Architecture Document
TuBuddy - Tuberculosis Treatment Application

Final Iteration (June 16, 2018)

Client: E.M. Koops,
e.m.koops@student.rug.nl

Teaching Assistants:

Charles Randolph
Frank te Nijenhuis

Team Members:

Teodor Ionut Oanca
Giacomo Casoni
Rutger Berghuis
Andrei Scurtu
Sten Sipma
Julius van Dijk
Noam Drong
Hidde Folkertsma
Marco Lu
Pieter Jan Eilers
Sytze Tempel
Roel Brandenburg
Robert Riesebos
Niek de Vries

Contents

1 Introdution

TuBuddy is the app that aids the patient with their medicinal intake for tuberculosis. It tries to increase medicine intake retention by using various tools to make intake less meddlesome as well as providing the user with an idea of what tuberculosis exactly is and the importance of medicine intake. Non-patients can also use TuBuddy for the informational aspects of the app.

The TuBuddy package also includes a webapp for the physician. The physician app allows for the physician of a patient to create a TuBuddy account so the patient can make use of the core features in the smartphone app. It also allows for creating/modifying the medication plan of a patient and allows the physician to read the questions a patient has asked in the mobile app.

The interaction between the physician webapp and the TuBuddy smartphone app is our back-end. The back-end receives information from both apps and sends updated information back.

2 Architectural Overview

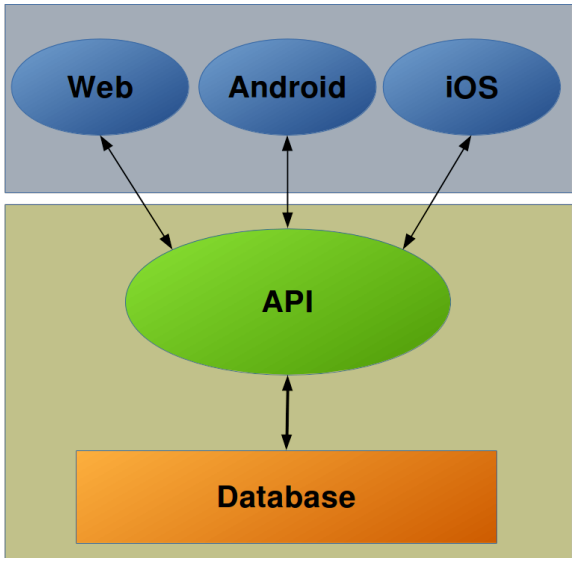


Figure 1: Communication between the different components

2.1 Physicians Web Application

-

2.2 TuBuddy smartphone app

The TuBuddy app is a smartphone app that aims to increase medicine intake retention. This is achieved by the use of a medication intake calendar along with notifications and various additional tools for information such as a FAQ and an information tab. The information we showcase in the app is retrieved from the database. The app aims to be intuitive, flexible and also have its core feature usable for the illiterate. In our design we explain how we aim to do this whilst also walking through the app itself.

