Bachelor’s Thesis

Development of a mobile routing application for local bus transportation in Göttingen

05.11.2015

Dieter Lechler
21165211
Contents

List of Abbreviations iv

List of Figures v

List of Tables vi

List of Listings vii

1 Introduction 1

2 Task 2

2.1 Initial situation 2

2.2 Approach delimitation 2

2.3 Implementation requirements 3

3 Concept 6

3.1 General concept 6

3.1.1 Android and Google Maps 7

3.1.2 Fragments 7

3.1.2.1 Google Maps 8

3.1.2.2 Time- and date-picker 10

3.1.2.3 Info 11

3.2 Data-modelling 11

3.2.1 Retrieving and conversion 11

3.2.2 Data-format 13

3.2.2.1 Time relevant data 15

3.2.2.2 Route relevant data 15

3.2.2.3 Station relevant data 16

3.2.2.4 Wrap-up 17

3.3 Use-case 18

4 Implementation 19

4.1 Used technologies 19

4.1.1 Android and Google Maps 19

4.1.2 Java and Javadoc 20

4.1.3 Android Studio 20

4.1.4 SQL and SQLite 21

4.1.5 XML 21
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>Android Development Tools</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>APP</td>
<td>Application</td>
</tr>
<tr>
<td>AVD</td>
<td>Android Virtual Device</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-Separated Values</td>
</tr>
<tr>
<td>DVM</td>
<td>Dalvik Virtual Machine</td>
</tr>
<tr>
<td>EPON</td>
<td>Einsatzplanung Für den Öffentlichen Nachverkehr</td>
</tr>
<tr>
<td>ERM</td>
<td>Entity-Relationship-Model</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>GÖVB</td>
<td>Göttinger Verkehrsbetriebe</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>iOS</td>
<td>iPhone OS</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RDMS</td>
<td>Relational Database Management System</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Implementation Requirements</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>An example of how two UI modules defined by Fragments can be combined into one Activity for a tablet design, but separated for a handset design. (Google, 2015e)</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Opened slide-drawer</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>A Google Map instance of Göttingen with bus-stations</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>InfoWindow of the station:&quot;Platz der Göttinger 7&quot;</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Time-picker on Android 5.1.1</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Date-picker on Android 5.1.1</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>Database structure</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Use-case diagram of the interactions between the user and the app</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>Results for a test with direct connection</td>
<td>29</td>
</tr>
<tr>
<td>11</td>
<td>Route for a test with direct connection</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>Results for a test with indirect connection</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>Route for a test with indirect connection</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description of the most important files</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Description of the most important files</td>
<td>22</td>
</tr>
</tbody>
</table>
### Listings

1. First 12 lines of the file LID_VERLAUF.X10
2. Interaction between `getMapAsync(OnMapReadyCallback callback)` and `onMapReady(GoogleMap map)` from mapFragment.java
3. First four stations with both directions in the database
4. SQL-query inside the function `getCertainTime(int dayNumber, int start, int stop, int actualTime)` from mapFragment.java
5. SQL-query inside the function `completeChangeTwo(int time, int day, int start, int stop)` from mapFragment.java
1 Introduction

About 20 million passengers yearly are using the public bus transportation in Göttingen (Fuhrmann 2014). Today, in the year 2015 passengers don’t look up the drive-time for buses in timetable-books anymore. With the technology revolution in the area of mobile devices, there also a revolution in the area of information-management took place. Nowadays, passengers want to receive their information independent of place, independently of time and as easy as possible on their mobile devices.

Since the winter-semester 2014/15, the Georg-August-University of Göttingen introduced a bus-semester-ticket, that allows students of Göttingen to use the public bus transportation for free (Ebert 2014). The semester-ticket is valid for about 25000 students and adds many new potential passengers to the bus transportation of Göttingen. Students often move to foreign cities for to study and therefore often are not familiar with street-names or station-names in the new city.

In many cases people don’t know the street-names of their desired destinations, but they likely know the rough direction on a map or at least some big places or popular stores nearby. A mobile application that helps users to find bus-stations in Göttingen without the knowledge of concrete street-names would help to improve the situation of the students.

The aim of this thesis is to design and implement a routing application for mobile devices, that can be used without a concrete knowledge about street- and station-names. The application should work for the complete public bus transportation in Göttingen. The main target are the students of the university of Göttingen.

To show the importance of this thesis the initial situation and the corresponding implementation requirements are presented in Chapter 2. The aim of this chapter is to find a clear requirements-catalogue that can be used for design- and implementation-phases.

In Chapter 3 a design-concept will be created from the prior defined requirements-catalogue and, furthermore, it will be described how the single parts of the software will work together. In addition, the format and the details of the inner structure of the used data will be shown.

Afterwards, in Chapter 4 the used tools are shortly presented and a closer view into the concrete implementation is given. Moreover, some initial test on several devices will be done. Finally, in Chapter 5 there will be analysed if the created product fulfils the requirements-catalogue. Additionally, a lookout for further developments is given.

The implementation of the mobile app with the complete source-code and all used databases are stored on an attached CD.
2 Task

Due to several influencing factors it is necessary to structure the task. First the initial situation is described, which creates the basis of the task. Later the delimitations of the approach are shown. Furthermore, the limitations are checked against the initial situation, so that no problems between these two states will occur. In the last part of the section the implementation requirements are revealed. Together these statements will clearly show the requirements for the design and concept of the app.

2.1 Initial situation

Since the winter-semester 2014/15, the Georg-August-University of Göttingen has a bus-semester-ticket for their students. This ticket allows students of the University of Göttingen to drive freely with the public buses inside Göttingen and vicinity (Ebert, 2014). Due to this change more and more students use the public bus transportation and therefore the interest for an easy way of getting information about the bus-line has increased. At the moment there are two problems, which make it difficult to use these services conveniently. The main problem is that many of the students don’t know about the bus-lines and neither about the street names or rather the names of the bus-stations. The reason why the students don’t know about these details is that many of them didn’t grow up in Göttingen and only moved to Göttingen for their studies. A further problem to consider is that the students often want to receive information about the bus-lines ad-hoc and if possible independently from their location. At the moment there are no solutions, which solve all of these problems. Some solution exist, which are able to solve some of the mentioned problems, but not all of them. For instance, the web-based route search engine from the VSN (Verkehrsbund Süd-Niedersachen) (VSN, 2015), where the user have to enter the desired stations, day and time. Afterwards the engine will suggest some possible connections. Again here is the problem, that users need to know the exact names of the bus-stations.

Another important fact about the situation is, that Göttingen is quite a small town and is also build very compact. Due to these circumstances most of students in Göttingen just walk or ride by bike for transportation. Although students have the chance to drive freely with the bus, they often prefer bikes or just walk, because it is to inconvenient to find information about the bus-lines in Göttingen.

2.2 Approach delimitation

The aim of this thesis is to develop a mobile app, which should make it easy to get information about the timetable for public buses in Göttingen. Within this process the two phases
of modelling and implementation should be covered. The result of the thesis should be a working implementation of a mobile app, where users can get information about the bus-timetables in Göttingen in a convenient and fast way, even if they don't have any knowledge about station-names.

