
# 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

# PREAMBLE

# ACKNOWLEDGEMENTS

# Contents

# 1 Introduction

## 1.1 The Bachelor's Thesis Problem

People are becoming more interested in matters concerning physical and mental health. This is most likely attributed to the COVID-19 pandemic that has been circulating in recent years. Along with positive outcomes, such as increased care for fellow citizens and greater awareness of health issues, the consistent growth of interest in health issues also caused 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. This is also noticed by the patients: Overcrowded waiting rooms combined with long waiting periods and nerve-racking telephone loops are becoming the norm for doctor visits. The bachelor's thesis problem can be traced back to the preceeding 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 being forced to order patient stops and issue access bans. This also means that patients in need of 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). A survey was launched to highlight the problem in more detail (all questions included in the survey and the corresponding answers can be taken from the appendix). The results have shown that around 80% of those questioned have put off a visit to the doctor in recent years, even though they have suffered from symptoms.

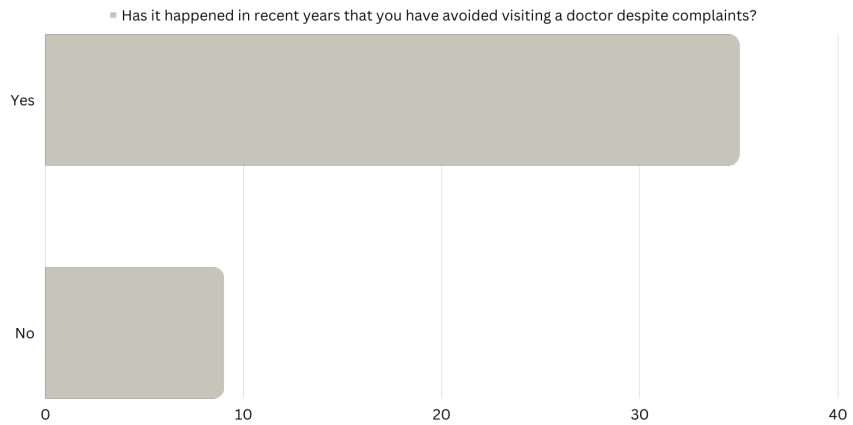


Figure 1.1: Survey Question - Avoiding to visit the doctor

Another question within this survey was which reasons were the reason for postponing these doctor visits, or why the respondents could imagine avoiding a doctor visit. Those reasons range from long waiting times, to not being able to make an appointment. Figure 1.2 shows the mentioned distribution of the answers.

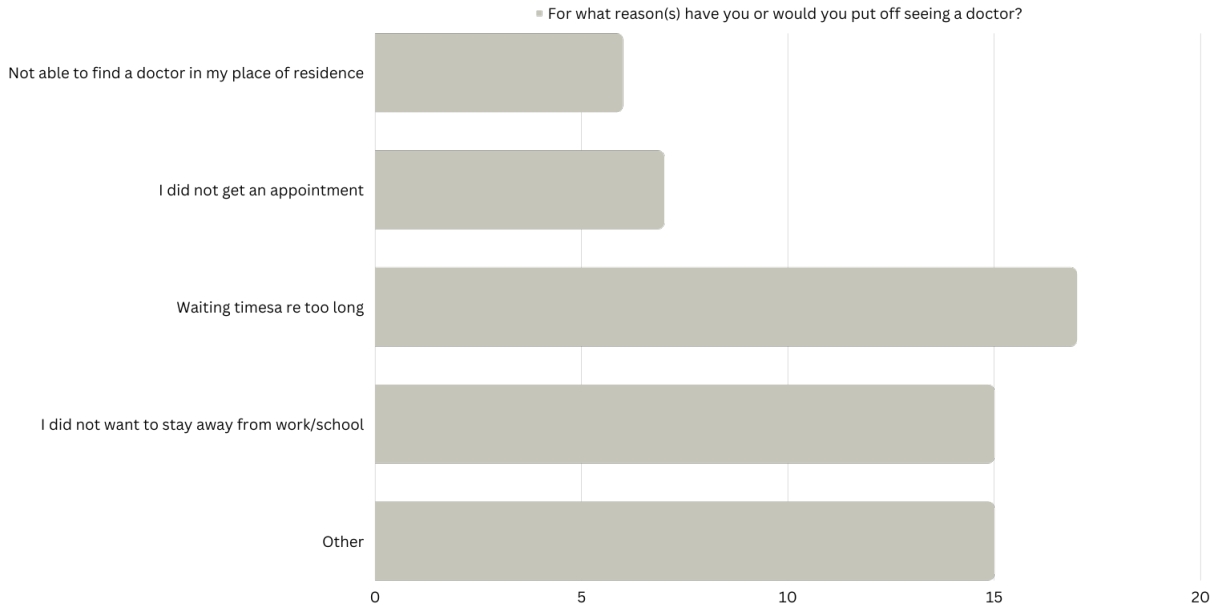


Figure 1.2: Survey Question - Reasons of avoiding to visit the doctor

## 1.2 Motivation

The mentioned problem imposes the question on how to address patients' concerns while also relieving the burden on doctors. Digitalization provides a solution to this problem. Online consultation hours and online 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 aforementioned problems through the use of mobile applications. Such an application can provide advice to a worried user and help alleviate their fears.

