servers and other infrastructure components. BaaS platforms offer various services required for applications, such as databases, authentication, storage, and APIs [22]. To use BaaS, developers must first create an account with a BaaS platform and register their application. The platform then provides a set of APIs and SDKs that developers can use to access the services and integrate them into their applications. Most BaaS platforms offer a web-based console that developers can use to manage their applications and configure the services, and so does Firebase. Something worth mentioning is that Firestore can be used with far more programming languages than Dart and is also compatible with representational state transfer (REST) and remote procedure call (RPC) APIs [23].

Cloud Firestore caches data that the connected app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device returns online, Cloud Firestore synchronizes any local changes to its servers [23]. It is also possible to access data stored in the cache of the smartphone. Listing 3.1 shows how this can be implemented in Dart.

3. Fundamentals

```
 1 Future<DocumentSnapshot> checkCacheBeforeServer() async {
 2   try {
 3     DocumentSnapshot snapshot = await this.get(GetOptions(source: Source.cache));
 4     if (snapshot == null) return this.get(GetOptions(source: Source.server));
 5     return snapshot;
 6   } catch e {
 7     print(e);
 8     return this.get(GetOptions(source: Source.server));
 9   }
10 }
```

Listing 3.1: Firestore and Dart - getting Cache-data

First, an attempt is made to get the data from the cache (line 3). If this attempt fails, the data stored in the server is accessed instead (line 4).

3.2.2. Document Databases

There are several different NoSQL databases, which all rely on different data models. Firestore makes use of the document-based data format. Data stored in the database is accessible via collections filled with documents [23]. For better understanding, one can imagine a collection in Firestore as a table in relational Databases, and a Document in Firestore equals a row in the relational schema. Documents in Firestore store their data in a key-value format, making it possible for a developer to store different documents in each collection. A quick view at an example makes this easier to understand: A developer wants to develop a restaurant-review application. For that, he created a collection named "restaurants" in firestore. Two of the three restaurants he now wants to add to the collection got a slogan with their brand, which he wants to add to the documents. The other restaurant does not have one. In a relational database, he still would have to fill the "slogan" column with at least NULL-data or an empty String (or whatever datatype the column has). Firestore, or document-based databases in general, allow it to just not add the slogan attribute to the third restaurant, which helps only to store relevant data to the database. Remember that even if the third restaurant gets a slogan one day, the developer can add that field to the document later. Cloud Firestore also stores subcollections or complex nested objects in documents. After working with Firestore for a while, you realise that Firestore has no option to store foreign keys in a document. A solution to nevertheless establish a connection to other documents in the database will be explained later in the thesis.

4. Requirements Engineering

In order to provide a functional and relevant application, it is necessary first to determine the stakeholders who influence the project in the form of a stakeholder analysis, from which a system context can then be determined. The functional and non-functional, as well as the optional requirements for the application, must then be described. A domain model can then be generated based on the information gained from all of this, which serves as a transition to database creation.

4.1. Target Group and User Group

One more target audience is medical professionals. The application should offer an easy-to-use interface for adding and editing data in the database, eliminating the need for technical expertise. Only the doctors' attitudes about the project can provide a more specific indication of this user group's limitations. The stakeholder analysis will go into greater depth on this subject. Despite the difficulties mentioned in the introduction, both senior persons and young people who would desire to have a diagnosis of their current health status situation are targeted by the system. Given the age distribution of smartphone users today, the user base will tend to be a younger group of people but will also be used by older individuals. In Germany, 94.2 percent of people aged 14 to 19 owned a smartphone by the year 2021, according to Statista statistics. Between the ages of 20 and 29, it is 95,5 %, and between 30 and 39, it is 96 %. The proportion of smartphone owners among the over 70s is still around 68 percent [24]. One more target audience is medical professionals. The application should offer an easy-to-use interface for adding and editing data in the database, eliminating the need for technical expertise. Only the doctors' attitudes about the project can provide a more specific indication of this user group's limitations. The stakeholder analysis will go into greater depth on this subject.

4.2. Stakeholder Analysis

The first step is to identify the project's interest and demand groups. This is done through stakeholder analysis. The societal influences on the project are looked at in the stakeholder analysis. The stakeholder analysis allows for predicting variables such as "power", "interest", and potential stakeholder behavior. Stakeholders are individuals (groups), organizations, and interest groups that have the power to affect a project's success significantly. Therefore, project managers need to understand their interests and potential for influencing the project goals [25, p. 28]. It is necessary to consider which individuals have a stake in the project's success and which individuals have the potential to influence the project in both positive and negative ways in order to identify the stakeholders. Persons affected by the project might be classified as internal or external stakeholders.

4.2.1. Internal Stakeholder

Developer

In this project, the internal stakeholder group is relatively small. The only significant internal stakeholder will be the personification of the developer, who is also the data bank administrator. Due to the positive effects, a successful and widely used application would have on his reputation as a developer, this person has a tremendous personal interest in the project's success. The power he wields over the project is extremely high. With him, application development is not possible.

4.2.2. External Stakeholder

Users

Customers, or users in the case of an application, are considered external stakeholders. They want to use a flawlessly functioning application and are keenly interested in the project's success. This can be attributed to the points mentioned in the introduction chapter. Their impact on the project appears to be significant, given that the success of an application cannot be guaranteed in the absence of a user group.

Doctors

General practitioners and specialists make up another stakeholder group. They have the option to log in to the application and change existing database entries, as well as create new entries. Their impact on the project is moderate because the internal database manager stakeholder can expand the database without them. However, the power factor can rise and improve when doctors talk to their patients about the application. A doctor's negative (or positive) impression of the initiative may deter (or pique) patients, resulting in the loss (or gain) of users. As a result, individual differences in interest in the project will also exist.

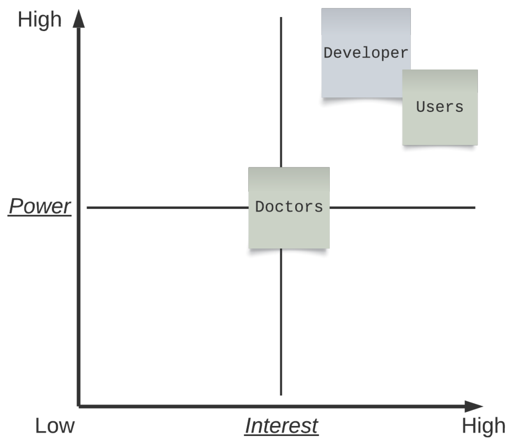


Figure 4.1.: Power/Interest Grid for Stakeholder-Analysis

4.3. System Context Diagram

The greater environment in which a specific system or process functions is known as the system context. It covers all the outside variables and influences that affect the system's function, such as the stakeholders impacted by its operation, external systems and processes with which it interacts, and the policies and regulations it must comply with [26]. The system context can be determined using the previously performed stakeholder analysis. Determining the system context helps to understand which components interact with the system. This includes both the stakeholders and systems that influence the system.

The stakeholder groups of users and doctors can access the application via a smartphone. The developer and the database communicate with the system via direct code-based access. The database is filled with data from an API, both have an impact on the system. There are some aspects of the privacy policy and the security of the user's personal data, especially during the verification process of the doctors. Since, in the context of this bachelor thesis, the application will not be launched on the PlayStore, these aspects should not be addressed in the system context.

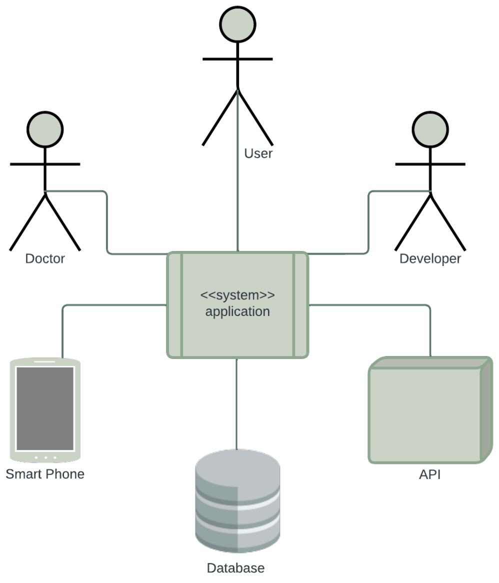


Figure 4.2.: System Context Diagram

4.4. User Requirements Specification

The requirements for an application can be divided into three categories: functional requirements, non-functional requirements and optional requirements [27, p. 51 ff.] [28]. In this section, the different types of requirements are considered and created project-specifically. An overview of all system requirements can be found in Appendix C.

4.4.1. Optional Requirements

As the name suggests, optional requirements of an application consist of requirements that do not necessarily have to be implemented. As can be seen from the survey, which can be found in Appendix A, users would like to be able to save their diagnoses. This wish forms the optional requirement 1 **([OR1])**. Another optional requirement for the system could be that a user gets the ability to save advice they browse as well - similar to the collection feature on Instagram **([OR2])**. This allows him to quickly filter and retrieve his stored advice.

4.4.2. Functional Requirements

Functional requirements are the specific capabilities, behaviors, and features that a system must have. They describe how the system must work to succeed and provide a basis for evaluating its performance and functionality. First, it should be possible for the user to switch back and forth between the different views of the application **([FR1])**. Furthermore, users shall be able to verify themselves as a doctor **([FR2]** and subsequently log in as such **([FR3])**. Once this is done, these doctors must be provided with the functionality to add new records to the database **([FR4])** and also to modify existing records **([FR5])**. Based on these records, a user should be able to start a new diagnosis **([FR6])**, which he can also save **([FR7])** and view again **([FR8])**. If the user wants to cancel the diagnosis, the system must be able to allow him to do so at any time **([FR9])**. After a diagnosis has been successfully made and saved, it must also be possible to delete the diagnosis **([FR10]1)**. Likewise, it should be possible for a doctor to cancel the addition or modification of data records at any time **([FR11])**.

4.4.3. Non-functional Requirements

After the functional requirements have been determined, the non-functional requirements are now considered. Non-functional requirements can be described as not dealing with a user's direct interaction with the application but with the system-specific properties. This includes, for example, the reliability of the application but also safety aspects [27]. For example, a non-functional requirement for the system is that it should be able to make correct diagnoses **([NFR1])**, and do so without requiring a user to log in **([NFR2])**. In addition, the graphical interface should be intuitive for the user **([NFR3])** to ensure user-friendliness.

4.5. Use Cases

The application's architectural goal is to provide an optimal user experience for both patients and doctors. In order to ensure this, it is necessary to determine, before the actual development, which uses cases the software has to cover, i.e., the externally visible interactions of the users with the system. This ensures that the application meets the users' wishes and that they use it. In addition, possible ambiguities are revealed, and required data structures are determined. Possible problems that may arise during the application use are likely to be found during the process, and technical solutions can then be worked out. Experience has shown that use cases also make it easier for a developer to create the objects that have to be created with an object-oriented programming language in the early development process more precisely and to recognize and implement inheritance options at an early stage. Some use cases can be identified from the functional and optional requirements above. Figure 3.3 shows the resulting use case diagram, which has been shortened to the most relevant use cases.

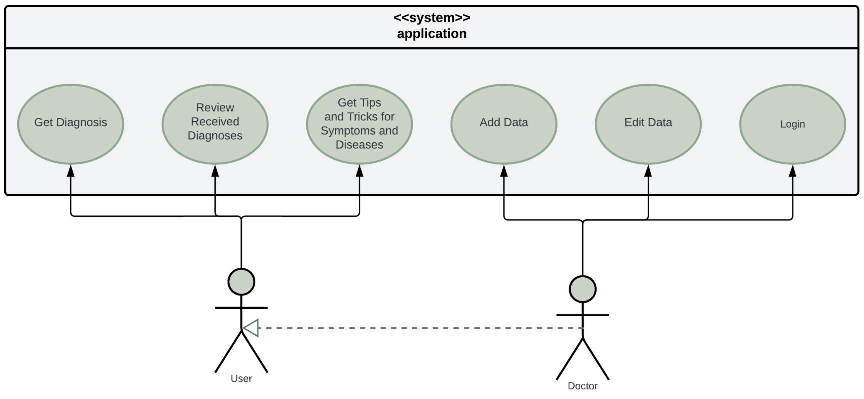

<<system>>
application
Get Diagnosis
Review Received Diagnoses
Get Tips and Tricks for Symptoms and Diseases
Add Data
Edit Data
Login
User
Doctor

Figure 4.3.: Use Case Diagram

The use cases are described in detail in the following sections. The resulting use case tables for each of them can be found in the appendix D.

4.5.1. Get Diagnosis

The first use case is described by the fact that a user would like to receive a diagnosis **[FR6]**. The actors who can carry out this use case are both patients, i.e., average users and doctors. In this application, doctors represent a subclass of users. In order to start the disease determination, the trigger, which will be implemented in the form of a button, must be pressed. The system then responds by displaying the view to enter and specify symptoms. As soon as the actor has finished specifying the symptoms, the system calculates the possible diseases and displays them to the actor as a diagnosis. The actor now has the opportunity to decide whether he wants to end the use case by exiting the diagnostic view or saving the diagnosis **[FR7]**, which is indicated by pressing the save button. The system then saves the diagnosis. Optionally, the actor should also be able to save the diagnosis as a PDF on his device **[OR1]**. During the process, the user can cancel the symptom statement at any time, which ends the use case by returning to the main page of the application **[FR10]**.

4.5.2. Review Received Diagnoses

After the user has received his diagnosis, he should be able to look at old diagnoses again **[FR8]**. The prerequisite for seeing such a diagnosis is that the actor had previously saved it when the diagnosis was made. A view is provided in the application in which saved diagnoses are displayed in a list. Clicking on such a diagnosis starts the use case, after which the system opens the detailed view of the diagnosis. The use case is ended by clicking on the back button provided for this purpose. While the diagnosis is being viewed, the user can delete the selected diagnosis. This is triggered by clicking on the icon provided for this purpose and is answered by deleting the diagnosis from the system. In this use case, too, the user is allowed to save his diagnosis as a PDF **[OR1]**. The procedure for this corresponds to that of the "Get Diagnosis" use case.

4.5.3. Login

As mentioned in the project goal, doctors should be allowed to log into the application **[FR3]**. The trigger for the use case is the click on the login button. Once the button is pressed, the application will display the login page. The actor can enter his credentials, at which point the system checks whether these are stored in the database. Should the result be positive, the doctor will be logged in, and the system will display the doctor's dashboard. The prerequisite for the use case to be carried out without errors is that the doctor has been able to verify himself as such beforehand **[FR2]**. If this has not yet happened, the user can click on the verify button, where he will be prompted to carry out this process and will be logged in if it is successful. Ould it is not successful because the user can not verify himself as a doctor, the system will continue showing error messages to the actor. Supposing the doctor is verified, but the system cannot find the entered credentials. In that case, the application displays an error message, and the user, suspecting that the credentials have been entered incorrectly, is asked to check his user data and try again after correcting possible input errors.

4.5.4. Add Data

Provided that he is logged in, a doctor can now add data to the database **[FR4]**. In order to do this, he must press the button provided for this purpose. The system then displays the blank template for a data record, in which the doctor can enter the data he wants to add. As soon as he has done this and pressed the confirm button, the system adds the data record to the database and saves it in his data list. Suppose the actor presses the confirm button without entering anything in each data field. In that case, the application will display an error notification on the screen, prompting the user to fill in all data fields. If the user wishes to cancel the process, he is free to press the button provided for this at any time, whereupon the system closes the view, and the use case ends **[FR11]**.

4.5.5. Edit Data

In addition to the functionality to create new datasets, the doctor should be able to expand and edit existing datasets **[FR5]**. The structure of this use case is similar to the previously described use case of adding new data. The doctor must first be in the view where all existing data records are displayed in list format. There he has the opportunity to click on one of these data sets, which signals to the system that it must now display the editing screen for the selected data set. In this view, the doctor can make the desired changes and press the confirm button. The application will then update the record in the database, and the use case will be terminated. As before, when adding new data records, the doctor can end the process at any time **[FR11]**.

4.5.6. Get Tips and Tricks for Symptoms and Diseases

The final use case worth mentioning is viewing advice on illnesses **[FR1]**. A user can go to the view for all advice that Doctors have uploaded. Once he has navigated there, and the system shows the predicted view, he can click on one of the pieces of advice there, which will be shown to him in detail afterward. Optionally, the user can save the advice as a favorite by clicking on the button provided for this purpose **[OR2]**.

4.6. Domain Model

Domain modeling is a vital modeling topic in Agile development at scale because there is frequently a gap between comprehending the issue domain and interpreting requirements. It depicts the solution as a collection of domain objects that collaborate to satisfy system-level scenarios [29]. The quintessence of the object-oriented analysis step is decomposing a domain into problem-relevant concepts or objects. A domain model is a visual representation of the problem-relevant domain classes of a domain. With the help of unified modeling language (UML) notation, a domain model is represented by a set of class diagrams in which no operations are defined. It presents a conceptual perspective and can show domain objects or classes, as well as associations between domain classes and attributes of domain classes.[29] [30]. Identifying domain entities and their connections, derived from a grasp of system-level requirements, offers a good foundation for understanding and supports practitioners in designing systems for maintainability, testability, and incremental development [29]. Finding conceptual classes by recognizing substantive phrases is an effective technique for domain modeling [30, p. 76].

- A person is a **user** of the application.
- A **user** chooses a **body part**.
  - A **body part** can be associated with different **symptoms**.
  - A **user** selects a **symptom** and specifies the selected symptom by narrowing down (selecting) **proposed symptoms**, the **time of occurrence** and the **symptom intensity**. The information obtained is summarized as a textbfuser-specified symptom.

- One or more **user-specified symptoms** lead to the calculation of one or more possible **diseases**.
  - A **disease** has different **symptoms**.
- A **diagnosis** consists of one or more **diseases**.
- **Doctors** are special **users** of the application.
- **Doctors** can add/edit **advices**, **symptoms**, **body parts** and **diseases**.
- **Users** can view **advices**.

The information just obtained makes it easier to create the domain model. The entities user, symptom, user-specified symptom, body part, diseases, doctor, advice and diagnosis can already be recognized. Based on that information a domain model can be created.

