UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

Development of a Chatbot for Tourist Recommendations Technical Documentation

Presentado por Jasmin Wellnitz
en Universidad de Burgos — July 1, 2017
Tutor: Bruno Baruque Zanón

Contents

List of Figures

List of Tables

Appendix A

Software Project Plan

A.1 Introduction

In following annex, the organizational aspects of the chatbot development will be examined. More precisely, the software development process is described as well as the tools that were used to manage the process, followed by a detailed examination of the course of the project. The second part of the annex examines the project's viability, including the calculation of involved costs and profit possibilities.

A.2 Project Management

Scrum

The project's management is inspired by *Scrum* [1], an agile software development framework which facilitates managing tasks in teams. The Scrum framework is based on the experience that most projects are too complex to be planned completely in the early stages and therefore provides an agile and iterative alternative by structuring the project in smaller iterations.

However, the applied process during this project is only loosely based on the Scrum framework since there are several concepts of the original framework that were not applied. Scrum defines roles which describes the different members of a group (*Scrum Master*, *Project Owner* and *Development Team*) as well as several artefacts organizing the team's interaction. Due to the nature of the project not having a development team in the proper meaning of the word, but only a single person realizing the bachelor thesis, only some artefacts were applied. The most important applied concept is the *Sprint*, a two-week time slot which is used to structure the project. At the beginning of each sprint, a *Sprint Planning* is realized between the author and the coordi-

nator of project. The project's coordinator can be seen as a Product Owner in the Scrum framework, prioritizing tasks and guiding the project's direction. In the Sprint Planning, it is decided which tasks are going to be worked at during the two-week sprint. At the end of each sprint, a *Sprint Review* takes place where the development team, meaning the student, presents its results to the Product Owner.

Managing the project in GitHub

In order to facilitate the project's management, the online project hosting tool *GitHub* [2] with *ZenHub* as an extension was used. ZenHub provides several useful collaboration features such as a board to visualize tasks as well as an overview of the remaining workload and velocity [3]. The tasks are defined in GitHub's *Issues* where tasks can be named, described and estimated. Story Points are used to estimate the workload of each issue. In this project, one story point is seen as the equivalent of 2 - 3 hours of work. The sprints are planned and defined using *Milestones* to which the different issues are assigned.

A.3 Time Planning

The Kick-Off Meeting took place at 5th December 2016, where the elemental concepts of the project were discussed. Due to the fact that this project was held in the course of the semester parallel to usual classes and exam periods, there were some time breaks between the two week sprints in which the development on the project paused. The project consists of 11 iterations, the last one ending a day before the submission date of the thesis, the 2th July, 2017.

The following section gives an overview of the iterations during this project, finally using Burn-Down-Charts and velocity tracking to visualize the project's progress.

Iteration 1 (30/12/2016-17/01/2017): Planning and Research

The first iteration mainly centered on setting up the project's infrastructure, including the establishment of the repository structure and the virtual machine setup. First research steps were taken by getting familiar with the used geo-information system.

Iteration 2 (17/01/2017 – 07/02/2017): Further Research

In this iteration, the knowledge of the used geo-information system was deepened by investigating how to extract essential tourist information from the

database. On the other hand, a chatbot library was chosen after comparing different possible candidates. Besides, the project's documentation will be extended by describing theoretical concepts, used tools and the project's objectives.

Iteration 3 (07/02/2017 – 21/02/2017): Recommender System and Design

In this iteration, recommender system libraries were examined to find out which one was the most suitable for this project. Afterwards, the system's basic architecture was designed. On top of that, the project's documentation was extended.

Iteration 4 (21/02/2017 – 07/03/2017): Mockups

First mockups were implemented in which the design of certain components of the application were tested. At first, the interaction of the application's chatbot layer with its environment was set up, using the Telegram Bot API and the API.AI HTTP API.

The recommender component was examined in more depth, meaning that a recommender library was finally chosen and then used to mock up a POI recommender mechanism.

Iteration 5 (10/03/2017 – 24/03/2017): Recommender System Mockup

This iteration centered on the design and first implementation of the application's recommender system. The library Apache Mahout is used to implement a content-based filtering recommendation mechanism. The iteration includes the design of user and item profiles as well as the investigation of how to retrieve the data from OpenStreetMap into the recommender system.

Iteration 6 (03/04/2017 – 17/04/2017): Collaborative Filtering & Requirements

In this iteration, the collaborative filtering recommender was designed. Additionally, the project's requirements were examined as well as the user-chatbot interaction.

Iteration 7(18/04/2017 – 02/05/2017) Conversational Interface Design

In this iteration, the chatbot's conversational interface was designed by modeling a conversation flow graph. The resulting knowledge was explained in the

Theoretical Concepts chapter of the documentation. On the other hand, user ratings were generated for the collaborative filtering algorithm.

Iteration 8 (04/05/2017 -18/05/2017) Conversation Flow Implementation

The previously specified conversation flow was modeled in API.AI and the application's chatbot layer was implemented to handle the conversation. On the other hand, the recommender component was finished and documented.

Iteration 9 (18/05/2017 – 01/06/2017) Conversation Flow Refinement and Testing

The conversation flow was refined to improve the conversation between user and chatbot. The rating mechanism was implemented. Some performance tests for the recommendation mechanism were implemented and the software was evaluated using metrical code analysis tools.

Iteration 10 (03/06/2017 – 17/06/2017) Latex Setup

This iteration dealt with setting up the latex file for the project's documentation. Previously prepared drafts were modified and completed. Several chapters and parts of the annex were introduced into the latex document and the bibliography was formalized.

Iteration 11 (18/06/2017 – 02/07/2017) Documentation and Final Conversation Flow Adjustments

This iteration concentrated on completing the project's documentation, especially outlining the project's relevant aspects and come to final conclusions. The application's implementation was only adjusted in refining the conversation flow and small refactorings in order to improve the code quality.

Burn-Down Chart and Velocity Tracking

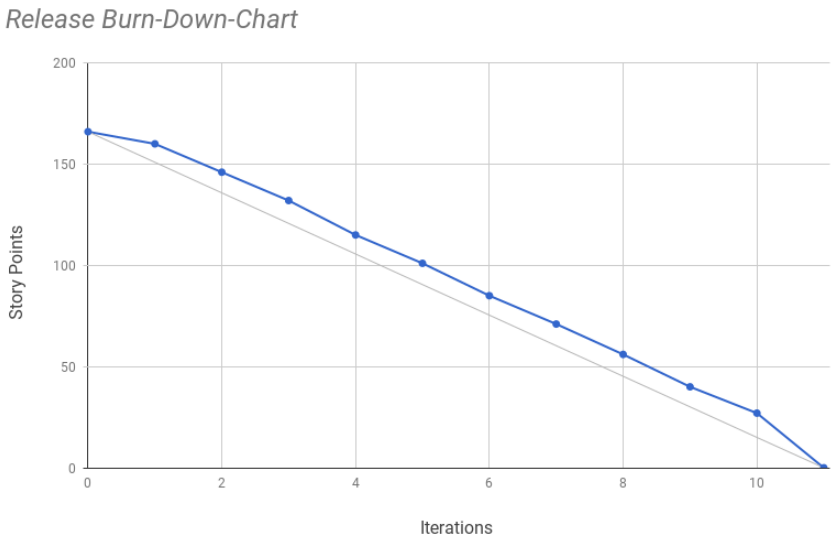

Figure A.1: Release Burn-Down-Chart

The Burn-Down-Chart shows the progression of story points in the development process. The ideal progression is represented in the grey straight line. As we can see, the actual story point progression does not differ largely from the shown ideal line, as the work load was divided equally to the different iterations.

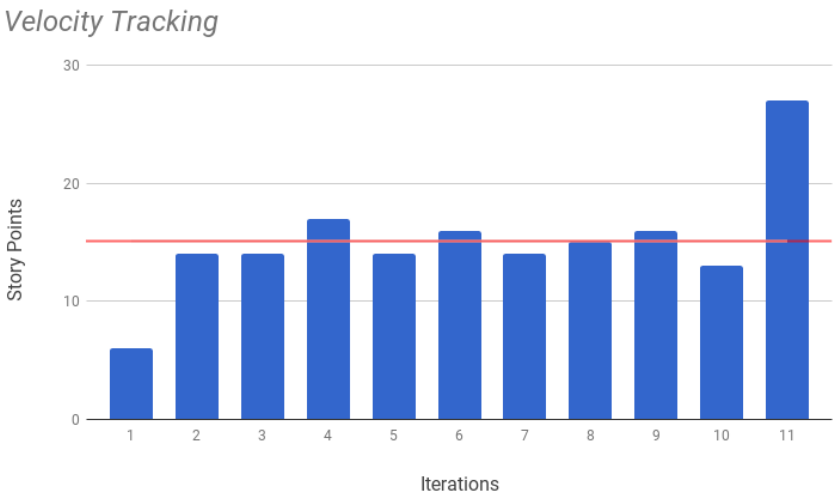

Figure A.2: Velocity Tracking

The workload divided to the different iterations is shown in detail in the velocity tracking diagram. The average work load per iteration is measured with 15 story points, which can be seen in the red trend line. Most iterations do not differ largely from this average measure, except for the first and the last iteration. This observation can be explained as in the first iteration, research was done and a first overview of all the different aspects of the project was made. After this iteration, the necessary tasks to be done became clearer and a first task overview was created.

In contrast, the last iteration is outstandingly bigger than the previous ones. This can be explained by the fact that the student had terminated all of the exams and semester's course work at that time and could completely focus on the preparation of the project.

A.4 Viability Study

In this section of the documentation, a viability study is made. It is examined whether the project is economically feasible, comparing the involved costs and potential sources of income. Additionally, the legal viability is checked to ensure that a realization of the application can be done without legal risks.

Economic Viability

Personnel Costs

The application's development was realized by the author of the presented bachelor thesis. As the author is a student of computer sciences without completed university degree, the hourly wage is estimated at 14 €/h. In order to calculate the total development costs, the time invested in the project is examined. As the realization of the bachelor thesis is evaluated with 12 ECTS points and a workload of 30 hours per ECTS point is assumed, a total amount of 360 hours were dedicated to the project's development. This also coincides with the results of the velocity tracking, where a total number 166 story points was estimated to finish the project, one story point being roughly equivalent to 2-3 hours of working.

All in all, this leads to the following cost calculation:

$360h * 14\frac{€}{h} = 5040€$

As the developer is a German citizen, German tax law is applied in the following scenario: Assuming that the produced application is developed in a freelance project on behalf of a customer, the developer only charges the hourly fee. There are no additional social security or tax costs for the customer as freelancers are liable to pay income taxes themselves. However, the charged fees are below the tax allowance for unmarried persons in 2017, which is 8820

€. Therefore no income taxes have to be paid, assuming that the student does not have any other freelance income in the respective year.

Software Costs

In the following, the costs of the services used in the project are examined.

Software	Costs	Comments
Ubuntu 16.04.2 LTS	0,00 €	-
OpenStreetMap	0,00 €	-
API.AI	0,00 €	According to the API.AI terms of use, "services include basic services ("Basic Services") provided free of charge and enhanced services ("Enhanced Services"), which, if available, must be purchased." [4] However, at the time of development, no fees or potential upgrades were evident.
Foursquare	0,00 €	According to Foursquare's developer documentation [5], "the Foursquare API has a default limit of 1000 free requests per 24 hour period" and "500 requests/hour", "whichever occurs first". An enterprise option must be booked if more requests are needed.
Telegram	0,00 €	-
Apache Mahout	0,00 €	-
Apache Maven	0,00 €	-
Java Spark	0,00 €	-
Heroku	0,00 €	A "Free Plan" was used to host the application in Heroku. There are several pricing options, adjusted to the developer's needs. The free plan comes with certain inconveniences, such as "sleep after 30 minutes of inactivity". [6]

Table A.1: Software Costs

As we can see, no software costs were charged in the way the third-party

products are used in this project. However, there could be additional costs with an increasing number of users accessing the chatbot, as some services that are used apply a pricing model that is based on the number of user requests.