## 1.3 The Bachelor Thesis Goal

The project's ultimate objective is the successful creation of a mobile application that enables users to receive a diagnosis based on the symptoms they reported, even if they were unable to schedule an appointment with a doctor, due to busy practices, or did not have the opportunity to see one, due to lack of time or long waiting times. 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 the database can be expanded by doctors, whereby a verification possibility must be provided. They should be provided with the possibility to add either disease-related data or pieces of advice regarding diseases and illnesses for users. The detailed determination of how all of this should be made possible will be determined and implemented during the development process.

# 2 Fundamentals

## 2.1 Flutter and Dart

The framework used to develop the disease-detection application will be Flutter, which is an open-source-UI-Kit developed by Google. Flutter uses the open-source programming language Dart, which at first was designed for building Google Chrome browser-applications and later on benefited greatly from various improvements since it was released back in 2011. The programming language consequently evolved from having a lot in common with JavaScript to sharing many features with C# and Java.

The Flutter and Dart ecosystem, which is brimming with open-source packages created by other developers from around the world, is one of the framework's best features. It enables programmers to create visually stunning applications in the shortest amount of time, by including packages from developers all over the world. Also, Dart is a client-optimized gneral-purpose programming language that supports cross-platform development. This implies that this application will be created with a single code base yet will run on both Android- and iOS-smartphones. 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 be entirely focused on Android development. Dart is also a statically-typed language, which means that the type of each variable 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 as 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 code that is safe and predictable. [QUELLE:DART/OVERVIEW]