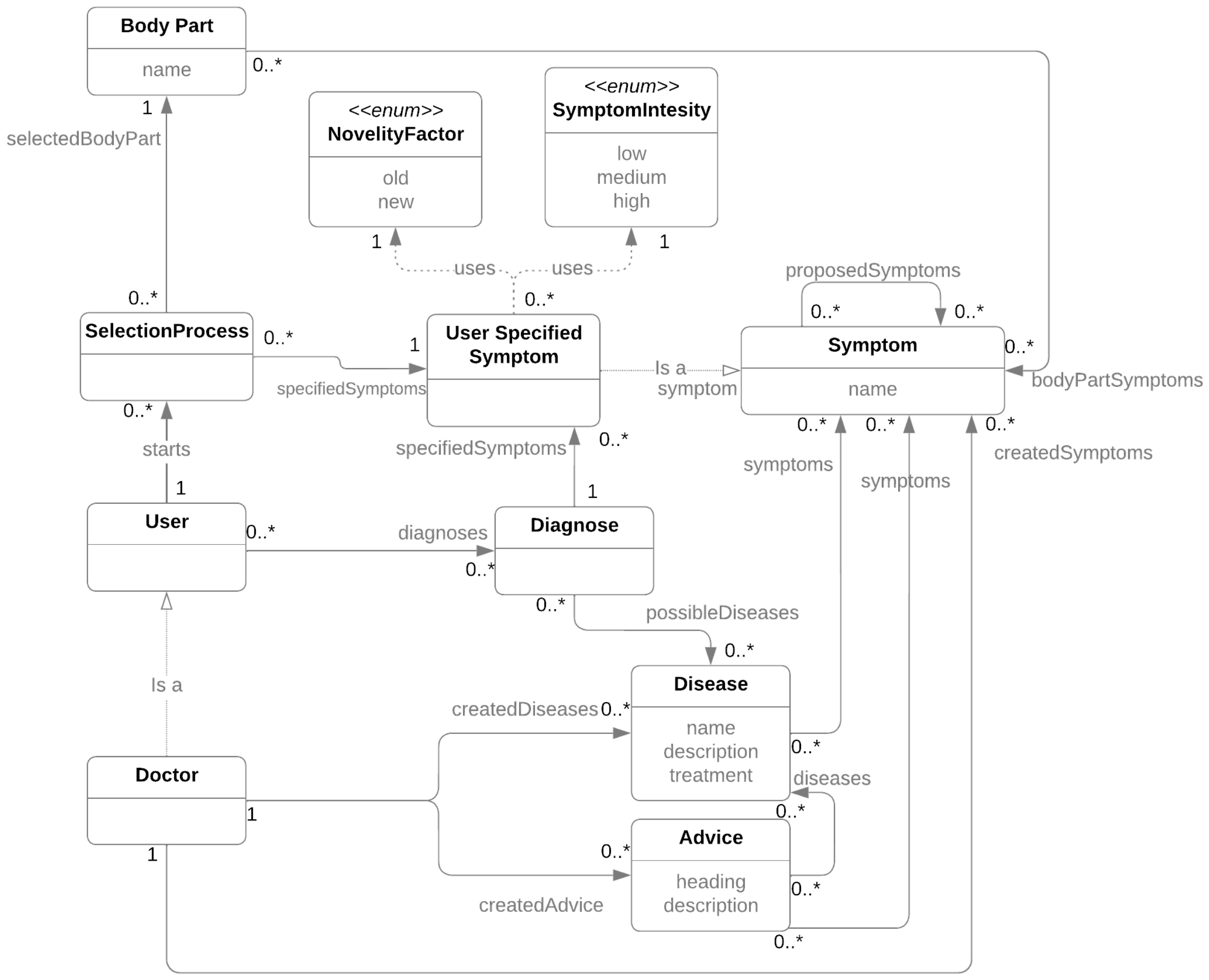


Figure 4.4.: Domain Model

Based on the domain model and the previously obtained information, the database development can now be started. Later on, the chapter optimizations also addresses some aspects that have been neglected at the moment.

5. The Database

The following chapter is devoted to creating the Firestore database for the application. After the successful data conception based on API responses, the data sets are integrated into Firestore via a Python script. This procedure is also documented using a notebook (Jupyter Notebook) and is described there in more detail.

5.1. Options for adding Datasets using APIs

The aim of this work is the conception and implementation of the described system. In order to achieve this goal, the application must access a database filled with relevant data. The information from an existing API is used here since no medically trained project participants could contribute their specialized expertise to the database. The API selection also influences the database's final data structure. For this purpose, the two most fitting APIs are considered, and their suitability for the project is determined.

5.1.1. NHS Health A to Z API

The NHS, or National Health Service, is the publicly funded healthcare system of the United Kingdom. It was established in 1948 and provides a wide range of medical services to the population of the UK, including general practitioners, hospitals, and community health services. The NHS website provides many different APIs, free for use [31]. The NHS Health A to Z API is the first API that will be considered to generate data for the database.

The API offers medical information about various diseases, their symptoms, and available treatments. A user account that is provided with a subscription key must be created in order to receive data from this API. The next step is to make an call with the hypertext transfer protocol (HTTP) to **https://api.nhs.uk/conditions**. An example of a possible request in the programming language Python looks like this:

```python
urlDiseases = "https://api.nhs.uk/conditions/acne"
header = {
  "subscription-key" : "YOUR_SUBSCRIPTION_KEY"
}
responseDiseases = requests.request("GET", urlDiseases, headers=header)
responseData = responseDiseases.json()
```

Listing 5.1: Example Python Request for the Health A to Z API

If the request was successful and the response contains the data in the javaScript object notation (JSON) format. Appendix C.1 shows an abbreviated sample response from the API. A positive aspect of this API is that all data is described in great detail, and a large amount of knowledge can be obtained in a single query. However, it must also be mentioned that the extent of the server response just mentioned entails the difficulty of storing the data accordingly in a separate database. For example, symptoms are supplied for a disease, but only in string format as a complete sentence. This means that a symptom is described in different ways in several diseases. In order to automatically scrape this data, one would have to recognize all variations in the description of a symptom. With the amount of data that the Health A to Z API brings, this is almost impossible. Another limitation is that a maximum of 6 requests can be made to the interface per minute, which proves to be a severe problem in terms of runtime.

5.1.2. ApiMedic Symptom Checker API

The ApiMedic API is the second interface to be considered. ApiMedic is powered by priaid [32], a company that combines medicine, IT, and business administration. Thanks to their highly specialized team composition, they offer expertise in all areas mentioned. API calls can be done with two different accounts:

- **Sandbox API Account:** It is possible to get unlimited data via the sandbox account. However, the data supplied is only dummy data.
- **Live Basic API Account:** The live account allows one to get the actual medical data of the API. However, there is a limitation concerning the possible calls: ApiMedic only allows 100 calls per month to be made without charge, and further requests cost money.

Although the data that can be received from this API is less detailed than that of the NHS API, using the data to create a data structure is more straightforward. It is possible to acquire diseases, bodily components, and symptoms as well as proposed symptoms and symptoms based on each body part. The following code example shows a request made with the live account to retrieve all diseases, the response of the API can be found in appendix C.2.

```python
stringURLIssues = "https://healthservice.priaid.ch/issues?token=YOUR_TOKEN"
responseIssues = requests.request("GET", stringURLIssues)
dataIssues = responseIssues.json()
```

Listing 5.2: Example Python Request for the ApiMedic API (all issues)

It is now possible to execute an API request that returns detailed information about each disease using the provided IDs. Appendix C.3 shows a sample response from the ApiMedic API.

```python
stringURLIssue = "https://healthservice.priaid.ch/issues/105/info?token=
  YOUR_TOKEN"
responseIssue = requests.request("GET", stringURLIssue)
dataIssue = responseIssue.json()
```

Listing 5.3: Example Python Request for the ApiMedic API (single issue)

The value of the "PossibleSymptoms"-key returns an enumeration of all the disease symptoms. These symptoms are listed using the values of the "Name"-key for the respective symptom when querying all symptoms. This makes it easier to scrape the data accordingly and store it in a database.

5.1.3. API Solution

The question that now arises is which of the two interfaces to choose. Both APIs have advantages and disadvantages. While the NHS API provides very detailed results, it is challenging to use the data for the purposes intended in this work. NHS also provides data on the causes of various symptoms and conditions, which can be an essential factor in making a diagnosis in the form of disease detection. ApiMedic delivers the data in an optimal format but far less detailed than NHS. One available option is to use the symptom data provided by ApiMedic as scrape material for the symptom list in the NHS. However, after an attempt to do so, only a minimal amount of symptoms has been recognized. This is because the same symptom is named differently in both APIs. Generating more appropriate scrape material would require a full inspection of all NHS API data to ensure getting a decent amount of data. This is not possible within the scope of this work but should be considered for future system optimizations. In the context of the bachelor thesis, the ApiMedic API is preferred from the point of view of a clean database structure.

5.2. Data Structure

The data structure of the database is based on the decision that the ApiMedic API will be the main supplier of the data. The basic data structure can already be guessed from the domain model, which is shown in Figure 4.4. Firestore supports the following datatypes: String, Number, Boolean, Map, Array, Null, Timestamp, Geopoint and Reference. With a closer look at the API's JSON responses, the data structures can be formed.

5.2.1. Examination of the JSON-Structure of ApiMedic

ApiMedic does not provide any actual documentation. Instead, they give interested users the opportunity to test the HTTP requests directly via the ApiMedic website. To access the API, a HTTP request must be sent. The URL of the endpoint, the HTTP method and, if necessary, other parameters and headers must be specified. All inquiries can be made via a request to the following URL: **https://healthservice.priaid.ch/** in connection with the corresponding path to the desired file resource. The API responses can then be extracted from the HTTP message body in the form of a JSON object.

Body Part

- **Get all Body Regions: /body/locations?token={your_token}**
  If the request is successful (status code 200), all body regions that are stored in the API form the output.

Name of Field	Content	Datatype
ID	ID of the body region	Integer
Name	Name of the body region	String

  With the help of the ID of the body regions, the individual body parts of a region can be determined.

- **Get all Body Parts: /body/locations/{body_region_id}?token={your_token}**
  If the request is successful (status code 200), all body locations that are stored in the API form the output.

Name of Field	Content	Datatype
ID	ID of the body location	Integer
Name	Name of the body location	String

- **Get all Body Parts: /body/locations/{body_region_id}/{gender_id}?token={your_token}**
  If the request is successful, all body locations that are stored in the API form the output. The IDs of the genders are values ranging from 0 to 3.

Name of Field	Content	Datatype
ID	ID of the symptom	Integer
Name	Name of the symptom	String
HasRedFlag	Indicates whether the symptom has been classified as critical	Boolean
HealthSymptomLocationIDs	IDs of the body locations that are affected by this symptom	Array[Integer]
ProfName	Professional name of the symtom	String
Synonyms	Synonyms of the symptom	Array[String]

Instead of storing all body areas, all explicit body parts and their possible symptoms will be stored in Firestore. Based on the information obtained from successful HTTP requests, the Firestore structure can be defined as shown below.
Collection: body_parts

Name of Document-field	Content	Datatype
name	Name of the body part	String
symptoms	List of all symptom ids of the body part	Array[String]

The ids of the documents are formed from the name of the respective object. Here, spaces are simply replaced by a "_" and the name string is stripped of leading and trailing spaces using the Python function .strip(). The id generation method should never be used for real world applications. However, it simplifies the legibility in the context of this bachelor thesis. Firestore saves all the data in form of documents which can be created using .doc({document_id}).set({...}). When performing this operation, the document is automatically assigned the specified ID, making it unnecessary to store the id in the document itself.

Ressource Symptom

- **Get all Symptoms : /symptoms?token={your_token}**
  If the request is successful, all symptoms that are stored in the API form the output.

Name of Field	Content	Datatype
ID	ID of the symptom	Integer
Name	Name of the symptom	String

- **Get all proposed Symptoms: /symptoms/proposed?symptoms=[{symptom_ids}] &gender={gender_name}&year_of_birth={birthyear_YYYY}?token={your_token}**
  Unlike the query regarding all symptoms of a body part, the values of the genders are not to be given as an ID, but in the form of full names, i. e. "male" and "female".

Name of Field	Content	Datatype
ID	ID of the proposed symptom	Integer
Name	Name of the proposed symptom	String

Since all ids of the symptoms for each body part are already stored in each body part document, there is no need to add the body part ids to the symptom anymore. Because of that, all that is needed to be stored is the name of the symptom and the ids of the proposed symptoms.
Collection: symptoms

Name of Document-field	Content	Datatype
name	Name of the body part	String
proposed_symptoms	List of all proposed symptoms ids of the symptom	Array[String]

Ressource Disease

- **Get all issues (diseases): /issues?token={your_token}**
  If the request is successful (status code 200), all diseases that are stored in the API form the output.

Name of Field	Content	Datatype
ID	ID of the body region	Integer
Name	Name of the body region	String

  With the help of the ID of the body regions, the individual disease can be determined.

- **Get a single issue: /issues/{issue_id}?token={your_token}**
  If the request is successful (status code 200), the requested issue forms the output.

Name of Field	Content	Datatype
Description	Description of the disease	String
DescriptionShort	Short description of the disease	String
MedicalCondition	Description of the symptoms	String
Name	Name of the disease	String
PossibleSymptoms	All symptoms of the disease, comma separated string	String
ProfName	Professional name of the disease	String
Synonyms	Synonyms of the disease	String
Treatment	Treatment steps for the disease	String

The resources provided by the issues are stored in the Firestore database under the diseases collection. For the purposes of the work, only the name, description, symptoms and treatment recommendation are extracted and stored.
Collection: diseases

Name of Document-field	Content	Datatype
name	Name of the body part	String
description	List of all symptom ids	Array[String]
treatment	Treatment recomendation of the disease	String

Advice

The advice generated by doctors is not part of the ApiMedic API data resources. They are created directly by users (doctors) of the application and require a title, description, associated symptoms and associated diseases. Taken together, this results in the document structure shown in Table x.x.
Collection: advices

Name of Document-field	Content	Datatype
name	Title of the advice	String
description	Description of the advice	String
symptoms	List of associated symptom ids	Array[String]
diseases	List of associated disease ids	Array[String]

Doctor

Doctors have the opportunity to register with the database. To do this they will be asked to enter an email and a password. Firestore allows all of this to happen without involving a dedicated collection of users. Nevertheless, it makes sense to do this, since user data could possibly be expanded at a later date. An example of this is that records added by a doctor should also be referenced in his account. This requires a collection that stores the user data of the respective doctor in separate documents. The user ID is automatically generated by Firestore in the form of a hash value during registration.
Collection: doctors

Name of Document-field	Content	Datatype
email	E-Mail-Adress of the doctor	String

User, User Specified Symptom, Diagnose, SymptomIntensity and NovelityFactor

The user-defined symptoms are not stored in the database, but are only generated temporarily and locally in the system during the diagnosis. The reason for not saving the user-specific symptoms is that a user does not have to log in to the application a nd therefore no user document is created in the database through which the symptoms could be traced back to the user. Only the individual attributes of the object are stored in the diagnosis in order to be able to understand the diagnosis when it is called up again. The classes users and diagnoses, as well as the enumerations symptoms intensity and novelty factor will not be stored in the database, but will be, too, modeled internally in the system. There, the diagnoses are to be stored locally on the smartphone, which ensures that no personal data of the user is stored in the database.

5.3. Inserting the Data into the Database

Adding the records to the database requires a few steps, which have been summarized with the help of Jupyter Notebook. The notebook can be downloaded by following the link of the QR code in the appendix x. After the successful data generation, the database can now be integrated into the application. For this, the scheme provided by the Flutter developers is followed.

6. Conception and Design

This section deals with the conception of the graphical user interface and the implementation options of the application.

6.1. Graphical User Interface

First, a look is taken at whether the graphical user interface plays an important role in applications at all. For this purpose, a question on this topic was also included in the survey already mentioned. The answers on this question revealed that the graphical interface of an application, in fact, plays a role in terms of the trustworthiness of the application. Figure 6.1 shows the results of the survey-question.

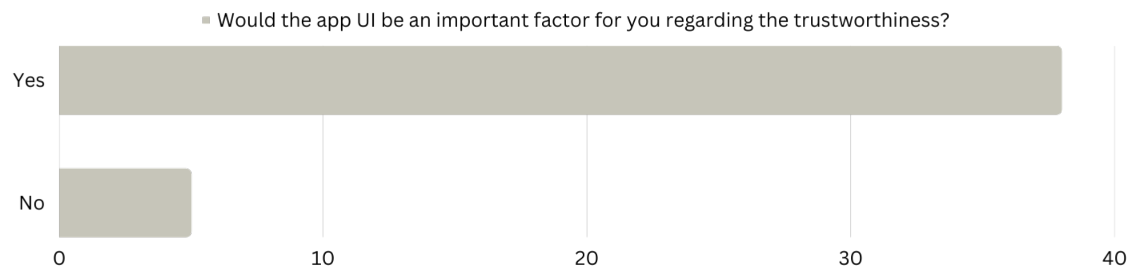


Figure 6.1.: Survey Question - Trustworthiness of the Application based on the UI

Respondents of the survey were also given the opportunity to choose between three different UIs. The graphical interface should be adapted to the choice made by the people interviewed.

6.1.1. Conceptual Design of the Graphical User Interface

The UI can be designed based on the requirements analysis of the application and the information obtained from it. As already defined there, the application should be designed to be as intuitive and easy to use as possible. The implementation of the design of the user interface, for the android version, is based on suggestions from the material.io website [33], an open-source design system developed by google. The integration of the design components into the Flutter application is simplified thanks to the pre-installed material.dart package. This package is based on the source mentioned and currently supports version 2 of material.io. Support for the current version 3 of Material Design is still being worked on. One goal of this application is to be designed to be intuitive. In his book "UI is COMMUNICATION", Everett N. McKay described an intuitive UI as follows [34]:

> *"A UI is intuitive when target users understand its behavior and effect without use of reason, memorization, experimentation, assitance, or training."* - Everett N McKay [34, p. 22]

In order to meet this definition, goals that the user interface should meet could be set before the graphic user interface is designed.

On the one hand, the user should be able to recognize immediately that a component of the screen is a widget that he can operate. The system must then show the user that his action has triggered something. This can be done, for example, through feedback in the form of a loading circle if required data still needs to be loaded or in the form of a new screen, pop-up, or similar design components. [34] Another step that can be taken to make the application more intuitive is the correct (short) labeling of the respective components. For example, a corresponding text, or even better imagery [35, p. 172], should be stored on a confirmation button. In addition, it should be ensured that surface components do not change their positions. This ensures that users can rely on their previous interactions and thus learn how to use the application. Minimalism is becoming a trend not only in the real world but also in the digital world. This can also be recognized by looking at the graphical development of various icons and web interfaces of large companies. This can also be seen by looking at the most popular graphical interface of the survey, shown in Figure 6.2. The associated surfaces can be found in Appendix x.

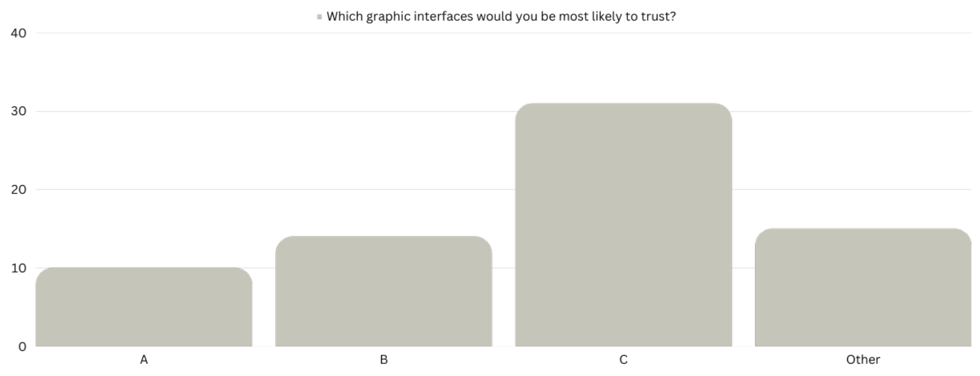


Figure 6.2.: Survey Question - Which UI would respondents like the most

Designing the application as minimalistic as possible has the advantage that the graphical interface is not overloaded; thus, the user is not overwhelmed by different widgets. This may make it easier for the user to interpret specific elements better. Another factor that plays a role in user-friendliness is to use of familiar design patterns. This ensures that users are already familiar with handling the app elements and that everything is clear regarding the use.

6.1.2. Description of the Views and Thoughts on the Implementation in Flutter

This section presents the individual views that are made available to users. In addition, first thoughts on possible implementation ways for those views, working with Flutter, are considered.