Hardware Costs

Two different hardware devices were used during the project duration. For the software's development, a laptop was used, in particular an Asus Zenbook, bought in 2015 for 899 €. In order to use the application and check the functionality in the productive environment, a mobile device is used, more precisely the Motorola Moto G4 Plus, bought in 2017 for 199 €. Due to the fact that the mentioned devices were not exclusively purchased for this project, their costs can be amortized as seen in the following calculation. A total amortization period of 3 years for the hardware is assumed:

Total Hardware Cost $= 899€ + 199€ = 1098€$

Amortization period in hours $= 3years * 8760h/year = 26280h$

Development period in hours $= 360h$

Hardware Cost per hour $= 1098€/26280h = 0,04€/h$

Hardware Costs during development $= 0,04€/h * 360h = 14,40€$

So all in all, total hardware costs of 14,40 € can be assumed.

Total Costs

The previously calculated partial costs result are summarized in the following table:

Component	Costs
Personnel Costs	5040,00 €
Software Costs	0,00 €
Hardware Costs	14,40 €
Total Costs	**5054,40 €**

Table A.2: Total Costs

It must be noted that occurring office costs were not included in the calculation as it is assumed that the developer has worked in a home office which can be set off against tax liability.

Income Analysis

The following section provides an understanding of possible ways to make profit from the developed application.

Due to the fact that the chatbot is publicly accessible in an instant-messenger, it is not very common to profit from the user directly, as there are no easy-to-use payment mechanisms integrated in the messenger.

A more interesting concept is to win tourist agencies, cultural institutions or restaurant owners as investors. The chatbot can be seen as an advertisement platform for those who want to promote their offers to tourists which in fact are a very promising target group as people tend to spend a lot of money on vacation.

Essentially, the idea is to present points of interest that belong to a customer in a more prominent, attractive way to the potential client. This could happen, for example, when a user asks for a recommendation and is situated in close proximity to a promoted point of interest. The presentation of the promoted point of interest could differ from the usual, non-promoted locations, in a way that more precise descriptions or more photos of the location are provided than usual. Additionally, the points of interests could be highlighted visually, for example by applying different font sizes or colors for the messages.

In return for this service, a monthly payment of the local investors would be conceivable. This way of promoting points of interests is quite interesting for the local tourist industry as the chatbot collects user interests for the recommendation, therefore, the marketers are provided with a direct way to target exactly the user group they are interested in.

If we charged a customer a fee of 10€/month to be presented in the chatbot, then we would need 42 customers to achieve the return of investment of 5040 €. Considering the large number of restaurant proprietors and tourist agencies in a city of touristic relevance, this could be seen as a realistic aim.

However, it must be noted that the commercial use of the application leads to a possible upgrade of pricing plans, for instance using the Foursquare API.

Legal Viability

The following legal viability concentrates on the third-party software licenses used in the project and in turn, an analysis of which license can be applied for the developed application. The table A.3 contains an overview of the used services and their licenses.

As some of the used third-party software is nonsublicenseable, for instance API.AI, it is not possible to apply a permissive free software license such as

Apache License 2.0 or GNU General Public License v2.0 to the application. However, if the designed application aims to make profit, publishing the application under a free software license is not an option anyway. Therefore suitable terms of service need to be designed in order to publish the application.

Software	License	Comments
OpenStreetMap	Open Database License (ODbL) 1.0	According to the ODbL license, the rights to use the OpenStreetMap data "explicitly include explicitly include commercial use, and do not exclude any field of endeavour." [7]
API.AI	API.AI End User License Agreement	According to API.AI's license agreement, the "use of the Services within the commercial enterprise for internal purposes is expressly allowed." [4]
Foursquare	Foursquare Labs, INC. API and Data License Agrement	For commercial use, an enterprise license is needed. The fees that are charged are not publicly available and are communicated directly by contacting Foursquare.
Telegram	GNU General Public License v2.0	According to Telegram, the commercial use of the Telegram API is permitted for anyone except for "large corporations, publicly traded companies and other businesses that exist to maximize shareholder value or sell stock." [8]
Heroku	Heroku License Agreement	-
Apache Mahout	Apache License, Version 2.0.	Permissive free software license that allows, among other things, sublicensing, commercial use, modifying and distributing of the software.
Apache Maven	Apache License, Version 2.0.	See above
Java Spark	Apache License, Version 2.0.	See above

Table A.3: Software Licenses

Appendix B

Software Requirement Specification

B.1 Introduction

The following chapter provides a specification of the software's overall requirements. Showing its functions, limitations and user interaction, the catalogue can be considered as a contract between clients and developers. Aside from giving textual descriptions, use case diagrams help to clarify the requirements and interactions in more detail.

B.2 General Objectives

The following steps and objectives are related to the software's requirements:

- Design of a conversational interface: the conversation flow between user and chatbot is mapped to the natural language processing platform api.ai which then parses the user input into formalized data. The parsed input is interpreted by the chatbot and triggers the desired behaviour, such as recommendation or storage of important user information.
- A recommender system must be implemented to provide personalized tourist recommendations. The recommender is based on data the user has shared with the chatbot and additional data of similar users. To overcome the problem of initially sparse user data, different recommendation methods are combined as well as retrieving existing user data from other sources and/or generating data.

- Extracting tourist data from a spatial information database based on OpenStreetMap: the data is filtered so that only data of touristic importance is evaluated by the recommender and presented to the user.

B.3 Software Requirement Catalogue

This section outlines the software requirements. At first, the participating actors are introduced. Then, the requirements are examined, dividing them into functional and non-functional requirements.

Participating actors

This software has only one main actor, the user. Using the Telegram messenger, users interact with the chatbot. There are no different user roles, so each user has the same range of features to use.

Functional Requirements

This section describes the software's full range of features:

- Using the Telegram interface, messages to the chatbot can be introduced which are answered accordingly.
- Using the Telegram interface, a location can be introduced which is used as the basis for the chatbot recommendations.
- The proximity of the recommendations can be specified by the user. If no radius is entered, the default value of 1 km is used as a distance between the user location and examined point of interests.
- The chatbot provides a recommended point of interest within the chosen proximity. The recommendation result contains the name, location, point of interest category as well as an additional picture retrieved from Foursquare (if available).
- Recommended points of interests are rated by the users. The rating is saved and used as a basis for future recommendations.
- The user can access the information the chatbot has collected about him, such as already recommended points of interests or saved interests.
- To refine recommendations, user chat messages are evaluated using natural language processing and saved in the user profile.

Non-functional Requirements

This section describes the so-called non-functional requirements which contain technical as well as operational requirements.

- To use the chatbot, the Telegram messenger app has to be installed on a smart device (smartphones or tablets). Although Telegram is also available as a desktop application, these versions do not support sending locations and are therefore not suitable.
- The device must be connected to the internet to use the chatbot.
- The device must be capable of receiving GPS information to calculate its current location.
- The conversational interface should be intuitive, so the user is able to communicate with the chatbot without previously reading an exhaustive tutorial. To facilitate user decisions, mutually exclusive keyboard buttons are used.
- The chatbot should be able to handle user requests adequately. Questions and demands concerning travelling should be understood and answered satisfyingly. Other requests are rejected politely.

Limitations

There are multiple limitations present due to the fact that the software can be still considered as a prototype. In a future enhancement, most of these limitations should be remedied.

- The used OpenStreetMap data was downloaded once and then used offline. To keep the data up-to-date, an automatic update mechanism should be set up.
- Due to performance reasons and sparse user rating data, the prototype only gives recommendations for the city of *Barcelona*, Spain. In a future enhancement, a bigger OSM region should be covered.

B.4 Requirement Specification

Use Case Diagrams

General Use Case Diagram

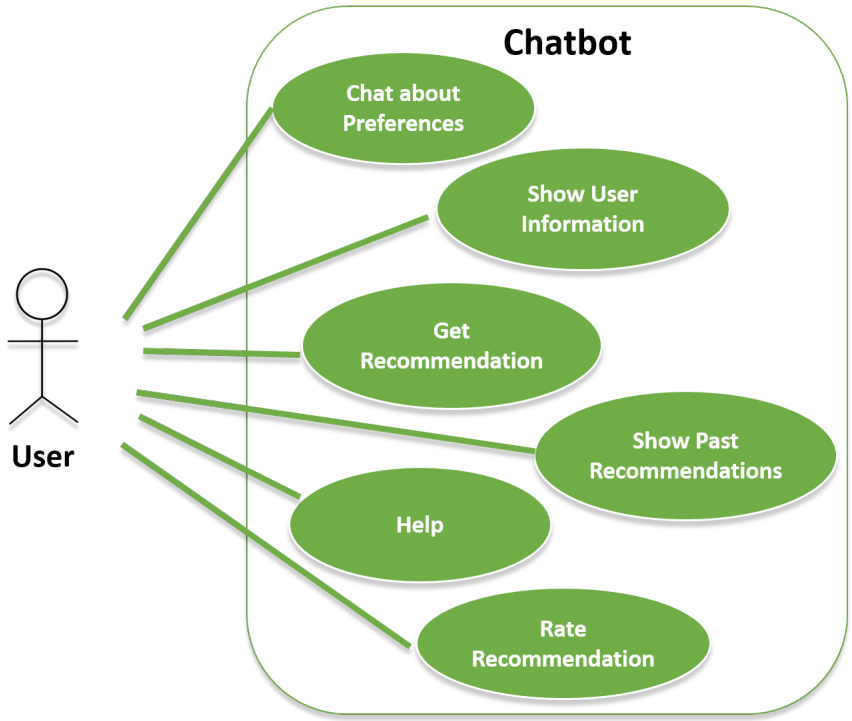

Figure B.1: General Use Cases

The shown use case diagram contains the six main interactions the user performs using the chatbot. The use case "Help" is used to show the user the main features of the chatbot. In "Chat about Preferences" the chatbot collects information about the user by chatting with him. This data is used to complete the user profile for recommendations. On the other hand, "Rate Recommendation" shows how the user rates previously recommended points of interests.

The use cases "Show Past Recommendations" and "Show User Information" both aim to provide the user an understanding of the data the chatbot has already collected of the user.

The use case "Get Recommendation" represents the interactions between user and chatbot that lead to the delivery of user adapted recommendations. Because of its complexity, this use case is shown in more detail in the following section.

Use Case Diagram - "Get Recommendation"

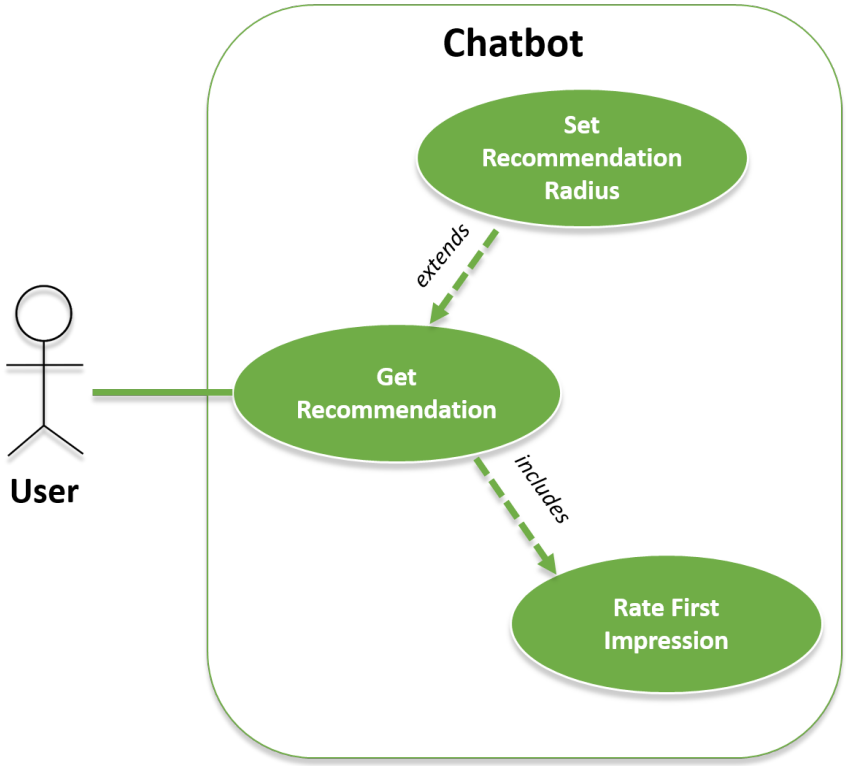

Figure B.2: Use Case - "Get Recommendation"

This use case shows the involved components in providing the user with recommendations. In order to get recommendations, the user is able to specify a recommendation radius to set the maximal distance between himself and the point of interest. This step is optional. After showing the user the recommended point of interest, he is asked to give a first impression of the point of interest in order to refine future recommendations.