### 2.1.1 Flutter: Everything is a Widget

When researching how Flutter functions, it's 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 [flutter in action, s 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, which means nested inside of one another [Flutter in action, s61], so rather than simply returning the widget it describes, a widget's build method really returns a tree of widgets. The DOM in any web browser is comparable to this widget tree. A sample widget returned by a build method is shown in Listing 2.1. Figure 2.1 illustrates the generated widget tree in detail.

```
import 'package:flutter/material.dart';

void main() {
  runApp(const TestApplication());
}

class TestApplication extends StatelessWidget {
  const TestApplication({Key? key}) : super(key: key);

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

Listing 2.1: Flutter example

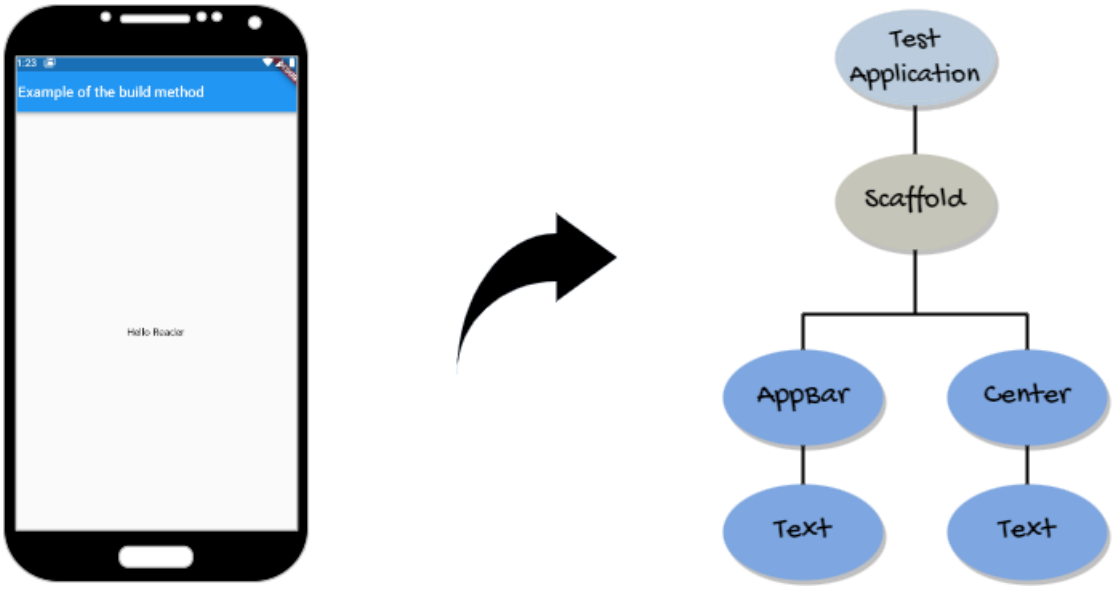


Figure 2.1: Widget Tree of Listing 2.1

### 2.1.2 Flutter: Architectural Layers

Application architecture refers to how the various components of a mobile or web application are organized and how they interact with each other. A well-designed application architecture helps improve the systems performance, maintainability, scalability but also makes it more modular. [Buch: Software arhcitecture seite 28] Many different application architectural patterns can be used, including layered architecture of which Flutter makes use. In the context of software development, layered architecture is a common design pattern in which the application is divided into different layers, each layer plays a specific role in the overall functionality of the application. No layer has privileged access to the layers below, and every part of the framework layer is designed to be optional and interchangeable. [Flutter.dev architectural layers]

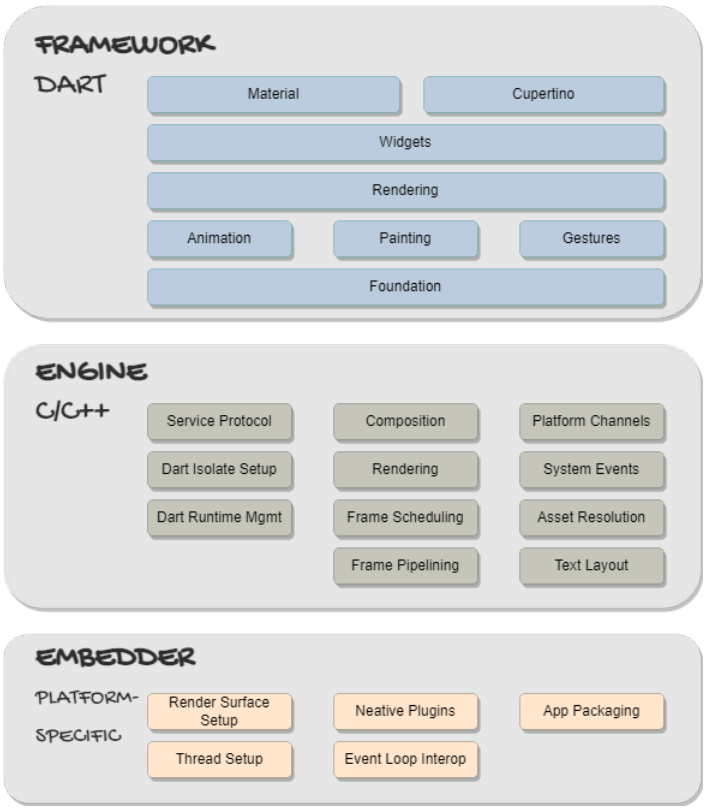


Figure 2.2: Layered Architecture in Flutter[based on Flutter page and TakingFluttertotheWeb page 50/]

### 2.1.3 Programming Paradigm

The programming paradigm of the Dart programming language is object-oriented programming (OOP). This means that it uses objects, 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, which enables developers to write code that can run concurrently and handle multiple tasks at the same time. 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. They were

introduced by Robert C. Martin 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 to develop software that is prone to fewer bugs and has cleaner source code. [https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/]

- **Single Responsibility Principle (SRP):** The single responsibility principle instructs the developer to develop classes and software components in such a way 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 to reduce complexity and improve the maintainability, testability, and extensibility of a software system. Another positive side-effect of following this principle is that the written code is easier to understand and error-testing can be done more efficiently.
- **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 in such a way that it can 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 existing functionality. This suggests that these classes or software components ought to be developed in a way that allows 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. [Software architecture with c++] One can also say, that it is an extension of the OCP. A subclass should be able to be used wherever its superclass is expected, without breaking the functionality of the program. [stickify, solid design liskov] The Liskov Substitution Principle helps to improve the flexibility and reusability of a software system.
- **Interface Segregation Principle (ISP):** The Interface Segregation Principle ensures that a clients of a class should not implement an interface that contains methods that are not relevant 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.
- **Dependency Inversion Principle (DIP):** The key 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.

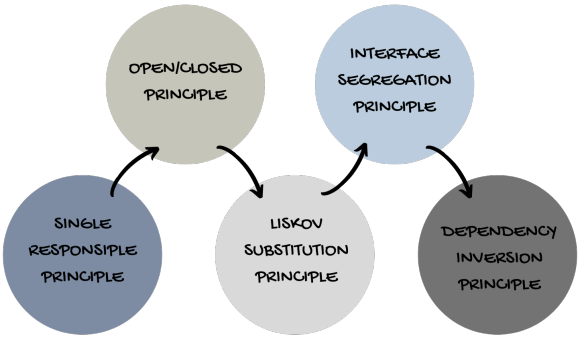


Figure 2.3: SOLID Principles

## 2.2 Symptom and Disease APIs

The aim of this work is to develop a functional disease detection application. In order to achieve this goal, the application must access a database that is filled with suitable data. The information from an existing API is used here because there are no medically trained project participants who could contribute their specialized expertise to the database. The final data structure of the database is also influenced by the API selection. For this purpose, the two most suitable APIs are considered and their suitability for the project is determined

### 2.2.1 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 (GPs), hospitals, and community health services. The NHS websites provides many different APIs, free for use [NHS website]. 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 HTTP call to https://api.nhs.uk/conditions. An example of a possible request in the programming language Python looks like this:

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

Listing 2.2: Example Python Request for the Health A to Z API

If the request was successfull and the response contains the data in JSON format. Listing 2.3 shows a sample abbreviated response from the API.