The required mobile app should have a graphical interface, which is usable with common known smart-phone handling-techniques. The app should provide a map of Göttingen with all public bus-stations in Göttingen in it. Users are supposed to select desired start- and destination-stations by clicking on them and set them as start- or stop-station. With this special representation users now just need to know the rough location of their destination and can choose their bus-stops easily. Because of the special situation in Göttingen, results need to be accessed and shown fast and easy. This requires a fast handling of the app and also a fast routing-algorithm.

Respecting to the targeted platform, several limitations are given. First of all the result of this thesis needs to be a mobile app, which is designed and implemented for mobile devices. Regarding to the Operating System (OS) of the target platform two possible options are given. Android or iOS, but in both cases a native app is demanded. This means that the app will be specifically programmed for a certain OS and will be only compatible for this OS.

The geographical position of the user is more or less an optional feature, which is not necessarily required, but nice to have. The geo-position is supposed to help the user getting a better orientation on the displayed map of Göttingen. It also could help the user to find the start-station by just choosing the station just next to the current position.

At the beginning of this thesis no data is given at all. All required information about the bus-infrastructure inside Göttingen and it’s vicinity were not available. This includes positions of the stations, routing of the bus-lines and also all relevant data about arrival- and drive-times of the buses.

The way of retrieval, and additionally, the way of storing this data remained unspecified and therefore hat to be considered as well. However, the focus of this thesis is not to deal with searching and updating relevant data, but to use them in an efficient and appropriate way, so users can retrieve desired information easily. Also the way how data is stored or used, can be chosen freely. Furthermore, the way of processing and evaluation is free to decide. Or in other words, it has to be developed without help, which is probably the biggest part of this thesis.

2.3 Implementation requirements

According to the preconditions and the given delimitations, now it is possible to derive concrete requirements for the implementation. The basis for the implementation is a mobile app with a map of Göttingen, where all streets and bus-stations of Göttingen are visible. Addi-
tionally, the related names of the stations and streets are required. The app could be developed for Android or iOS, but deferentially as a native app. In terms of targeted platform and additional software, there are no more restrictions, which means that any additional software or framework can be part of the app.

Users should be able to choose stations for routing without knowing their respective names, but only by knowing their rough locations. This doesn’t necessarily mean that users with knowledge of street-names and bus-station-names should have a disadvantage compared to users without any knowledge. After choosing two stations the app shall process a possible route between these stops. This process includes two complex steps. The first step is to find a possible route between those stops. Afterwards it is necessary to find the very next combination of buses, which fits best for the requested query. Both steps imply that several optimisation algorithms for routing and time-querying are needed. In addition the suggested route and the related times need to be shown to the user. As there is no data or neither data-structures are considered, it is not clear yet how these algorithms will look like. Furthermore, it remains unclear on which level of abstraction this will happen.

As mentioned before the app should be easy to use and show results quite fast, otherwise students will still use bikes or walk. This will require a fast and somehow slim implementation of data to achieve fast results within the routing.

Although results are needed ad-hoc, users still should be able to choose different times and dates for their queries, so they can plan their transportations early for future trips.

The current geographical-position should be visible for a better usability. Beyond the geographical-position a compass would help the user to get a better orientation on the displayed map as well.

There are no limitations in terms of used techniques. It is not predetermined whether data is stored locally or is retrieved over the internet. This also implies that existing web-service with local-bus-routing could be used. So that the app will just send queries from the user and send those to some web-service and retrieve the computed results. In addition there are no limitations about third party software, like editors, frameworks or IDE’S. Furthermore, there are no special requirements about the exact look of the app.

In Figure 1 one is able to see the criteria-catalogue of implementation requirements, which are derived from the current situation and the approach delimitations. This catalogue creates the basis for the further design and implementation choices for the app.
FIGURE 1: Implementation Requirements
3 Concept

In this chapter the concept of the app is modelled. As a guideline for creating the product, the previously created implementation requirements are regarded. Tools like mock-ups, use-case diagrams and database schemes are used to create and model the concept. The creation of this concept has two intentions. First of all this concept has the aim to give an overview of functionality and the used data. Secondly, this concept has the purpose to give a prototypical implementation of the product. The first part of the chapter will deal with the used platform and the coarse structure of the app. Later the used data and the data-format is shown in detail within an ERM (Entity Relationship Model). At last, a use-case shows the expected interactions between the user and the app.

3.1 General concept

The following subchapters will deal with the general design of the app. First of all a brief overlook for chosen the platform will be given. Afterwards a general design concept of the individual parts of the app are shown.

3.1.1 Android and Google Maps

As the implementation requirements give the option to choose between a native Android or a native iOS app, Android was picked. There are different reasons for this decision. One of the main reasons is the good Google Maps implementation on Android. Concerning that Google Maps is the most used mobile map app in the world and even one of the most popular mobile apps at all, Google Maps was picked to show the map of Göttingen and bus-stations of Göttingen on it (Statista, 2015b). Due to the high popularity of the mobile Google Map app, one can assume that most smartphone-users are familiar with the app. Concerning this fact, a Google Maps implementation should make a good basis for the app. Due to this familiarity users should have a high chance to easily handle the app.

As Android and Google Maps are both developed by Google, one can assume, that the Google Maps implementation on Android is at least as good as on iOS. In addition, Android-systems clearly dominate the global market and therefore many more students will be able to use the app (IDC, 2015).

Another reason to prefer Android is that Android is an open source project, which is free to use and moreover free to develop for (Google, 2015). However, iOS is a proprietary software, which means that it is a non-free software. It costs at least 99$ to get a developer licence for one year and this would strongly increase the project costs (Apple, 2015).
3.1.2 Fragments

For a structured, flexible and dynamic user-interface (UI), Android Fragments are used for this thesis. An Android Fragment is a class allowing a modular and interactive design. Android is able to show several different Fragments at once and also process data from different Fragments in parallel. Moreover, Android is able to process data between different Fragments and therefore create a fluent data communication inside apps. These flexible design options lead to a very specific and individual Fragments creation for each implementation requirement.

Android Fragments are placed inside an Android Activity, which is a class to create UI's. "An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface." (Google, 2015a).

These two components one can develop interactive apps, which are suitable for different smartphones with different Android versions and different screen-sizes. Figure 2 shows how Activities and Fragments work together.

![Image of Activities and Fragments](Google, 2015e)

The Figure 2 nicely shows how Fragments and Activities can adapt to individual screen-sizes. Furthermore, it shows the versatile possibilities to use Fragments and Activities for different scenarios.

As now the power of Fragments and their flexible options are presented, the single Fragments...
are used to fulfil all implementation requirements. The idea behind this Fragment-design is that users are able to switch between desired Fragments. With this mechanic developers can easily and quickly implement changes between contents. To implement this variable design, a slide-drawer will be used. Inside this slide-drawer all Fragments can be chosen from a list. The slide-drawer opens and closes on interaction with the menu-toggle on the left upper corner. The last selected item is displayed with a green background, as you can see in Figure

For a better focus on the slide-drawer, the right half of the app will turn darker.

In the following subchapters the designs for the different components are created.