Use Case Templates

The previously defined use cases are explained in the following templates, showing the circumstances in which a use case is handled. Additionally to the given description, the flow of events will be shown in detail in the Design Specification, illustrated by sequence diagrams.

Use Case ID	**UC-01**	
Use Case Name	Chat About Preferences	
Description	The user chats with the chatbot about his preferences.	
Trigger	User connects to the chatbot for the first time or shows proactively the intention to chat about his preferences.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	Chatbot: "So, tell me, what are you interested in when you visit a new place?"
	2	User answers and interest is filtered
	3	The interest is saved and the user is told to repeat the process whenever he likes.
Alternate Flow	Step	Action
	3a	User response is not understood, so the user is asked to rephrase his answer.
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently. The user profile is updated with user interests.	
Exceptions	Step	Action
	2	User answers "No", so the use case is aborted.
Frequency of Use	Medium	
Importance	High	

Table B.1: UC-01 - Chat About Preferences

Use Case ID	**UC-02**	
Use Case Name	Get Recommendation	
Description	The user asks for a recommended point of interest. Based on the user information, the recommender returns points of interests close to the user.	
Trigger	The user asks for a recommendation.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	Chatbot asks for the user's current location.
	2	User enters his location
	3	Chatbot presents user a recommended point of interest.
	4	Use Case → Rate First Impression (see B.5)
	5	Chatbot: "Do you want to see another recommendation?"
	6	User enters No.
Alternate Flow	Step	Action
	1a	User starts recommendation process by entering its current location which is followed by step 3.
	6a	User enters Yes, so steps 3-5 are repeated.
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently. Recommended points of interests the user liked are saved and marked as unrated.	
Exceptions	Step	Action
	3	Chatbot does not find any (more) points of interests for the user and cancels recommendation process.
Frequency of Use	High	
Importance	High	

Table B.2: UC-02 - Get Recommendation

Use Case ID	**UC-03**	
Use Case Name	Rate Recommendation	
Description	The user rates a points of interest that was previously recommended to him.	
Trigger	The user greets the chatbot or wants to see past recommendations and has unrated recommendations.	
Precondition	The user was given a recommendation before that he has not rated yet.	
Flow of Events	Step	Action
	1	Chatbot asks how the user liked the first unrated recommended point of interest.
	2	User chooses from mutually exclusive rating buttons (e.g. 1 stars to 5 stars rating)
	3	Chatbot: *"Thanks for the rating!"*
Alternate Flow	Step	Action
	-	-
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently. The maximal radius for that user is saved and used for the next recommendations.	
Exceptions	Step	Action
	2	The user states he does not want to rate the point of interest, so the use case is aborted.
Frequency of Use	Medium	
Importance	High	

Table B.3: UC-03 - Rate Recommendation

Use Case ID	**UC-04**	
Use Case Name	Specify Recommendation Radius	
Description	The user sets the radius in which he wants the recommended points of interests to be in.	
Trigger	The user asks to set the recommendation distance.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	Chatbot asks the user about his preferred maximal recommendation radius.
	2	User answers with a positive numeric value.
	3	Chatbot repeats: *"Fine, I set the maximal radius to (repeat value)"*.
Alternate Flow	Step	Action
	-	-
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently. The rating is saved in the ratings file and the corresponding recommendation is marked as rated.	
Exceptions	Step	Action
	1	User doesn't answer with a positive numeric value, so the chatbot asks again or aborts the use case.
Frequency of Use	Low	
Importance	Medium	

Table B.4: UC-04 - Specify Recommendation Radius

Use Case ID	**UC-05**	
Use Case Name	Rate First Impression	
Description	When the user is given a recommended point of interest, he is immediately asked of his first impression to refine future recommendations.	
Trigger	-	
Precondition	The user has just received a recommended point of interest (see B.2)	
Flow of Events	Step	Action
	1	Chatbot asks the user about his first impression of the recommended point of interest.
	3	User chooses from mutually exclusive buttons (e.g. *"Sounds good!"* or *"Don't like it"*).
Alternate Flow	Step	Action
	-	-
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently. The rating is saved in the ratings file and the corresponding recommendation is marked as rated.	
Exceptions	Step	Action
	1	User does not answer accordingly, so the point of interest is discarded and no rating is saved.
Frequency of Use	High	
Importance	Medium	

Table B.5: UC-05 - Rate First Impression

Use Case ID	**UC-06**	
Use Case Name	Show Past Recommendations	
Description	The chatbot provides the user with information about the points of interests that were recommended to him previously.	
Trigger	The user asks to see his previous recommendations.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	The chatbot lists the past recommendations the user was interested in.
Alternate Flow	Step	Action
	2	When there are unrated items left, the user is asked to rate a previously recommended point of interest (see B.3)
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently.	
Exceptions	Step	Action
	1	There are no past recommendations so far, so the user is told to ask for a recommendation first in order to use this feature.
Frequency of Use	Medium	
Importance	Medium	

Table B.6: UC-06 - Show Past Recommendations

Use Case ID	**UC-07**	
Use Case Name	Help	
Description	The user asks for help and gets an overview of the chatbot features.	
Trigger	The user asks for help.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	The chatbot gives an overview of the chatbot features (rating, chatting, recommendations).
Alternate Flow	Step	Action
	-	-
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently.	
Exceptions	Step	Action
	-	-
Frequency of Use	Low	
Importance	Low	

Table B.7: UC-07 - Help

Use Case ID	**UC-08**	
Use Case Name	Show User Information	
Description	The user is provided with the personal information the chatbot has collected so far.	
Trigger	The user asks to see his personal information.	
Precondition	The chatbot is in a state in which the user is allowed to type messages independently, meaning that the user does not conduct another predefined conversation.	
Flow of Events	Step	Action
	1	The chatbot shows the user his stored interests and the current specified recommendation radius.
Alternate Flow	Step	Action
	-	-
Postcondition	The chatbot is in a state in which the user is allowed to type messages independently.	
Exceptions	Step	Action
	-	-
Frequency of Use	Low	
Importance	Low	

Table B.8: UC-07 - Show User Information

Appendix C

Design Specification

C.1 Introduction

This annex serves as a detailed description of the application's implementation, structure and way of functioning. Besides providing structural information such as data model, system topography and architecture, some peculiarities of the application are examined that took significant effort in design, for example the implementation of the recommender system and the conversation flow design. Additionally, the chatbot's flow of events is shown by examining the procedural steps of some main use cases.

C.2 Data Model

Package Structure

The application's source code is divided into six packages, which also reflects the application's main component structure. Figure C.1 shows the packages and relationships between them.

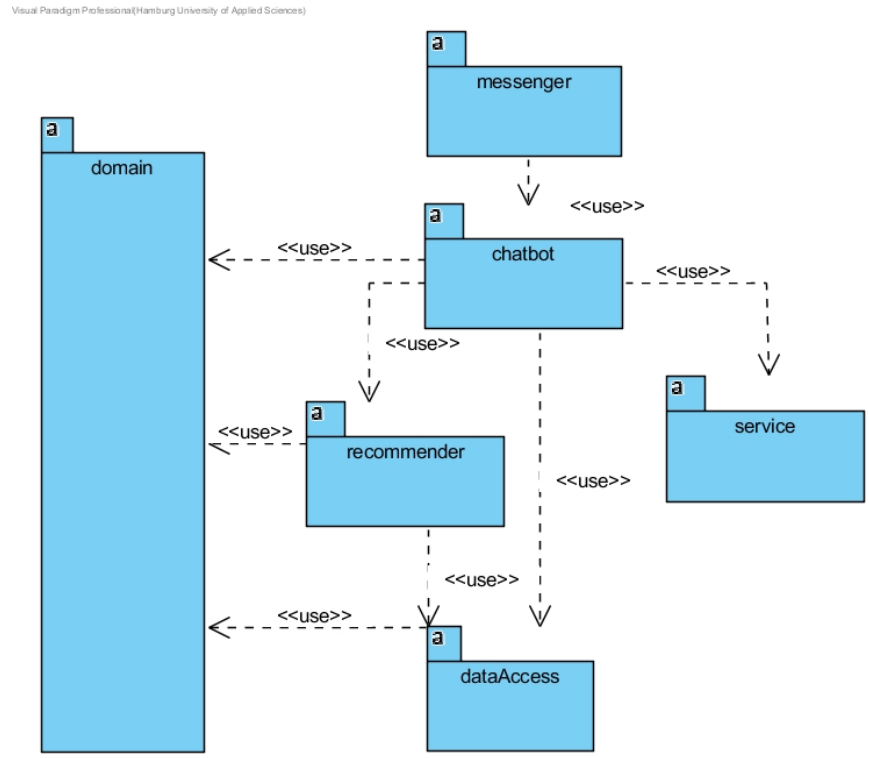

Figure C.1: Package Structure

A description of the packages containing the application's classes is provided in the following sections. For reasons of clarity and comprehensibility, the classes are not shown in a single diagram but are split into several class diagrams based on their respective package.

Messenger

The messenger is the application's entry point. The web service is set up in *Main* and all of the system's components are initialized using dependency injection. The *TelegramBotHandler* encapsulates the connection to Telegram, receives messages from the messenger and forwards it to the *TouristChatbot*. It sends messages back to the messenger and triggers the modification of the messenger's appearance if needed. To do so, a Telegram Bot API [9] is used.

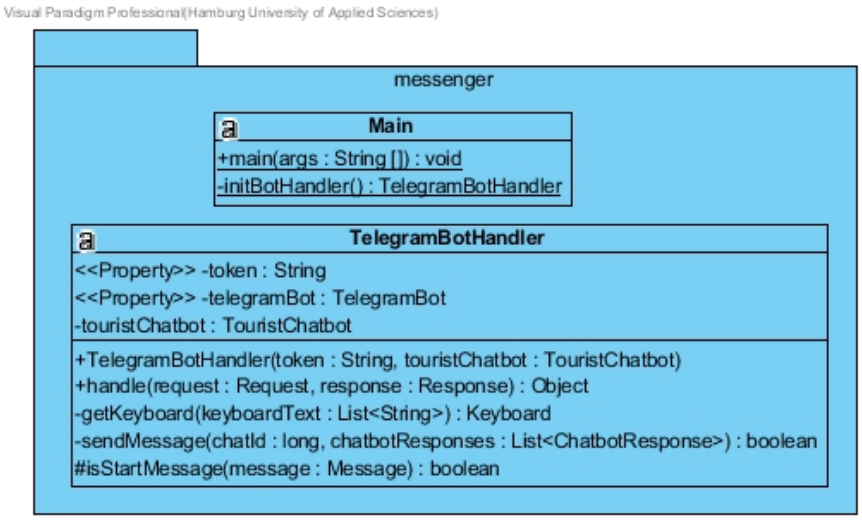

Figure C.2: Messenger Package

Chatbot

The *TouristChatbot* controls the chatbot's conversation flow. It forwards messages to the natural language platform and triggers recommendations. The class *ChatbotResponse* is a container specyfing the information that is forwarded back to the messenger. The enumeration *Action* is used to distinguish between the different actions to be taken in the *TouristChatbot*.