```
{
  "@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 2.3: Example Response for the Health A to Z API

An entire text document with the server's response to the request made in listing 2.2 can be viewed by scanning the QR code shown in the appendix x.x. 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 with it, 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 serious problem in terms of runtime.

### 2.2.2 ApiMedic Symptom Checker API

The ApiMedic API is the second interface to be considered. ApiMedic is powered by priaid, a company that focuses on bringing together medicine, IT and business administration. Thanks to their highly specialized team composition, they offer expertise in all of the areas mentioned. The API can be addressed in two different ways:

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

Although the data that can be received from this API is not as detailed as that of the NHS API, using the data to create a data structure is simpler. 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 an request, made with the live account, to retrieve all diseases followed by the response of the API.

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

Listing 2.4: Example Python Request for the ApiMedic API (all issues)

```
[
  {
    "ID": 130,
    "Name": "Abdominal hernia"
  },
  {
    "ID": 170,
    "Name": "Abortion"
  },
  {
    "ID": 456,
    "Name": "Abscess of the tonsils"
  },
...
```

Listing 2.5: Response 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. Listing 2.7 shows a sample shortened response from the ApiMedic API.

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

Listing 2.6: Example Python Request for the ApiMedic API (single issue)

```
{
  "Description": "Measles is caused by a virus a...",
  "DescriptionShort": "Measles, also known as morbilli, ...",
  "MedicalCondition": "The infection begins with flu-like symptoms ( ...",
  "Name": "Measles",
  "PossibleSymptoms": "Burning eyes,Burning in the throat,Cough,...",
  "ProfName": "Morbilli",
  "Synonyms": "Red measles",
  "TreatmentDescription": "To prevent measles, an effort to vaccinate ..."
}
```

Listing 2.7: Response for the ApiMedic API (single issue)

The value of the "PossibleSymptoms"-key returns an enumaration of all the symptoms of the disease. 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.

### 2.2.3 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 makes it difficult to use the data for the purposes intended in this work. NHS also provides data on the causes that various symptoms and conditions can have, which can be an important factor in making a diagnosis in the form of disease detection. ApiMedic, in turn, delivers the data in an optimal format to use, but far less comprehensive than NHS. One option that is available 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, it has been found that only a very small amount of symptom has been recognized. In the context of the bachelor thesis, the use of the ApiMedic API is preferred from the point of view of a clean database structure. For optimizations in the future, a combination of the two sets of data can be considered.

# 3 Requirements Engineering

In order to provide a functional and relevant application, it is necessary to first 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 determined. Based on the information gained from all of this, a domain model can then be generated, which serves as a transition to the creation of the database.

## 3.1 Stakeholder Analysis

The first step is to identify the project's interest and demand groups; this is done through a stakeholder analysis. The societal influences on the project are looked at in the stakeholder analysis. The stakeholder analysis allows for the prediction of variables such as "power," "interest," and potential stakeholder behavior. Stakeholders are individuals (groups), organizations, and interest groups that have the power to significantly affect a project's success. Therefore, it is essential for project managers to understand their interests and potential for influencing the project goals. [Quelle] 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.

### 3.1.1 Target Group and User Group

Both senior persons and young people who, despite the difficulties mentioned in chapter[], would desire to have a diagnosis of their current health status situation are targeted by the mobile application for diagnosing diseases. One may assume that, given the age distribution of smartphone users today, the user base will be evenly split between the young and the old. 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 %. Over 70 percent of smartphone owners still make up about 68 percent of the total. [https://de.statista.com/statistik/daten/studie/459963/umfrage/anteil-der-smartphone-nutzer-in-deutschland-nach-altersgruppe/] One more target audience are 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 limitation. The stakeholder analysis will go into greater depth on this subject.

### 3.1.2 Internal Stakeholder

In this project, the internal stakeholder group is relatively small. The most significant internal stakeholder will be the personification of the developer, which is also the administrator of the data bank. Due to the positive effects a successful and widely used application would have on his reputation as a developer, this person has a great, personal, interest in the project's success. The power he wields over the project is extremely high. Without him, the application development would not be possible.

### 3.1.3 External Stakeholder

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 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.

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. The power factor, however, can both 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.

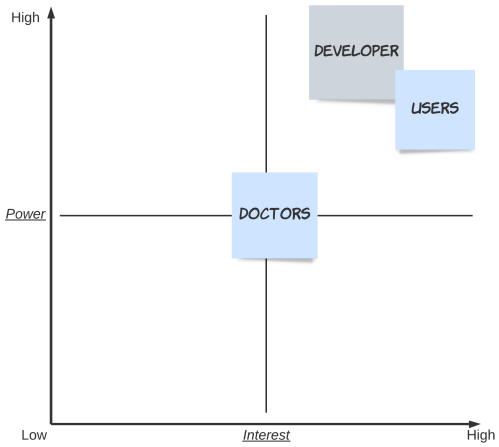


Figure 3.1: Power/Interest Grid for Stakeholder-Analysis

## 3.2 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 who are impacted by its operation, external systems, and processes with which it interacts and the policies and regulations that it must comply with. The system context can be determined using the previously performed stakeholder analysis. Determining the system context helps to get an understanding of 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 communicates with the system via direct code-based access. The database, which is integrated externally, also communicates with the system via a code-based interface. There surely are some aspects about privacy policy and the security of personal data of the users, especially during the verification process of the doctors. Since, in context of this bachelor thesis, the application will not be launched on the PlayStore these aspects are neglected in the system context.