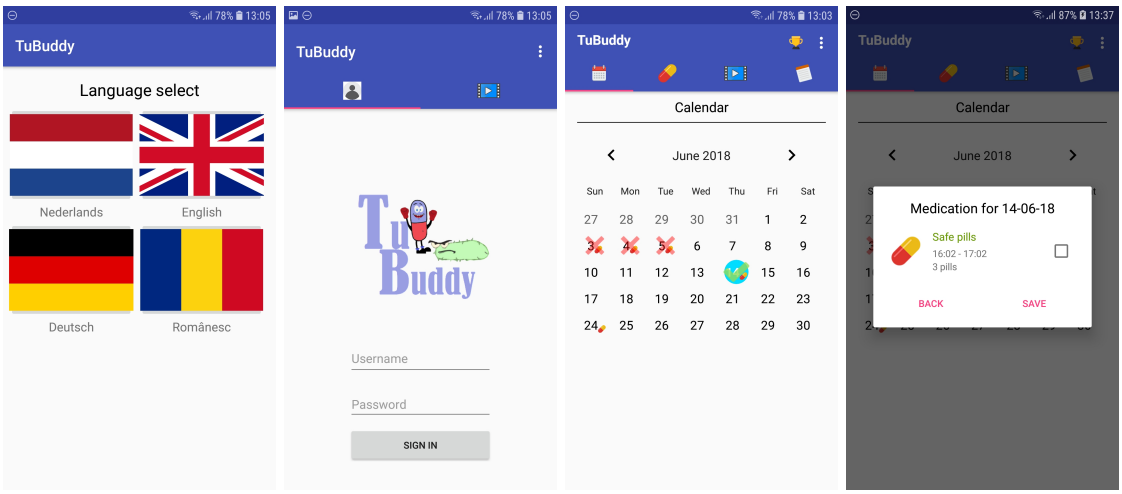
2.2.1 Design - Android

To make our app as usable for illiterate users as possible we largely make use of images in our app to display information.

Language Select The first screen that greets the user is Language select. We want our user to be able to select a different language than their phone's current language as some people don't have their phone set in their native language. This language select screen is only on the initial launch of the app and will disappear in subsequent launches as its further usage is trivial.

Sign in The sign in is relatively simple as our app is not in charge of creating accounts. Noteworthy is that we can already access the information tab. This design choice was made as it was important that features would still be available despite not being logged in. You can also go back to the language select from this screen using the icon on the top right in case you have selected the wrong language.

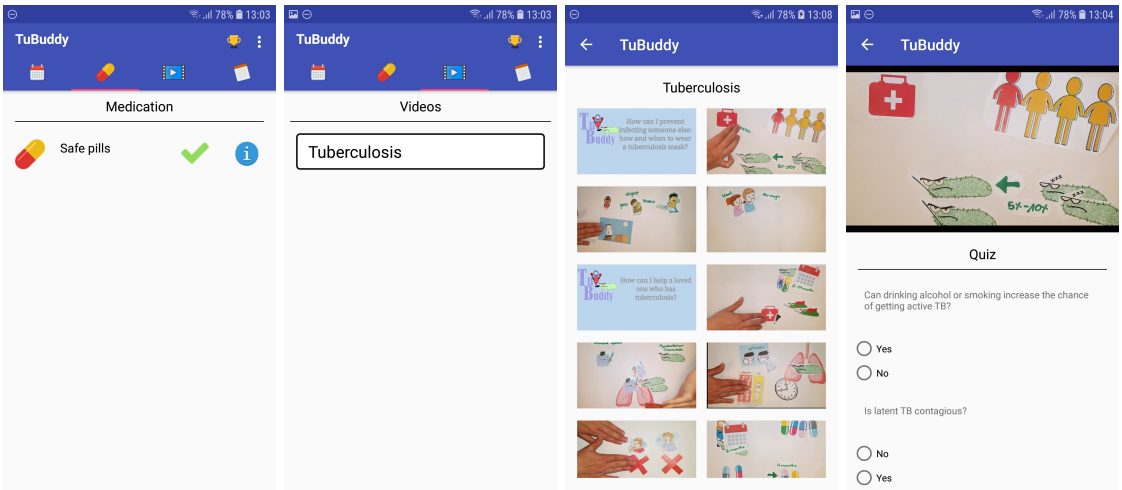
Calendar The calendar is the core feature of the app and also the first screen you see upon signing in. On the calendar screen you can see on which days you have or haven't taken your medicine as well as update your medicine intake of the current day or edit the medicine intake of yesterday. The ability to update the intake of yesterday is implemented as the possibility is high that you simply forget to update your intake in the app. When you forget to take medicine in the specified time frame in the calendar; The app will give you a notification that you still need to take your medicine. The calendar is one of the mechanics we use to boost medicine intake retention by simply making it easier for the patient to know which pills to take at what date. Updating is simply done by clicking on the date and then checking the box for the specific medicine you have taken.



(a) Language select (b) Sign in (c) Calendar tab (d) Calendar Day

Medicine The medicine tab is a simple addition for the calendar. The medicine tab's main usage is to provide additional data regarding the intake for the current day.