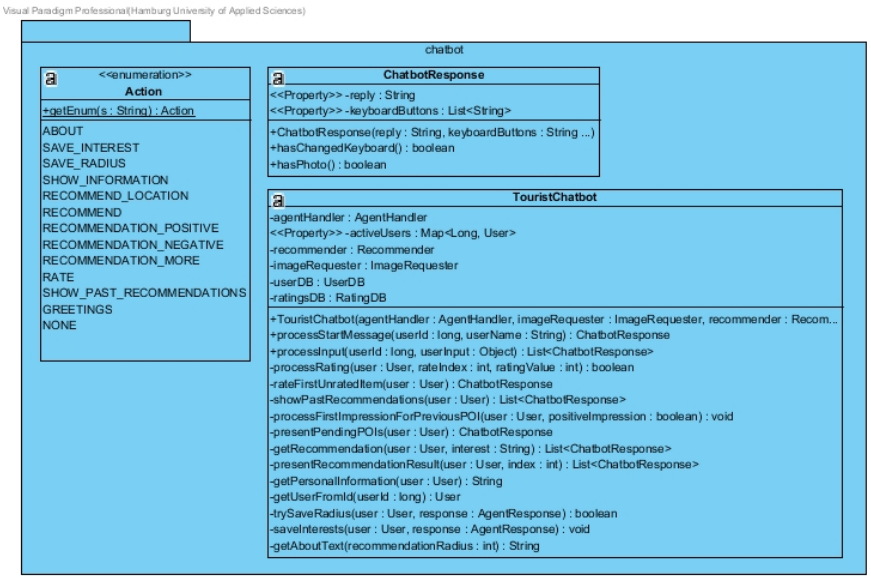

Figure C.3: Chatbot Package

Services

The service package controls how the APIs are accessed in the project. The package contains a subpackage, the agent package, which contains all logic related to the natural language platform API.AI, including the access of the API and processing of API.AI's output. Additionally, the Foursquare API is accessed in the class *ImageRequester*. As the way to access HTTP APIs is the same for both services, an inheritance is implemented. The super class *ServiceRequester* uses the library OKHTTP to encapsulate the GET requests. The subclasses *ImageRequester* and *AgentHandler* define the specific way of accessing the APIs.

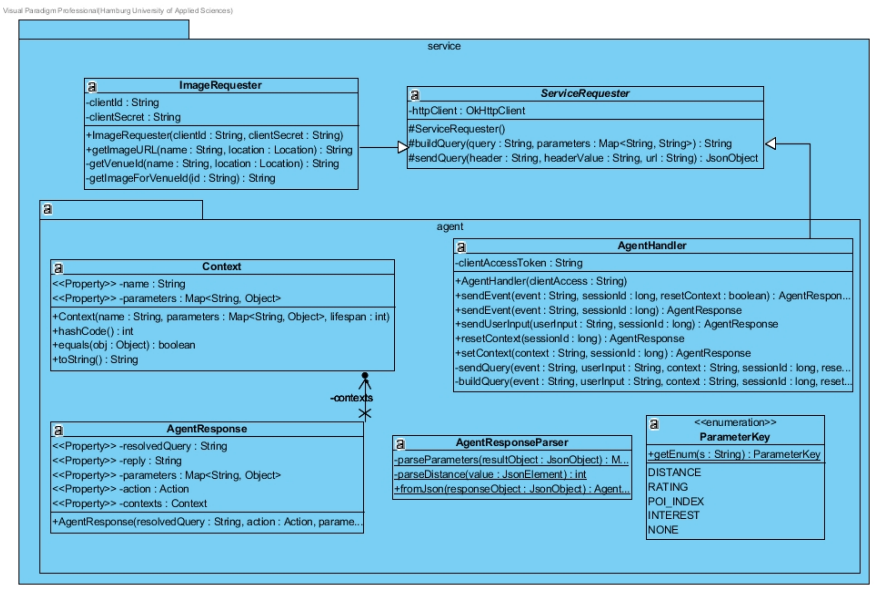

Figure C.4: Service Package

Recommender

The recommender package is one of the most central components of the application. The recommendation process itself takes places in the class Recommender which uses the machine learning library *Apache Mahout* to compute recommendations. The classes *ProfileSimilarity* and *POIDataModel* override Mahout functionalities to adapt the recommendation process to the project's needs. The *POIProfile* contains the tourist categories that show the user interests or POI characteristics. The recommendation mechanism is explained in more depth in the section C.4.

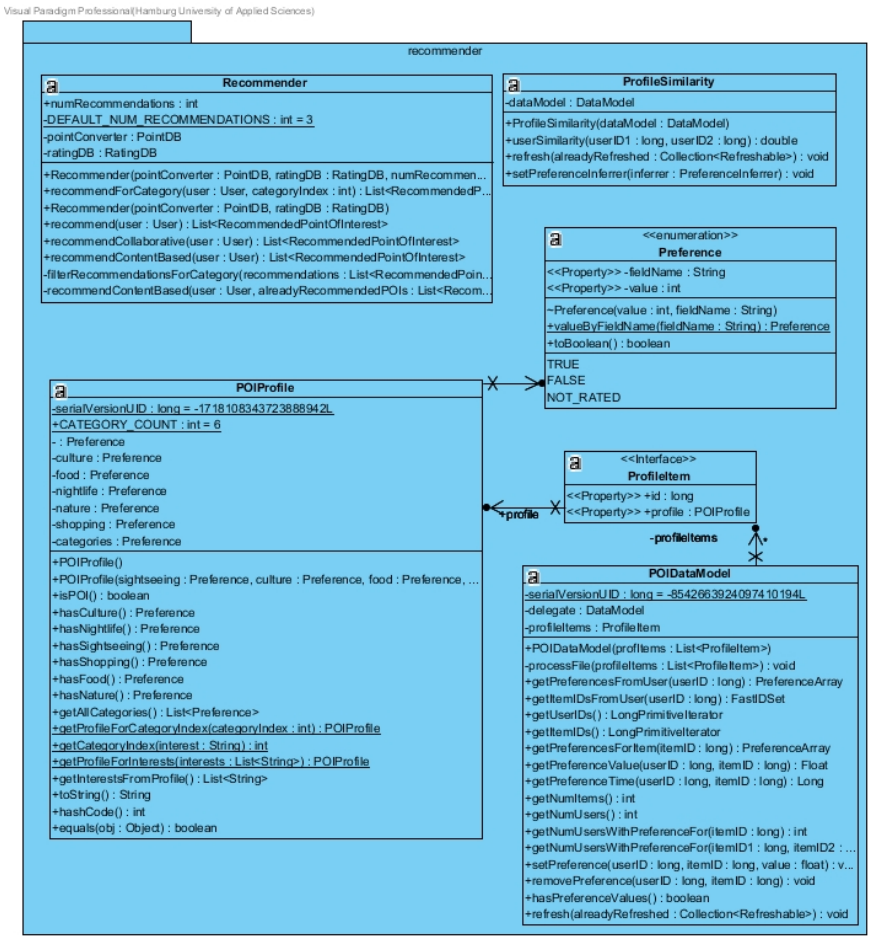

Figure C.5: Recommender Package

Data Access Package

The data access package connects the application to the PostgreSQL database. It consists of the classes *UserDB*, *RatingDB* and *PointDB* which encapsulate the access to the three respective database tables. Due to the fact that their principal way of working is similar, an inheritance was implemented, handling the database connection and methods to execute queries in the *DatabaseManager*. The classes *UserDB*, *RatingDB* and *PointDB* are subclasses of the *DatabaseManager*.

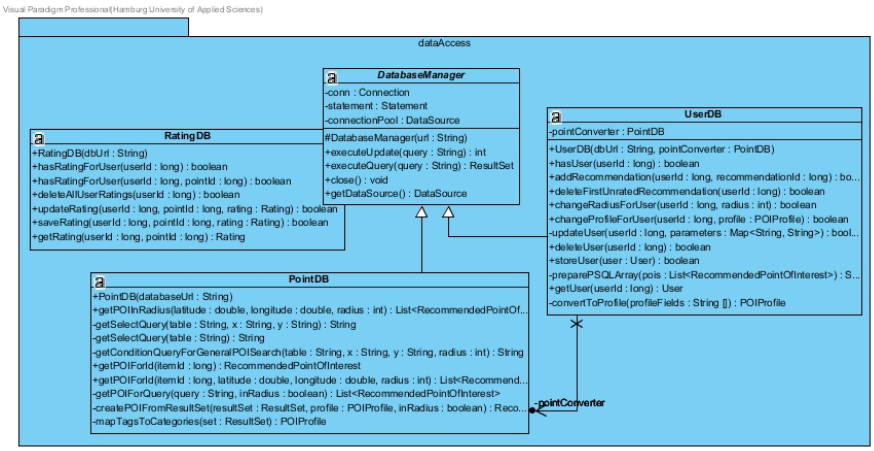

Figure C.6: Data Access Package

Domain

The domain package contains the business objects of the project, meaning natural objects that have a representation in the real world. They do not possess any complex logic or provide any services, their single function is to contain data. The domain objects are used system-wide and are shared between the components of the data model.

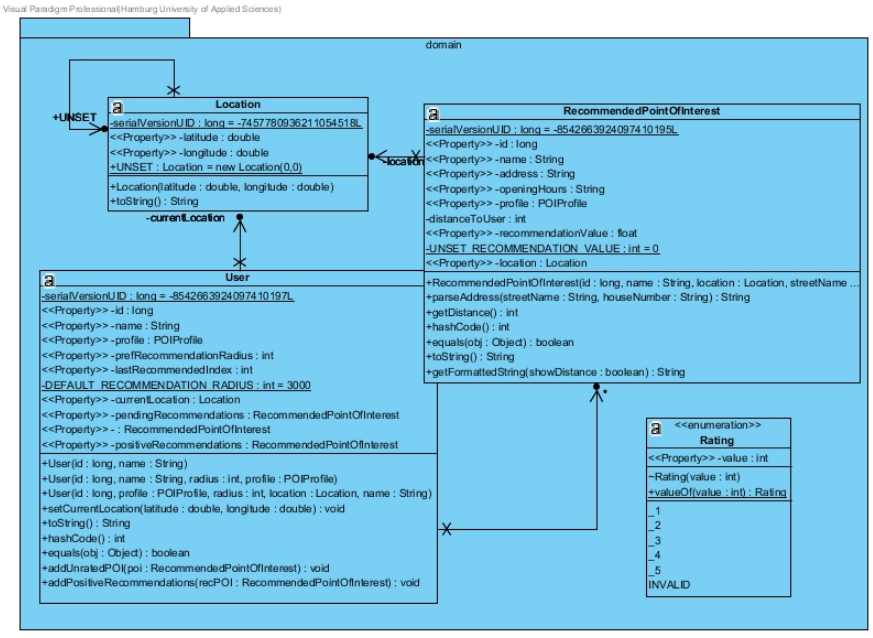

Figure C.7: Domain Package

C.3 Architecture

System Overview

In this section, the application's architecture is examined in depth. To get a first overview of the system's topography, figure C.8 is shown which gives an impression of the system's main components.

Figure C.8: System Overview

The figure shows the developed program in green and the external components that are used in the system, visualized in grey, as well as the used technologies to access them.

The system's heart is a Java web application that was set up with the help of the micro framework *Spark* [10]. The web application and PostgreSQL database are both deployed to the Platform-as-a-service *Heroku*.The Telegram Bot is connected to the application via webhook, specifically to the URL that was assigned to the web application by Heroku. Every time the user sends a message to the chatbot via Telegram, the web application is notified. The user input is then forwarded to the natural language parsing platform API.AI which interprets the meaning of the user input. The Foursquare API is used to provide images to the recommended points of interest.

Multitier Architecture

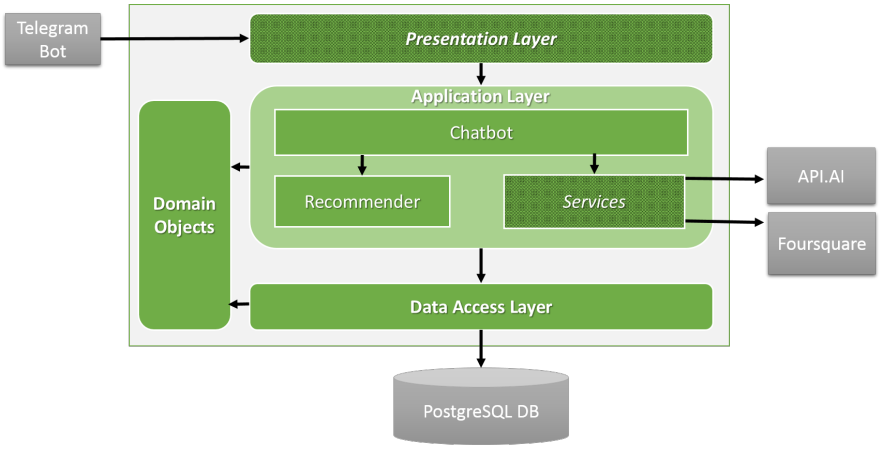

Figure C.9: The multitier architecture

As we can see, the application is designed as a typical 3-tier architecture and consists of a presentation layer, an application layer, a data access layer as well as a horizontal domain object layer. The multitier architecture is a well-known architecture pattern which ensures loose coupling and high cohesion. The direction of access between the layers is from top to bottom which avoids cycles in the dependency graph.