3.1.2.1 Google Maps

First of all the Fragment for showing a Google Maps instance is created. Inside this Fragment most of the user-interaction will take place. Almost all functions of the app are implemented inside this Fragment and therefore it is the most important one.

When we recap the implementation requirements, then one sees that firstly a map of Göttingen should be visible. In addition, the public bus-stations of Göttingen also need to be shown. The concrete implementation and functionality of the map and bus-stations will be developed later in the chapter 4 Implementation.

Furthermore, this Fragment has to show the current position of the user and also a compass for a better orientation on the displayed map.
In Figure 4 and 5 the Google Map Fragment with a snippet of Göttingen can be seen. In the upper left corner a compass shows the current direction of the mobile device, where the red end directs to north. In the upper right corner is the geographical-position button. When the button is clicked, the map snippet of the current position of the user will be shown. Each green custom icon represents a bus-station. When a user clicks on a station an info window opens and shows the name of the selected station. This process is visible in Figure 5. On the bottom of the InfoWindow is a button, which enables the user to set the selected station as start or end station.

At the bottom of the map Fragment is the search button can be found. After the start- and stop-station were selected, the user can search for possible bus-lines between the chosen stops. The results of the request should be displayed within a new window. The belonging route of the bus should be displayed directly on the map, to provide the user with better feeling of the picked route. Furthermore, two extra icons should represent the start-, stop-station and all needed changeover stations.
3.1.2.2 Time- and date-picker

As the implementation requirements says, users should be able to change time and date for requests. Thereby users can retrieve routing-information for future bus-lines. Android provides some time- and date-pickers, which simplifies this process. Those pickers are actually dialogs, which can be displayed inside a Fragment. Since the time- and date-pickers are actually very similar functions in terms of usage, implementation and behavior, these are merged in this chapter. Users can select one of the fragments from the slide-drawer. After choosing the time- or date-picker, the demanded picker-dialog inside a Fragment will appear.

The specific style and look of the pickers will differ with different version of Android. The reason for this is that those pickers are Android core-functions and rely on the installed Android version on the smartphone.

This two Fragments should not fully fill out the complete screen-size, as for instance the Google Map Fragment does. For the pickers it is enough, when a dialog will appear in the foreground. Around the created dialog the rest of the app is visible.

Two possible implementations can be seen in Figure 6 and 7. Both figures show an implementation on an Android 5.1.1 test-system. The used Android version is a clean and original OS from Google without any external third-party software.
Both pickers should show the actual time and respectively the actual date as a default vie, when the pickers are created.
After the user has chosen a specific time or date, a small dialog should appear and confirm the requested change and display the chosen values.

3.1.2.3 Info

The last Fragment is the info Fragment. It’s the only Fragment without any interactive functions. The info Fragment only shows some text elements and a small icon. The only function of this Fragment is to give the user some additional information about the app, which aren’t necessary for the functional usage of the app. The info Fragment provides information about the developer, legal rights and warranty conditions.

3.2 Data-modelling

The chapter data-modelling will deal with the used databases. After a short lookup how the data is retrieved, a description of how and why the new data has to be converted, is given. Afterwards the format of the data will be described. A database scheme will illustrate the inner structure of the data.
It is really important to understand the structure of data, otherwise it is not possible to receive the required information correctly.

3.2.1 Retrieving and conversion

As already mentioned in the implementation requirements, no data is given about the bus-lines in Göttingen. All required data had to be collected somehow before.
After contacting the Göttinger Verkehrsbetriebe (GÖVB), which is the bus company for the public bus-system in Göttingen, they send me a complete data-set for all public buses in Göttingen.
The GÖVB uses the EPON (Einsatzplanung für den öffentlichen Nachverkehr) system, which is a common tool to manage public bus transportation systems. EPON uses the data format VDV 452-format. The data-structure of the VDV is quite similar to the Comma separated values (CSV) format.
The complete data-set consists of 30 files of the type X10. The file-type X10 is the common output format of the EPON system. At the beginning of each X10 file there is some metadata, which gives information about the used format and the type of the given information inside the document.

**Listing 1:** First 12 lines of the file LID_VERLAUF.X10

```plaintext
1  mod;DD.MM.YYYY; HH:MM: SS; free
2  src;"epon−VDV−Schnittstelle −91";"27.02.2015";"11:28:24"
```
As you can see in the Listing 1 line 1-8 are EPON only specific information, which gives information about used standards in the file. In line 9 the names of the data-entries are given. The names are directly mapped to the values-entries, which start at line 11. At line 10 the data types of the entries are declared. From line 11 to line 4516 are datasets saved. All other 29 X10 files have the same structure, but with different data.

Since this is format is not very convenient to use in systems, which are different from EPON system, a new data-format is required. As databases are one of the most used technologies for storing, managing and modifying data, it is reasonable to convert the data into databases. Furthermore, Android has several tools and functions for working with databases.

As mentioned earlier in this chapter the format of the X10 is quite similar to CSV files. There are only a few steps that are necessary the convert X10 documents to databases.

First of all the EPON relevant data needs to be deleted. Now the file has a CSV format and just needs to be renamed from .X10 to .CSV. Afterwards the modified files can be transformed easily to databases. There are many tools, which are able to convert CSV files to databases. For this thesis the the "SQLite Database Browser" from the Mozilla Foundation is used.

For converting CSV files, it is required to know which data is actually needed in the end. To make this decision it is necessary to initially get familiar with the data-structure first.
3.2.2 Data-format

In this subchapter the inner structure of given data is analysed. Moreover, the relations between single data-sets are important for further structural decisions. Due to the fact that the retrieved data from the GÖVB is a complete set of all data for public transportation, there are many non-relevant data referring to routing. For instance there are some files, which deal with the management of the buses and the drivers. After filtering the non-relevant data, only eight files are remaining. In fact each file represents an own database and holds data for different issues. Since it is easier to maintain one big database, the eight databases are merged to one. Each individual database is now represented as a table in the new merged database. In Figure 8 the schema of the created database can be seen. The used tables are arranged into three categories. This arrangement represents the purpose of the individual tables more easily. Actually, in Figure 8 are twelve tables, but not eight. The first reason is, that the table "android_metadata" is automatically generated by Android. Since the data inside this is table not relevant for the inner structure of the data from the GÖVB the details of the table are not considered. Another reason are the three tables outside of the categories, which are compounded tables from the original data-set. Therefore, these three tables are not actually part of the database, but those represent the queries that are needed to get specific information about time, routes and stations. Each of these three tables only consist of entries from their belonging tables. The connections between those tables can be seen as grey arrows. Next to the arrows the shared entry can be seen. Each of these entries inside the created tables are foreign-keys, which are marked with a yellow arrow on the right side of a table. The magnifiers on the left in the tables are symbolizing a reference to another table.
FIGURE 8: Database structure
3.2.2.1 Time relevant data

The category "Time relevant data" holds three tables: REC_FRT, REC_FRT_HZT and SEL_FZT_FELD. All temporary values are stored in seconds, which means that e.g. the time 23:39 would be stored as 85140. Whereas weekdays are represented as integers from 1 to 7. The first table REC_FRT holds information about the starting-time of the single routes. For instance the route 103 with bus-number 12 starts at 28200 and day 6. Unfortunately, the table only shares information about the day and start-time of a certain route, but not at which station and, moreover, not at which time the bus will arrive at upcoming stations.