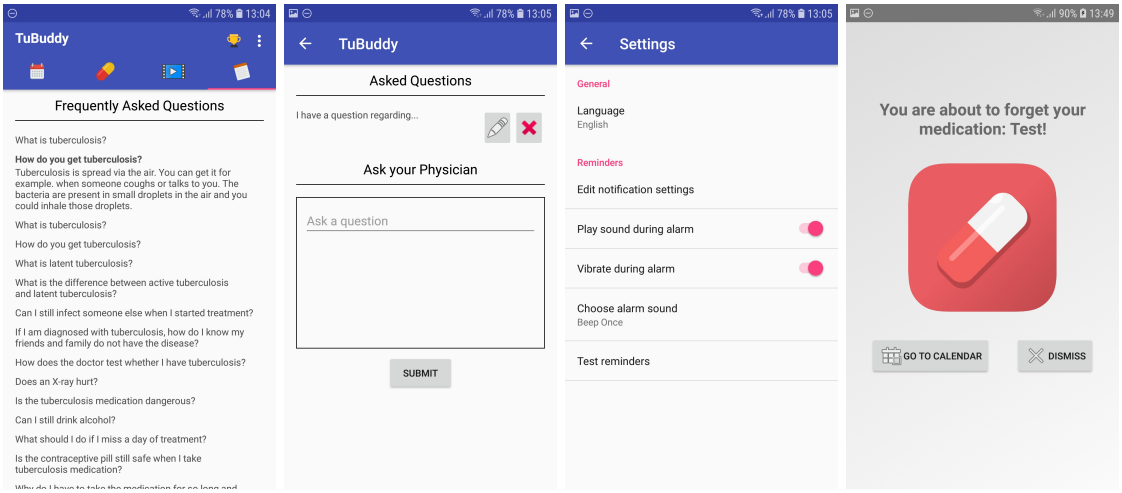
Video The video tab is used to display information regarding tuberculosis to the patient. Video form was chosen as it's a more passive form for information accumulation compared to text, which is a lot more appealing. You can simply click through to a video so you can watch it as well as make a quiz which is displayed under the video in case the user wants to know if they have comprehended the video correctly.



(a) Medication tab (b) Videos tab (c) Video category (d) Video and quiz

Notes It's hard to setup a meeting with your physician regarding every single question you might have. The FAQ tries to remove all those question marks by providing the answers. You can take a note which the physician can see and discuss with you during an appointment if the question is not answered in the FAQ. In case you want to rephrase the question or you have figured out the answer yourself you can also edit or simply delete your note. The importance of notes is that when a user has forgotten medicine or taken too many he can easily check what the correct course of action is.

Settings The settings are mostly of functional nature. In settings you have the ability to change language, as well as changing the nature of your notifications and alarms. In case you want to check if you have set up your reminder correctly you can use the test reminder functionality.



(a) Notes tab (b) Ask physician screen (c) Settings screen (d) Alarm screen

2.3 Patient App - iOS

-

2.4 Back-end

The back end of the TuBuddy Tuberculosis Treatment Application consists of a database and an API as a layer of communication.