Being usually the layer defining the graphical user interface, the application's presentation layer is rather small in this case. As the UI is defined externally in the Telegram messenger, the presentation layer in this application only consists of the connection to Telegram and methods to change Telegram's appearance (e.g. keyboard buttons). This behaviour is mainly controlled by the class *TelegramBotHandler*.

The data access layer connects the application to the PostgreSQL database. It consists of the classes *UserDB*, *RatingDB* and *PointDB* which encapsulate the access to the three database tables.

Certainly, the application layer can be considered as the heart of the software. It contains the two major components *Chatbot* and *Recommender* as well as the helper component *Services*. In contrast to the vertical 3-tier layers, a horizontal layer is introduced that contains the domain objects of the application, such as *User*, *Rating* or *RecommendedPointOfInterest*. Besides using these objects in the application layer, they are also used in the persistence of the data access layer.

A main focus of the architecture design was the interchangeability of components that access external services, marked as cross-hatched areas in figure C.9. A reason for this is the recent importance of conversational interfaces these days and therefore the constant appearance of new technologies. In this

application, the presentation layer is completely adjusted to the Telegram messenger. If the decision is made to use a different messenger or include an additional one, only the presentation layer has to be adapted whereas the rest of the architecture remains untouched. Similarly, the *Services* component of the application layer is easily replaced if the natural language parsing platform is changed.

C.4 Procedural Design

The followings section serves to show the main flow of events and usage of components in the application.

Flow of Events

The principal message flow is shown in sequence diagram C.10. It includes all objects that are involved in forwarding and processing user input. This flow is always the same for any user input. As already explained previously, the input is received from the *TelegramBotHandler* and forwarded to API.AI which filters the meaning of the user input. This process is based on the agent model that was implemented in API.AI in which intents and concepts were defined. The response from the NLU platform is parsed from JSON to Java and stored in an object of the class *AgentResponse*. Based on this AgentResponse, the *TouristChatbot* decides which action needs to be taken and processes the action accordingly. The calculated answer is transmitted back to the messenger.

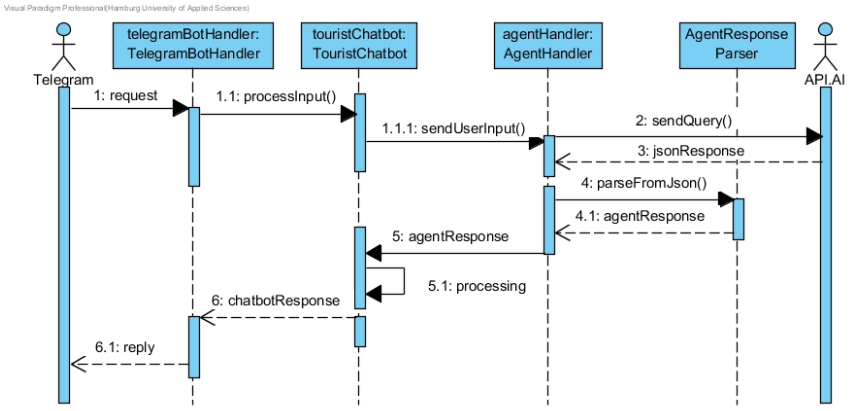

Figure C.10: Message Flow Sequence Diagram

The actual operations that are done in step 5.1 processing depends on the user input and ranges from triggering recommendations to storing user

interests or ratings in the database. Certainly, the most complex action to be taken is a recommendation, which is shown in the figure C.11.

Recommendation

This use case is used as an example as it involves all components of the program. For the sake of simplicity and easier understanding, the previous shown natural language parsing and message forwarding from and to Telegram are left out.

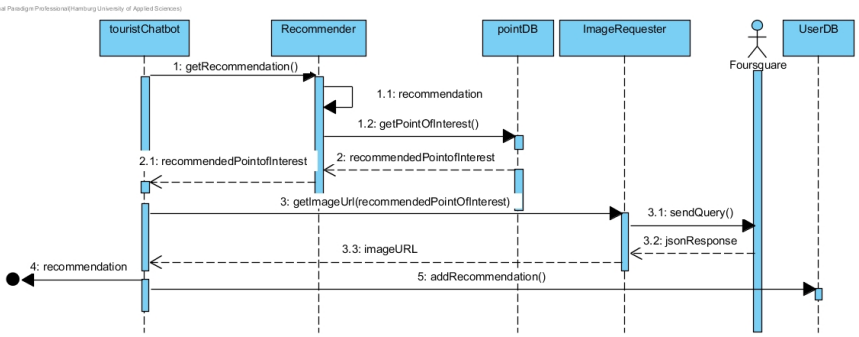

Figure C.11: Recommendation Sequence Diagram

The actual recommendation mechanism itself is presented in a simplified scheme by step 1.1. As already explained in the relevant aspects of the thesis, the recommender combines collaborative filtering and content-based filtering approaches.

In total, three points of interest are retrieved for the user, all of them situated in the recommendation radius. The aim is to retrieve as many recommendations as possible from the collaborative filtering mechanism as this is the mechanism that is more likely to output suitable recommendations. The collaborative filtering mechanism is based on user ratings which are retrieved from the PostgreSQL database using Mahout's *PostgresJDBCDataModel*. If the collaborative filtering mechanism outputs less than three different recommendations (for example caused by a lack of user ratings for the specific user), the content-based fallback mechanism is used.

In order to apply the content-based mechanism, some of Mahout's functionalities were overridden to fit the project's needs: The classes *ProfileSimilarity* and *POIDataModel* are created that are derived from Mahout interfaces. The *POIDataModel* is used to formalize the relevant data for the recommender, including both the user and all points of interest in the specified recommendation radius. More precisely, the *POIDataModel* introduces *ProfileItems* into the recommender. The *ProfileItem* interface is created to standardize objects that contain a *POIProfile*. The *POIProfile* defines the

preference to the tourist categories. The classes *User* and *Recommended-PointOfInterest* both implement the interface as they either show interest in the tourist categories or are characterized by the profile. The similarity measure made to compute the points of interest most similar to the user's preferences is specified in the class *ProfileSimilarity*. In this class, the user interests are compared with the properties of the points of interest based on their *POIProfile*.

Additionally to the general recommendation, the user is able to trigger a category specific recommendation. When the category specific recommendation is used, only recommendations are returned that match the given interest, e.g. sightseeing.

Conversation Flow Design

In order to visualize the conversation flow, a graph was designed defining the chatbot-user interactions based on the application's use cases. In the following, the general conversation graph is shown as well as the subgraphs defining the interface's main features. According to this conversation flow, corresponding intents and contexts were modeled in API.AI.