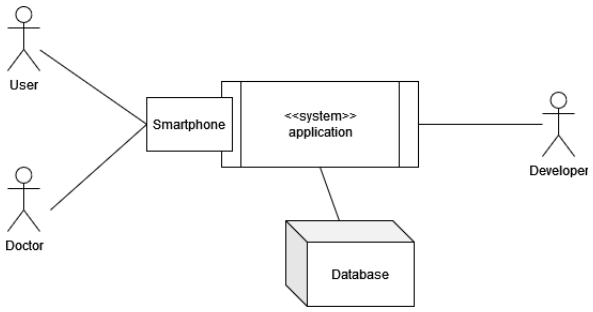


Figure 3.2: System Context Diagram

## 3.3 User Requirements Specification

The requirements for an application can be divided into three categories: functional requirements, non-functional requirements and optional requirements. In this section, the different types of requirements are considered and created project-specifically.

### 3.3.1 Functional Requirements

Functional requirements are the specific capabilities, behaviors, and features that a system must have in order to fulfill its intended purpose. They describe what the system must do in order to be considered successful, and provide a basis for evaluating the system's performance and functionality. Functional requirements describe the functionality that an application should fulfil, i. e. the expected behavior of the system is captured by the functional requirements.

| ID | Description |
|---|---|
| FR1 | The application must allow the user to switch between the diagnostics and the advice 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 |
| FR10 | The application must allow a user to abort his diagnostic procedure at any time |
| FR11 | The application must allow a doctor to view his added records |
| FR12 | The application must allow a user to delete saved diagnoses |
| FR13 | The application must allow a user to view tips and tricks for symptoms and diseases |
| FR14 | The application must allow a doctor to abort adding/editing data at any time |

Table 3.1: Functional Requirements

### 3.3.2 Non-functional Requirements

After the functional requirements have been determined, the non-functional requirements are now considered. Non-functional requirements can be described by not dealing with the direct interaction of a user with the application, but with the system-specific properties. This includes, for example, the reliability of the application, but also safety aspects. [Buch google]

| ID | Description |
|---|---|
| NFR1 | The application must make correct diagnoses |
| NFR2 | The application must make correct diagnoses |
| NFR3 | The application must be usable for patients without registration |
| NFR4 | The application must be usable for the user groups described |
| NFR5 | The application should have a graphical interface that can be used intuitively |

Table 3.2: Non-Functional Requirements

### 3.3.3 Optional Requirements

As the name suggests, optional requirements of an application consist of requirements that do not necessarily have to be implemented. Their implementation is just a kind of bonus for the user.

| 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 3.3: Optional Requirements

## 3.4 Use Cases

The architectural goal of the application is to be designed 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 use 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 wishes of the users and that they actually use the application. In addition, possible ambiguities are revealed and required data structures are determined. Possible problems that may arise during use of the application are most 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 set out above. Figure 3.3 shows the resulting use case diagram, which has been shortened to the most relevant use cases.

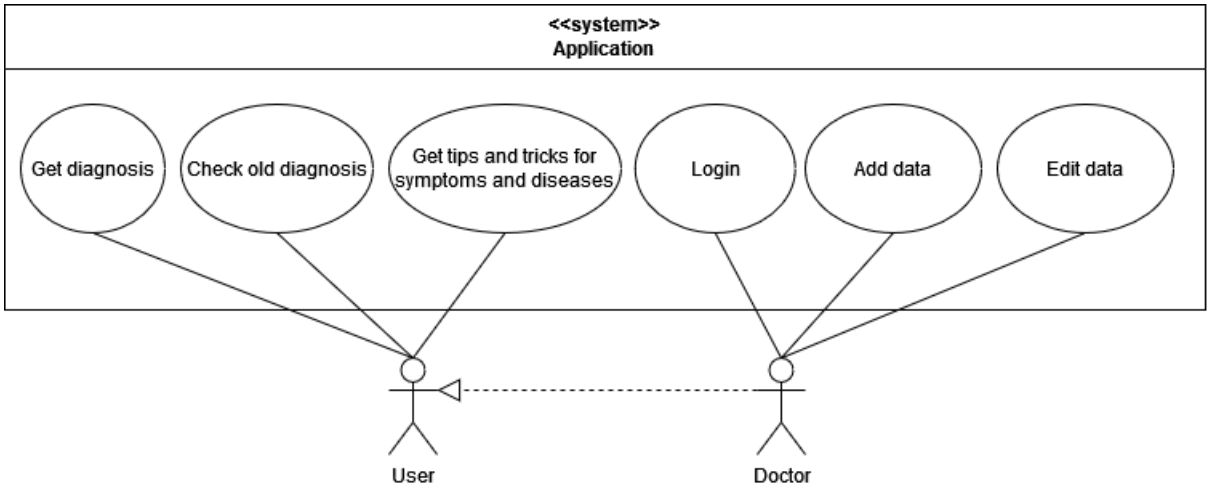


Figure 3.3: Use Case Diagram

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 []. Furthermore, screens that are described in the use cases can also be found in the appendix [].