The other two tables inside the time category are very similar. SEL_FZT_FELD and REC_FZT_HZT provide the needed time to calculate the drive-time within a route. SEL_FZT_FELD stores the drive-times between every two adjacent stations, in contrast REC_FZT_HZT stores the stop-time at each individual station.

This way of storing time relevant information causes several extra computations to retrieve the needed drive time for a certain bus. In practice this means that if one wants to know the drive-time between two stops A and Z it is necessary to individually calculate the time between each stop in between (A : B + B : C + ... + Y : Z). Overall the table "Time" shall now show all relevant data to calculate specific times inside the system.

Here one can see the important information from "Time relevant data" category:

- Start-time and day of routes
- Drive-time between adjacent stations
- Stop-time at each individual station

3.2.2.2 Route relevant data

The second category "Route relevant data" holds information about the all routes of the system. In this data-set a route describes all stations within a trip of a bus from the very first start-station to the final destination. The category "Route relevant data" also has three original tables: "LID_VERLAUF", "REC_LID" and "REC_SEL". From these three tables the compounded table "Routes" was created.

The table "LID_VERLAUF" is the main-component of the route-system. Inside this table are all stops saved for each specific route. The system even distinguishes between the direction of a bus and start-time of the route. Therefore every time a bus starts a new route, there is a individual route-number for this and every station inside this route is saved. This means that for each direction, day, time and bus-number a route is stored. For example the bus-number 12 owns 227 different routes.

The second table "REC_LID" offers information about the lines. In this data-set a line is defined as a group of routes for a bus-number with the same start- and destination-station,
but with different start-times. In most cases there are two lines for each bus-number, as the bus serves most stations into two directions. Sometimes buses have several lines. This usually happens when a bus-number additionally serves special lines, like late night-buses. E.g. the bus-number 1 has three different lines: Holtenser Berg - Zentrum, Holtenser Berg - Straßburgstraße and Zentrum - Holtenser Berg.

So "REC_LID" stores data for the name of a line, the corresponding bus-number, the belonging route and the direction of a line.

The last table "REC_SEL" holds information about the distances between adjacent stations. The way of storing this distances is similar to the format in the table "SEL_FZT_FELD", where drive-time between adjacent stations is stored. Therefore, if one want to calculate these distance between two stations A and Z, one need to calculate all sub-distances in between \((A : B + B : C + \ldots + Y : Z)\).

Overall the table "Routes" now holds all required information for a route.

Here, one can see the most important information from the category "Route relevant data":

- All stations inside a route
- Lines of specific routes
- Direction of lines
- Names of directions
- Distances between adjacent stations
- Route-numbers

### 3.2.2.3 Station relevant data

The third and last category "Station relevant data" deals with data for all stations inside the data-set. Inside this category there are only two original tables: "REC_HP" and "REC_ORT". For this table also one compounded table was created from the originals. The constructed table "Stations" shall hold all required informations for stations.

The first table "REC_HP" shows data about the hold-points in Göttingen. In this data-set hold-points are defined as points at stations where passengers are able to enter the bus. In most cases there are two hold-points per station, because most buses serve stations into two directions. In some exceptions a station has three or more hold points. This mostly happens when stations are very big and provide several hold-points like central-stations usually do.

The second table "REC_ORT" is the main component of category "Stations relevant data". The table holds information about the specific position of stations and their names. In the data-set each station is treated individually even if the stations have the same name. Therefore every station is least twice in the data-set with the same name, but with different
geographical-positions.
An important remark here is that the geographical data inside the set is stored in World Geodetic System 84 (WGS84) with the format (GGG/MM/SS/NNN). Here the G’s represent the degrees, M’s the minutes, S’s the seconds and N’s the decimal places of the seconds. Unfortunately, Google Maps uses a decimal-degree format. Therefore a conversion of the complete geographical data need to be made. The conversion is done by this formula:

\[
\text{Decimal degree} = \left( \frac{\text{seconds}}{60} + \frac{\text{minutes}}{60} \right) + \text{degree}
\]

Here are the most important information that can be retrieved from the category "Station relevant data":

- Exact geographical position of each station
- Names of stations
- Hold-point at stations
- Station-numbers

3.2.2.4 Wrap-up

Now all relevant data for the mobile app are summed up in one big database. Of course it is insufficient to store the the required information. At least there are some efficient queries and routing algorithms needed, to handle complex user requests. At the moment there are eight tables inside the database which hold information about the bus transportation system and to get the right results many Structured Query Language (SQL) queries are needed. The concrete implementation of these functions are later shown in the chapter Implementation. The three categories (time-, route- and station relevant data) often need to be queried in parallel and within this queries very big tables can be created temporally. Since the database holds 36369 entries, it is not unusual that with complex queries tables with more than 200000 rows and 10 columns can occur. This will create a table with 2 million entries, which need to be created, ordered and searched for. Even worse, all these computations and comparisons have to be done on a mobile device.

The performance of the developed system with the created database has to be tested later. At the moment only predictions about the runtime are possible.
### 3.3 Use-case

To create a better view on the interactions between users and computer-systems it is reasonable to create a use-case diagram with the modelling language Unified Modelling Language (UML). Furthermore, the use-case diagram should create a better understanding of the required functionalities and requirements of the app. [Schalles (2013)]

In Figure 9 the interaction between a user and the developing app can be seen in a form of a use-case diagram. The diagram shows the typical use-case, where users query for possible routes between given stations. Within this process it is necessary to select two stations from the map. The app shall now compute possible routes for the given query. This should happen with respect to the fastest available connections. Moreover, the very next available connections shall be found. Furthermore, the system displays the results to the user. Those should be displayed as a driving-route on the map.

Additionally, users have the option to change the time or the date for querying. Since this function is an optional feature, this case extends the use-case. An additional extension-point was created for this extension.
4 Implementation

In this chapter the concrete implementation of the earlier created concept in Chapter 3 is shown. Firstly, the used technologies for the developing process are introduced and described. Afterwards, the concrete Android implementation with the developed functions and algorithms are presented.

4.1 Used technologies

4.1.1 Android and Google Maps

Android is an open source platform and an OS for mobile devices. Android was established 2003 and bought by the Google company 2005. Since then Google is developing the Android system together with the subsidiary company Open Handset Alliance. The Android core is based on a Linux kernel and runs on the Dalvik Virtual Machine (DVM). The DVM runs code, that was translated for a Java Virtual Machine (JVM) and creates machine code for the underlying hardware. Since the DVM is Java-based and expects Java-code, all Android app’s are written in Java (Google 2015b). Design and layout related components in Android app’s are done with Extensible Markup Language (XML) code.

From 2011 until today the Android OS is the world market leader on mobile devices. Today Android has a global market share over 82% (Statista 2015a). Every developer can feel free to develop any mobile application he wants to, as long as the application doesn’t contain any malicious functionality (Rick Rogers 2009a).