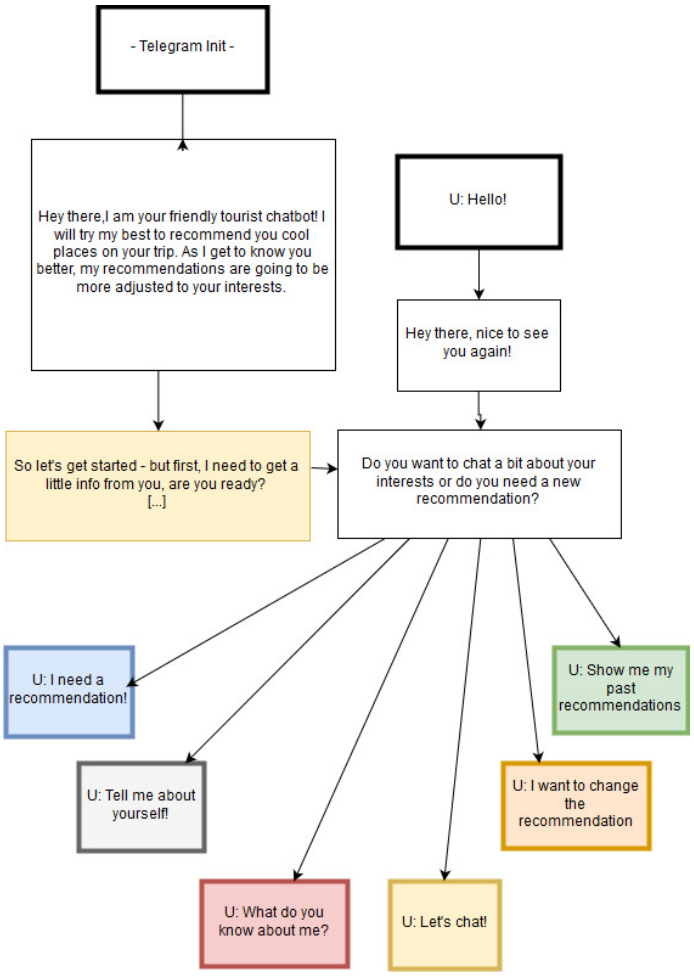

Figure C.12: The basic conversation flow

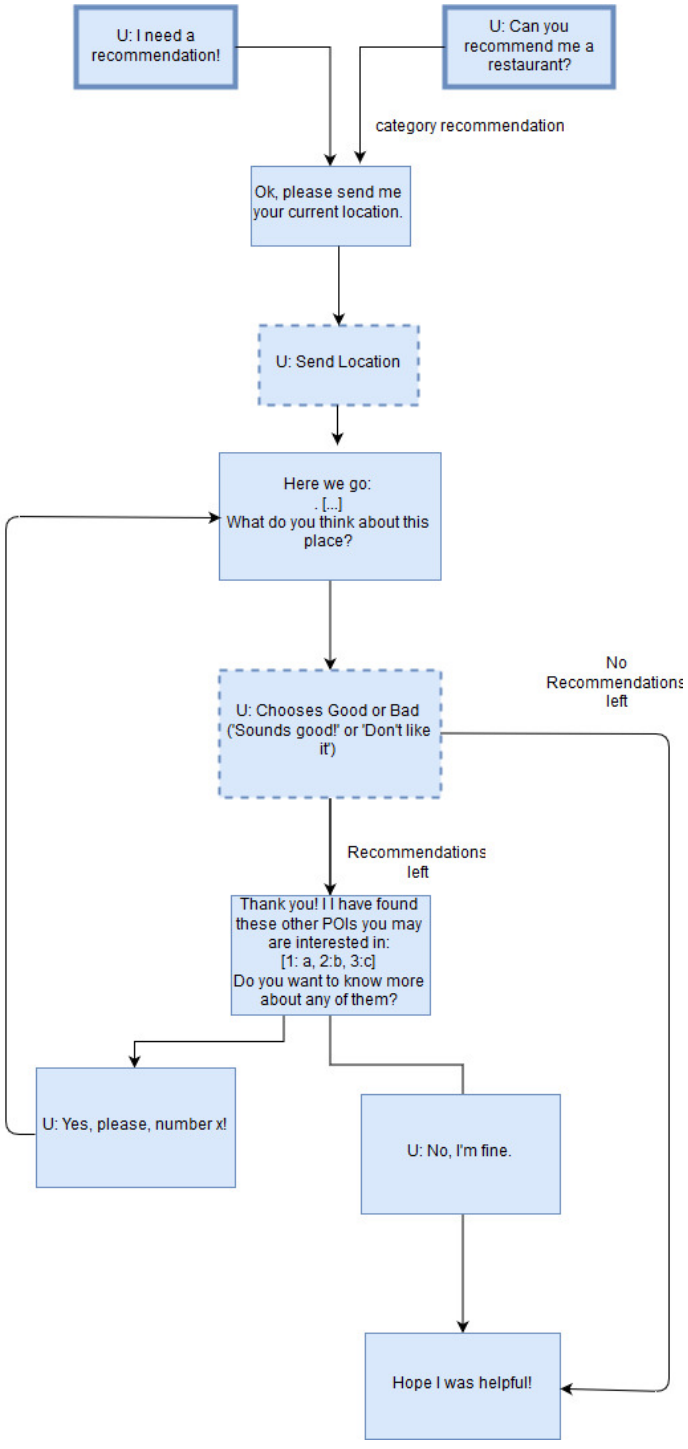

Figure C.13: The recommendation conversation flow

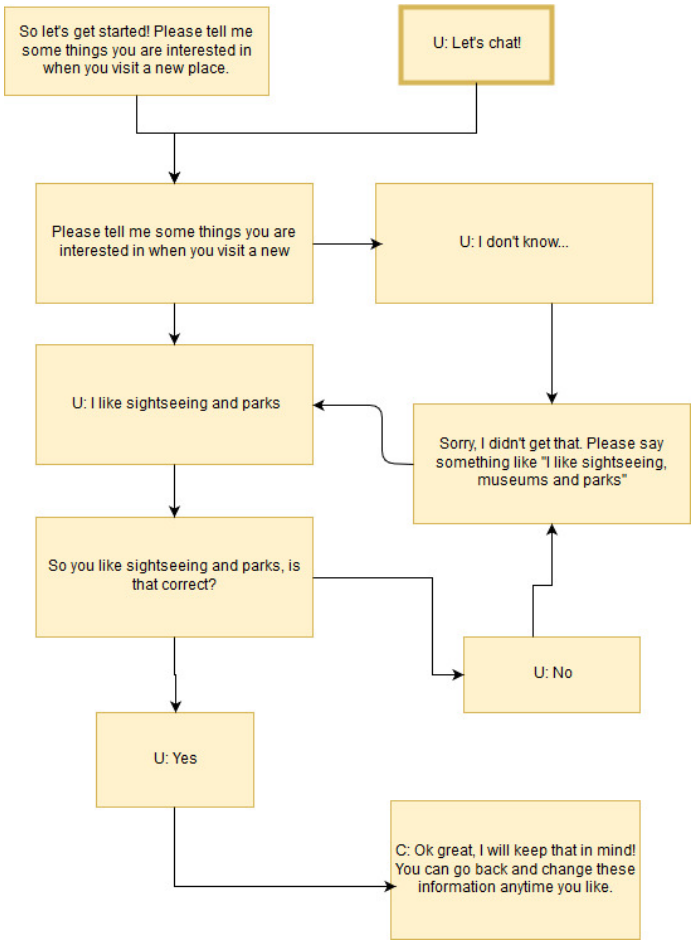

Figure C.14: Asking for the user's preferences.

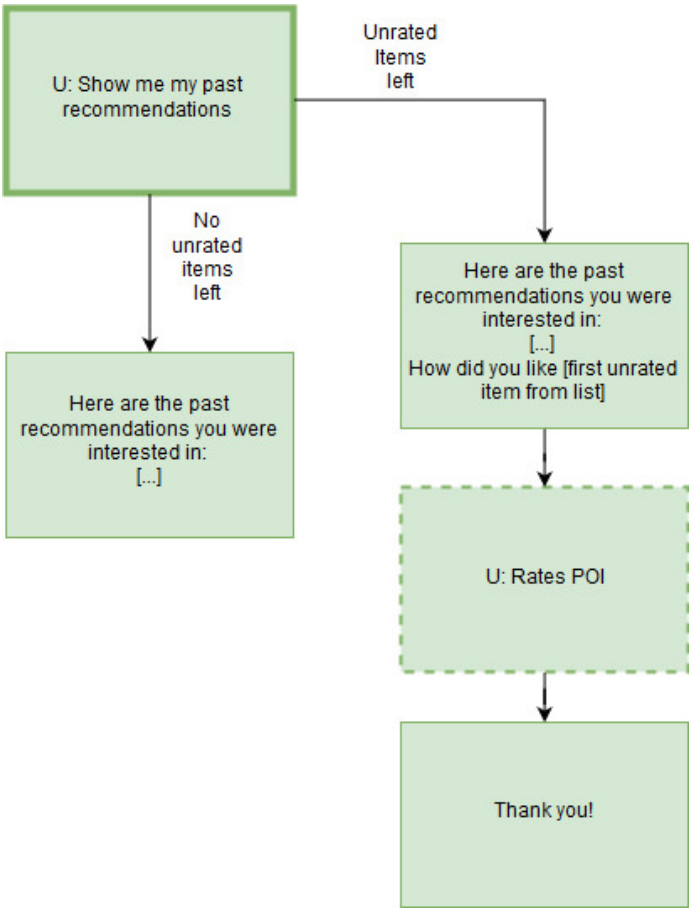

Figure C.15: Showing the past recommendations and asking the user to rate

API.AI Agent

Based on these graphs, intents and contexts were designed in API.AI. The user input is categorized in intents which loosely match the basic use cases. Every time the user chooses between multiple possible answers, a context is set to ensure an appropriate follow-up reaction. The majority of intents trigger actions that are defined in the agent. After parsing the user input with API.AI, the Java web application performs functionalities based on those actions, such as information storage in the database or a recommendation. For further examination, the agent model can be found in the delivered data and introduced into the API.AI (see Technical Programming Documentation).

Appendix D

Technical Programming Documentation

D.1 Introduction

This section contains the technical programming documentation, describing the directory structure of the presented CD, an installation guide as well as a documentation of the realized tests to measure the code quality.

D.2 Directory Structure

This report is handed in along with a CD containing the essential data to examine the application in more depth. The presented CD contains the following directories:

Documentation Contains the project's documentation, saved as both .pdf and .doc format.

Software Contains the executables of the tools needed to run the virtual machine.

Application Contains all essential data of the developed software, subdivided in the following directories:

- **Virtual Machine** The virtual machine image containing the infrastructure that was set up in the course of the project
- **Source Code** The source code of the application
- **Javadoc** The source code's documentation in Javadoc format.
- **Agent** Contains the exported api.ai agent as .zip
- **Data** Contains the raw geographic data and dumps of the initial database

D.3 Developer Manual

This section serves as an installation guide describing which steps to take to set up the application.

Database Setup

The geographic database is used by the chatbot application to retrieve and manage geographical data. In this project, the database runs on a Virtual Machine using Ubuntu 16.0.4 LTS. Therefore, Ubuntu's terminal is used to install most of the software. In order to make sure Ubuntu has access to the current package index, it is advised to execute an update command before installing the software:

```
sudo apt-get update
```

Set up PostgreSQL and PostGis

The first step is to install the data management system, PostgreSQL. To install the version used in this project, the following command is used:

```
sudo apt-get install -y postgresql=9.5+173
    postgresql-contrib=9.5+173
```

Then, the database "touristdb" is created as well as the managing user, which is called "touristuser". The createuser command will prompt for a password which can be chosen by the developers.

```
sudo -u postgres createuser -P touristuser
sudo -u postgres createdb -owner touristuser touristDB
```

Now we have set up the database, the PostGIS extension is installed and added to prepare the database for geospatial data.

```
sudo apt-get install -y postgis
    postgresql-9.5-postgis-2.2
sudo -u postgres psql -c "CREATE EXTENSION postgis;
    CREATE EXTENSION postgis_topology;" touristDB
```

The next step is optional, but seems convenient if the developers want to manage their database with the help of a user interface. The managing tool

pgadmin facilates running and editing SQL queries and viewing the stored data.

```
sudo apt-get install pgadmin3
```

To access the database in pgadmin3, a connection to the server must be added, which can be realized by clicking the plug button in the upper toolbar and then entering the following values.

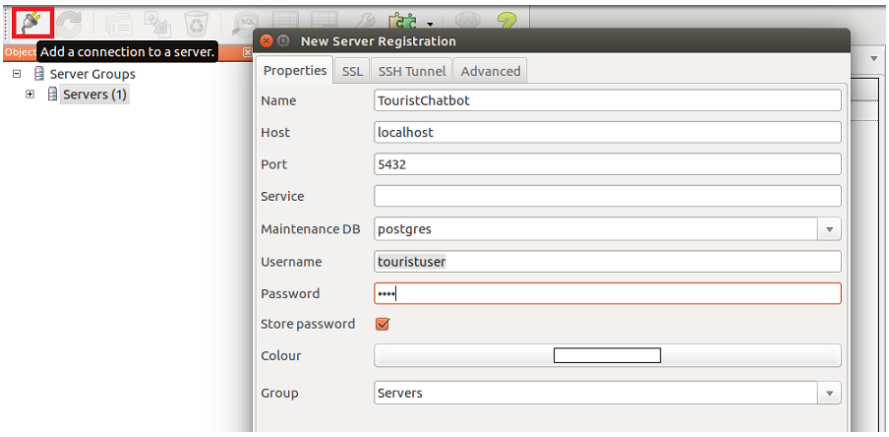

Figure D.1: pgadmin3: A server connection is added

In the object browser, the database schemas can be viewed accessing TouristChatbot -¿ databases -¿ touristDB.

Import Data into Database

Now that we have set up the database, it needs to be filled with geospatial data. In this project, the recommendations are based on test data of Barcelona. The required data is downloaded as a .pbf file from the website https://download.bbbike.org/osm/bbbike/Barcelona/. After that, the tool Osmosis is used to import the OSM data which can be installed using the following command:

```
sudo apt-get install osmosis
```

The next commands prepares the database for the osmosis import. It sets the hstore extension and the pgsnapshot database schema which causes that all relevant tag data are stored in a hstore column.

```
sudo -u postgres psql -c "CREATE EXTENSION hstore;"
    touristDB
psql -U touristuser -d touristDB -f
    /usr/share/doc/osmosis/examples/pgsnapshot_schema_0.6.sql
```

After that, the import itself is realized. Remember to execute this command in the folder where the downloaded .pbf file is situated and to add the corresponding password (which is set by the developer in the previous step of this manual).

```
osmosis --read-pbf file="Barcelona.osm.pbf"
    --write-pgsql host="localhost" database="touristDB"
    user="touristuser" password=password
```

In order to see if the import was successful, pgadmin3 can be used to take a look at the now imported data. Again, this step is optional. In the object browser on the left hand side of the user interface, the database tables can be viewed accessing *TouristChatbot* → *databases* → *touristDB* → *Schemas* → *public* → *Tables*.

Store User Information

In order to store information of the users or of the ratings they made, the existing users table must be modified. To do so, the following POSTGRESQL queries are executed so that new columns are added to our table. These modifcations can either be made using the psql command via bash or pgamin3's query tool.

```
alter table users add column recommendations bigint[];
alter table users add column unrated bigint[];
alter table users add column radius integer;
alter table users add column name ;
```

The ratings are stored in a newly created table:

```
create table ratings(
userId bigint,
pointId bigint,
ratings integer,
PRIMARY KEY(userId, pointId))
```

Access to External Services

In the following, it is described how to set up and access the external services used in this project: To access the conversational interface, the messenger Telegram as well as our natural language parsing platform api.ai are used. Additionally, the FourSquare API is used to retrieve images for recommended Points of Interests.

Telegram Bot

The messenger Telegram is used to provide an interface to our tourist bot. After installing Telegram on a mobile device and setting up an account, the bot can be created using Telegram's *BotFather*. The Botfather can be accessed using the messenger's search function. After that, the creation of the bot is triggered by entering /newbot in the input field.