### 3.4.1 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. normal users, and doctors. In this application, doctors represent a subclass of a user. 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 in the form of 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 has the option to cancel the symptom statement at any time, which ends the use case by returning to the main page of the application **[FR10]**.

### 3.4.2 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 has previously saved it directly 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, whereupon 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 has the option of deleting the selected diagnosis, which 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 given the opportunity to save his diagnosis as a PDF **[OR1]**, the procedure resulting for this corresponds to that of the "Get Diagnosis" use case.

### 3.4.3 Login

As already mentioned in the project goal, doctors should be given the opportunity 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. There, the actor has the opportunity to enter his credentials, whereupon the system checks whether these are stored in the database. Should the result be positiv, 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 has the opportunity to click on the verify button, where he will be prompted to carry out this process and will be logged in if it is successful. Should it be not successfull, because the user can not verify himself as a doctor, the system will continue showing error messages to the acteur. If the doctor is verified, but the system cannot find the entered credentials, an error message is displayed by the application and the user, suspecting that the crednetials have been entered incorrectly, is asked to check his user data and try again after making a correction.

### 3.4.4 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 as well. If the actor presses the confirm button without entering anything in each data field, 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 **[FR12]**.

### 3.4.5 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 in which 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 now 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 has the option to end the process at any time **[FR12]**.

### 3.4.6 Get Tips and Tricks for Symptoms and Diseases

The final use case worth mentioning is viewing advice on illnesses **[FR13]**. A user has the option to go to the view for all advice that Doctors have uploaded. Once he's navigated there and the system shows the predicted view, he can click on one of the pieces of advice there and it will be shown to him in detail. Optionally, the user can save the advice as a favorite by clicking on the button provided for this purpose **[OR2]**.

## 3.5 Domain Model

Domain modeling is a major modeling topic in Agile development at scale because there is frequently a gap between comprehending the issue domain and the interpretation of requirements. It depicts the solution as a collection of domain objects that collaborate to satisfy system-level scenarios. [internetseite SAFe] The quintessence of the object-oriented analysis step is the decomposition of 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 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. [UML 2 Buch] Domain modeling is a major modeling topic in Agile development at scale because there is frequently a gap between comprehending the issue domain and the interpretation of requirements. 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. [internetseite SAFe] Finding conceptual classes by recognizing substantive phrases is an effective technique to domain modeling. [Buch UML2]

- A person is a **user** of the application.
- A **user** chooses a **symptom** and specifies it.
  - A **symptom** occurs in different **parts of the body**, has different **proposed Symptoms** and belongs to different **diseases**.
  - A **user** specifies the selected symptom by narrowing down (selecting) **body parts** of the symptoms occurrence.
- One or more **user-specified symptoms** lead to the calculation of one or more possible **diseases**.
  - A **disease** occurs in different **parts of the body**, has different **symptoms** and has a name, treatment advice and description.
- A **diagnosis** consists of one or more **diseases**.
- **Doctors** are special **users** of the application.
- **Doctors** can edit **advices**, **symptoms** and **diseases**.
- **Doctors** can add **advices**, **symptoms** 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. The domain model will also be based on the apimedic data.