2.4.1 File Structure

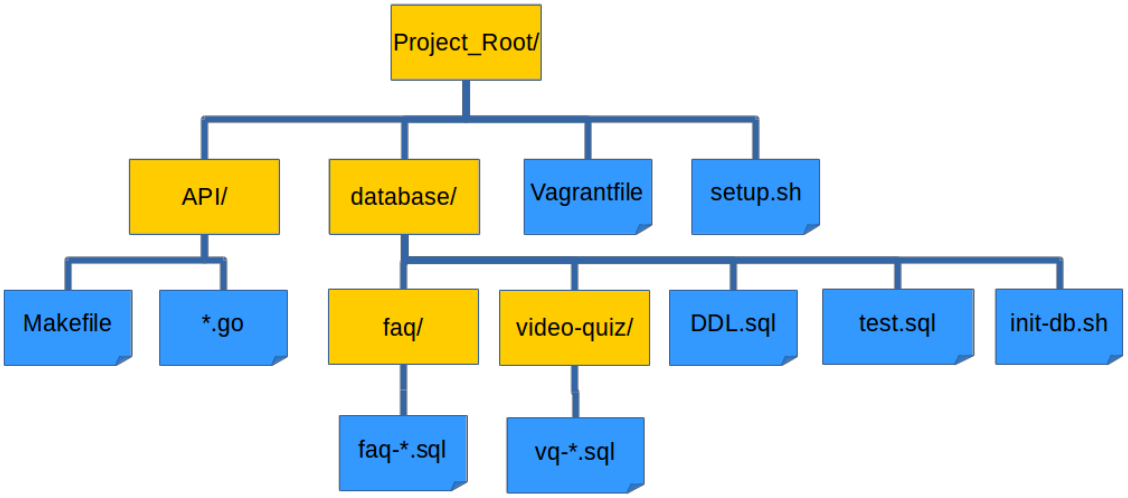


Figure 5: File Structure of the back-end project

The file structure of the project is basic, but clear. A visual representation can be seen in figure 5. Overall, the structure makes sure all files are where they're supposed to be and can be easily found. The actual content of the file is mainly described by its name. For example, all request functions which were specific to `notes`, could be found in the file `notes.go`, all `wrappers` in `wrappers.go` etc.

Because the API is spread among multiple files, we have a Makefile in the same directory which makes the process of compiling, running checks and downloading dependencies easier. We can just run `make` in the commandline to format all code, run all checks and compiles all files.

2.4.2 Database

The database has been fully modelled in ORM, so we have a good overview of all database constraints to make it easier to spot potential mistakes or errors. Having a model-first approach results in a higher quality database.

Overall the database is not very complicated. Often, the main object has their own identifier and some required fields. To discuss the individual designs in more detail, we will describe each component separately:

Accounts (Figure 6) An account consist of some basic information, such as a username, password etc. There are two subtypes: Patients and Physicians. A Patient must is attached to his personal Physician, in addition to the normal account information. A Physician has an email address for contact and a creation token, which is required for creating a new account for Patients. More on the latter can be found under the API section.

Dosages (Figure 7) Dosages are the most important object in our database, for they describe what medicine, how much of the medicine and when it needs to be taken for the specific patient. It also tracks whether the patient has taken the medicine on the specified day. To be most efficient in space we decided to split the dosages up into two objects: dosages and scheduled dosages.

Dosages contain all constant information, which is specific to the dosage. For instance for which patient the dosage is meant, which medicine etc. For each dosage, the medicine always has to be taken at the same time interval (for example between 10am and 11am). Therefore it is also attached to a dosage, and not a scheduled dosages.