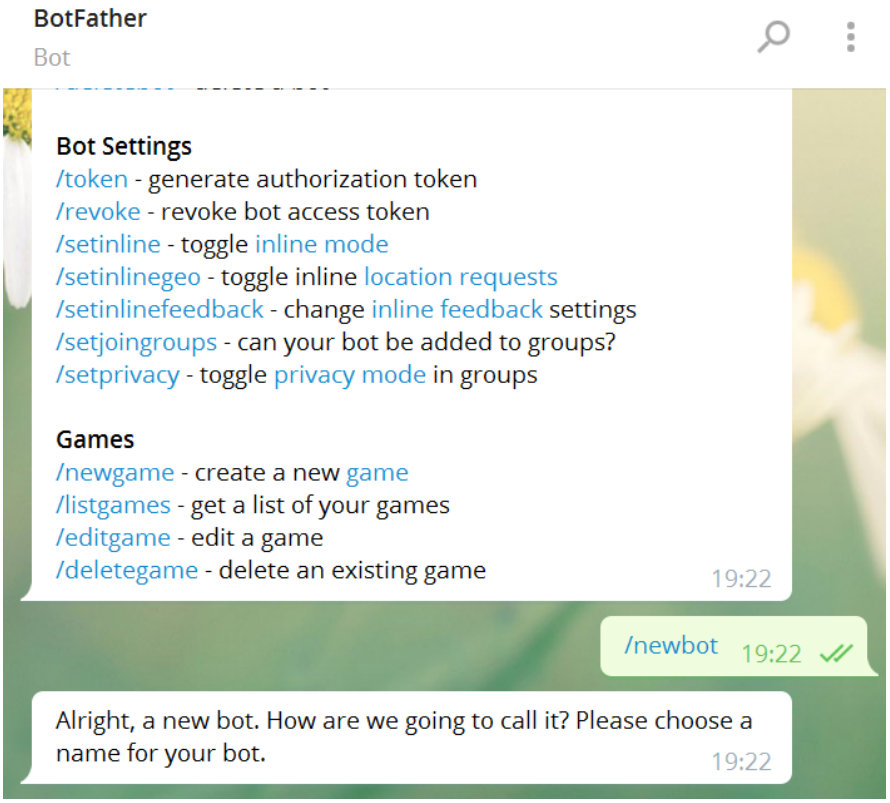

Figure D.2: The *Botfather* is used to create the Telegram Bot

After choosing a name and a username, the BotFather provides you with the authorization token for your bot. This authorization token is needed to access the Telegram bot from our web service. To do so, the token is saved

as an environment variable (see System Environment Variables). After saving the token, the web service is able to receive updates from and send messages to the Telegram bot via a webhook.

api.ai agent

In order to use the NLU platform api.ai, we need to set up an account. After doing so, the api.ai agent modeling interface can be accessed. First, we need to create a new agent and enter an agent name.

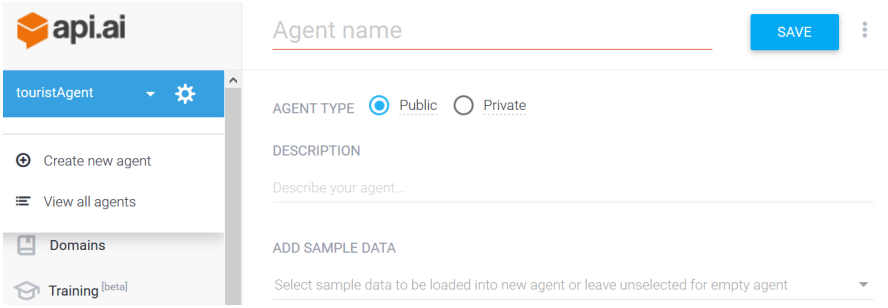

Figure D.3: api.ai agent creation interface

If the creation is successful, the entered agent name will appear on the left sidebar. In order to access the agent from our web service, api.ai's HTTP API is used. Therefore, the agent's API key is needed which can be accessed by clicking on the gear icon right to the agent name. The client access token is be copied and, again, introduced into the system environment variables (see System Environment Variables).

To restore the agent created in the course of this project, a.zip file containing the modeled agent can be found in the project's documents (Application/Agent). By clicking on the already mentioned gear icon right to the agent name, a subtab called "Export and Import" provides the possibility to import the agent from zip.

Foursquare

To retrieve images using the Foursquare API, a Foursquare account has to be created first. After that, the application has to be registered. If the registration is successful, the API access token can be found in the application overview. Again, these tokens are saved in the System Environment Variables).

Web Service Setup

Prerequisites

This project runs on Java 8 which is a requirement for the web service framework Java Spark as well as the used cloud service Heroku. The JDK can be downloaded from Oracle. Git is used to manage the project's source code as well as to deploy the code to the Platform as a Service application. In order to manage a Git repository, you have to sign up on GitHub and install Git using the following command.

```
sudo apt-get install git-all
```

On top of it, all libraries used during this project are included using the build-management tool Apache Maven. This program is installed by executing the following command in Ubuntu's command line. Maven is also needed for deploying a Java application to Heroku.

```
sudo apt-get install maven
```

Deployment to Heroku

The web service is deployed using Heroku, a cloud Platform as a Service (PaaS). In order to use the application, an account has to be created previously, following https://signup.heroku.com. At first, the Heroku command line interface has to be installed. Using Ubuntu, this is achieved executing the following commands:

```
sudo add-apt-repository "deb␣
    https://cli-assets.heroku.com/branches/stable/apt␣./"
curl -L https://cli-assets.heroku.com/apt/release.key |
    sudo apt-key add -
sudo apt-get update
sudo apt-get install heroku
```

After installing the command line, execute the following command and enter the Heroku credentials when asked.

```
heroku login
```

In Ubuntu's file system, change into the project directory touristbot and execute the following command in order to create a Heroku app.

```
heroku create
```

As we can see, a random application name is assigned, in this case e.g. https://arcane-fjord-43759.herokuapp.com/. This name has to be copied and inserted as our HEROKU_URL to the System Environment Variables). In doing so, the Telegram bot is hooked to the Heroku app later.

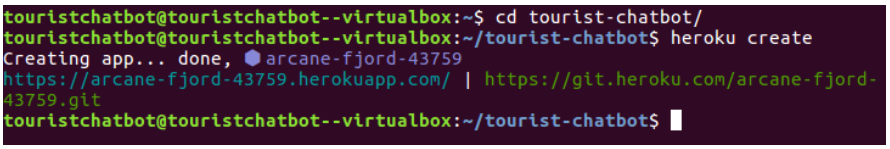

Figure D.4: Heroku Web App Creation

The source code can be now deployed using the command:

```
git push heroku master
```

Heroku Postgres Setup

In order to access our geospatial database online, Heroku Postgres is used to set up a productive PostgreSQL database. The following commands are executed from the Heroku repository:

```
heroku addons:create heroku-postgresql:hobby-dev
```

After that, the psql command is used in Heroku to enable sending POSTGRESQL queries:

```
PGUSER=postgres heroku pg:psql
```

The following queries are executed to enable the PostGis support in the PostgreSQL database.

```
CREATE EXTENSION postgis;
CREATE EXTENSION hstore;
CREATE EXTENSION postgis_topology;
```

The touristdb that was set up in the Database Setup is then pushed to the just created database:

```
PGUSER=postgres heroku pg:push touristdb DATABASE_URL
```

System Environment Variables

In order to manage the access tokens of our external components and not push them publically into a repository, system environment variables are used. These are set differently according to whether tests are run locally on the virtual machine or the application runs productively in Heroku. To set the system environment variables locally, the file /etc/environment is modified to contain the following variables:

```
HEROKU_URL=INSERT HEROKU URL
DATABASE_URL="postgres://touristuser:password@localhost:5432/touristdb"
TELEGRAM_TOKEN=INSERT TELEGRAM TOKEN
API_AI_ACCESS_TOKEN=INSERT API.AI CLIENT ACCESS TOKEN
F_CLIENT_ID=INSERT FOURSQUARE CLIENT ID
F_CLIENT_SECRET=INSERT FOURSQUARE CLIENT SECRET
```

The placeholders are replaced with the respective tokens from the external services. Concerning the variable DATABASE _ URL, the POSTGRESQL password has to be entered that was set in the Database Setup). For the productive runtime environment, the bash is used to set the environment variables on Heroku, entering the following command:

```
heroku config:set TOKEN_PARAMETER=VALUE
```

This command's execution is repeated for all of the above mentioned tokens with the exception of the variable DATABASE _ URL as it is an already predefined environment variable.

Integrated Development Environment

The project is developed using Eclipse Neon as an IDE. The 64-bit installer can be downloaded from Eclipse's website. After installing Maven and Eclipse, start Eclipse in order to import the source code. This can be easily done by importing a Maven project, executing *File* → *Import* → *Existing Maven Projects* and then choosing the project's source folder.

The project's source code can now be accessed and modified using Eclipse. Additionally, Eclipse is used to run the Junit tests for this project.

D.4 Program Compilation, Installation and Execution

The presented application was designed for online usage and is deployed to the Platform as a Service Heroku. Therefore, no further compilation, installation or execution steps are needed as this is managed by the PaaS. Changes to the previous source code are published by using the Git workflow, meaning to commit the changes and then push them to Heroku using the command:

```
git push heroku master
```

D.5 Tests

Tests were made during this project to ensure the project's quality. For this reason, several measures were taken to concentrate on different aspects of quality assurance.

Automation Testing using JUnit

Java's unit testing framework JUnit is used to design automated tests. Using JUnit, the test-driven development paradigm was applied in this project to ensure the code's correctness constantly during development. The tests can be found in the project folder tree navigating to *src/test/java*. The folder is subdivided into the packages *bot, dataAccess, poiRecommendation* which match the main components in the project. In this project, two different kinds of tests were designed:

Unit tests that concentrate on ensuring the proper functioning of the code on a class level.

Integration tests covering the essential use cases of the chatbot (including all of the system's relevant components and therefore demonstrating the proper interaction of the components). These integration tests can be found in the JUnit test class *src/test/java/bot/TouristChatbotTest*

The code coverage tool EclEmma is used to show how much of the source code is actually tested by the JUnit tests. EclEmma is integrated into the IDE Eclipse. The following results are obtained by executing all of the project's JUnit tests:

java (Jun 9, 2017 5:06:57 PM)

Element	Coverage	Covered Instructions	Missed Instructions
touristbot	80.0 %	5,220	1,304
src/main/java	80.0 %	5,220	1,304
chatbot	76.5 %	1,796	553
model	77.4 %	1,496	437
dataAccess	85.7 %	1,408	235
recommender	86.8 %	520	79

Figure D.5: Code Coverage Analysis Result

As we can see, 80 % of the productive source code is tested. The test coverage mainly centers on the proper functioning of the service classes of the chatbot, meaning the classes that provide important functionalities and are error prune due to their complex structure. On the other hand, model classes are not as extensively tested as most of them follow a simple design, providing only getter, setter and field-based equals implementations. Furthermore, classes using code from external libraries are not a focus of the tests as it is assumed that their proper functioning is ensured by the developers of the respective libraries (e.g. hooking the application to the Telegram bot by using a third party Telegram library).

Recommender Evaluation

To ensure that the computed recommendations of the chatbot are actually adjusted the users' preferences, an evaluation is made using the *Mahout* framework. The evaluation is limited to the user-based part of the recommender which is based on user ratings. The reason for this is that the content-based mechanism is used as a fallback that provides recommendation when the user data is too sparse for the user-based recommender to perform properly. In fact, the content-based mechanism is not really a recommender but rather a similarity measure. The proper functioning of this mechanism is tested using unit tests (see the JUnit test class *RecommenderTest*).

The actual recommender evaluation can be found in *src/test/java/poiRecommendation/RecommenderEvaluation*. Mahout's *RecommenderIRStatsEvaluator* is used which splits the available user data automatically into training and test sets. To evaluate the recommender performance, information retrieval metrics are computed. More precisely, the metrics precision and recall are examined as well as the F-Measure which is a harmonic mean of the other mentioned metrics [11].

$$Precision = \frac{|\text{relevant items retrieved}|}{|\text{items retrieved}|} \tag{D.1}$$

$$Recall = \frac{|\text{relevant items retrieved}|}{|\text{relevant items in collection}|} \tag{D.2}$$

$$F - Measure = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{D.3}$$

During the recommender development, Mahout provides a variety of similarity and neighborhood functions to choose from. Depending on the applied functions, the recommender computes the similarity between two items differently and considers different items to be suitable for a similarity measure. Using the evaluation results, the combinations of the following similarity and neighborhood functions can be tested to determine which one of them performs best on the given data. The following table shows the f-measures of the 16 investigated combinations:

	Threshold User	Nearest 2 User	Nearest 5 User	Nearest 10 User
Euclidean Distance	0.5	0.4	0.5	0.5
Pearson Correlation	NaN	NaN	NaN	NaN
Loglikelihood	0.55	0.29	0.73	0.5
Spearman Correlation	NaN	NaN	NaN	NaN

Table D.1: F-Measures of the recommenders using different similarity and neighborhood functions

It quickly becomes clear that the Pearson Correlation Similarity and Spearman Correlation Similarity are not suitable for this recommender as they do not output valid performance results at all. A reason for this is that the applied user data is too sparse to achieve significant results using these similarity functions. The best f-measure is achieved by a recommender using Loglikelihood Similarity and Nearest-5-User as neighborhood function. Examining this recommender's performance in detail, we see that it achieves a precision value of 0.8 and a recall of 0.67. As we can see, the precision value is higher than the recall value. The precision tells us the fraction of retrieved items that are relevant whereas the recall tells us the fraction of relevant items that are retrieved. In this project, the precision value is considered as more important than the recall value as the user has to be provided with recommendations that fit his interests. Yet, the fact that the user-based recommender may not find all possible recommendations for the user, is rather negligible as the

content-based mechanism is used as a fallback in this case. Also it is assumed that the overall performance of the recommender will rise with increasing data as more users use the chatbot (cold start problem of a recommender).