Home Screen

The home screen is the first view a user will encounter when installing or opening the application. There, he should be allowed to switch between the different views necessary for his use: diagnosis creation, diagnosis overview, and advice view. In addition, every user should be able to access the login screen for doctors since they can verify themselves there. The navigation to create a diagnosis is implemented as a button and labeled with a corresponding, clearly formulated text. To get to all other screens, a list view of all options is offered, in which various clickable fields trigger navigation to the corresponding screen when clicked. In Flutter, this view can be achieved through several different components. Essential widgets can be used as children in a column. In order to implement the clickable forwarding elements to other views, one should use a ListView, storing children of the widget

GestureDetector, which allows any widget to be clickable. The children can be stored in an external modifiable list, which simplifies the integration of new categories at a later point in time. A mock-up of a possible home screen can be found in Appendix x.

View for the diagnostic process

The screen to start a diagnosis should be designed as simply as possible. Users can address on which body part they are experiencing their symptoms. Based on their selection, the application shows all symptoms associated with the body part, which can be selected and afterward specified by the user. The widget *Stepper* is a possible implementation option. It allows the developer to implement a type of step-by-step tool quickly. For this purpose, only all steps for generating a diagnosis are created and filled with the appropriate data. The complete code for an example Stepper widget can be found in appendix x.

Login Screen (Doctor)

The login screen is displayed to the user immediately after clicking on the navigation tool labeled "Doctor Panel". There he will be offered the option to log in or register. This is achieved by adding editable text fields for the needed user input: email, password, and confirmation password. If the user has already created a user account in a previous session, has verified himself as a medical professional, and has then been logged in, he will be forwarded to the actual doctor panel without needing to log in. An exception is given if he has previously logged out. With the help of the flutter_login package, a developer can easily implement the login process in Dart. Appendix x demonstrates the use of the FlutterLogin-plugin. There are also options to customize the appearance of the login form.

Screen with all body parts, diseases, advices and symptoms (Doctor)

The doctor panel should contain all the necessary data related to the records stored in the database. Here, a doctor can switch between the different collections. A suitable implementation method for this would be the BottomNavigationBar. It contains various navigation elements, which all change their data view when clicking on them. The data views show a list of stored data in the associated Firestore collection. Care should be taken to ensure that it is recognizable that one can click on the list elements. When the user clicks on such an element, the system redirects to the edit view for the associated record. In order to be able to store new data quickly, a FloatingActionButton can also be integrated, which opens a small menu by clicking on it, where all the data add options for the doctor are listed. Clicking on a listing item then opens the associated add view.

View for adding and editing data (Doctor)

In order to make the views as intuitive as possible, it is advisable to make the graphical interface as uncluttered as possible. Because of this, only text fields are included for the data records where possible. For the selection of related data for a data set (e.g., symptoms for an illness), a button is placed on the view, which, when clicked, generates and displays a dialog with the content of all symptoms in the database. Ideally, this dialog should also contain a search bar so that the user is not forced to scroll through many data records. Both the edit view and the view to add new records consist of a form widget. There, it is possible to add text fields as children, which users can modify. The distinction here is only between the initial value of the text fields. While the data stored in the database for the associated data record is displayed in the edit view, only an empty text field with a hint regarding the data field is specified when a new data tuple is created.

View with all saved diagnoses

A vertical list of all diagnoses represents the diagnoses view. A ListView is used here. This ListView contains all diagnostic data that the user has previously saved on his device. Clicking on a diagnosis element in the list opens a detailed view of this diagnosis. It shows the date on which the diagnosis was made, which symptoms the user specified, and which illnesses resulted from calculating the probability of diseases. This is represented in the form of a bar chart. At the end of the diagnosis, the diseases are again listed in list form, with their description and treatment recommendations.

View with all advices stored in the database (User)

The advice screen can be designed similarly to the diagnosis screen. Here, too, the data from Firestore is displayed in the form of a vertical list, and a click opens the detailed description of the advice. This detailed view consists of text fields containing the document fields' data.

6.1.3. Mock-ups and Survey regarding the Trustworthiness of Mock-up Designs

With the description of the screens and the first development approaches, mock-ups can already be created with the help of Canva. These can be found in Appendix x. As part of a survey, it was found out whether the basic idea of the graphic design would speak for trustworthiness and whether the participants would have instinctively known how to interact with the interface. The results are presented below.

7. Implementation

Based on the specifications from the previous chapter, it is now possible to open the app to implement. This chapter describes the technologies used for the development of this application. The project structure is also examined in more detail.

7.1. Project Structure

When developing software systems in the client-server, it is advised to make sure that the project and class structures are divided into three different layers.

- **Data Layer**
  The data layer takes care of retrieving the (raw) data from the database. For this purpose, a class named DatabaseService is generated as part of the Flutter application, which can be instantiated.

```
    class DatabaseService {
      /* CREATE AN INSTANCE OF THE DATABASE */
      DatabaseService._();
      static DatabaseService _instance = DatabaseService._();
      static DatabaseService get instance => _instance;
    }

```

Listing 7.1: Stepper for Body Part Selection

  Using DatabaseService.instance._methodName, each method of the service can now be called globally from anywhere in the project.

- **Business Logic Layer**
  With the help of the business layer, the data that is now received from the database service can be converted into models with which the system can work accordingly. For this purpose, a ConvertService can be created similar to Listing 7.1, which can be instantiated in the same way. This now queries the data using the DatabaseService.instance and converts the data which is supplied by the Firestore API in the form of DocumentSnapshots, Streams or QuerySnapshots. The data models for this are presented in the Data Models section.

- **Presentation Layer**
  This layer takes care of mapping the data to the graphical interface. To do this, it queries the model data of the ConvertService and creates the widgets that are to be displayed accordingly.

In project development, it is increasingly interesting how applications receive their data. As part of this work, this will happen through API queries to the Firestore database. The whole procedure is also known as the client-server model. An example of the communication process, together with the services that just got described, is shown in Figure 7.1.

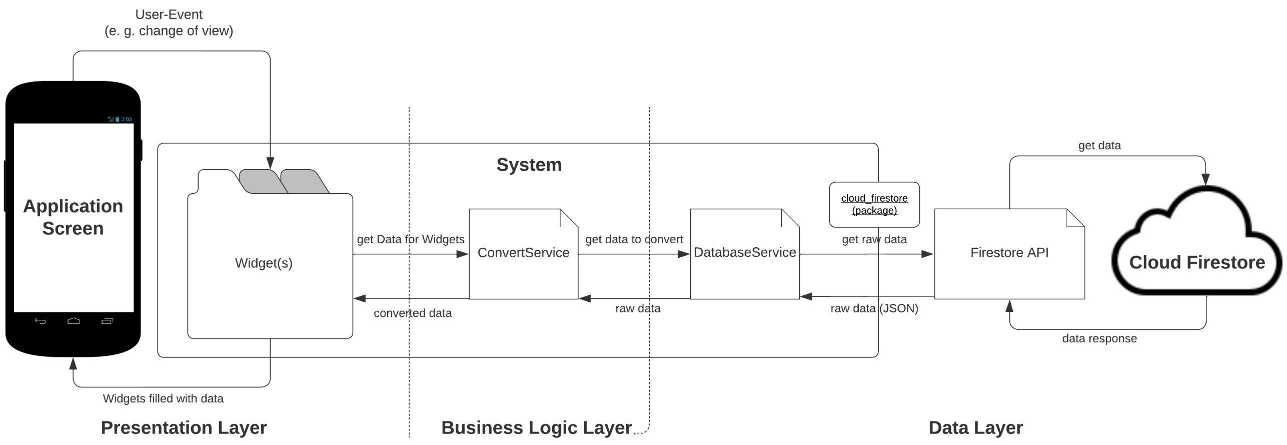


Figure 7.1.: Client-Server Request Flow

7.1.1. Routing in the Application

Another step that can be taken to keep the project structure as clean as possible is to lay out the navigation in an extra class. In this work, a class that is named AppNavigator is created for this purpose. This navigator can then be called globally using a defined method AppNavigator.push(Route pRoute). The route parameter is an element of an enumeration which refers to different views of the application. All AppNavigator code can be found in Appendix x.

7.2. Data Models

The data model into which the ConvertService converts the raw data of the DataService is based on the document structure stored in the database. When scraping the data of the JSON responses, the data is converted as follows: String → String, Array → List, Map<key, value> → Map<String, dynamic>, Number → num. Listing 7.2 shows the resulting model for symptoms. All other Models can be found in the appendix x.

```
class Symptom {
  final String? id;
  String name;
  List<String> symptoms;

  Symptom({this.id, required this.name, required this.symptoms});

  Map<String, dynamic> toMap() {
    return {
      'name': name,
      'proposed_symptoms': List<String>.from(
      symptoms.map((x) => x),
      ),};}

  Symptom.fromDocumentSnapshot(DocumentSnapshot<Map<String, dynamic>> doc)
    : id = doc.id,
    name = doc.data()!["name"],
    symptoms = List<String>.from(
    doc.data()!["proposed_symptoms"].toList(),);
}
```

Listing 7.2: Model for Symptoms

7.3. Algorithms for the Attribution of Symptoms to potential Diseases

7.3.1. List Matching

The first methodology to be considered is based on purely theoretical aspects. By specifying the symptoms given by the patient, the application compiles a list of them. When adding the diseases to the database, a list of symptoms was also generated in each disease document. An intuitive idea would be to compare these two lists with each other according to a matching algorithm, which calculates the percentage of word matching in the lists. The so-called word matching is described in source [36]. However, it should be noted that no medical data is included in the calculation. Accordingly, the probability of the correctness of the diagnosis decreases. In this sense, another, more exact method is to be considered.

7.3.2. Machine Learning: Bayes Theorem and Bayesian Networks

The so-called Bayes theorem is the foundation of the machine learning technique known as naive Bayes [37, p. 403]. According to the Bayes theorem, a principle of probability theory, one may determine the likelihood of an event (such as the existence of a disease) based on the likelihood of occurrences that are connected to it (e.g., certain symptoms) [37, p. 403 ff.] [38]. The Bayes theorem is applied in the case of Naive Bayes to forecast classification issues. The model's components are thought to be completely independent of one another [39].

On the other hand, Bayes networks are a particular probabilistic graph type that employs the Bayes theorem to determine the likelihood of occurrences that depend on numerous independent variables [40]. In Bayes networks, the edges between variables represent the relationships between them, with the edges' probabilities describing the link's strength. Similar to Naive Bayes, Bayes networks can be used to forecast classification problems, but they can also be used to simulate more intricate relationships between variables [40]. Generally speaking, the Bayes theorem-based methodologies of Naive Bayes and Bayes Networks are used to forecast occurrences. The fundamental distinction is that although Bayes networks represent and model the relationships between variables, Naive Bayes implies that all features are independent.

To create a Bayesian network for disease calculation, it is first necessary to identify the variables that should be included in the model. These variables will include the various symptoms that are associated with different diseases, as well as the diseases themselves. Once they are identified, one can create a graphical representation of the relationships between them by creating a directed acyclic graph (DAG) [40] [41]. In this DAG, the nodes represent the variables, and the edges represent the relationships between the variables. For example, a node for a particular symptom, such as a fever, and another node for a particular disease, such as the flu. Then an edge between the nodes can be created to represent that the presence of a fever is often associated with the flu. It is also possible to assign probabilities to the edges in a Bayesian network to represent the strength of the relationship between the variables. For example, if a fever is a strong indicator of the flu, it might be wise to assign a high probability to the edge between the fever and flu nodes. On the other hand, if it is known that a fever is only a weak indicator of the flu, one might assign a lower probability to the edge.

Once the Bayesian network is created, and probabilities are assigned to the edges, one can use it to make predictions about the likelihood of an individual having a particular disease based on the presence or absence of specific symptoms. For example, if one has a Bayesian network that includes nodes for the symptoms of fever, cough, and sore throat, as well as nodes for the diseases of the flu and the common cold, which is information that the patient entered: In that case, one can use the probabilities assigned to the network's edges to calculate the likelihood that the individual has the flu or the common cold.

In summary, Bayesian networks are a powerful tool for calculating the likelihood of an individual having a particular disease based on the presence or absence of certain symptoms.

7.3.3. Algorithmic Solution

Lastly, the symptom classification described in the conference paper "towards a ranking of likely diseases in terms of precision and recall" [42] is examined. Although the paper was published in 2012, it addresses problems that are still present today with certain disease classification methods and a solution, developed with medical experts, on how these methods can be cleanly replaced. The information which can be taken from the paper still offers a good basis for an algorithmic solution of the disease determination.

The paper addresses, among other things, problems that bayesian networks bring with them: the pure information regarding the probabilities of certain diseases is not widespread and accessible or even available [42]. The solution described in the paper defines a variant in which one does not have to fixate purely on the probability of a disease based on symptoms, but other values are also included.

Influencing Factors

First, the influencing factors are considered and assigned in the context of the application.

- **Classification of the Symptoms**
  After the user has entered his symptoms in the application, related diseases can be filtered from the database based on the information provided. This provides all possible symptoms of the diseases and two sets can be formed: present and absent symptoms. This information is later used to determine the probability of a disease based on the patient's symptoms.

- **Probability of Occurrence of a certain Disease**
  For the probability of occurrence of a disease, dummy data are used in this work. The procedure for storing these data in the database is carried out in JupyterNotebook, which can be found in Appendix x as already described.

  In addition to this information, the probability of how severe a symptom actually is is needed. In the paper, this value is described as default importance. The factor is also stored in the form of dummy data.

- **Patient-specific Factors**
  Patient-specific factors include characteristics such as age, gender, or whether or not the patient smokes. A query for these factors can be made during the symptom specification. However, in the context of the implementation of the algorithm in this work, these factors are left out.

  In addition to these factors, the time of occurrence of a symptom and the intensity with which a patient experiences this symptom must also be included. This information is provided by the user as part of the symptom specification and is stored temporarily in the form of a user specified symptom. The factors of intensity and novelty are stored in the program code by means of enumerations.

- **Relation of Diseases and Symptoms**
  The calculation of relatedness requires, first, the calculation of the conditional probability P(s|d) of a symptom and a disease, and second, the factor of whether the symptom is a leading symptom of the disease. The implementation in this work determines the factor, whether it is a leading symptom also, based on dummy data.

By means of the described factors the values precision and recall can be determined [42, p. 9]. With the help of these, in a further step, it can be determined to what degree the patient's symptoms can be mapped to the disease symptoms. Finally, the ranking of the diseases can be calculated as follows:

$$r(d) := F(d) + \lambda + P_p(d) \qquad \text{from [42]} \tag{7.1}$$

Where F(d) stands for the factor of symptoms matching the disease symptoms and P_(d) stands for the incidence proportions. In case of this work the incidence proportions will be the probablity assigned to each disease. The lamba value is used to reduce the disease probability value, as this is considered less important according to the authors [42, p. 10].

In the context of the bachelor thesis, the focus will lay on this methodology, since it is basically based on the variant of Bayes Networks but in a more algorithmic way instead of machine learning.

7.4. Evaluation of the Algorithm

Within the scope of the work, the last mentioned algorithm was implemented using the Dart programming language. In the next step, it will be evaluated. For this purpose, 3 different symptom compilations were generated. How these look like can be taken together with the exact test results from the appendix F. There one can also get information about the system on which the emulator is running.

7.4.1. Calculation Time

With every compilation, the program took more 30 seconds up to over a minute to calculate the individual values for all the specified symptoms and associated diseases. This means that a user has to wait an average of one minute to receive a diagnosis. Considering that the application should be designed to be as user-friendly as possible, this value is clearly too high. In some cases, it is even claimed that a user waits a maximum of 3 seconds for an application to respond [43].

7.4.2. CPU Performance

On an Android emulator with Android Studio, the CPU usage of the parent system was 3.2 - 3.7% continuously. It was always about 0.5% in idle mode. An impairment of the emulator usage during the computing process could not be determined here.

7.5. Implementation of the functional Requirements

7.5.1. Doctor Login and Verification [FR2], [FR3]

The specification of a doctor can only be done by a simple e-mail verification in the context of this bachelor thesis. For this, one can make use of the already described possibilities of the BaaS Firebase. The plugin that is included in the application to use Firebase and Firestore has a method that allows to send a verification email to the specified user email. As soon as the user follows the sent link he will be marked as verified. Also the login can be realized over the described package, for this already an example from appendix H.4 can be regarded (line 35 ff.)

7.5.2. Creating the Diagnoses

To generate a diagnosis, the user must first work through the process for making a diagnosis. This includes specifying their symptoms and evaluating them. Subsequently, the algorithm calculates the most probable disease. Once this is done, there is the possibility of narrowing down the actual possible diseases based on the probabilities. For example, one can say that only diseases with a probability of at least 20% are included in the diagnosis. These diseases can be described in detail in the diagnosis screen already described. This can be done by simple database queries to the Firestore database, which deliver the diseases as a result. In this way, the fields of the documents can be filtered and displayed on the graphical user interface. In the project additionally a class diagnosis should be created which serves as a template for such a class. The received data fields are assigned to this class. This makes it possible to store objects of this class locally, but also delete them by refering the key.

Saving the Diagnoses [FR7]

In order to store the created instances of the Diagnosis class on the user's smartphone, the methods .toMap(), .encode(), .decode() and .fromJson() should be created in the Diagnosis class.

- **.toMap():**
  The .toMap() method of the class ensures that the object is internally converted to a map with JSON structure. This ensures that this JSON can be converted internally.
- **.encode():**
  When the object is successfully converted to a map, the .encode() method is used. This ensures that the map is converted to a string, since JSON documents are usually long strings. The string just created can now be stored locally using the dependeny shared_preferences [44]. SharedPreferences stores the data in a key-value format. So one could assign a key to each diagnostic and then retrieve it using the key. This is done as described in Listing 7.3 (based on [45]).

```
    Future<void> _saveData() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    await prefs.setString(_diagnoseListKey, _diagnoseListString);
  }

```

Listing 7.3: Storing Data to Locally with SharedPreferences

- **.decode():**
  If one wants to call the diagnosis again, the system only has to convert the saved string into a JSON object. For this purpose the method .decode() is created.
- **.fromJson():**
  The JSON object can now be converted back to the correct instance of the Diagnosis class. The method .fromJson() is used for this purpose. The resulting object can now be used again in the application.

Examples of the methods described can be found in Appendix J.