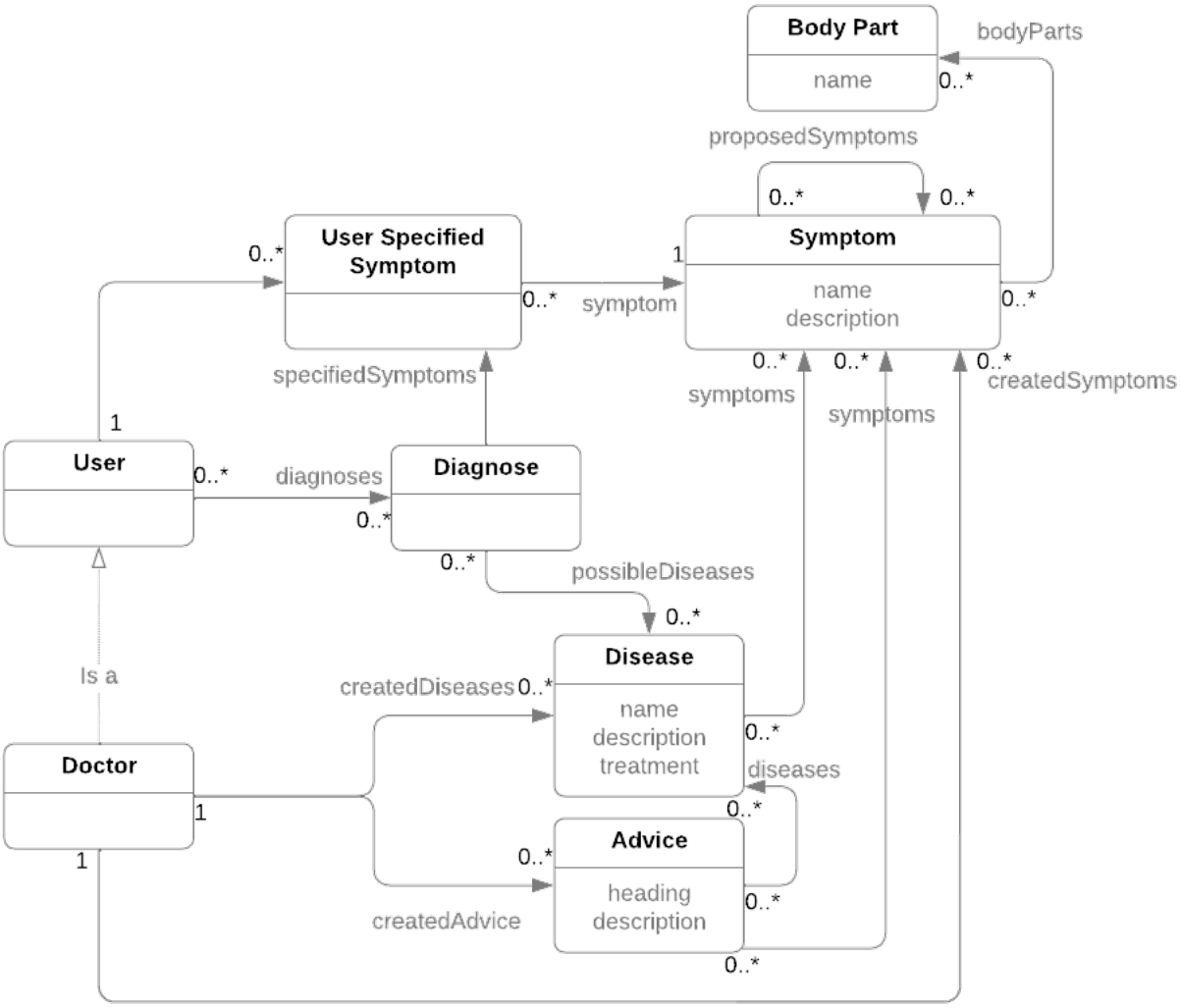


Figure 3.4: Domain Model

Based on the domain model and the previously obtained information, the development of the database can then begin. Chapter x also addresses some aspects that have been neglected at the moment. These are, for example, the causes of diseases and symptoms, but also the time when the symptoms appear. In the mentioned chapter, various possibilities for optimizing the disease detection and other aspects are discussed.

# 4 The Database

## 4.1 Introduction to NoSQL Datbases

The term NoSQL describes database systems that, unlike 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 on the basis of two aspects: In contrast to relational databases, which present their data storage in table format, NoSQL databases benefit from different database models: document-oriented, key Value, graph and column databases.[Image] This wide range of different data models gives developers the benefit of being able to choose the model that best suits their application design. The resulting result is a minimization of the code to be developed for an application. In addition, NoSQL databases allow administrators to scale their data both on one machine and on hardware clusters, so data volumes can be expanded without an expensive investment in new servers.

## 4.2 Firestore

Cloud Firestore, more often called Google Firestore, is a cloud-hosted NoSQL database option which enables developers to store and synchronize their data in realtime, meaning that data which just got added to the database and changes made on already existing data are instantly shown to the application users. Since Firestore is published by Google, it comes with peak reliablitly and great performance. Firestore can easily be integrated in a Flutter Application with help of public packages and via the native SDKs. Something worth to mention is, that Firestore can be used with far more programming languages than Dart and is also compatible with REST and RPC APIs. Cloud Firestore caches data that your app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device comes back online, Cloud Firestore synchronizes any local changes back to Cloud Firestore. To keep your data safe, firestore offers the opportunity to create security rules based on an individuals needs, this includes Identy and Access Management (IAM).

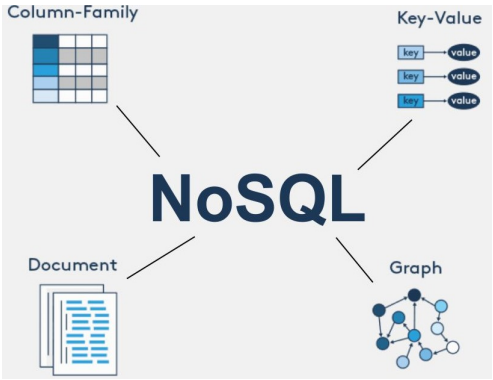


Figure 4.1: NoSQL Data Models

With Firestore in combination with Flutter one can retrieve data via the package cloud_firestore and the StreamBuilder Widget provided by Flutter. The named package also allows the developer to make use of the authentication functionality provided by firestore, which makes it possible for a user to register and login in an application. A detailed example will be given later on, when adding and retrieving data becomes relevant in the diagnose-tracker-application.