Appendix E

User Manual

E.1 Introduction

This user manual serves as a guide to show the user how to use the presented chatbot via the instant messenger Telegram. In the following, it is explained how the user accesses the chatbot as well as presenting the main features of the chatbot exemplary.

E.2 User Requirements

In order to use the chatbot, the user needs to have a smartphone which is capable of connecting to the Internet. The messenger Telegram has to be downloaded and installed on the device, the app can be found using the respective App Store of the phone. In order to create a Telegram account, a real mobile phone number has to be introduced which links the Telegram messenger to the smartphone's sim card.

E.3 Installation

The chatbot is accessed via Telegram. In order to contact the chatbot, its Telegram user name (**touristrecommenderbot**) is introduced in the Telegram search function. Pressing *Start* activates the user-chatbot conversation. After pressing the button, the conversation should look as follows:

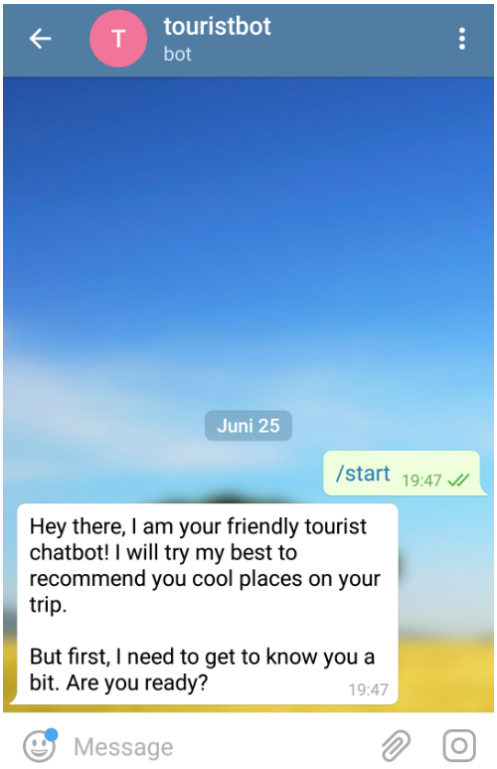

Figure E.1: Starting the chatbot

E.4 User Manual

The chatbot can be acessed by sending messages using the Telegram keyboard. Users are free to send the chatbot any message they like – however, the chatbot may reject a request if the user's intention is not tourism-related. The following section shows the chatbot's main features and examples of how to trigger them.

Help

The user shows interest in the chatbot's functioning, so a guide is returned in which the main features are outlined.

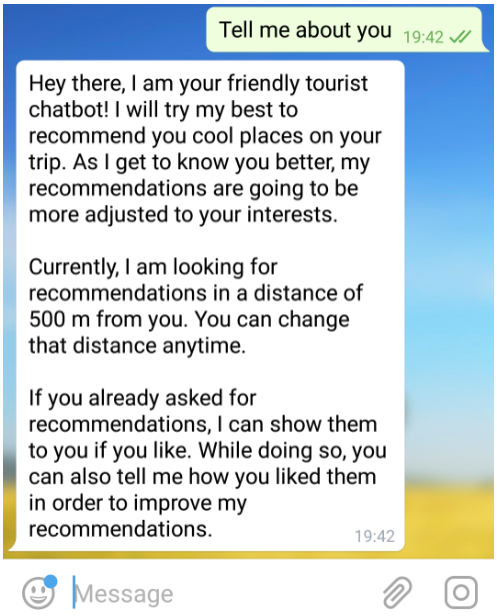

Figure E.2: The user asks for help.

Chat About Preferences

The user shows the intention to talk and the chatbot asks the user about his tourist preferences. If the chatbot is told by the user that the interests were not filtered correctly, the chatbot asks the user to rephrase his interests.

Figure E.3: The user tells the chatbout about his preferences.

Recommendation

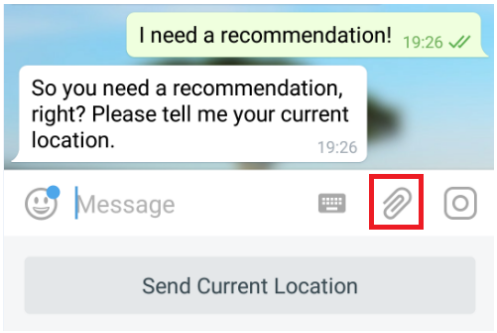

Figure E.4: The user triggers the recommendation.

The user triggers the recommendation and is asked to send his location. The shown button helps sending the user's current location. However, due to the fact that the presented chatbot only recommends points of interest for Barcelona, the recommender will not be able to recommend a point of interest when the user is not situated in that city. Therefore Telegram's manual location attachment function can be used, accessed by the highlighted paper clip button. The location can be changed using drag and drop.

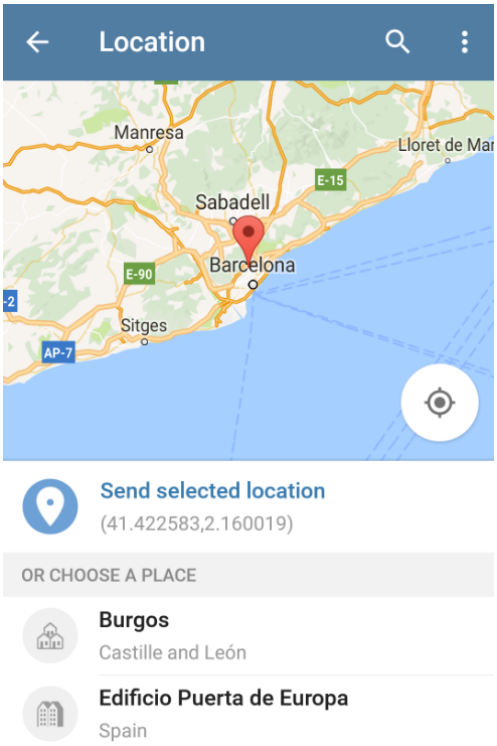

Figure E.5: A location in Barcelona is chosen.

After selecting a location, a point of interest that matches the user's interest the most is presented, alongside with a photo of the point of interest (if available). The user is asked to send his first impression of the point of interest, using one of the mutually exclusive buttons or typing a message on his own.

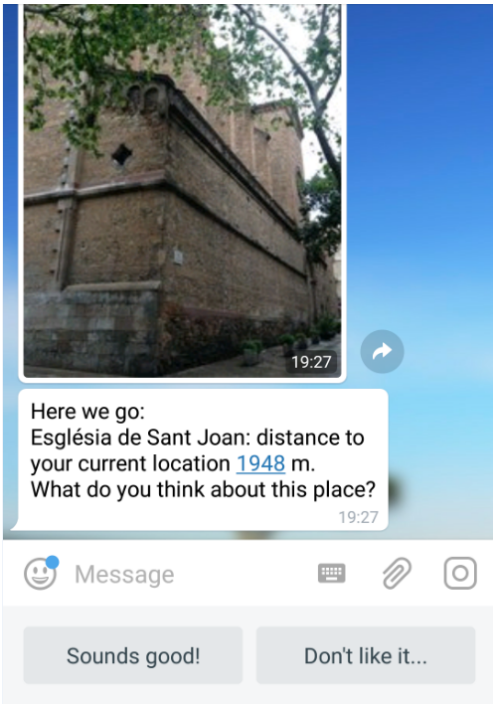

Figure E.6: A point of interest is recommended.

The chatbot provides other points of interest that match the user's interest and are situated in the given radius. The user can choose to see one of them by entering the point's index or decline.

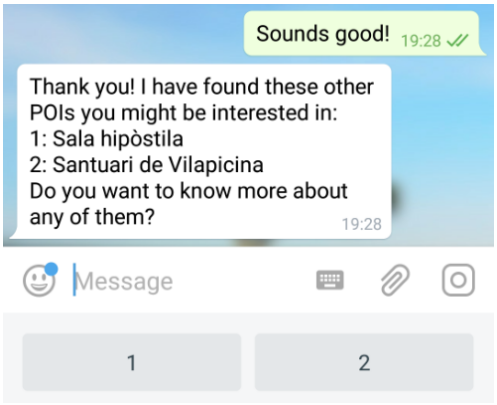

Figure E.7: Other points of interest the user might like are shown.

After all points of interests were shown or the user declined to see more, the recommendation process is completed.

Category-based Recommendation

If the user mentions his interest for a specific tourist category, only points of interest that match the stated categories are recommended. Such a category-based recommendation can be triggered as follows:

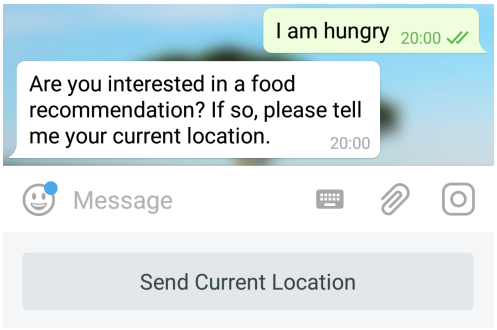

Figure E.8: Category-based recommendation

Show Past Recommendations

The user asks to see his past recommendations and is presented with a list of the recommendations he has seen before and were interested in (by showing a positive first impression during recommendation). If there is a recommendation the user has not rated yet, he is asked of his final impression by rating the point with stars from 1 to 5.

Figure E.9: The user's past recommendations

Specify Recommendation Radius

The recommendation radius can be specified by entering the preferred maximal distance to the points of interest.

Figure E.10: The recommendation radius is specified

Show Personal Information

The user is provided with the personal information the chatbot has saved during the course of the conversation.

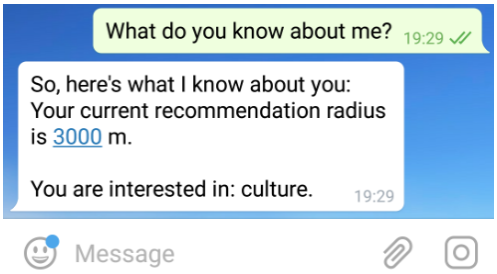

Figure E.11: The collected user information is shown by the chatbot.

Error Handling and Reset

There is certainly no such thing as bug free software, despite extensive testing. In the rare case of the chatbot not responding appropriately or not responding at all, the chatbot conversation should be restarted by the user. In order to do so, the user has to enter the command *start* in the Telegram messenger which restarts the user interaction with the Bot and leads to the state previously seen in screenshot E.1.

This command can also be used if the user wants to reset the conversation, for example in order to make the chatbot delete the previous collected user data.

Bibliography

[1] Ken Schwaber and Jeff Sutherland. The scrum guide, 2014. [Internet; accessed 23/06/17].

[2] GitHub. How developers work, 2017. [Internet; accessed 30/06/17].

[3] ZenHub. Prioritize your projects and release better software, 2017. [Internet; accessed 30/06/17].

[4] API.AI. Terms of use and privacy policy, 2017. [Internet; accessed 25/06/17].

[5] Foursquare. Rate limits, 2017. [Internet; accessed 25/06/17].

[6] Heroku. Simple, flexible pricing - plans for the need of every app, 2017. [Internet; accessed 25/06/17].

[7] Open Data Commons. Open database license (odbl) v1.0, 2017. [Internet; accessed 25/06/17].

[8] Telegram. Api terms of use, 2017. [Internet; accessed 25/06/17].

[9] Telegram. Telegram bot api for java, 2017. [Internet; accessed 27/06/17].

[10] Spark. Spark - a micro framework for creating web applications in kotlin and java 8 with minimal effort, 2017. [Internet; accessed 27/06/17].

[11] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.