Generating the PDFs [OR1]

Flutter makes it really easy to create a PDF for the diagnosis. The dependency pdf can be used for this purpose. Among other things, the page format and the content of the PDF can be specified there. By referencing the attributes of the diagnostic elements, the necessary texts can be easily generated and subsequently saved as described in Listing 7.4 (based on [46]).

```
final file = File("diagnose_DD_MM_YYYY.pdf");
await file.writeAsBytes(await pdf.save());
```

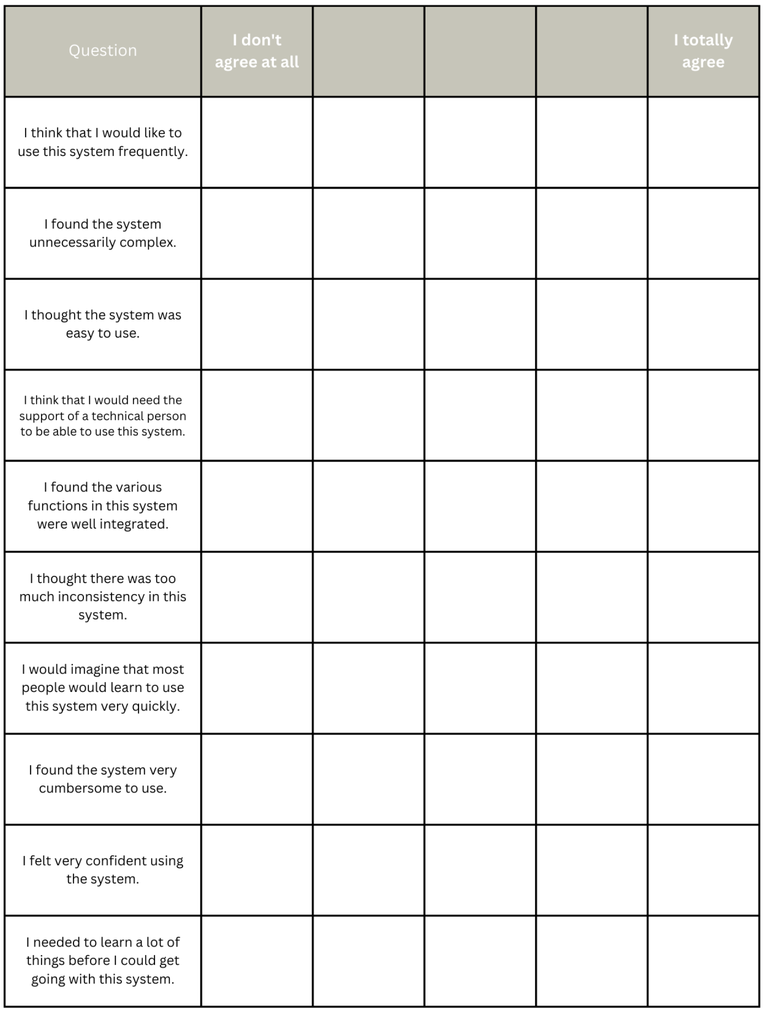
Listing 7.4: Saving a PDF

8. Evaluation

In the Evaluation chapter,the system usability is checked with help of the system usability scale (SUS). The extent to which the system requirements have been met is considered aftward. For the reasons already mentioned, the evaluation of the correctness of the diagnoses is not carried out in this work.

8.1. System Usability Scale

The SUS makes it possible to determine the usability of an application by means of a questionnaire [47]. The application is currently still in an early stage of development, as the focus was on the conception and a possible implementation during the work. The questions are asked based on graphical interfaces. The following questions [47] are asked of anonymous persons:

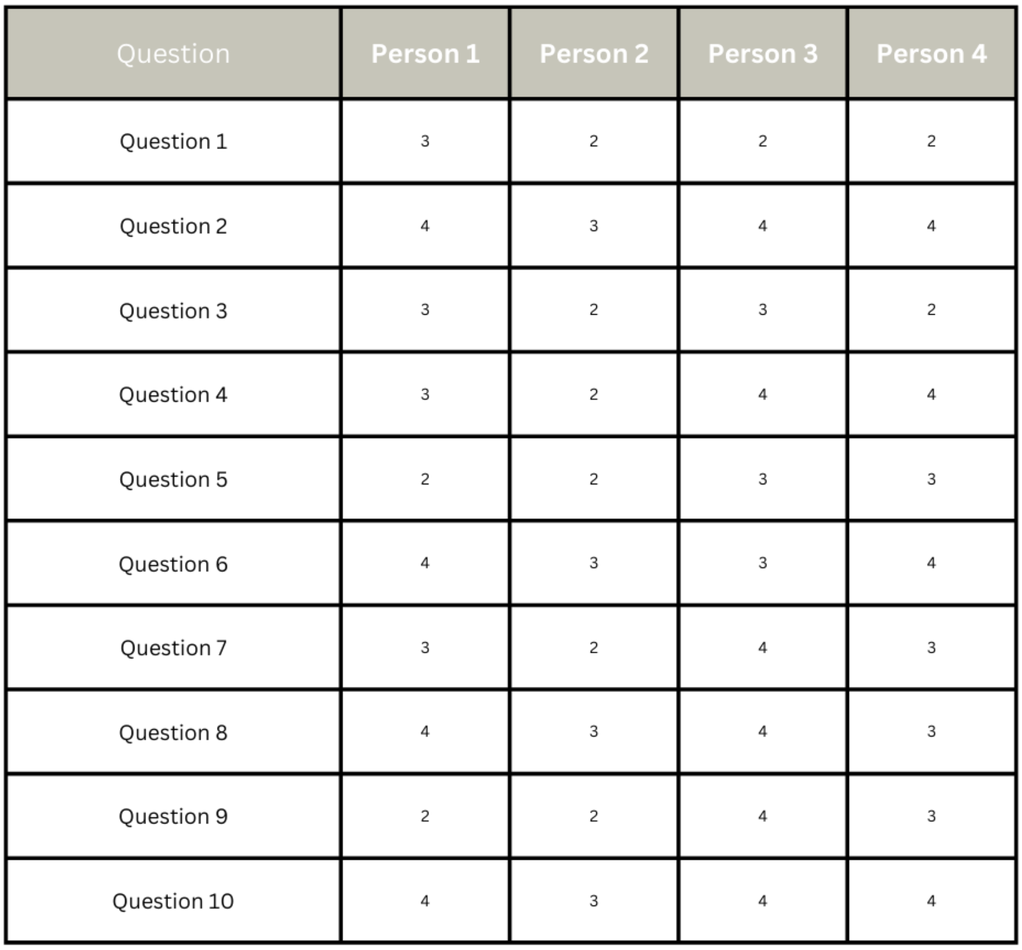


Question	I don't agree at all				I totally agree
I think that I would like to use this system frequently.					
I found the system unnecessarily complex.					
I thought the system was easy to use.					
I think that I would need the support of a technical person to be able to use this system.					
I found the various functions in this system were well integrated.					
I thought there was too much inconsistency in this system.					
I would imagine that most people would learn to use this system very quickly.					
I found the system very cumbersome to use.					
I felt very confident using the system.					
I needed to learn a lot of things before I could get going with this system.					

Figure 8.1.: SUS Template

Evaluation of the SUS

The possible answers are given values from 0 to 4, where 0 equals "I don't agree at all". Based on that the following values are the result:



Question	Person 1	Person 2	Person 3	Person 4
Question 1	3	2	2	2
Question 2	4	3	4	4
Question 3	3	2	3	2
Question 4	3	2	4	4
Question 5	2	2	3	3
Question 6	4	3	3	4
Question 7	3	2	4	3
Question 8	4	3	4	3
Question 9	2	2	4	3
Question 10	4	3	4	4

Figure 8.2.: SUS Answers

The SUS score is calculated as described in [47] (all answers are added up and multiplied by 2,5). The following values are created for the individual persons:

- Person 1: 80 %
- Person 2: 60 %
- Person 3: 87,5 %
- Person 4: 80 %

Altogether this results in a SUS score of 76.875%. This can be rated as above average [47] and the usability of the system, based on the current design status, can be considered satisfactory.

8.2. Requirements

- **[OR1]: The application should make it possible to save and download a diagnosis in PDF format**
  This requirement can be fulfilled as described in chapter 7.5.2.

- **[OR2]: The application should make it possible for a user to save tips on a favorites list**
  Similar to the methodology of how diagnoses are stored in the system, it is also possible to store advice. These also get their own class whose instances can be stored in a list and using methods described in 7.5.2, the data is converted and stored as shown in Listing 7.3.

- **[FR1]: The application must allow the user to switch between the diagnostics and the advices view**
  Switching between screens can be implemented via the described AppNavigator.

- **[FR2]: The application must offer the user the option of being able to verify themselves as a doctor, and and [FR3]: The application must offer a doctor the opportunity to log in**
  The steps taken for this were described in detail in Chapter 7.5.1. However, it should be noted that another method should be used for the verification of doctors.

- **[FR4] and [FR5]: The application must offer a doctor the opportunity to add a new record in the database and to edit old datasets**
  To enable the doctors to do this, the data models described in Chapter 7.2 are used. The doctor can use this to create new objects and add them to the database using the DatabaseService described in Chapter 7.1. The database service also ensures that the data that is already available is displayed to the doctor and can be edited.

- **[FR6]: The application must allow a user to start a new diagnosis**
  This is done by clicking on the button provided for this purpose on the start page, which has also been marked with a suitable text.

- **[FR7]: The application must enable a user to save a diagnosis**
  This requirement can also be met using the methodology described in 7.5.2.

- **[FR8]: The application must enable a user to view his saved diagnoses again**
  By accessing the data entries stored in the SharedPreferences, the system can access the associated key and display the data accordingly on the screen.

- **[FR9]: The application must allow a user to abort his diagnostic procedure at any time, and [FR11]: The application must allow a doctor to abort adding or editing data at any time**
  This is given by clicking on the "back" button indicated in mock-up x. This is displayed to the user throughout the whole process.

- **[FR10]: The application must allow a user to delete saved diagnoses**
  SharedPreferences allows it to access and delete saved data.

- **[NFR1]: The application must make correct diagnoses**
  In the context of this work, this requirement cannot be met because there is no correct data for the calculation. As already mentioned, dummy data was used here.

- **[NFR2]: The application must be usable for patients without registration**
  This is ensured because the user can access all the views he needs without restrictions and no restrictions have been integrated there. Only the Doctor Panel has to be blocked for users who are not verified and logged in using the login mechanism.
- **[NFR3]: The application should have a graphical interface that can be used intuitively**
  The results from the second survey show that the graphical interface is generally intuitive to use.

In general, it can be said that most requirements can actually be implemented. The use of the Flutter framework in particular allows an easy implementation.

9. Conclusion and Outlook

Based on the growth of interest regarding health issues and the contexts of busy medical practices discussed, this work was generated as a template for current and future medical apps. The aim of the present work "Development of a mobile application for the algorithmic attribution of symptoms to potential diseases" was to design a smartphone application for the described situation and to generate first development ideas with the framework Flutter and to evaluate the generated results. The main aim was to design a platform that would relieve the burden on doctors' offices and reassure worried patients.

By using the personas, the requirements and also the use cases that can occur during the use of the application could be determined. The survey conducted at the initial stage of the work also helped to obtain more precise information on non-functional and functional requirements. In the context of this work, it made sense to interact with the potential user group at an early stage and to involve them in the development and planning process. However, the doctor verification should be discussed in the future. As part of the bachelor thesis, only the possibility of e-mail verification was discussed. However, this should not be used in the real world, since any user could do this, even if he is not a doctor. One possibility would be to provide manual verification, which allows the app developers themselves, or a designated authority, to create and verify doctor accounts. In cooperation with the right authorities, it can be ensured that only the right people have access to the medical area.

A positive aspect during the processing was, among other things, that almost all functional and optional requirements can be fulfilled. The use of the Flutter framework is particularly helpful here, as many functionalities can be implemented with fewer lines of code thanks to the wide range of different dependencies.

By determining the system context, it could be recognized that an external API will also affect the system. In the following, two different APIs were evaluated and a decision was made. In the following it can be said that the choice of the ApiMedic API made sense for this work. However, it should be considered in the future whether to take the step of scraping the data from the NHS API. These turn out to be more detailed and could lead to a more detailed application. On the other hand, it can be argued that the database should be expanded with data sets created by doctors.

With the generated domain model and the data of the API, classes for implementation could already be generated and a database structure determined. For a simplified representation within this work, only the names were used as ID for the documents within the firestore database. In reality, however, this is not recommended. A hash functionality would be more suitable for this to ensure that nobody from outside can manipulate the diseases and symptoms. The classes created and also the structuring of the project, which was carried out in Chapter 6, help to implement clean design principles, such as the SOLID principles.

As part of the first survey mentioned, it was found that the graphical user interface of the application plays an important role. Accordingly, design principles were considered and created based on these first conception ideas. These were equipped with implementation proposals for Flutter. Subsequently, mock-ups could already be generated. These were evaluated in a second survey for their user-friendliness and intuitiveness. It turned out that the design solutions provided by the respondents were mostly intuitive. As part of an analysis using SUS, it was also found that the graphical user interface is above average in terms of usability. With regard to the diagnosis process, the graphical user interface could be improved. Currently, the user is forced to select their body parts and symptoms by scrolling through large lists. It would be possible to simplify this using a search bar. However, the user would have to know the name of his symptom for this, which proves to be difficult with technical terms. For this reason, the search algorithm must be expanded to such an extent that it can also assign related terms.

A similar search facility should also be made available to doctors to assist them in their search for specific symptoms and diseases.

Finally, possible algorithmic solutions for the implementation of the diagnosis were compared. It could not be determined whether the algorithm that was chosen has a high rate of correctness. A bad point, however, was that the time it took to perform the calculation turned out to be too high for a normal user, even though the CPU usage turned out to be low. This long calculation time can probably also be attributed to the queries to the database, since these can also be time-consuming with a large number of queries. Regarding the algorithm, the dummy data with which the application is currently working should be replaced with real data. It is also urgently necessary to optimize the calculation time. Here you can think about describing symptoms in more detail. For example, the user could be asked questions about their reported symptoms, possible causes (e.g., not drinking enough, not sleeping, medications the patient is taking, etc.). Based on these selections, certain diseases may be excluded in advance while others gain in importance. In addition, medical professionals should be consulted to ensure the correctness of the diagnoses.

In general, it can be said that a user-friendly application can be developed using the approaches discussed. With regard to the performance of disease detection, however, improvements should clearly be made. It is not possible to predict whether a change in the data structure will be necessary for this.

A. Details of Survey 1

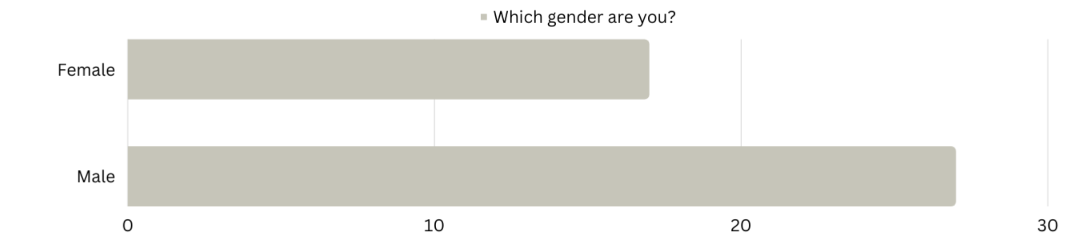


Figure A.1.: Survey 1, Question 1

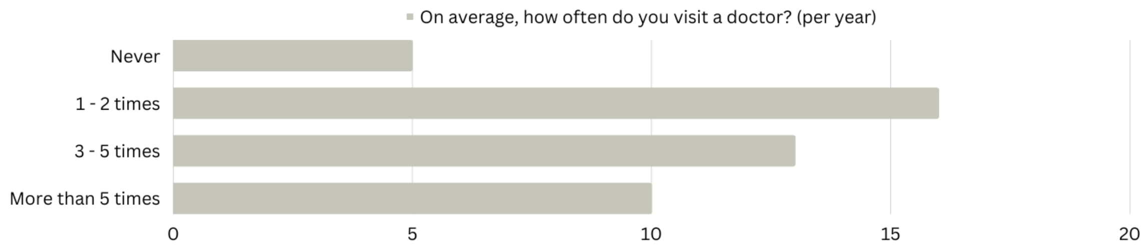


Figure A.2.: Survey 1, Question 2

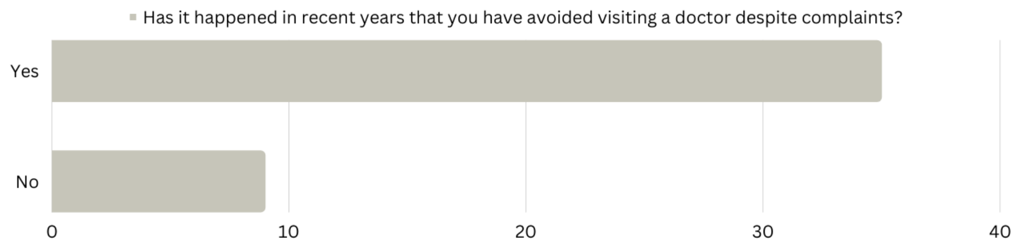


Figure A.3.: Survey 1, Question 3

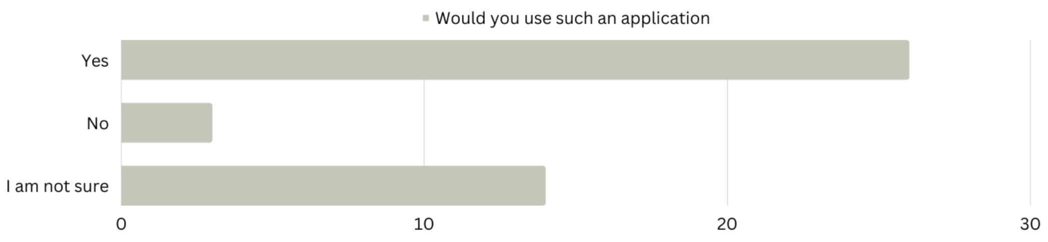


Figure A.4.: Survey 1, Question 4

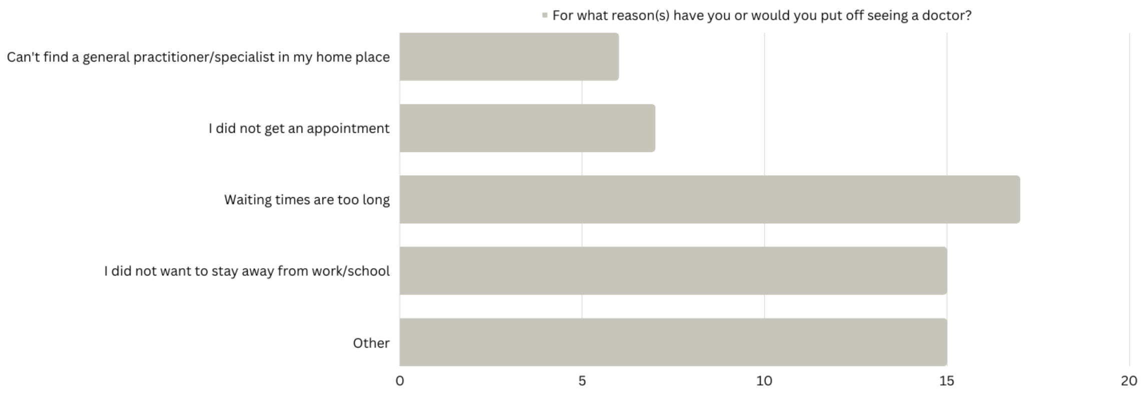


Figure A.5.: Survey 1, Question 5

Below are the answers given for the "other" option.