Android applications usually are built of from four basic components: Activities, Services, Broadcast- and Intent Receivers and Content providers. Activities are similar to standalone code-fragments, that are possible to be executed and terminated by the OS. Activities are often related to single screens and with changing to another screen mostly different activity will be loaded. Activities can perform interactions with the user and can also contain services, broadcast- and intent Receivers and Content providers (Rick Rogers 2009b).

Android provides services, which run in the background of the system and are similar daemons on regular OS’s. Those services provide functions like reading the geographical position from the installed GPS-sensor.

Broadcast- and intent receivers are necessary to respond on service request from other applications or functions of the system. E.g. activities and services use intent receivers to provide access to their functions to other applications (Rick Rogers 2009b).

Content providers are able to share data with other activities or services. Moreover, content providers can store, manage and interact with local databases on the system and even with databases on different non-local systems.

As Google Maps is another project of Google, of course there is an implementation of Google Maps for Android. Furthermore, there is an application programming interface (API) for de-
velopers to include a Google Maps instance in Android applications. Since Android is open source and free to use, everyone can use those features including all meta-information of maps like street names, stores, restaurants, etc.. The Google Maps API also provides functions to retrieve the current geographical position of the system and display those information on the map instance. A Google Maps instance can be easily implemented with a few lines of Java and XML Code (Xianhua Shu, 2009).

As one can see Android is a powerful platform with many different functionalities and services. Android provides a good basis for this application and enables to fulfil the implementation requirements in Chapter 2.3.

4.1.2 Java and Javadoc

Java is a class-based and object-oriented programming language, which was founded in 1995 by Sun Microsystems and owned by Oracle since 2010. Today, Java is one of the most used programming language worldwide in many different sectors like datacenters, game consoles, scientific supercomputers, smartphones, etc. (Oracle, 2015b). Java is a multi-platform system and every Java code runs inside a JVM, which sends code to the hardware. With this design one can execute the same Java code on almost every system (Oracle, 2015c).

Within this thesis Java is used for two different purposes. The main usage of Java for this work is the Android development. Despite some XML files for structural and design aspects and SQLite commands for working with databases, the app completely consists of Java code. The second usage of Java appears in the documentation. Since every software project should be documented well to achieve a high level of software quality.

The tool Javadoc from Oracle enables developers to automatize the process of documentation. Javadoc can generate a documentation from the given Java source code and the developers comments inside the code. The output format of Javadoc is a structured Hypertext Markup Language (HTML) code.

Javadoc can help to increase the code understanding of for external developers. Moreover, the understating of the involved developers can be increased, which is very useful for updates when structural details need to be recapped (Oracle, 2015a).

4.1.3 Android Studio

Android Studio is a powerful and free to use IDE for Android development. Android Studio is developed by Google, but the IDE is based on IntelliJ IDEA, which is an IDE for Java programming created by the company JetBrains.

The final version of Android Studio was released in December 2014. Since Android Studio is written in Java it is a cross-platform software and can be used on Windows, Linux and OSX (Google, 2015d).
Due to the fact that Android Studio and Android are both developed by Google, there are several circumstances that make it comfortable to develop Android applications on Android Studio. For instance, all required Universal Serial Bus (USB) drivers and Android Development Tools (ADT) can be easily downloaded and configured with Android Studio. Furthermore, the required Software Development Kit (SDK) for the targeted Android version and the corresponding API are also easy to set-up. Another important feature is the Android Virtual Device (AVD) Manager, that can emulate a complete virtual Android system, where developers can test their app’s on. Additionally, every available Android version can be used to set-up the AVD, in order to that developers can test their products on several different systems (Google, 2015c).

Android Studio has many features that help to implement the mobile app and even test the end-product in detail. Because of all these facts and the good Android integration, Android Studio is used for this project.

4.1.4 SQL and SQLite

SQL is a programming language for accessing and manipulating database systems. In 1970 the work with SQL has started and in 1987 SQL became an official International Organization for Standardization (ISO) standard (Auer, 2015).

In this thesis SQL is used to manage and access the database, which was created in Chapter 3.2.2. Android, unfortunately, does not directly support SQL-statements, but has a working implementation of SQLite in the API. Therefore, SQLite is used in this theses to handle the database. SQLite is a free to use and cross-platform relational database management system (RDMS) (SQLite, 2015). In difference to most other RDMS implementations, SQLite is an embedded system and is built into the end software.

4.1.5 XML

XML is a free open standard, which was defined by the World Wide Web Consortium (W3C)in 1996. The W3C is an international standards organization for the internet. The format of XML is textual and human readable. The main focus of XML is to create a simple and usable language, that can be used in the whole internet (Tim Bray, 2008). As already mentioned in Chapter 4.1.1 Android uses XML files for structural and design purposes. Therefore XML is necessarily one of the used programming languages in this thesis.

4.2 Technical realisation

In the following chapters the concrete technical realisation is shown. Since the developed product is far too big to show every single line of the written code, only the most important parts will be described. First of all, the inner structure of the app and implementation of the
Fragments from Chapter 3.1.2 will be shown. Afterwards, the data-connection between the app and the Google Maps API will be described. Finally, the algorithms for finding direct and indirect connections, as well as the calculation for drive-times, will be presented. All additional information about the app, classes, functions and methods are stored in the generated Javadoc, which can be found in the appendix [C].

4.2.1 Inner structure

In this Chapter the coarse inner structure of the mobile app is presented. First of all, the most important files of the system and their main function will be described. The files can be seen in the Table 1.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity_maps.xml</td>
<td>Is one of the layout-files and defines the design of the main-components.</td>
</tr>
<tr>
<td>AndroidManifest.xml</td>
<td>Holds basic information about the app, e.g. permissions and targeted SDK-version.</td>
</tr>
<tr>
<td>BUS_ALL.db</td>
<td>Compounded database with all required information about the bus-system.</td>
</tr>
<tr>
<td>ButtonAdapter.java</td>
<td>Creates a custom InfoWindowAdapter to show the belonging station-name and adds a button for setting the stations as start- or stop-station.</td>
</tr>
<tr>
<td>ImportDB.java</td>
<td>Handles all relevant functions and classes for working with the database.</td>
</tr>
<tr>
<td>impressumFragment.java</td>
<td>Holds the info-Fragment.</td>
</tr>
<tr>
<td>mapFragment.java</td>
<td>Contains the Google Map instance and connects to the Google-API. Furthermore holds all algorithms and SQL-queries for routing and time calculations.</td>
</tr>
<tr>
<td>MapsActivity.java</td>
<td>Only Activity in the app and is responsible for displaying and changing fragments.</td>
</tr>
<tr>
<td>NavDrawer.java</td>
<td>Sets the title and icon for the entries in the slide-drawer</td>
</tr>
<tr>
<td>NavDrawerAdapter.java</td>
<td>Creates the slide-drawer and places the entries.</td>
</tr>
<tr>
<td>placeHolderFragment.java</td>
<td>Creates a date-picker with the current date.</td>
</tr>
<tr>
<td>selectStop.java</td>
<td>Creates a custom DialogFragment, which appears while setting a station as start- or stop-station.</td>
</tr>
<tr>
<td>setTimeFragment.java</td>
<td>Creates a time-picker with the current time.</td>
</tr>
</tbody>
</table>

TABLE 1: Description of the most important files
The file MapsActivity.java keeps all Fragments together, whereas this is the only activity in the app and all content is shown through Fragments to the user. The calling, destroying and replacement of new Fragments is done by the MapsActivity.java. In addition, the MapsActivity.java creates the slide-drawer and handles the user interactions with the drawer. The implementation of the drawer can be seen in Figure 3.