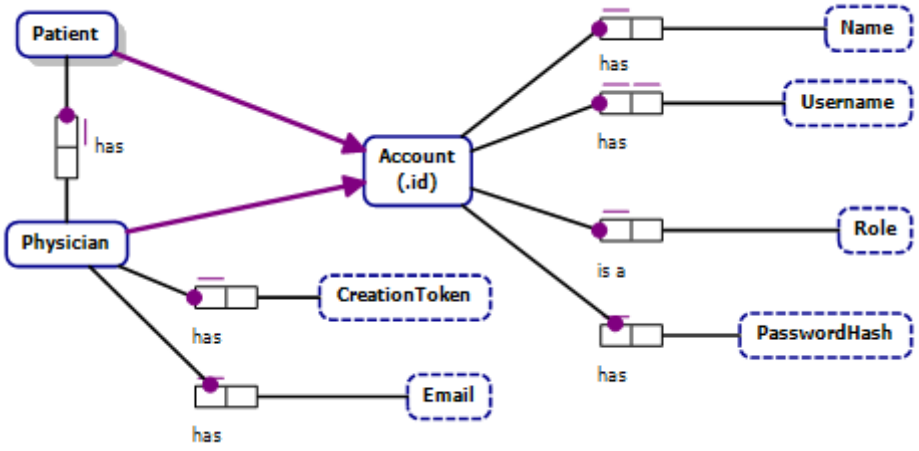


Figure 6: ORM Database model for accounts

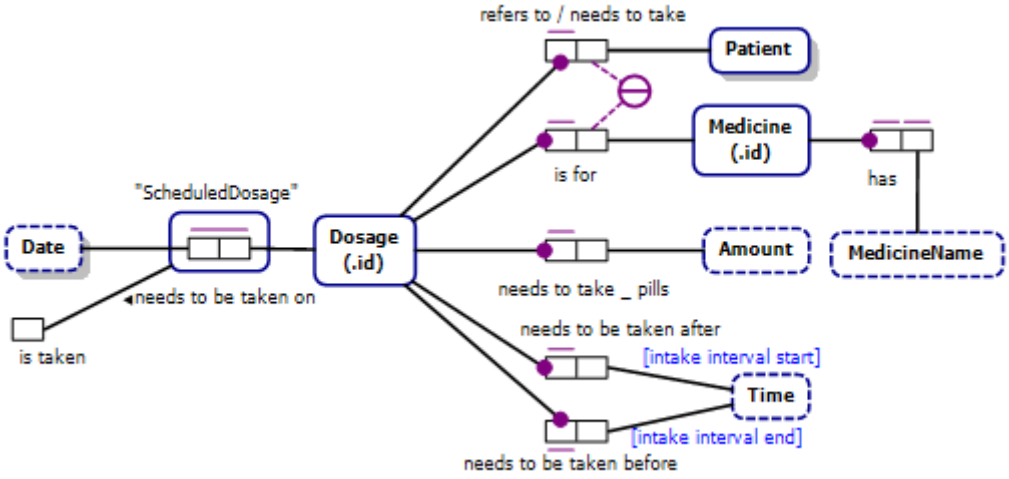


Figure 7: ORM Database model for dosages

Scheduled dosages, are a dosage (as described above) in combination with a date. So literally it is a scheduled version of a dosage, hence the name. Also, whether a dosage has been taken can vary between scheduled dosages, so it also stores whether this specific scheduled dosage has been taken.

This decision allows us to create the dosage once, and later add and remove dates on which it needs to be taken. Also, when something needs to be updated, for instance the intake interval, we can just update the Dosage object and it will automatically update for all scheduled dosages, since they all reference the same object.

The current medicine object only has a name, however more (general) information about the medicine can be added to it in the future.

In general, the dosage is identified by its identifier number, however, the combination of a patient and a medicine is also unique, which means it can be used to identify the dosage.

Notes (Figure 8) The note object is simply a question the patient has asked to the physician. It contains the question, and the creation date. This object is not very special and does not need any complicated relations.

Videos & Quizzes (Figure 9) The video and quiz objects have some constraints attached to them, which will need an explanation. Videos are identified by an id number, are in a specific language (English, Dutch, German or Romanian), and have a link to their location (in our case

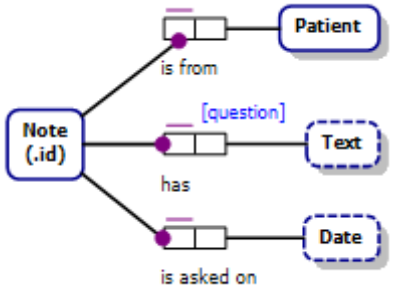


Figure 8: ORM Database model for notes

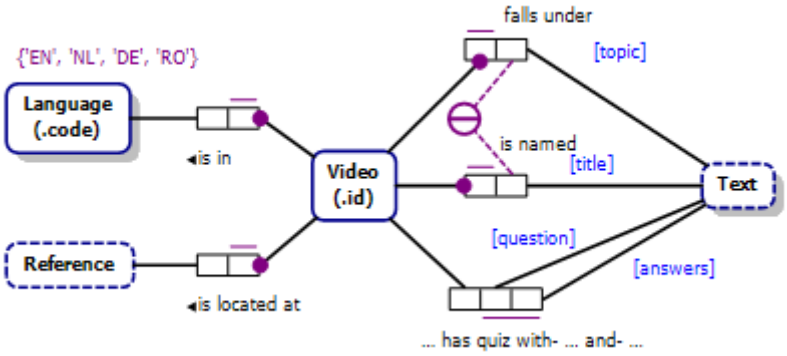


Figure 9: ORM Database model for videos and quizzes

a YouTube link). Each video also falls under a topic and has a title. The combination of title and topic is also unique, because it would be confusing to have two different videos with the same name.

A video can also have zero or more quizzes attached to them. This quiz then consists of a question and its possible answers. Because there is no fixed amount of options, our current implementation is a list of answers (separated by a semicolon, as mysql doesn't support lists). The first item in the list is always the correct answer. A downside to this implementation is that the front end always has to randomise the order.