- I thought I would make it without a doctor
- I had no time to visit a doctor
- Too much effort to get an appointment
- Was able to assess and treat my symptoms myself
- Social Anxiety
- Know nurse and can ask there for small problems

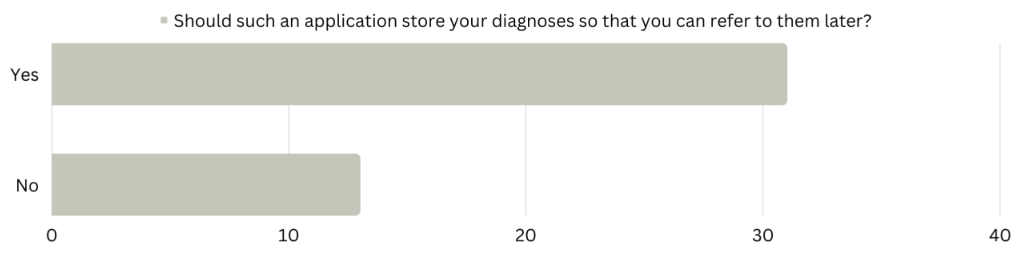


Figure A.6.: Survey 1, Question 6

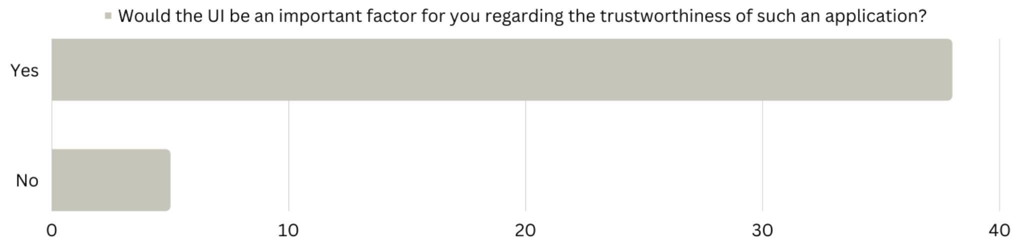


Figure A.7.: Survey 1, Question 7

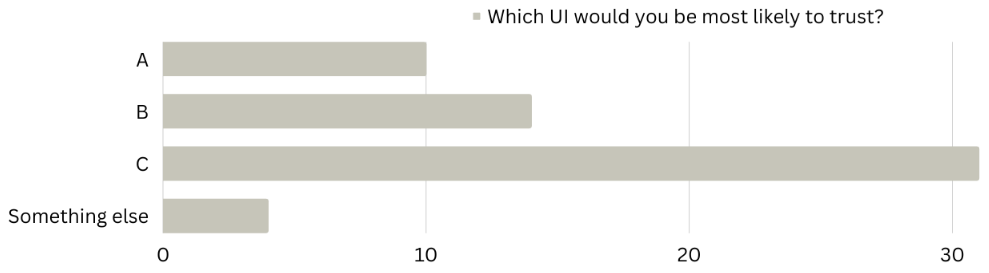


Figure A.8.: Survey 1, Question 8

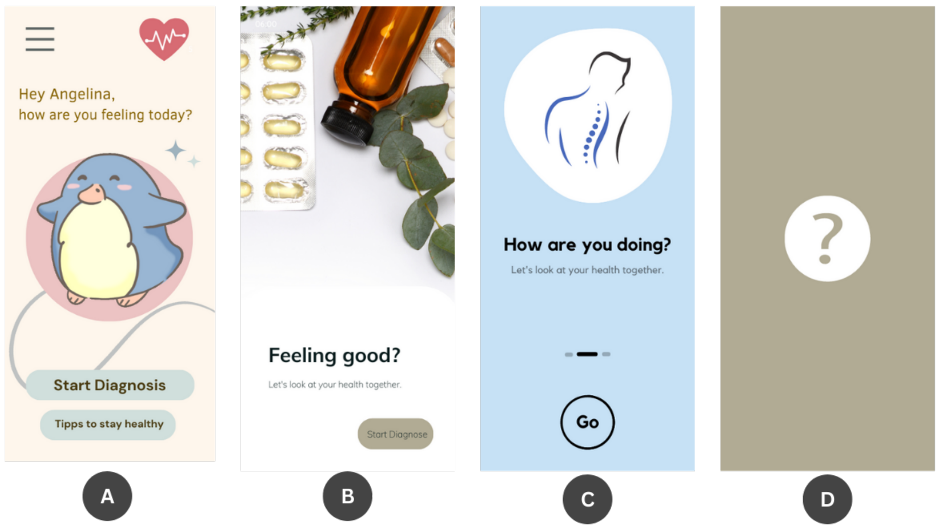


Figure A.9.: Survey 1, Question 8 (Pictures)

B. Code Snippet of Widget Tree

```
Widget build ( BuildContext context ) {
   return Scaffold(
     appBar: AppBar (
      title : const Text('Example of the build method') ,
     ),
     body : const Center(
      child : Text('Hello Reader') ,
     ),
   );
}
```

Listing B.1: Code Snipped of Widget Tree

C. Overview of the Requirements

C.1. Optional Requirements

ID	Description
OR1	The application should make it possible to save and download a diagnosis in PDF format
OR2	The application should make it possible for a user to save tips on a favorites list

Table C.1.: Optional Requirements

C.2. Function Requirements

ID	Description
FR1	The application must allow the user to switch between the diagnostics and the advices view
FR2	The application must offer the user the option of being able to verify themselves as a doctor
FR3	The application must offer a doctor the opportunity to log in
FR4	The application must offer a doctor the opportunity to add a new record in the database
FR5	The application must offer a doctor the possibility of editing data records in the database
FR6	The application must allow a user to start a new diagnosis
FR7	The application must enable a user to save a diagnosis
FR8	The application must enable a user to view his saved diagnoses again
FR9	The application must allow a user to abort his diagnostic procedure at any time
FR10	The application must allow a user to delete saved diagnoses
FR11	The application must allow a doctor to abort adding or editing data at any time

Table C.2.: Functional Requirements

C.3. Non-functional Requirements

ID	Description
NFR1	The application must make correct diagnoses
NFR2	The application must be usable for patients without registration
NFR3	The application should have a graphical interface that can be used intuitively

Table C.3.: Non-Functional Requirements

c

Name	Get Diagnosis **[FR6]**
Description	The user wants to get a diagnosis, based on their symptoms
Result	The user receives a diagnosis
Actors	User, Doctor
Trigger	The user clicks on the new diagnosis button
Preconditions	None
Steps	1. The user clicks on the new diagnosis button 2. The system displays the view to enter and specify symptoms 3. The user selects his symptoms and specifies them 4. The user indicates that he is finished 5. Diagnosis: the system determines possible diseases and presents them to the user and the use case ends
Alternate flow	AF1a. The user wants to cancel the diagnosis and presses the stop button **[FR10]** AF1b. The system returns to the main page of the application AF2a. The user wants to save the diagnosis **[FR7]** AF2b. The user presses the save button Af2c. The system saves the diagnosis

Table C.4.: Use case get diagnosis

Name	Review received diagnosis **[FR8]**
Description	The user wants to review a diagnosis one more time
Result	The system shows the selected diagnosis to the user
Actors	User, Doctor
Trigger	Click on the diagnosis
Preconditions	The user has previously saved the diagnosis
Steps	1. The user selects the diagnosis from a list of previously stored diagnoses 2. The system shows the selected diagnosis to the user 3. When finished the user clicks on the back button
Alternate flow	AF1a. The user wants to delete the diagnosis **[FR12]** AF1b. The user clicks on the delete button AF1c. The system deletes the diagnosis AF2a. The user wants to download the diagnosis as PDF AF2b. The user presses the download button

Table C.5.: Use case review received diagnoses

Name	Get Tips and Tricks for Symptoms and Diseases **[FR13]**
Description	The user wants to see tips and tricks regarading their symptoms
Result	The system shows the tip-view to the user
Actors	User, Doctor
Trigger	Click on the tip tab
Preconditions	None
Steps	1. The user selects the tip tab on the bottom navigation bar 2. The system displays the tip-dashboard 3. The user clicks on a tip to see the whole tip-description 4. The system displays the tip-detail-page and the use case ends
Alternate flow	AF1a. The user wants to add a tip to his favorites **[OR2]** AF1b. The user clicks on the favorite icon of the tip AF1c. The system saves the tip to the users favorites

Table C.6.: Use case get tups and tricks for sympstoms and diseases

Name	Login **[FR3]**
Description	The user, a doctor, wants to log into the application
Goal	The doctor successfully logged into the system
Actors	Doctor
Trigger	Click on the login button
Preconditions	User is verified as a doctor **[FR3]**
Steps	1. The doctor clicks on the login button 2. The systems shows the login form 3. The doctor enters his personal details and presses the okay button 4. The system checks for the credentials in the database 5. The system displays the Add screen and the use case ends
Alternate flow	AF1a. The system could not find the given credentials in the database AF1b. The user entered wrong credentials AF1c. The system displays an error message AF1d. The user retries AF2a. The user is not verfied as doctor yet **[FR3]** AF2b. The doctor enters his personal details and presses the okay button AF2c. The system starts the verification method AF2d. The doctor is verified as doctor AF2e. The system displays the Add screen and the use case ends
Alternate flow (failure)	AFF1a. The user is no doctor AFF1b. The user is not able to verify himself as doctor AFF1c. The system shows an error

Table C.7.: Use case login

Name	Add data to the databas **[FR4]**
Description	The actor wants to add new data to the database
Goal	The data is added to the database
Actors	Doctor, Developer
Trigger	Click on the addData button
Preconditions	Actor is logged in
Steps	1. The actor clicks on the addData button 2. The system shows the add form 3. The actor enters the required data and presses the ok button 4. The system adds the disease, symptom, cause or tip to the database
Alternate flow	AF1a. The actor missed to enter data AF1b. The system displays an error message AF1c. The actor retries AF2a. The actor wants to cancel the process **[FR12]** AF2b. The actor clicks on the cancel button AF2c. The system closes the add form

Table C.8.: Use case add data

Name	Edit Data **[FR5]**
Description	The actor wants to edit old data
Goal	The edited data is uploaded to the database
Actors	Doctor, Developer
Trigger	Click on the edit button
Preconditions	Actor is logged in
Steps	1. The actor clicks on the edit button on the data he wants to edit 2. The system shows the edit form 3. The actor edits the data and presses the okay button 4. The system updates the data and the use case ends
Alternate flow	AF1a. The actor wants to cancel the process **[FR12]** AF1b. The actor clicks on the cancel button AF1c. The system closes the edit form

Table C.9.: Use case edit data

D. Response of the APIs

D.1. Response of the NHS API

The total length of an NHS response from the server is a JSON with approximately 466 lines. Because of this, only the abbreviated response from the server is shown here. A full response can be viewed by scanning the QR code shown below.

```json
{
  "@context":"http://schema.org",
  "@type":"MedicalWebPage",
  "name":"Acne",
  "copyrightHolder":{...},
  "license":"https://developer.api.nhs.uk/terms",
  "author":{...},
  "about":{...},
  "description":"...",
  "url":"https://api.nhs.uk/conditions/acne/",
  "genre":[...],
  "keywords":[...],
  "lastReviewed":[...],
  "breadcrumb":{...},
  "dateModified":"2022-05-30T14:30:18+00:00",
  "hasPart":[...],
  "relatedLink":[...],
  "contentSubTypes":[...],
  "mainEntityOfPage":[...],
  "alternativeHeadline":"Overview"
}
```

Listing D.1: Abbreviated Response of NHS API

Figure D.1.: QR-Code NHS API Response, GitHub

The QR code redirects to a GitHub repository created for this bachelor thesis. If the code does not work anymore due to an error, you can simply go to the following address in your web browser:

`https://github.com/petzolan/disease_detection/blob/main/data.json`

D.2. Response of the ApiMedic API

Responses to requests to the ApiMedic API always look similar. The response for the request to retrieve all diseases (abbreviated) and a response regarding an explicit disease are shown below.

All diseases:

```
[
  {
    "ID":130,
    "Name":"Abdominal hernia"
  },
  {
    "ID":170,
    "Name":"Abortion"
  },
  {
    "ID":456,
    "Name":"Abscess of the tonsils"
  },
  {
    "ID":577,
    "Name":"Absence seizure"
  },
  {
    "ID":584,
    "Name":"Accident injury"
  },
...
```

Listing D.2: Abbreviated Reponse of all Diseases ApiMedic API

Disease with the ID 105:

```
{
  "Description": "Measles is caused by a virus and is very contagious. Measles
    can be very unpleasant and can lead sometimes to serious complications.
    Anyone can get measles if he has not been vaccinated. People who did not have
     measles before can also get it if they come in contact with an infected
    person. However, the condition is most common in young children. The
    infection disappears usually in around 7 to 10 days.",

  "DescriptionShort": "Measles, also known as morbilli, is a viral infection that
     commonly affects children and causes a skin rash, fever, and swelling of the
     lymph nodes. It is very contagious and there is a vaccine for it.",

  "MedicalCondition": "The infection begins with flu-like symptoms (runny nose,
    fever, cough) and subsequently develops into a fever and a rash with large
    spots that spread from the head downward. The lymph nodes are also swollen
    enough to be felt. Because the measles weakens the immune system, other
    dangerous secondary infections of the lung or brain can develop. The measles
    can remain undetected for several years after the initial attack and attack
    the brain, which can lead to death.",

  "Name": "Measles",

  "PossibleSymptoms": "Burning eyes,Burning in the throat,Cough,Eye redness,Fever
    ,Itching eyes,Pain in the limbs,Runny nose,Skin rash,Sore throat,Swollen
    glands in the armpit,Swollen glands in the groin,Swollen glands in the neck,
    Tiredness,Oversensitivity to light,Facial swelling,Flaking skin",
  "ProfName": "Morbilli",

  "Synonyms": "Red measles",

  "TreatmentDescription": "To prevent measles, an effort to vaccinate the entire
    population is being undertaken to protect from the consequences of the
    infection. Two shots are required, the first one at 12 months of age and the
    second at 15-24 months. The shot is combined with other vaccines for measles,
     mumps, and rubella (MMR). If an infection with measles occurs, the infected
    child should not be allowed contact with other children for 5 days after the
    outbreak of the rash to prevent the infection from spreading.  People who
    have come in contact with the infected child (classmates, for example) and
    who have not had measles or been vaccinated against them should stay at home
    for 14 days for the same reason. There is no treatment for measles. The only
    protection against measles is the vaccination or a previous measles infection
    ."
}
```

Listing D.3: Response ApiMedic API (issue ID 105)

E. Jupyter Notebook (QR-Code)

Figure E.1.: QR-Code JupyterNotebook, GitHub

The QR code redirects to a GitHub repository created for this bachelor thesis. If the code does not work anymore due to an error, you can simply go to the following address in your web browser:

`https://github.com/petzolan/disease_detection/tree/main/JuypterNotebook`

F. Details of Survey 2

G. Mock-ups

G.1. Home Screen

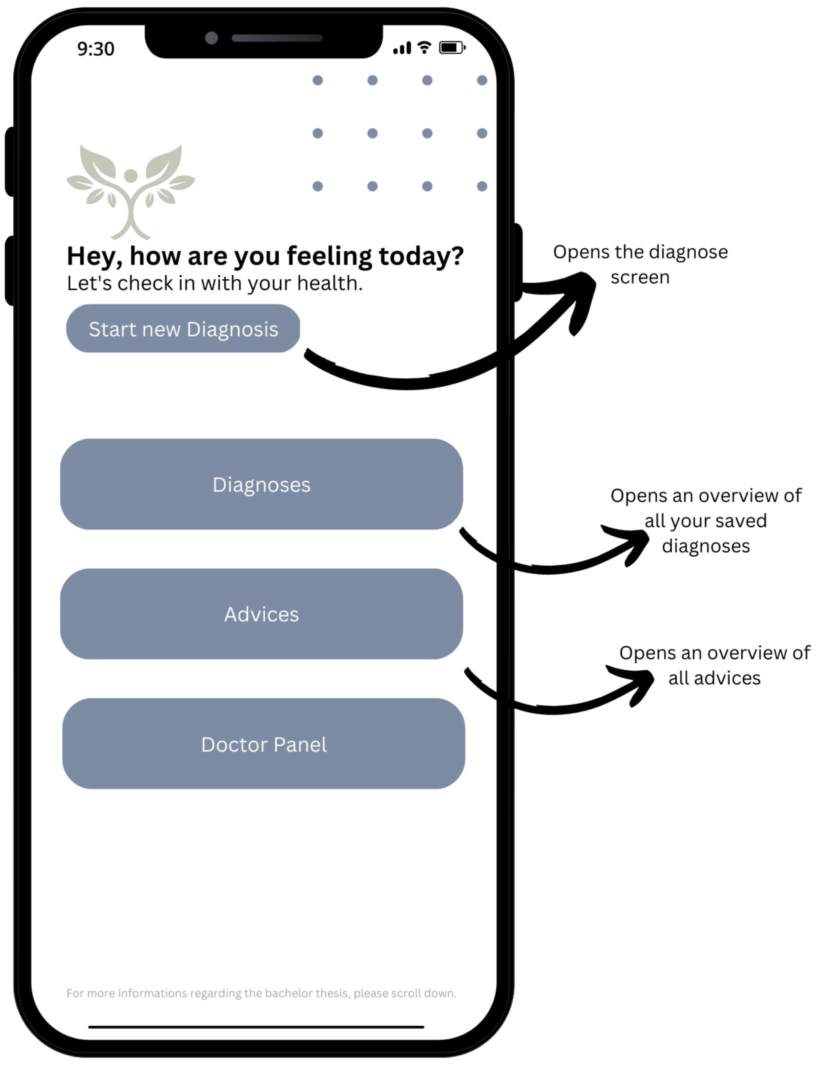
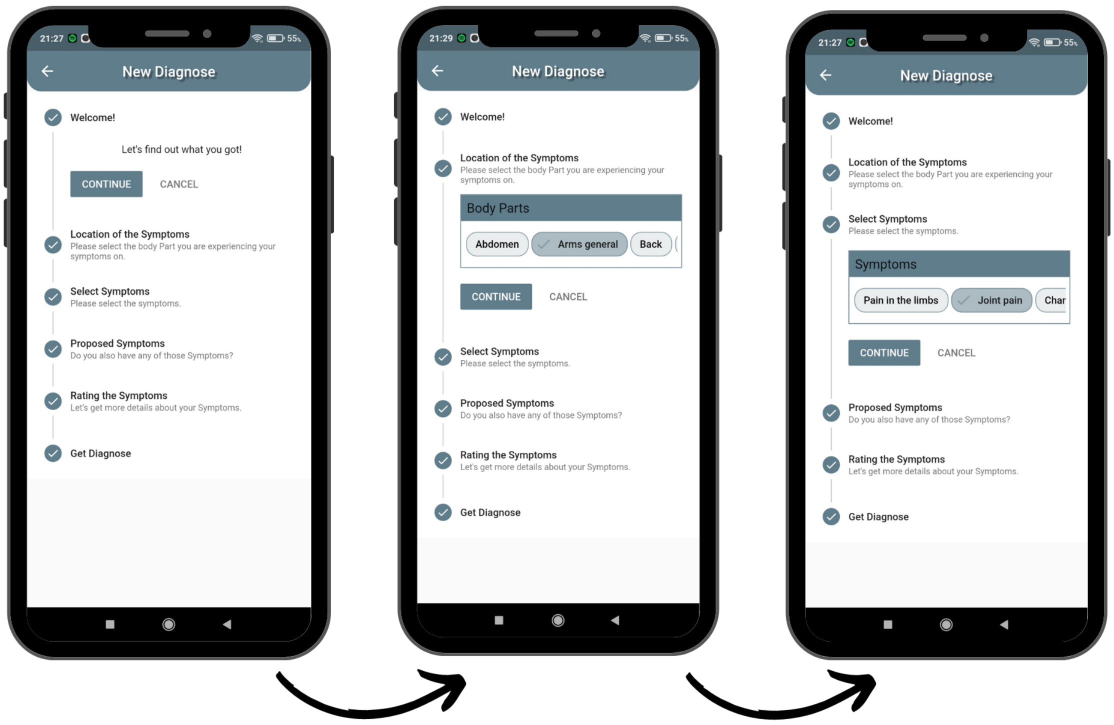

9:30
Hey, how are you feeling today?
Let's check in with your health.
Start new Diagnosis
Diagnoses
Advices
Doctor Panel
For more informations regarding the bachelor thesis, please scroll down.
Opens the diagnose screen
Opens an overview of all your saved diagnoses
Opens an overview of all advices

Figure G.1.: Mock-up Home Screen

G.2. Diagnostic Process Screen


21:27 55%

New Diagnose

Welcome!

Let's find out what you got!

CONTINUE CANCEL

Location of the Symptoms
Please select the body Part you are experiencing your symptoms on.

Select Symptoms
Please select the symptoms.

Proposed Symptoms
Do you also have any of those Symptoms?