## 4.3 Document Databases

As mentioned in the introduction to NoSQL databases, there are several different databases which all rely on different data models. Firestore makes use of the document-based data format. This means, that data stored in the database is accessible via collections, which are filled with documents. 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. An example of that is shown in the figure x.x. Documents in Firestore store their data in a key-value-format which makes it possible for a developer to store different sort of documents in each collection. A quick view at an example makes this easier to understand: A developer wants to develop a restaurent-review application. For that he creates 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 doesn't 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 to only store relevant data to the database. It is important to keep in mind, that even if the third restaurant one day gets a slogan, the developer has the possibility to add that field to the document later on. Cloud Firestore also allows it to store subcollections or complex nested objects to documents. When wanting to recieve data from Firestore, one can simply

## 4.4 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 3.4. 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.

## 4.5 Firebase Redundancy

## 4.6 Inserting the Data into the Database

JupyterNotebook - python code

## 4.7 Connecting the Database with the Flutter Project

# 5 Development

## 5.1 Architecture: The Domain model (Flutter, Layered Architecture

## 5.2 Graphical User Interface

### 5.2.1 Survey

### 5.2.2 Design of the Graphical User Interface

### 5.2.3 Development of the Graphical User Interface

## 5.3 State Management of the Application

## 5.4 Symptomanalysis

## 5.5 Generate Diagnose PDF

## 5.6 Store PDF Locally on Device

## 5.7 Design Tip and Trick View

## 5.8 Doctor Login

## 5.9 Doctor Add/Edit Illnesses

## 5.10 Doctor Add/Edit Tips and Tricks

# 6 Development of the Match Algorithms

## 6.1 The Symptom Graph

## 6.2 Weighting of the Symptoms within the Application

## 6.3 Development of the Algorithms

## 6.4 Evaluation of the Algorithms

### 6.4.1 Performance Evaluation

### 6.4.2 Scalability Comparison

# 7 General Overview of the Application

## 7.1 Testing the Application

## 7.2 Survey: Would Respondents use this Application and put their trust in it

# 8 Conclusion and Outlook

## 8.1 Symptom Detection Applications in the Future

## 8.2 Use of Flutter to Develop Applications

## 8.3 Outcomes of the Performance Comparisons

## 8.4 Overall Conclusion

# 9 Appendix

| 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<br>2. The system displays the view to enter and specify symptoms<br>3. The user selects his symptoms and specifies them<br>4. The user indicates that he is finished<br>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]**<br>AF1b. The system returns to the main page of the application<br><br>AF2a. The user wants to save the diagnosis **[FR7]**<br>AF2b. The user presses the save button<br>Af2c. The system saves the diagnosis |

Table 9.1: 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<br>2. The system shows the selected diagnosis to the user<br>3. When finished the user clicks on the back button |
| Alternate flow | AF1a. The user wants to delete the diagnosis **[FR12]**<br>AF1b. The user clicks on the delete button<br>AF1c. The system deletes the diagnosis<br><br>AF2a. The user wants to download the diagnosis as PDF<br>AF2b. The user presses the download button |

Table 9.2: 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<br>2. The system displays the tip-dashboard<br>3. The user clicks on a tip to see the whole tip-description<br>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]**<br>AF1b. The user clicks on the favorite icon of the tip<br>AF1c. The system saves the tip to the users favorites |

Table 9.3: 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<br>2. The systems shows the login form<br>3. The doctor enters his personal details and presses the okay button<br>4. The system checks for the credentials in the database<br>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<br>AF1b. The user entered wrong credentials<br>AF1c. The system displays an error message<br>AF1d. The user retries<br><br>AF2a. The user is not verfied as doctor yet **[FR3]**<br>AF2b. The doctor enters his personal details and presses the okay button<br>AF2c. The system starts the verification method<br>AF2d. The doctor is verified as doctor<br>AF2e. The system displays the Add screen and the use case ends |
| Alternate flow (failure) | AFF1a. The user is no doctor<br>AFF1b. The user is not able to verify himself as doctor<br>AFF1c. The system shows an error |

Table 9.4: 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<br>2. The system shows the add form<br>3. The actor enters the required data and presses the ok button<br>4. The system adds the disease, symptom, cause or tip to the database |
| Alternate flow | AF1a. The actor missed to enter data<br>AF1b. The system displays an error message<br>AF1c. The actor retries<br><br>AF2a. The actor wants to cancel the process **[FR12]**<br>AF2b. The actor clicks on the cancel button<br>AF2c. The system closes the add form |

Table 9.5: 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<br>2. The system shows the edit form<br>3. The actor edits the data and presses the okay button<br>4. The system updates the data and the use case ends |
| Alternate flow | AF1a. The actor wants to cancel the process **[FR12]**<br>AF1b. The actor clicks on the cancel button<br>AF1c. The system closes the edit form |

Table 9.6: Use case edit data

Figure 9.1: QR-Code for NHS API Response

# 10 Bibliography

# 11 List of Abbreviations

# 12 List of Tables

# 13 List of Figures