The three Fragments impressumFragment.java, placeHolderFragment.java and setTimeFragment.java are not important for the functionality of the app, but is still necessary to fulfil the requirements from Chapter 2.3.

The mapFragment.java and ImportDB.java create the core of the app and most of the developed functions are written in these files. The mapFragment.java contains all Google Map related code and also connects to the Google Maps-API. Furthermore, the the Google Map instance with all stations is created here. Both files are required to find routes and bus connections within Göttingen. The file mapFragment.java has all necessary queries to find these information, which are send with specific user-request to ImportDB.java, where the BUS_ALL.db is queried with the requests.

The The XML-file AndroidManifest.xml is related to technical Android specific details and for instance stores the Google Maps-API key for authentication at the Google Server. activity_maps.xml is relevant for design aspects of the MapsActivity.java and partly for the representation of all Fragments.

4.2.2 Setting up Google Maps

To set up a Google Map instance in a mobile app there are some initial steps required. Fortunately, this only needs to be done once. Primarily, a Secure Hash Algorithm-1 (SHA-1) fingerprint of the created app has to be generated and registered at the Google Maps Android API as a service for the project. With this fingerprint one can receive an unique Android-API key. The API key needs to be stored in the AndroidManifest.xml and is used every time for authentication when the app connects to the Google-API (Google, 2015f). Now the map instance can be implemented and used.

The class mapFragment extends the Android class MapFragment, which automatically creates a raw Google Map instance. Since the creation of the map instance is quite critical and can take some time, it is important to wait until this process is completely finished. This is done by the Android function getMapAsync(OnMapReadyCallback callback), which waits until the map Fragment is ready to use and then automatically calls onMapReady(GoogleMap map).
This can be seen in the Listing[2]

LISTING 2: Interaction between `getMapAsync(OnMapReadyCallback callback)` and `onMapReady(GoogleMap map)` from `mapFragment.java`

```java
public View onCreateView (LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    ...
    mapFragment.getMapAsync( this );
    return view;
}

@Override
public void onMapReady(GoogleMap map) {
    // checks if a connection to the google-api is possible
    int status = GooglePlayServicesUtil.
        isGooglePlayServicesAvailable( getActivity() );

    if ( status != ConnectionResult.SUCCESS ) {
        // Google Play Services are not available
        int requestCode = 10;
        // show alert-dialog with specific error
        Dialog dialog = GooglePlayServicesUtil.
            getErrorDialog( status, getActivity(),
                requestCode );
        dialog.show();

    } else { ...

    }
}
```

To use the those functions, the class `mapFragment` has to extend the Android class `OnMapReadyCallback`. After the app has connected to the Google-API, which happens in line 10, the map is ready to be used and edited. The reason for this careful treatment is that if the map isn’t completely finished, but still is used for editing a NullPointerException is possible and the app would immediately crash.

Now all bus-stations can be loaded from the database and added to the map with a custom image of a bus-station. As mentioned in Chapter [3.2.2.3] the geographical position needs to be converted for this process first.

All stations are set on the map with a custom `InfoWindow`, which overwrites the `InfoWindow` from Google Maps. The implementation of the custom `InfoWindow` can be seen in Figure[5]
After everything has loaded and ready to use, the app will automatically zoom to a segment of Göttingen and if the user enabled the geographical-position before, then the segment of the current position will be seen. This means that users have the option to use the product without having a GPS-sensor in their devices. Moreover, users can just disable the the geographical-position if they do not want to get tracked by Google.

The complete map is now ready to use. After selecting a station, users will see the station-name and therefore are able to find desired stations with just knowing their positions. As stations can be set as start- or stop-stations the focus in the next chapters will be on finding bus-lines, routes and drive-times for specific user requests.

4.2.3 **Finding direct connections**

To find out the needed drive-time it is first of all relevant to know the concrete route. Therefore we start with searching connections between the given stops. The finding of connections is divided into two stages. In the first step the app tries to find a direct connection between two stations. If this fails the app tries in a second step to find connections between stations with additional change stations in between.

Before the querying for routes can start, there is still one issue that needs to be considered. The database from the GÖVB has information about all public bus-stations in Göttingen and since every station is at least twice on the map it is hard for users to distinguish between stations. In addition, the stations with the same names are mostly very close to another, which makes the situation even worse. Because of this inconvenient situation for the users I decided that the app should not make a difference between stops with the same name.

In practice this means that when a user accidentally selects the a station with the correct name but into the wrong direction, the system should still find the correct connections. To ensure this feature it is necessary to always look for all four possible connections between two stations.

For example a user sets "Tulpenweg" with the ORT_NR 2241 and 2242 as start and sets "Landgericht/Bahnhof" with the ORT_NR 2011 and 2012 as stop. Now the app should search for connections between the stations 2241, 2242, 2011 and 2012, but with respect to the order of the stations. Since "Tulpenweg" was set as start only routes are concerned from "Tulpenweg" to "Landgericht/Bahnhof" and not the other way round. Since a bus can only serve "Tulpenweg" 2241 and "Landgericht/Bahnhof" 2011 at once or "Landgericht/Bahnhof" 2012 and "Tulpenweg" 2242, but that would be the wrong direction. Therefore here is only one possible solution of stations in this example, which is "Tulpenweg" 2241 and "Landgericht/Bahnhof" 2011. To implement this feature we need to have a closer look at the format of the data. The table "REC_ORT" from Chapter 3.2.2.3, where the information about the stations are saved, shows that the stations are numbered in a consistent system. An extract of this table can be seen in Listing 3.
LISTING 3: First four stations with both directions in the database

```
1 ... 2 1011; "Otto-Frey-Brücke"
3 1012; "Otto-Frey-Brücke"
4 1021; "Posthof"
5 1022; "Posthof"
6 1031; "Stadtfriedhof"
7 1032; "Stadtfriedhof"
8 1041; "Gotteslager"
9 1042; "Gotteslager"
10 ...
```

For a better overview only the names and the corresponding numbers ("ORT_NR") of the stations are displayed in this Listing. As one can see in Listing 3 each station has a complete range of ten and each direction of single stop has an individual number. When a user sets a station as start- or stop-station the system automatically looks for connections with +1 and -1 of the "ORT_NR" additionally. Hereby, missclicks of the users are fine and will not lead to wrong results.

When we recap the prior example with the two stations "Tulpenweg" 2041 and "Landgericht/Bahnhof" 2011, the system will search for connections within stations-numbers 2010, 2011 and 2012 to 2040, 2041 and 2042. Since the fact, that no stations uses a number with a 0 at the end, the system never shall pick a wrong station. The same holds for stations with a 3 at the end, which would happen when e.g. one would pick the station 2012.