Rating the Symptoms
Let's get more details about your Symptoms.

Get Diagnose

21:29 55%

New Diagnose

Welcome!

Location of the Symptoms
Please select the body Part you are experiencing your symptoms on.

Body Parts

Abdomen | Arms general | Back

CONTINUE CANCEL

Select Symptoms
Please select the symptoms.

Proposed Symptoms
Do you also have any of those Symptoms?

Rating the Symptoms
Let's get more details about your Symptoms.

Get Diagnose

21:27 55%

New Diagnose

Welcome!

Location of the Symptoms
Please select the body Part you are experiencing your symptoms on.

Select Symptoms
Please select the symptoms.

Symptoms

Pain in the limbs | Joint pain | Char

CONTINUE CANCEL

Proposed Symptoms
Do you also have any of those Symptoms?

Rating the Symptoms
Let's get more details about your Symptoms.

Get Diagnose

Figure G.2.: Mock-up Diagnostic Process Screen

G.3. Diagnose Screen

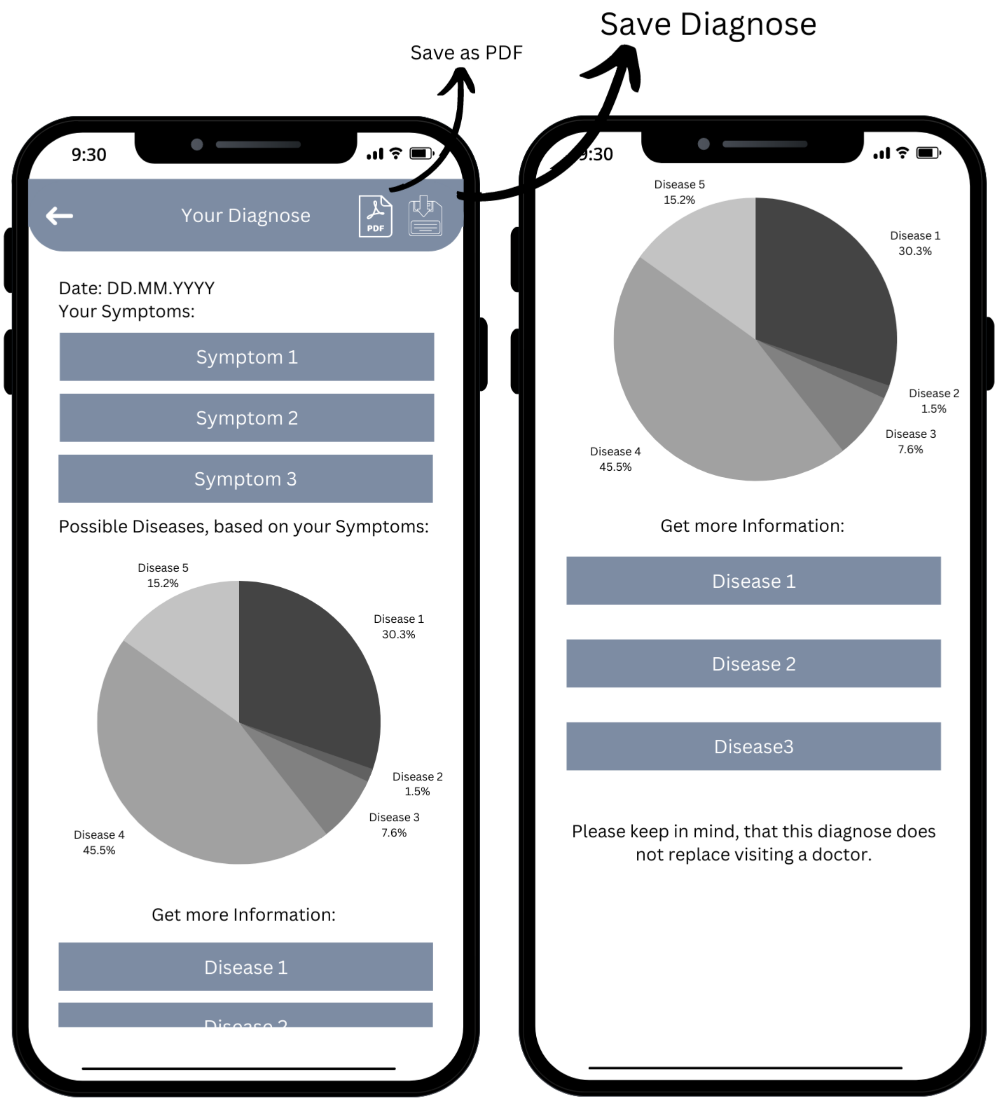


Figure G.3.: Mock-up Diagnose Screen

G. Mock-ups

G.4. Diagnose Overview Screen

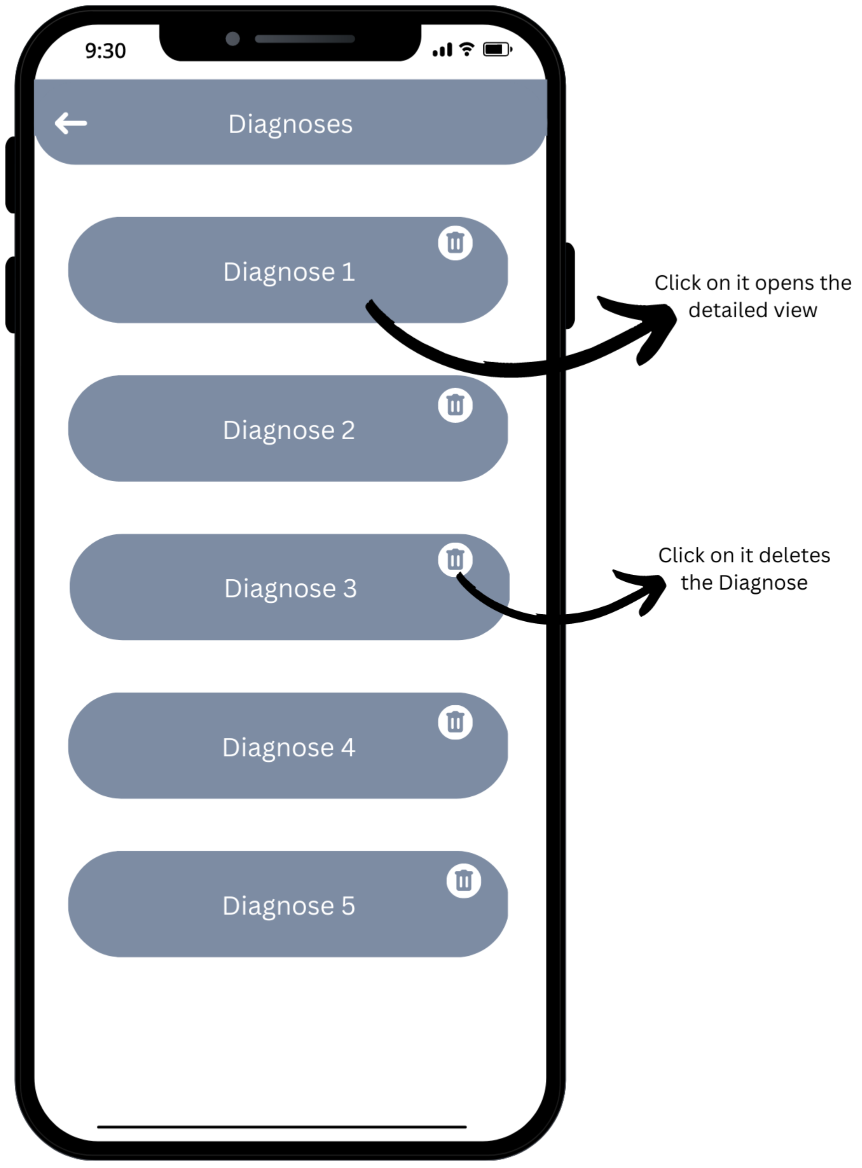

9:30
Diagnoses
Diagnose 1
Diagnose 2
Diagnose 3
Diagnose 4
Diagnose 5
Click on it opens the detailed view
Click on it deletes the Diagnose

Figure G.4.: Mock-up Diagnose Overview Screen

G.5. Advice Overview Screen

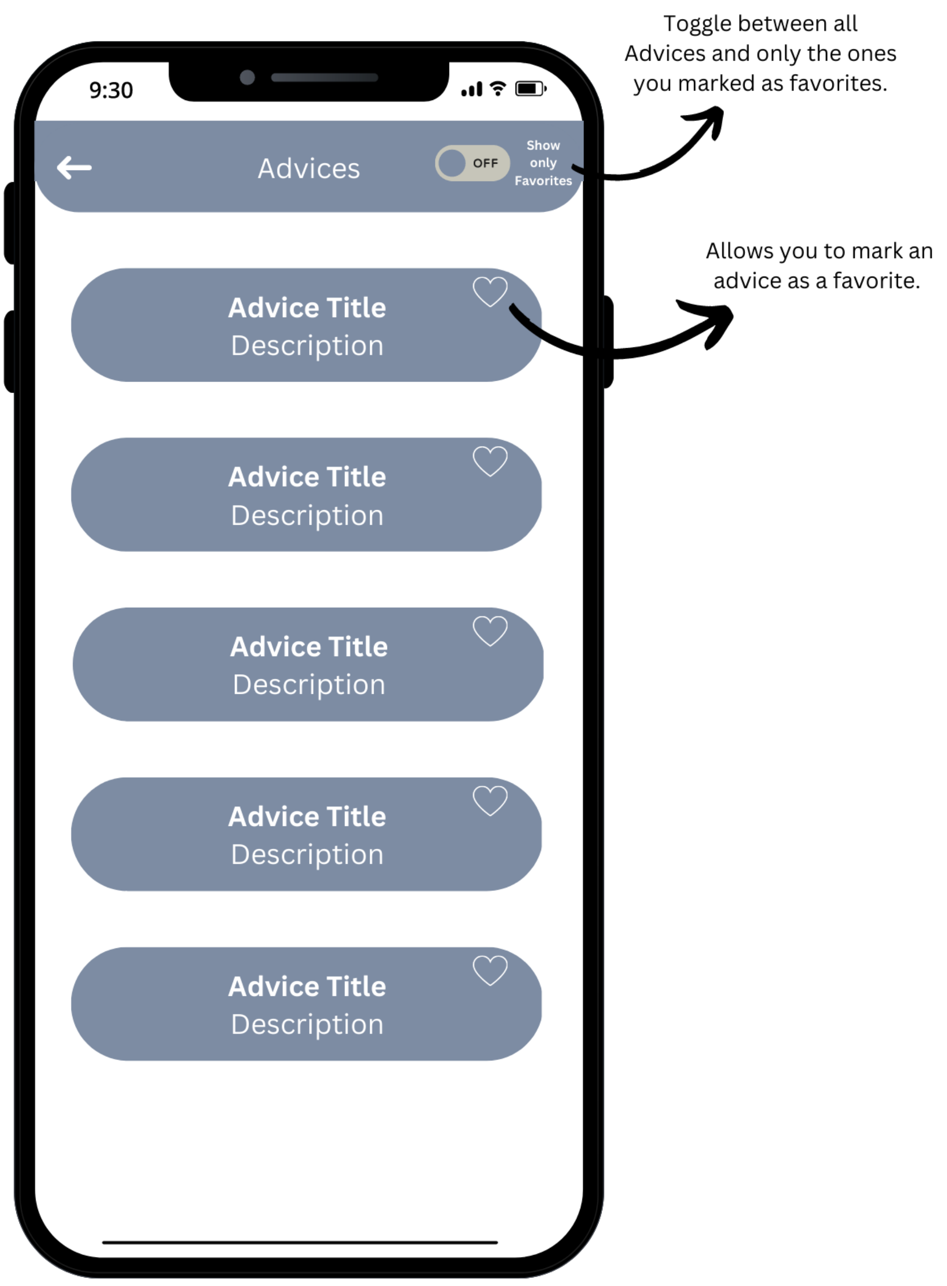


Figure G.5.: Mock-up Advice Screen

G.6. Login Screen

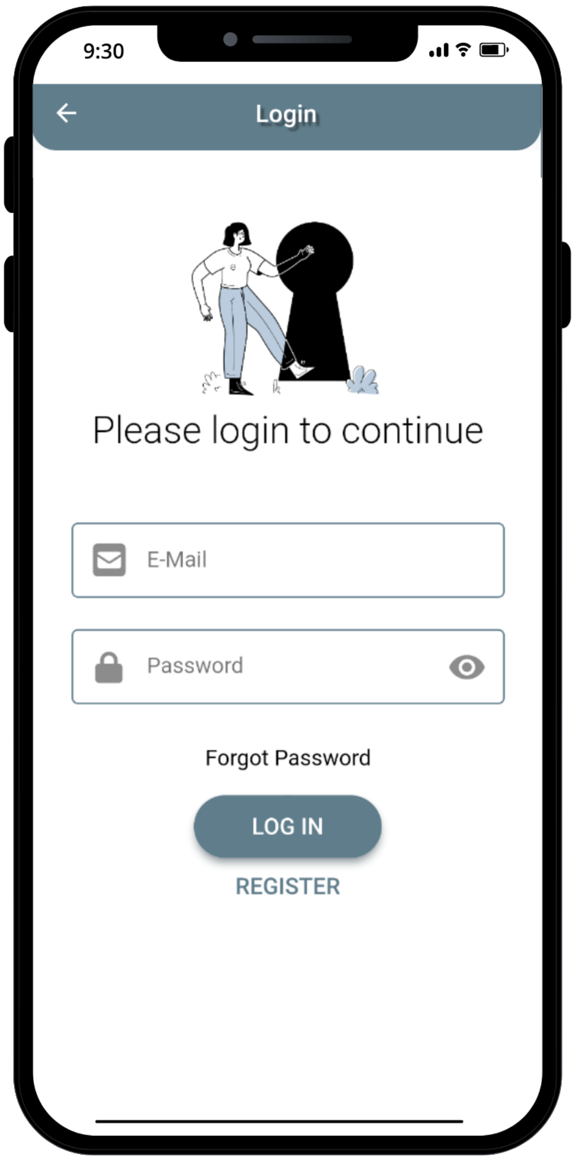


Figure G.6.: Mock-up Login Screen

G.7. Doctor Panel Screen

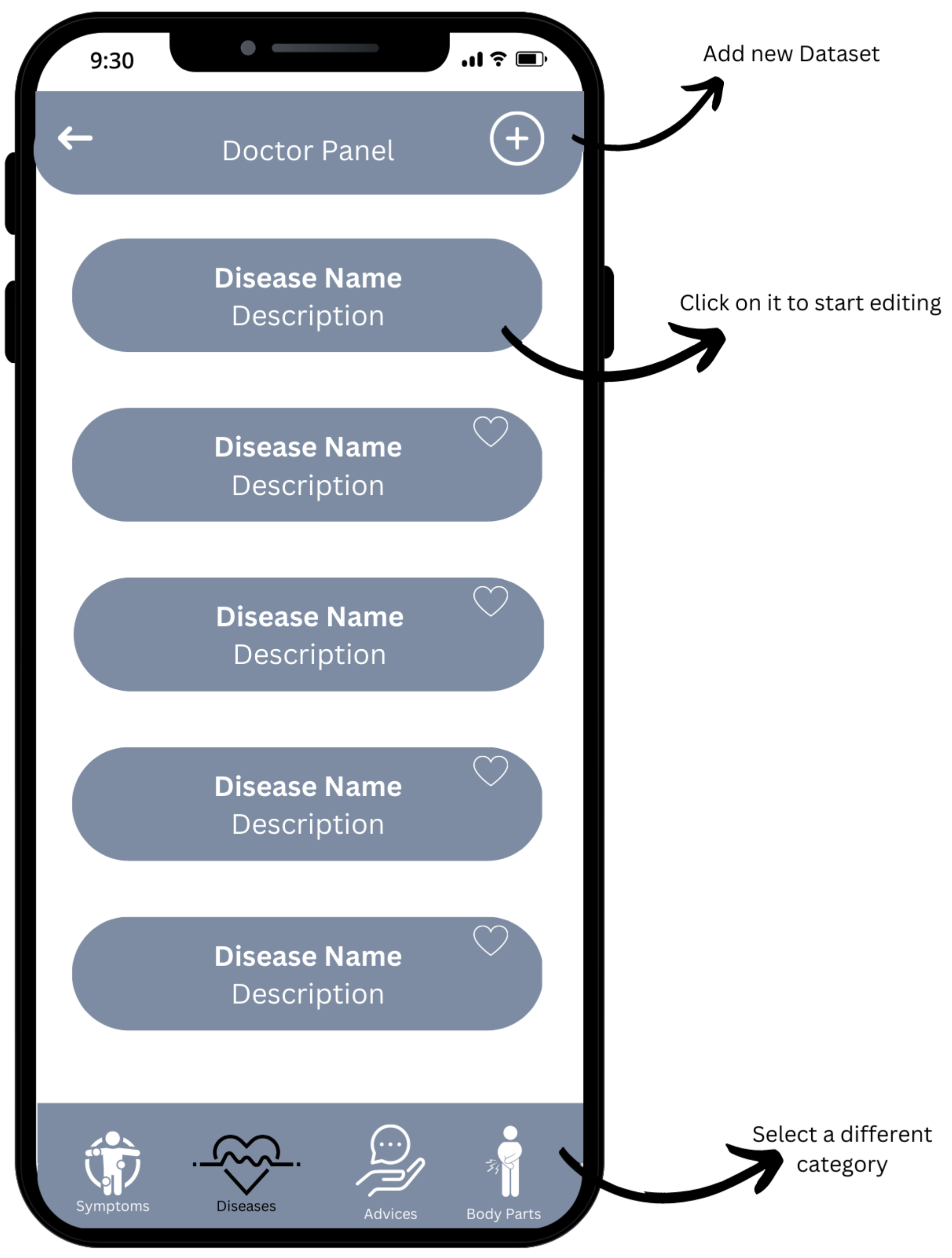


Figure G.7.: Mock-up Doctor Panel

H. Implemented Services

H.1. DatabaseService

```
class DatabaseService {
  /* CREATE AN INSTANCE OF THE DATABASE */
  DatabaseService._();
  static DatabaseService _instance = DatabaseService._();
  static DatabaseService get instance => _instance;

  /* GET REFERENCES */
  final CollectionReference _bodyPartReference =
  FirebaseFirestore.instance.collection('body_parts');

  final CollectionReference _symptomCollectionReference =
  FirebaseFirestore.instance.collection('symptoms');

  final CollectionReference _diseasesCollectionReference =
  FirebaseFirestore.instance.collection('diseases');

  final CollectionReference _adviceCollectionReference =
  FirebaseFirestore.instance.collection('advices');

  final CollectionReference _doctorCollectionReference =
  FirebaseFirestore.instance.collection('doctors');
  /* GET BODY PART DATA */
  getAllBodyParts() {
    return _bodyPartReference.snapshots();
  }

  Future<List<BodyPart>> retrieveBodyParts() async {
    QuerySnapshot<Map<String, dynamic>> snapshot =
    await _bodyPartReference.get() as QuerySnapshot<Map<String, dynamic>>;
    return snapshot.docs
    .map((docSnapshot) => BodyPart.fromDocumentSnapshot(docSnapshot))
    .toList();
  }

// MORE METHODS
...
```