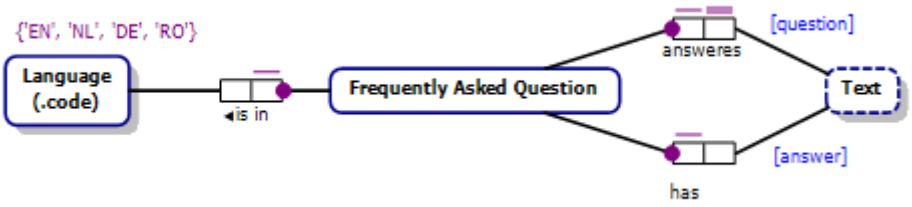


Figure 10: ORM Database model for frequently asked questions

Frequently Asked Questions (Figure 10) Similar to the Note object, the Frequently Asked Question object is very intuitive. It simply contains its language, the question and the answer to that question. The question is also the identifier of the object, which means there will not be any duplicate questions.

Languages / Translations Because our front end supports multiple languages (English, Dutch, German and Romanian), there need to be videos and faqs available for multiple languages. Currently, the database has the language as an object field, but there are some advantages and disadvantages to this. The main advantage is there are not a lot of constraints to enforce, and does not

complicate the model a lot. However, it might be desired to have a link to the same videos from different languages. Unfortunately, this is the main drawback of our current language model. An implication of this is that it will not be possible to change the translation while you have selected a video, since there is no way to know what the translated version of the selected video is.

Cascades A final database implementation are some added cascades, which are activated upon deleting an object. For example, a patient can have multiple notes, dosages and scheduled dosages connected to them. If you would want to delete this patient, you would need to delete all these rows manually. However, adding a `ON DELETE CASCADE` clause to a foreign key constraint, makes sure the object is deleted when its parent is deleted. Now, we can just delete the patient, and the database handles all other deletions.

One place where we left this out, is when you delete a physician. It would be bad if a patient account would be automatically deleted when its assigned physician is deleted.

2.4.3 API

The API is the largest component of the back-end. It handles all communication between the front end applications and the database, as can be seen in figure 1. For both the front end and the back end this is beneficial, as it makes retrieving data possible without needing to have knowledge on the underlying database structure. Also, it is easier to extend or adjust the database in order to fit specific needs, without much changes for the front end. A description on how the API works, can be found in the documentation. This shows how and to where requests should be made, and what should be expected in the response. Following is a brief description of some of the design choice made regarding the creation of the API:

Wrappers Go programming language supports and encourage the use of middleware functions. Similar to OOP Decorator Pattern, our middlewares (wrapper), add functionality to a given call. We currently use 2 wrapper functions:

1. *handleWrapper* is used to create the the HTTP response the API is going to send back upon being called. It can cal either an API function or the *authWrapper* middleware function. It can also return upon failure to create an HTTP response.

2. *authWrapper* is used for requests that need to have restricted access. It is always called by *handleWrapper* and it check the credentials of the user that made the request. It can either call an API function, or return with an appropriate HTTP error code upon failure to validate a user.

Authentication Authentication is carried out via JWT (JSON Web Tokens).
The login procedure (function), checks the password of a user against his/her username. If the check succeeds, the API creates an JWT, using password and username as claims, and signing it with the HS256 method (for development using the string "secret" as seed). The token is then encrypted (for development using a simple Cesar Chiper), and embedded in the body of the HTTP response, together with the ID that identifies the user on the database (needed for most GET requests).
When a access restricted request is made, it will have the released token passed as an header parameter, and the provided ID passed as URL variable.
The *authWrapper* function will check the validity of the token as follows.

- The token is decrypted (the seed is store in the database for each patient)
- The token is decoded and checked for JWT compatibility (to see if it has been tampered)
- The username is retrieved from the token and checked to see if it matched the ID passed as parameter (otherwise an authenticated user could see everyone's information)

If all the checks are passed, the function that retrieves the requested data from the database is called.

3 Technology Stack

3.1 Physicians Web Application

-

3.2 Patient App - Android

- **Android Studio**
  It was the clear best choice to work in android studio as it provides a vast amount of tools as well as having a massive amount of information available for it online.

- **InVision** When designing the app we used Invisio as it allowed us to give our customer layout design which you could click through. It's quick use allowed us to quickly revision the layout according to our customer's needs.

3.3 Patient App - iOS

- **Flutter**
  The iOS app is built using Flutter, an open-source framework developed by Google. Flutter utilises Google's programming language Dart, an object-oriented programming language that looks somewhat similar to Java and C#, but less verbose. Flutter code cross-compiles to Android and iOS with very little overhead, often outperforming similar cross-platform solutions such as Xamarin.
  The power of this framework lies in its fast development: Flutter's hot reload helps developers quickly and easily experiment, build UIs, add features, and fix bugs faster, by offering sub-second reload times, without losing state, on emulators, simulators, and physical hardware for both iOS and Android.