To actually realise now a function for finding connections between two given stops for a certain date and time, we need to query all three categories in the database from Chapter 8. The concrete SQL-query from the app can be seen in Listing 4, that is used in the function `getCertainTime(int dayNumber, int start, int stop, int actualTime)`.

LISTING 4: SQL-query inside the function `getCertainTime(int dayNumber, int start, int stop, int actualTime)` from mapFragment.java

```
1 ... 2 SELECT REC_FRT.FRT_START, LID_VERLAUF.ORT_NR,
3 LID_VERLAUF.STR_LI_VAR AS STA, REC_FRT.LI_NR,
4 LID_VERLAUF.LI_NR, LID_VERLAUF.LI_LFD_NR AS START_NR
5 FROM REC_FRT, REC_ORT, LID_VERLAUF WHERE
6 (LID_VERLAUF.ORT_NR = start OR
7 LID_VERLAUF.ORT_NR = (start + 1) OR
8 LID_VERLAUF.ORT_NR = (start - 1) AND
9 #sub-query starts
10 EXISTS (SELECT LID_VERLAUF.ORT_NR,
```
Note, that for a better overview the query was converted into a pure SQL-query, but in the app the format is different due to the required standards for SQLite.

The query searches for all possible routes where the selected stations are present. This means, that only direct connections between the given stations can be found with this query. In the lines 6-8 and 16-18 one can see the previously described feature that prevents wrong results due to accidentally wrong choices from users.

Since the system has to check the entries LID_VERLAUF.ORT_NR and LID_VERLAUF.LI_LFD_NR twice with different values, it is necessary to create a sub-query, which can be seen in the lines 10-25. Sub-queries enable the system to search the database for the same entries with different values simultaneously. The sub-query and the main-query search for the same entries, but for different stations. Later from line 26-30 the query merges the entries with same attributes.

Line 20 ensures, that only routes are picked, where the desired start-station occurs before selected stop-station.
Finding connections for a certain time is a little bit more complex and requires at least three independent queries in the database.

Firstly, the system looks for routes that start after the given time from the user and picks the route, which is closest to that time. This process happens in lines 23-25 and 30-33. This selected route is only an intermediate result and is needed to calculate the perfect solution later.

In the second step the app needs to calculate the drive-time from the very first start-station of the route until the entry-station of the user. For the test-setting from "Tulpenweg" to "Landgericht/Bahnhof" with the bus-number 11 the system calculates the time from the very first station of the route "Europaallee" until the entry-station "Tulpenweg", which is the eighth station in this route. This is necessary because the table "REC_FRT" only stores the start-time of routes. Therefore, the app needs to calculate the arrival of the bus at the entry-station.

The exact way of computing the drive-time is described in the Javadoc C.

After the arrival of the bus at the entry-point is known, the algorithm uses the query from Listing 4 again. The previously calculated arrival-time at the entry-station is used to find routes in this step. As in this query the exact time of arrival of the bus at the entry-station is used, the perfect fitting route can be found.

It is possible that the algorithm does not find any connections between the stations. This happens when there are no direct routes between the given stations. In this case the app automatically switches to another function, which is able to find indirect routes in the database. This process is documented in Chapter 4.2.5. At this point all needed times and stations of the route are known and are ready to be presented to the user. The app will draw the result of the route on the map. Furthermore, for a better orientation the start-station and stop-station are marked with a special icon. In addition, the arrival of the bus at the entry-station, the arrival-time of the bus at the end-station, the number of the bus and the direction are displayed to the user.

In Figure 10 the time and bus details for a test request are displayed. The request for this example was: start-station = "Tulpenweg", stop-station = "Landgericht/Bahnhof", time = 17:50 and day = Sunday. In Figure 11 the respective route of the results is displayed.

If the user queries for other connections, the old routes will be deleted from the map, so users are not confused with the several different routes on the map.
4.2.4 Database indexing

The first few initial tests on several test devices showed, that user requests need too much computational time. Simple requests like in the test setting from previous Chapter 4.2.3 need over 2 minutes. This gets even worse when the system later should be able to search for indirect connections. Since this process should probably take about three to four times more time, because of a much higher complexity. A Google Nexus 5 smartphone and a Samsung T520 tablet were taken as test devices. Both devices are quite powerful and listed in the top 40 of the most powerful Android devices (state: October 2015) (Primate-Labs, 2015). Since the implementation requirements demand a fast routing algorithm, some improvements are needed.

As already mentioned in Chapter 3.2.2.4 the load of computations on the database is huge. Most of the needed time for finding connections rely on finding entries in the database. A fast and efficient way of boosting databases and the related queries, are indexes. Indexes allow to create new tables, that consists of links to other table entries, which are needed often. Instead of searching every single entry in the database, the system looks in the new index table and only needs to search for link entries (W3C, 2015). Depending on the setting, this is able to highly improve the lookup time in the database up to 40000%. Furthermore, it
is possible to create indexes for entries, that are related and mostly searched together. This feature allows complex queries with several different entries to be processed even faster. These indexes are created on the following table entries:

- In **LID_Verlauf**: *ORT_NR*
- In **LID_Verlauf**: *STR_LI_VAR*
- In **LID_Verlauf**: *STR_LI_VAR* and *ORT_NR*
- in **REC_FRT**: *FRT_START*
- in **REC_FRT**: *STR_LI_VAR*
- In **REC_FRT**: *FRT_START* and *STR_LI_VAR*
- In **SEL_FZT_FELD**: *ORT_NR* and *SEL_ZIEL*

The entries were chosen by the frequency of searches in the database and their relations. Indexes unfortunately also have two drawbacks. They require a little more space on the hard drive and it is more complex to edit entries. None of the drawbacks create a problem for this thesis, since database entries remain unmodified, but they are completely replaced by new ones every few months by the GÖVB. Moreover, the extra space needed sums up to approximated 500 kB.

New tests with the improved database show the splendid effect of indexes. A request does not need about several minutes anymore, but under 1 second, which is an improvement of roughly 1500%. As an efficient implementation for finding direct connection is created now, the next chapter will show the finding for indirect connections.

### 4.2.5 Finding indirect connections

The finding of indirect routes is quite similar to the finding of direct routes and therefore many of the existing methods can be adapted. The app will only enter this mode, when the method for finding direct routes fails. Therefore, the app already knows, that there are no direct connections available.

In a first step the system tries to find a joint station between the two stations, where two routes crosses and the user can change the bus. For finding such a joint station it is necessary to look for all routes, that have a stop at the start-station or at the stop-station. This operation drastically increases the complexity and amount of processed data, because the number of possible routes that need to be considered are at least doubled. The query for this process can be seen in the following Listing 5.