Listing H.1: DatabaseService Code

H.2. ConvertService

```
class ConvertService {
  /* CREATE AN INSTANCE OF THE DATABASE */
  ConvertService._();
  static ConvertService _instance = ConvertService._();
  static ConvertService get instance => _instance;

  // SYMPTOMS
  Future<List<MultiSelectItem<String>>> getAllSymptoms() async {
    var data = await DatabaseService.instance.retrieveSymptoms();
    List<MultiSelectItem<String>> symptoms = [];
    for (var element in data) {
      symptoms.add(
      MultiSelectItem(element.name, element.name),
      );
    }
    return symptoms;
  }
// MORE METHODS
...
```

Listing H.2: AppNavigator Class

H.3. AppNavigator (NavigationService)

```
enum Routes { survey, home, advices, doctorPanel, detailScreen, diagnosesList }

class _Paths {
  static const String survey = '/';
  static const String home = '/home';
  static const String advices = '/home/advices';
  static const String doctorPanel = '/home/doctorPanel';
  static const String detailScreen = '/home/detailScreen';
  static const String diagnosesList = '/home/items';

  static const Map<Routes, String> _pathMap = {
    Routes.home: _Paths.home,
    Routes.survey: _Paths.survey,
    Routes.advices: _Paths.advices,
    Routes.doctorPanel: _Paths.doctorPanel,
    Routes.detailScreen: _Paths.detailScreen,
    Routes.diagnosesList: _Paths.diagnosesList
  };

  static String of(Routes route) => _pathMap[route] ?? survey;
}

class AppNavigator {
  static GlobalKey<NavigatorState> navigatorKey = GlobalKey();
  static Route onGenerateRoute(RouteSettings settings) {
    switch (settings.name) {
      case _Paths.survey:
      return FadeRoute(page: SurveyScreenFlutter());
      case _Paths.advices:
      return FadeRoute(page: AdviceScreen());
      case _Paths.doctorPanel:
      return FadeRoute(
      page: FirebaseAuth.instance.currentUser != null
      ? DoctorPanelScreen(title: "DoctorPanel")
      : LoginScreen(),
      );
      case _Paths.detailScreen:
      return FadeRoute(page: DetailScreen(null));
      case _Paths.diagnosesList:
      return FadeRoute(page: DiagnosesScreen());
      case _Paths.home:
      default:
      return FadeRoute(page: HomeScreen());
    }
  }

  static Future? push<T>(Routes route, [T? arguments]) =>
  state?.pushNamed(_Paths.of(route), arguments: arguments);
  static Future? replaceWith<T>(Routes route, [T? arguments]) =>
  state?.pushReplacementNamed(_Paths.of(route), arguments: arguments);
  static void pop() => state?.pop();
  static NavigatorState? get state => navigatorKey.currentState;
}
```

Listing H.3: AppNavigator Class

H.4. AuthService

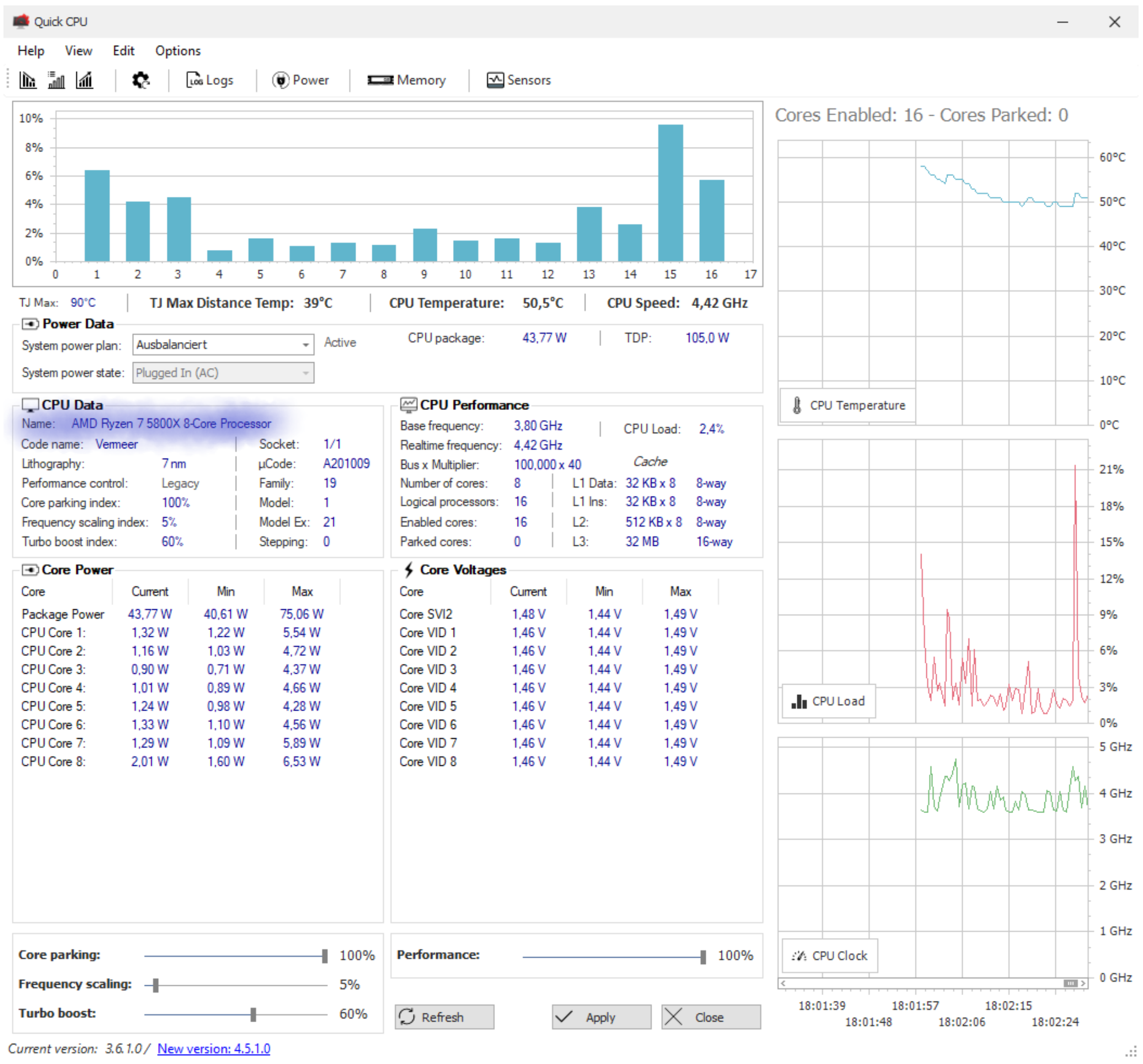
```
class AuthService {
  /* CREATE AN INSTANCE OF THE DATABASE */
  AuthService._();
  static AuthService _instance = AuthService._();
  static AuthService get instance => _instance;

  FirebaseAuth auth = FirebaseAuth.instance;

  Future<String?> registration(SignupData pData) async {
    try {
      UserCredential userCredential =
      await FirebaseAuth.instance.createUserWithEmailAndPassword(
      email: pData.name!,
      password: pData.password!,
      );

      Doctor d = Doctor(
      email: pData.name!,
      createdSymptomsIDs: [],
      createdDiseasesIDs: [],
      createdAdvicesIDs: [],
      );

      DatabaseService.instance.addNewDoctor(
      FirebaseAuth.instance.currentUser!.uid,
      d.toMap(),
      );

      print(FirebaseAuth.instance.currentUser);
    } catch (e) {
      print(e);
    }
  }

  Future<String?> signIn(LoginData pData) async {
    try {
      UserCredential userCredential =
      await FirebaseAuth.instance.signInWithEmailAndPassword(
      email: pData.name,
      password: pData.password,
      );
    } on FirebaseAuthException catch (e) {
      if (e.code == 'user-not-found') {
        print('No user found for that email.');
      } else if (e.code == 'wrong-password') {
        print('Wrong password provided for that user.');
      }
    }
  }
// MORE METHODS
...
```

Listing H.4: AuthService

I. Evaluation of the Algorithm

I.1. Information about the System

I.1.1. CPU



Cores Enabled: 16 - Cores Parked: 0

TJ Max: 90°C | TJ Max Distance Temp: 39°C | CPU Temperature: 50,5°C | CPU Speed: 4,42 GHz

Power Data

System power plan:	Ausbalanciert	Active	CPU package:	43,77 W	TDP:	105,0 W
System power state:	Plugged In (AC)					

CPU Data

Name:	AMD Ryzen 7 5800X 8-Core Processor		
Code name:	Vermeer	Socket:	1/1
Lithography:	7 nm	µCode:	A201009
Performance control:	Legacy	Family:	19
Core parking index:	100%	Model:	1
Frequency scaling index:	5%	Model Ex:	21
Turbo boost index:	60%	Stepping:	0

CPU Performance

Base frequency:	3,80 GHz	CPU Load:	2,4%	
Realtime frequency:	4,42 GHz			
Bus x Multiplier:	100,000 x 40		*Cache*	
Number of cores:	8	L1 Data:	32 KB x 8	8-way
Logical processors:	16	L1 Ins:	32 KB x 8	8-way
Enabled cores:	16	L2:	512 KB x 8	8-way
Parked cores:	0	L3:	32 MB	16-way

Core Power

Core	Current	Min	Max
Package Power	43,77 W	40,61 W	75,06 W
CPU Core 1:	1,32 W	1,22 W	5,54 W
CPU Core 2:	1,16 W	1,03 W	4,72 W
CPU Core 3:	0,90 W	0,71 W	4,37 W
CPU Core 4:	1,01 W	0,89 W	4,66 W
CPU Core 5:	1,24 W	0,98 W	4,28 W
CPU Core 6:	1,33 W	1,10 W	4,56 W
CPU Core 7:	1,29 W	1,09 W	5,89 W
CPU Core 8:	2,01 W	1,60 W	6,53 W

Core Voltages

Core	Current	Min	Max
Core SVI2	1,48 V	1,44 V	1,49 V
Core VID 1	1,46 V	1,44 V	1,49 V
Core VID 2	1,46 V	1,44 V	1,49 V
Core VID 3	1,46 V	1,44 V	1,49 V
Core VID 4	1,46 V	1,44 V	1,49 V
Core VID 5	1,46 V	1,44 V	1,49 V
Core VID 6	1,46 V	1,44 V	1,49 V
Core VID 7	1,46 V	1,44 V	1,49 V
Core VID 8	1,46 V	1,44 V	1,49 V

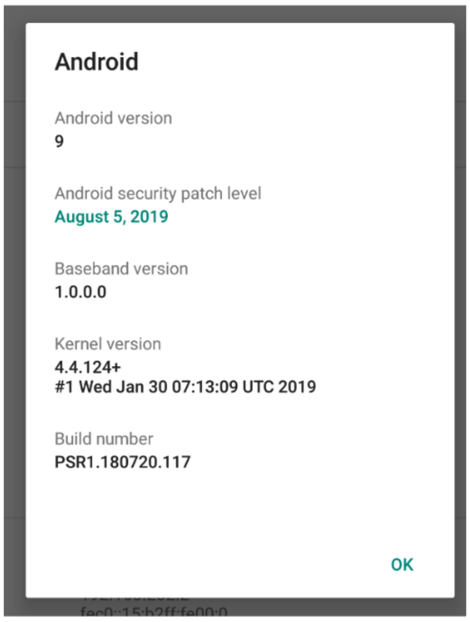
Core parking:	100%
Frequency scaling:	5%
Turbo boost:	60%
Performance:	100%

Current version: 3.6.1.0 / New version: 4.5.1.0

Figure I.1.: System CPU Information

I. Evaluation of the Algorithm

Emulator



Android

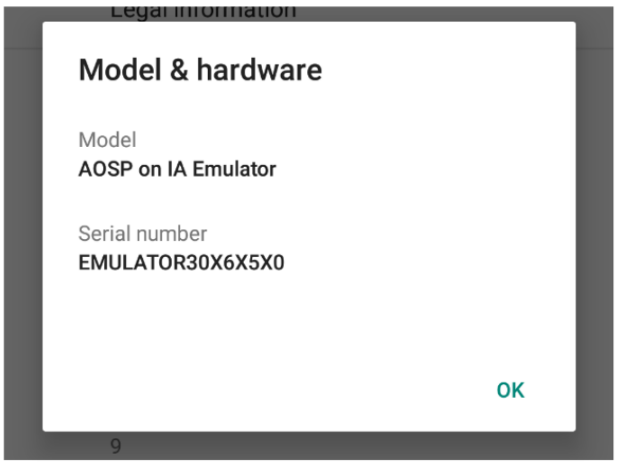
Android version
9

Android security patch level
August 5, 2019

Baseband version
1.0.0.0

Kernel version
4.4.124+
#1 Wed Jan 30 07:13:09 UTC 2019

Build number
PSR1.180720.117



Model & hardware

Model
AOSP on IA Emulator

Serial number
EMULATOR30X6X5X0

Figure I.2.: Emulator Details

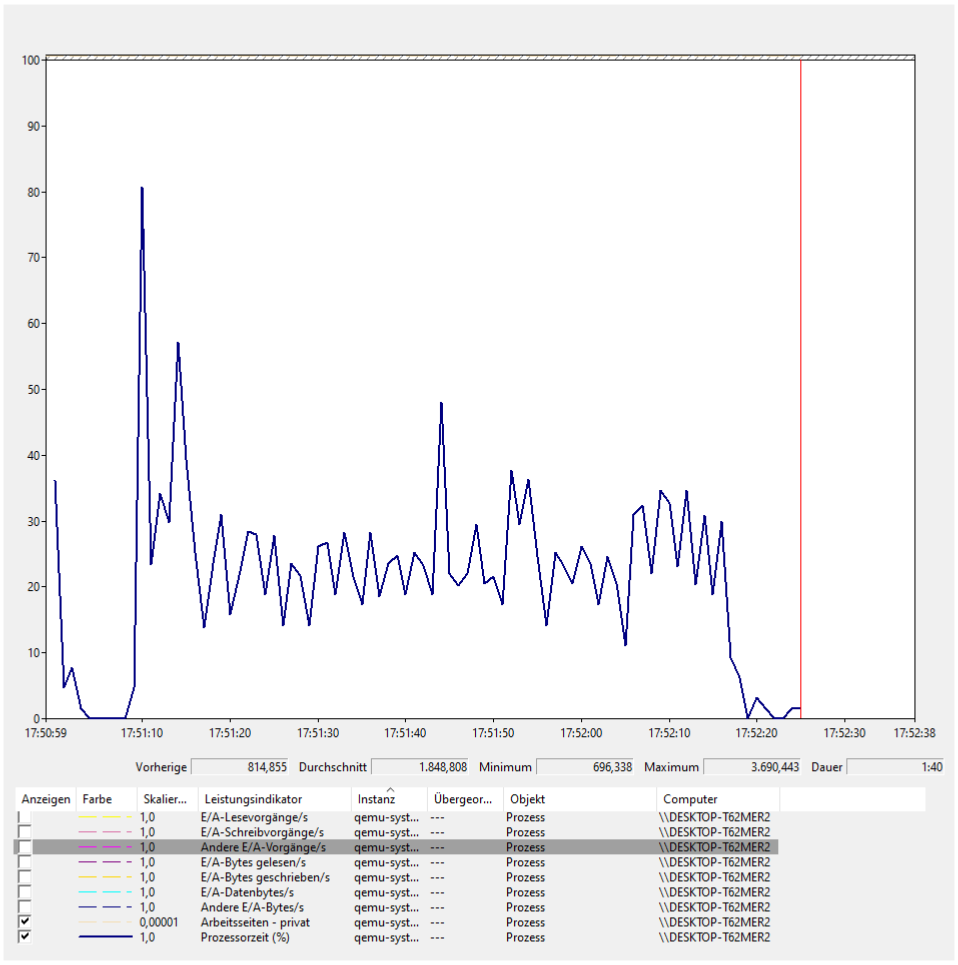
I. Evaluation of the Algorithm

I.1.2. Details of the Compilations

Compilation 1

- Body Part: Chest
- Symptoms: Cough, Neusea
- Proposed Symptoms: Runny Nose
- Execution Time: 0:01:10.896734

Processor time (in %)



Vorherige	Durchschnitt	Minimum	Maximum	Dauer
814,855	1.848,808	696,338	3.690,443	1:40

Anzeigen	Farbe	Skalier...	Leistungsindikator	Instanz	Übergeor...	Objekt	Computer
☐	🟨	1,0	E/A-Lesevorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟪	1,0	E/A-Schreibvorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟪	1,0	Andere E/A-Vorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟪	1,0	E/A-Bytes gelesen/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟨	1,0	E/A-Bytes geschrieben/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟦	1,0	E/A-Datenbytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	🟦	1,0	Andere E/A-Bytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	🟫	0,00001	Arbeitsseiten - privat	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	🟦	1,0	Prozessorzeit (%)	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2

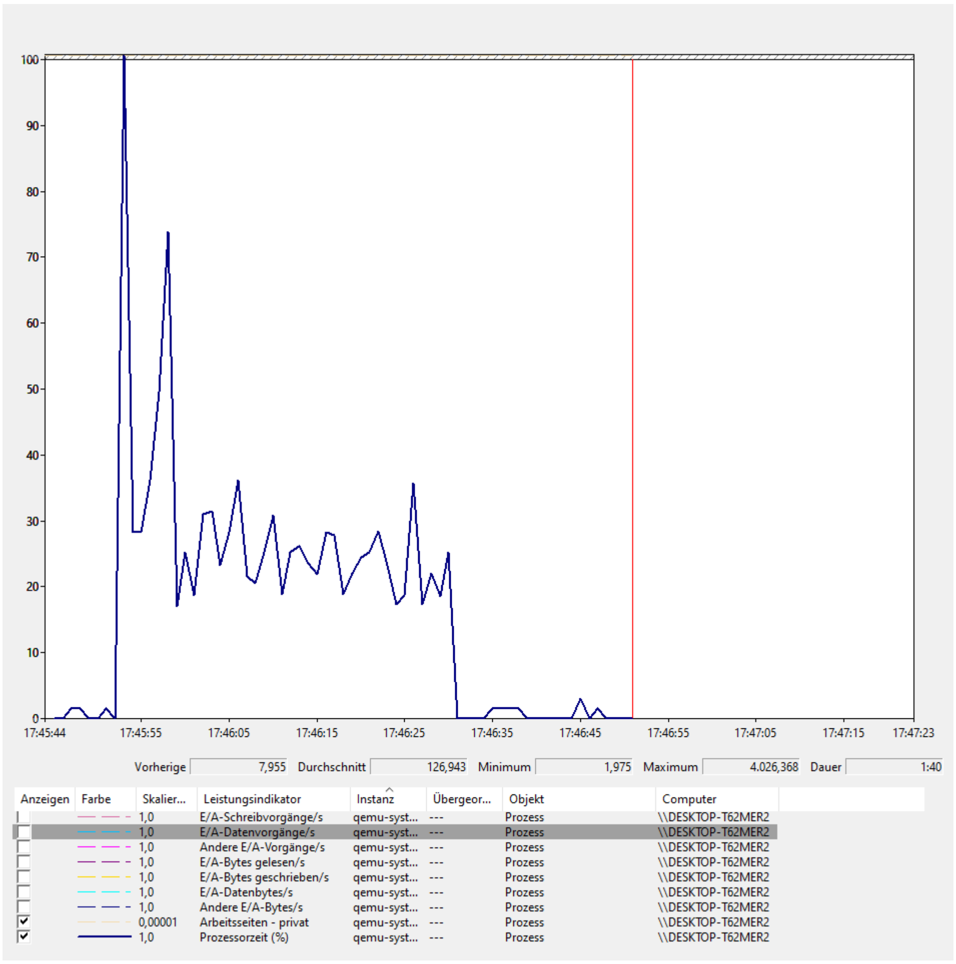
Figure I.3.: Processor Time Compilation 1