- **IntelliJ IDEA**
  The Flutter team offers a plugin for IntelliJ IDEA, the popular IDE by JetBrains. Since the members of the iOS team were already accustomed to IntelliJ, this made developing very user-friendly. The aforementioned plugin offers code completion, short-keys for common tasks such as formatting and linting, as well as buttons for deploying the app to emulators, simulators and physical devices.

- **Travis CI**
  In order to prevent the introduction of faulty code into the code base, we made use of Travis CI for continuous integration. Travis allowed us to automatically run unit and integration tests against the committed code, blocking pull requests if one of the tests fails.

3.4 Back-end

- **Golang (Go)**
  To create the API, we used the programming language Go. Go is developed by a team at Google and is completely open source. From the official Go website:

  "Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."

  Within Go, we used some tools and packages. These are some worth mentioning.

  - **Mux** : A package we used to make handling requests and making the listening server easier.
  - **Go fmt** : A tool which formats your code. For example, it removes double whitespaces, adds whitespaces after comma's and neatly aligns columns in a struct.
  - **Go vet** : Detects some likely mistakes in your project before compiling.
  - **Go lint** : This assured some styling of our code. For example it assures all variables are named with camelCase and each exported function/method/struct is documented with a comment.

  - **Errcheck** : This makes sure all errors are being checked, and not just silently fail.

- **Travis CI**
  Continues Intergration software we used. The main use case was verifying all the formatting / linting (some described above), verifying the database DDL statements and whether the API code compiles successfully.

- **MySQL**
  The database is created in MySQL. The main reasons for choosing MySQL are: it is SQL-based (therefore we could use previous knowledge), it is easy to use and has a large community which means there is a lot of help available.

- **Postman**
  Postman makes it easy to make requests to the API, which makes testing (manual and a tiny bit of automated) easier. In addition to testing, we also used Postman to document the requests which can be make to the API.

- **Vagrant**
  Vagrant is software with a similar purpose as Docker, however it is more general. Vagrant can be used in combination with different Virtual Machine or Container programs. Vagrant normally is run in combination with VirtualBox, so we decided to use this.
  It is possible to create a new VM with the command 'vagrant up', which then also runs a setup script downloading and installing all dependencies (e.g. Go, MySQL, etc). This way, every member of the team can easily test the API locally, without having to install all the technologies.

  A slightly more extensive guide on how to use Vagrant for our project can be found in the README of the Github project.

4 Team Organization

4.1 Front-end

Physicians Web Application

- Teodor Ionut Oanca
- Rutger Berghuis
- Andrei Scurtu

Responsibilities: *Pysicians Web Application.*

Patient App - Android

- Marco Lu
- Pieter Jan Eilers
- Roel Brandenburg
- Robert Riesebos
- Niek de Vries

Responsibilities: *Patient Mobile App for Android & User Interface Design for Patient App.*

Patient App - iOS

- Julius van Dijk
- Noam Drong
- Hidde Folkertsma

Responsibilities: *Patient Mobile App for iOS.*

4.2 Back-end

- Giacomo Casoni
- Sten Sipma
- Sytze Tempel

Responsibilities: *Database & API*

5 Change log

Date	Contributor(s)	Section(s)	Description
22/03	Sten Sipma	All	Created first basic layout for document + added team members & responsibilities
22/03	Marco	All	created basic introduction and filled in most of the android stuff
30/03	Rutger Berghuis	All	Updated the introduction and wrote the part on the webapp
12/06	Sten Sipma	Overall layout + Section 3 for Backend	Recreated the layout for the final version of the document. Changed the team compositions around. Added the technology stack for the Back-end team.
13/06	Sten Sipma	2	Added the Architecture Overview of the database, the file overview for the back-end and a small start of the API overview.
14/06	Marco Lu	Introduction + Android sections	Added a new introduction, added the TuBuddy design for android, added the general TuBuddy mobile philosophy and added the android technology stack.
14/06	Niek de Vries	Android sections	Added images to android section, fixed typos and grammar