---

5
Listing 5: SQL-query inside the function `completeChangeTwo(int time, int day, int start, int stop)` from mapFragment.java

```
1 SELECT DISTINCT LID_VERLAUF.STR_LI_VAR,
2 LID_VERLAUF.ORT_NR as a
3 FROM LID_VERLAUF
4 WHERE
5 # subquery starts
6 EXISTS ( SELECT DISTINCT LID_VERLAUF.STR_LI_VAR,
7 LID_VERLAUF.ORT_NR as b
8 FROM LID_VERLAUF
9 WHERE (b = startPos OR b = (startPos + 1) OR
10 b = (startPos - 1) OR b = stopPos OR
11 b = (stopPos + 1) OR b = (stopPos - 1))
12 AND a=b ) # subquery ends
```

Again, a sub-query is needed for this query. In line 12, all doubled and irrelevant entries are filtered. After all possible routes are found, the system processes iteratively through every single route pair and checks, whether they have a joint stop or not. When such a pair of routes with a joint stop can be found, the app tries to find two simple direct routes like in Chapter 4.2.3. Firstly from the start-station to joint-station and from the joint-station to the stop-station.

For example a user sets "Tulpenweg" 2041 as start-station and "Guldenhagen" 5151 as stop-station. As the system does not find any direct connections between these two stations, it will switch into the mode for finding indirect connections. Usually, in this case "Landgericht/Bahnhof" 2241 should be chosen as joint-stop, because of faster connections due to specific day and time settings this can differ. The app now calculates routes from "Tulpenweg" to "Landgericht/Bahnhof" and from "Landgericht/Bahnhof to "Guldenhagen". Afterwards, the results are displayed in a similar way as for direct connections. Also, to the normal info-window, the waiting time is displayed and the joint station is indicated with a special marker on the map.

A test request for this setting with the following: day = Monday and time = 10:28 can be seen in Figure 12 and 13.

Note that in Figure 12 the direction for bus 71 is not displayed, because this bus has no explicit direction. Of course it is possible to zoom in and out to get a better overview of the situation, since Figure 13 is zoomed out for demonstration reasons. Fortunately, this process only requires about one second and is only possible with the help of indexes.

Unfortunately, at this point some problems with the finding of routes are possible. The first thing is, that two routes may cross at a joint-station, but there are no available connections for the requested time and date. So if the system has found a possible joint-station, but cannot
find any possible connections for the user, the app puts the joint-station inside a temporary blacklist. After this a recalculation for the user request is done, but this time without the station on the blacklist. This process iteratively adds unusable stations to the blacklist and recomputes the request until a solution was found.

Of course, this function will stop after several retries, unless the system would end in an infinite loop. The abortion of the function means that there are no possible connections for the specific request with only one joint stop. For reasons of usability the system does not support finding of routes with two or more joint-stations. The computational time for finding this grows exponentially and is not able to processed on most smartphones. In this case the user is informed over a dialog that suggests to change the station or the time and date. Furthermore, the Göttingen’s bus system is quite small and almost every desired route can be served with one joint station.
5 Conclusion and outlook

The aim of this thesis was to design and implement a mobile routing app for the public bus system in Göttingen. For the implementation several requirements had to be considered as one can see in the Figure 1. The implementation requirements are divided into three groups. In the following the requirements will be recapped and showed how they have been implemented in this thesis.

The first category "Developing of a native app" demands a mobile and native app, that allows users to find bus-lines within Göttingen without knowing concrete street- or station-names. To fulfil these requirements a native Android app was developed that shows a Google Map instance with all bus-stations inside Göttingen. Users are able to choose stations on this map and search for possible connections between those stations. Furthermore, the current geological-position and a compass are displayed for a better orientation. The usage of the geological position is an optional feature and therefore users can use the system without any GPS-connections or any internet-connection at all. Moreover, users also can search for future bus-lines, by changing time and date values freely. Overall, the app is designed quite simple, so everyone is able to use it easy and convenient.

The second category "Routing algorithms" requires a fast and efficient routing algorithm, that is able to find the fastest available connections. This was solved in several different steps, which are adapted to the complexity of the user requests. Therefore, the load of the system and the computational time is hold low in many settings. The system is able to find direct connections between stations and connections with one extra joint-station for changing the bus. Unfortunately, the system is not able to find connections with two ore more joint-station due to the limited computational power on smartphones. The implemented algorithms work fast and also find the very next available connections for given requests.

The last category ,"Data", demands the retrieval of the required data for this project and a proper data-management. The data for this project was directly received from the GÖVB, which includes the positions of all bus-stations and all routing relevant data in Göttingen. Irrelevant data was filtered and dropped. The rest of the data was converted into a database. Later, indexes were added to the database to improve the lookup times inside the database. An interesting setting for future modifications would be a live communication between the GÖVB and the app, to retrieve live-information about delays and failures of buses. Sadly, the GÖVB initially wants to adept a compatible live system earliest in 2016.
References


   URL http://www.goettinger-tageblatt.de/Nachrichten/Goettingen/Uebersicht/100-000-Schwarzfahrer-pro-Jahr-in-Goettingen

   URL http://developer.android.com/guide/components/activities.html


   URL https://developers.google.com/maps/documentation/android-api/signup

   URL https://source.android.com/

   URL http://www.idc.com/prodserv/smartphone-os-market-share.jsp
   URL http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html

   URL https://www.java.com/en/about/

   URL http://www.oracle.com/technetwork/java/compile-136656.html

   URL https://browser.primatelabs.com/android-benchmarks


   URL http://sqlite.org/features.html


   URL http://www.w3.org/TR/REC-xml/

   URL http://fahrplaner.vsninfo.de/hafas/query.exe/dn?externalCall=yes

   URL http://www.w3schools.com/sql/sql_create_index.asp

The complete appendix is stored on an additional CD, which will be handed over together with this thesis. The three main directories are described in the following sections. All of the three directories are placed on the root directory of the CD. Furthermore, a PDF-file of this thesis is stored in the root directory of the CD.

A Javadoc

In this directory are all files stored, that have been created with Javadoc. The entry-point for starting the Javadoc is the file `index.html`.

Most relevant directories for Javadoc:

```
- javadoc
  - index.html
  - overview-tree.html
  - overview-summary.html
  ...
  - com
    - example
      - dima
        - maps_test
          - Manifest.html
          - MapsActivity.html
          - mapFragment.html
          ...
  ...
  - index-files
    - index-1.html
    - index-2.html
    ...
  ...
  - resources
    - ibackground.gif
    - tab.gif
    ...
  ...
  - info
    - slidemenu
      - adapter
        - NavDrawerAdapter.html
        ...
      - model
        - NavDrawerAdapter.html
        ...
```
B  Gö Bus source-code

In this directory are all source-code files for the created app.
This are most important directories for app:
Created databases: /Gö Bus/app/main/assets/
Important source-code files: /Gö Bus/app/main/java/com/example/dima/maps_test/
The apk of the app is stored directly inside this folder: /app-debug.apk
C  Data from the GÖVB

In this directory all files from the GÖVB are stored.

Dataset
  ├── LID_VERLAUF.X10
  │    └── REC_FRT.X10
  │        └── REC_SEL.X10
  │            └── REC_SEL_ZP.X10
  │                └── ...

...
Certification

I hereby certify that this thesis entitled “Development of a mobile routing application for local transportation in Göttingen” represents my own original work, except where acknowledged by general and specific references.

____________________________________  ______________________________________
Date                                          Signature