I. Evaluation of the Algorithm

Compilation 2

- Body Part: Hand & wrist
- Symptoms: Joint pain(intensity: high, novality: old), Hand swelling (intensity: medium, novality: old)
- Proposed Symptoms: Overweight (intensity: medium, novality: new)
- Execution Time: 0:00:36.914100

Processor time (in %)



Vorherige 7,955 Durchschnitt 126,943 Minimum 1,975 Maximum 4.026,368 Dauer 1:40

Anzeigen	Farbe	Skalier...	Leistungsindikator	Instanz	Übergeor...	Objekt	Computer
☐	— — -	1,0	E/A-Schreibvorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	E/A-Datenvorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	Andere E/A-Vorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	E/A-Bytes gelesen/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	E/A-Bytes geschrieben/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	E/A-Datenbytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	— — -	1,0	Andere E/A-Bytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	— — -	0,00001	Arbeitsseiten - privat	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	——	1,0	Prozessorzeit (%)	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2

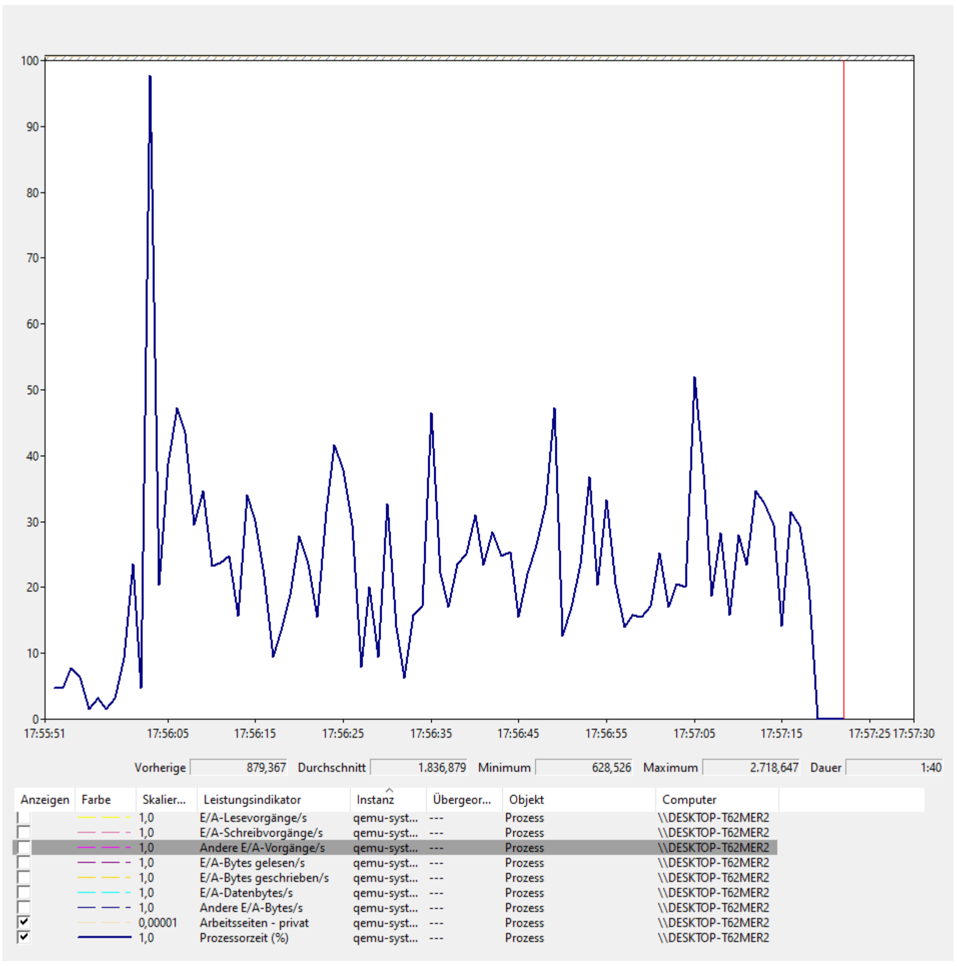
Figure I.4.: Processor Time Compilation 2

I. Evaluation of the Algorithm

Compilation 3

- Body Part: Foot
- Symptoms: Cold feet(intensity: high, novality: old), Cramps (intensity: medium, novality: old), Tremor at rest (intensity: medium, novality: old)
- Proposed Symptoms: Tiredness (intensity: medium, novality: new)
- Execution Time: 0:01:15.182124

Processor time (in %)



Vorherige 879,367 Durchschnitt 1.836,879 Minimum 628,526 Maximum 2.718,647 Dauer 1:40

Anzeigen	Farbe	Skalier...	Leistungsindikator	Instanz	Übergeor...	Objekt	Computer
☐	- - -	1,0	E/A-Lesevorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	E/A-Schreibvorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	Andere E/A-Vorgänge/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	E/A-Bytes gelesen/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	E/A-Bytes geschrieben/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	E/A-Datenbytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
☐	- - -	1,0	Andere E/A-Bytes/s	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	- - -	0,00001	Arbeitsseiten - privat	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2
✓	—	1,0	Prozessorzeit (%)	qemu-syst...	---	Prozess	\\DESKTOP-T62MER2

Figure I.5.: Processor Time Compilation 3

J. Methods to Store Data Locally

J.1. Convert to Map

```
  static Map<String, dynamic> toMap(Diagnose pDiagnose) => {
  'id': pDiagnose.id,
  ...
};
```

Listing J.1: Data toMap

J.2. Encode Data

A list of diagnoses can even be saved directly. Since SharedPreference works with key-value values, these can also be easily deleted from the list by simply removing the associated key.

```
  static String encode(List<Diagnose> pDiagnoses) => json.encode(
  pDiagnoses
    .map<Map<String, dynamic>>((diagnose) => Diagnose.toMap(diagnose))
    .toList(),
  );
```

Listing J.2: Encode Data

J.3. Decode Data

Here the variant is displayed to directly convert a list of diagnoses.

```
static List<Diagnose> decode(String pDiagnoses) =>
  (json.decode(pDiagnoses) as List<dynamic>)
  .map<Diagnose>((item) => Diagnose.fromJson(item))
.toList();
```

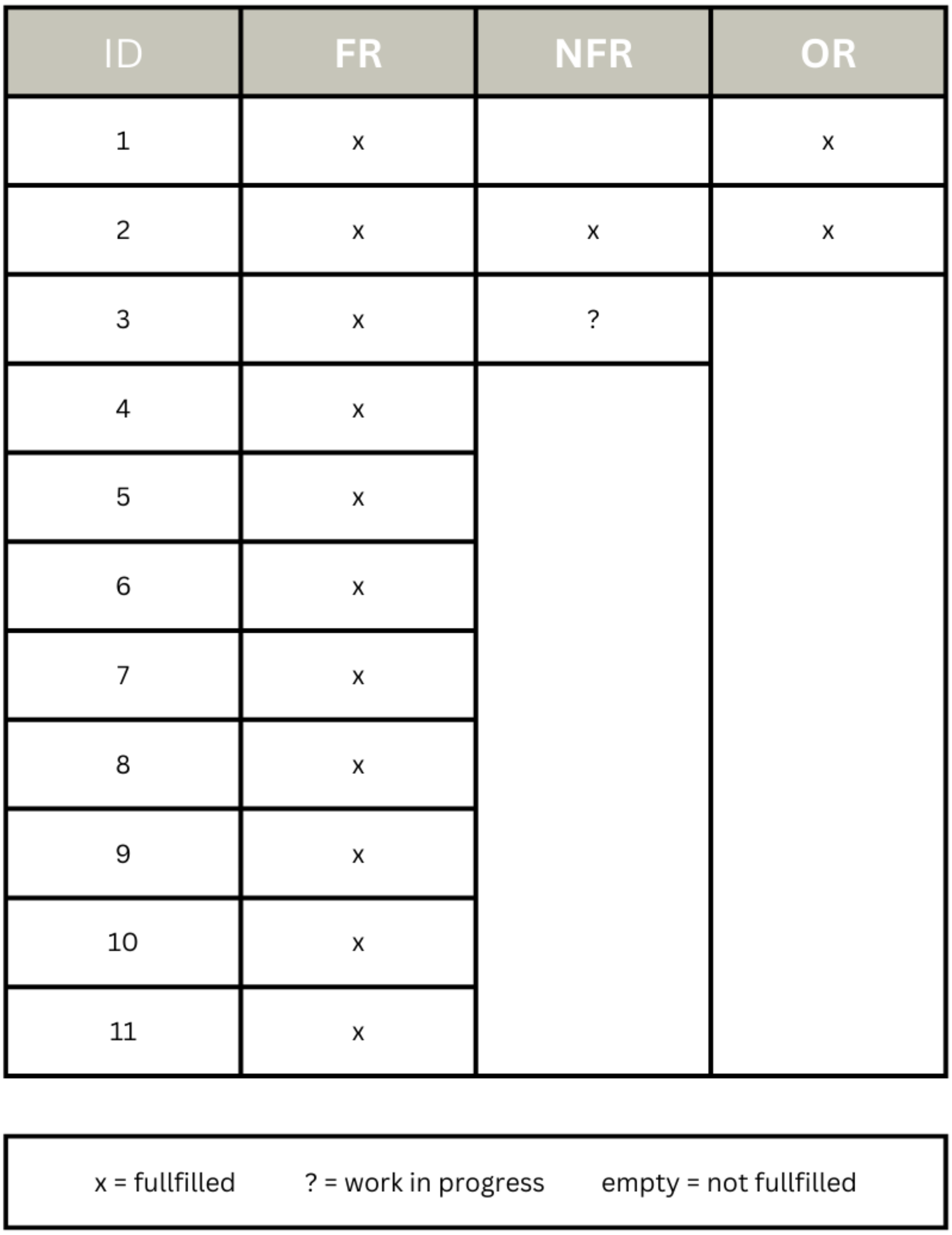
Listing J.3: Decode Data

J.4. Convert to Object (from Json)

```
  factory Diagnose.fromJson(Map<String, dynamic> jsonData) {
    return Diagnose(
    id: jsonData['id'],
    ...
    );
  }
```

Listing J.4: Data fromJson

K. Status of the Requirements



ID	FR	NFR	OR
1	x		x
2	x	x	x
3	x	?	
4	x		
5	x		
6	x		
7	x		
8	x		
9	x		
10	x		
11	x		

x = fullfilled ? = work in progress empty = not fullfilled

Figure K.1.: Functional Requirements

List of Figures

List of Tables

Bibliography

[1] *Corona macht rücksichtsvoller: Thema Gesundheit bei den Deutschen immer wichtiger*. Berlin, 8/4/2022. URL: https://www.bah-bonn.de/presse/bah-gesundheitsmonitor/presse-detailseite/corona-macht-ruecksichtsvoller-thema-gesundheit-bei-den-deutschen-immer-wichtiger/.

[2] Sokollu Senada. "Überforderung der Arztpraxen: Corona-Überlastung: Patienten schildern Grenzsituationen bei Hausärzten". In: *SWR Aktuell* (5/1/2022). URL: https://www.swr.de/swraktuell/baden-wuerttemberg/patienten-im-stich-gelassen-in-corona-zeiten-100.html.

[3] Deutscher Ärzteverlag GmbH, Redaktion Deutsches Ärzteblatt. *Bundesärztekammer warnt vor Ärztemangel*. 2022. URL: https://www.aerzteblatt.de/nachrichten/134020/Bundesaerztekammer-warnt-vor-Aerztemangel.

[4] Rober Koch Institut. *COVID-19: Fallzahlen in Deutschland und weltwei: Fallzahlen in Deutschland*. Ed. by Robert Koch Institut. 19.08.2022. URL: https://www.rki.de/DE/Content/InfAZ/N/Neuartiges_Coronavirus/Fallzahlen.html.

[5] Ada. *Health. Powered by Ada*. 12/27/2022. URL: https://ada.com/de/.

[6] Ada. *Wie funktioniert Adas Symptomanalyse?* 12/27/2022. URL: https://ada.com/de/help/360000319469/%7D,%20urldate%20=%20%7B12/27/2022%7D.

[7] Ada. *Bietet ihr API oder SDK an?* 12/27/2022. URL: https://ada.com/de/help/360000309665/.

[8] Ada Developer Documentation. *Ada Developer Documentation*. 12/27/2022. URL: https://developers.ada.cx/.

[9] *Android-Apps auf Google Play*. 12/27/2022. URL: https://play.google.com/store/apps/details?id=com.programming.progressive.diagnoseapp&gl=DE%7D.

[10] Shady Boukhary and Eduardo Colmenares. "A Clean Approach to Flutter Development through the Flutter Clean Architecture Package". In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2019, pp. 1115–1120. DOI: 10.1109/CSCI49370.2019.00211.

[11] *Everything You Need to Know About Flutter App Development*. 12/28/2022. URL: https://www.cometchat.com/blog/flutter-app-development.

[12] AfafMirghani Hassan. "JAVA and DART programming languages: Conceptual comparison". In: *Indonesian Journal of Electrical Engineering and Computer Science* 17.2 (2020), pp. 845–849.

[13] *Dart overview*. 12/27/2022. URL: https://dart.dev/overview%7D.

[14] E. Windmill. *Flutter in Action*. Manning, 2020. ISBN: 9781638356431.

[15] *Flutter architectural overview*. 12/27/2022. URL: https://docs.flutter.dev/resources/architectural-overview#architectural-layers%7D.

[16] C. Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. ISBN: 9783960888970. URL: https://books.google.de/books?id=2TrCDwAAQBAJ%7D,%20year=%7B2019%7D,%20publisher=%7Bdpunkt.verlag%7D.

[17] R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445. URL: https://books.google.de/books?id=0HYhAQAAIAAJ.

[18] A. Ostrowski and P. Gaczkowski. *Software Architecture with C++: Design modern systems using effective architecture concepts, design patterns, and techniques with C++20*. Packt Publishing, 2021. ISBN: 9781789612462. URL: https://books.google.de/books?id=GrAmEAAAQBAJ.

[19] Thorben Janssen. "SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples". In: *Stackify* (11/4/2018). URL: https://stackify.com/solid-design-liskov-substitution-principle/.

[20] KAUR AMANDEEP and DHINDSA KANWALVIR SINGH. "PERFORMANCE EVALUATION FOR CRUD OPERATIONS IN NoSQL DATABASES". In: *i-manager's Journal on Cloud Computing* 3.2 (2016), p. 1. ISSN: 2349-6835. DOI: 10.26634/jcc.3.2.8164.

[21] S. Alessandria and B. Kayfitz. *Flutter Cookbook: Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart*. Packt Publishing, 2021. ISBN: 9781838827373. URL: https://books.google.de/books?id=zDE0EAAAQBAJ.

[22] *What Is BaaS (Backend as a Service)? Definition and Usage | Okta Australia*. 12/28/2022. URL: https://www.okta.com/au/identity-101/baas-backend-as-a-service/.

[23] Firebase. *Firestore &nbsp;|&nbsp; Firebase*. 12/15/2022. URL: https://firebase.google.com/docs/firestore.

[24] Statista. *Smartphones - Penetrationsrate in Deutschland nach Altersgruppe 2021 | Statista*. 12/28/2022. URL: https://de.statista.com/statistik/daten/studie/459963/umfrage/anteil-der-smartphone-nutzer-in-deutschland-nach-altersgruppe/.

[25] M. Pirozzi. *The Stakeholder Perspective: Relationship Management to Increase Value and Success Rates of Projects*. CRC Press, 2019. ISBN: 9780429591754. URL: https://books.google.de/books?id=87G2DwAAQBAJ.

[26] Tim Weilkiens. *How to model a simple system context with SysML - Model Based Systems Engineering*. 2012. URL: https://mbse4u.com/2012/07/23/how-to-model-a-simple-system-context-with-sysml/.

[27] *The Requirements Engineering Handbook - Google Books*. 11/22/2022. URL: https://www.google.de/books/edition/The_Requirements_Engineering_Handbook/Rkulpi4N3JsC?hl=de&gbpv=1&dq=requirements+engineering+functional+and+nonfunctional+requirements&pg=PA46&printsec=frontcover.

[28] *Functional and Non-functional Requirements – Francois Botha*. 12/30/2022. URL: https://francoisbotha.io/2018/05/15/functional-and-non-functional-requirements/.

[29] Scaled Agile Framework. *Advanced Topic - Domain Modeling - Scaled Agile Framework*. 10/5/2022. URL: https://www.scaledagileframework.com/domain-modeling/.

[30] V. Khononov. *Learning Domain-Driven Design*. O'Reilly Media, 2021. ISBN: 9781098100100. URL: https://books.google.de/books?id=qAtHEAAAQBAJ.

[31] nhs.uk. *The NHS website*. NaN. URL: https://www.nhs.uk/.

[32] *Symptom Checker API | Integrate medical diagnosis into your app today*. 12/29/2022. URL: `https://apimedic.com/`.

[33] *Material Design*. 1/1/1980. URL: `https://m3.material.io/`.

[34] E.N. McKay. *UI is Communication: How to Design Intuitive, User Centered Interfaces by Focusing on Effective Communication*. Elsevier Science, 2013. ISBN: 9780123972873. URL: `https://books.google.de/books?id=wNozxtKuOKcC`.

[35] C. Banga and J. Weinhold. *Essential Mobile Interaction Design: Perfecting Interface Design in Mobile Apps*. Usability. Pearson Education, 2014. ISBN: 9780133563474. URL: `https://books.google.de/books?id=yDOkAwAAQBAJ`.

[36] Oracle Help Center. *Oracle Enterprise Data Quality Online Help*. 2018. URL: `https://docs.oracle.com/en/middleware/enterprise-data-quality/12.2.1.3/edqoh/comparison-word-match-percentage.html`.

[37] Shoba Ranganathan, ed. *Encyclopedia of bioinformatics and computational biology: Volume 1 Methods; Volume 2 Topics; Volume 3 Applications*. Amsterdam: Elsevier, 2019. ISBN: 978-0-12-811432-2.

[38] In: ().

[39] scikit-learn. *1.9. Naive Bayes*. 12/29/2022. URL: `https://scikit-learn.org/stable/modules/naive_bayes.html`.

[40] *Introduction to Bayesian networks | Bayes Server*. 12/21/2022. URL: `https://www.bayesserver.com/docs/introduction/bayesian-networks/`.

[41] Jason Brownlee. "A Gentle Introduction to Bayesian Belief Networks". In: *Machine Learning Mastery* (10/10/2019). URL: `https://machinelearningmastery.com/introduction-to-bayesian-belief-networks/`.

[42] Heiner Oberkampf et al. "Towards a Ranking of Likely Diseases in Terms of Precision and Recall". In: Aug. 2012.

[43] "Drei Sekunden sind zu lang – Auswirkung der Ladezeit von Webseiten auf die User Experience". In: *Computer Science Blog* (17.1.2022). URL: `https://blog.mi.hdm-stuttgart.de/index.php/2022/01/17/drei-sekunden-sind-zu-lang-auswirkung-der-ladezeit-von-webseiten-auf-die-user-experience/`.

[44] Dart packages. *shared_preferences | Flutter Package*. 12/31/2022. URL: `https://pub.dev/packages/shared_preferences`.

[45] *Store key-value data on disk*. 12/30/2022. URL: `https://docs.flutter.dev/cookbook/persistence/key-value`.

[46] Dart packages. *pdf | Dart Package*. 12/31/2022. URL: `https://pub.dev/packages/pdf`.

[47] Affairs, Assistant Secretary for Public. *System Usability Scale (SUS)*. 2013. URL: `https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html`.