# Annex 3 – Design Specification

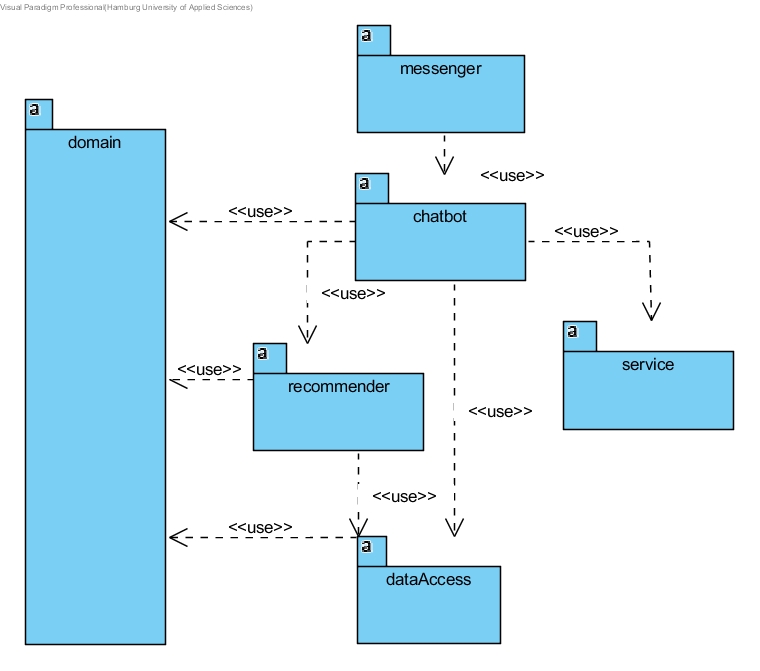
## Introduction

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 more detailed by examining the procedural steps of the main use cases.

## Data Model

**Package Structure**

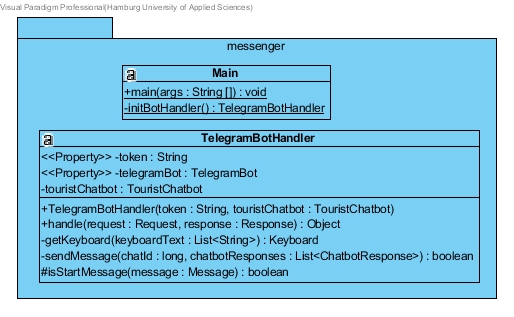
The application’s source code is divided into six packages, which also reflects the application’s main component structure. The figure shows the packages and relationships between them.



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.

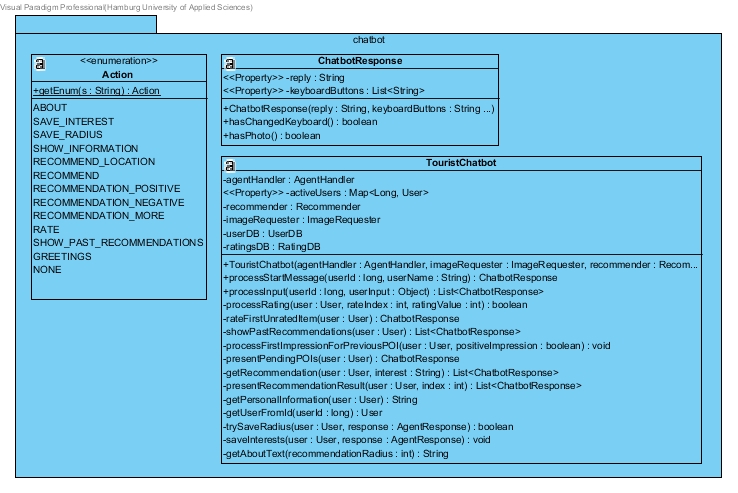
**Messenger:**

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 [x] is used.



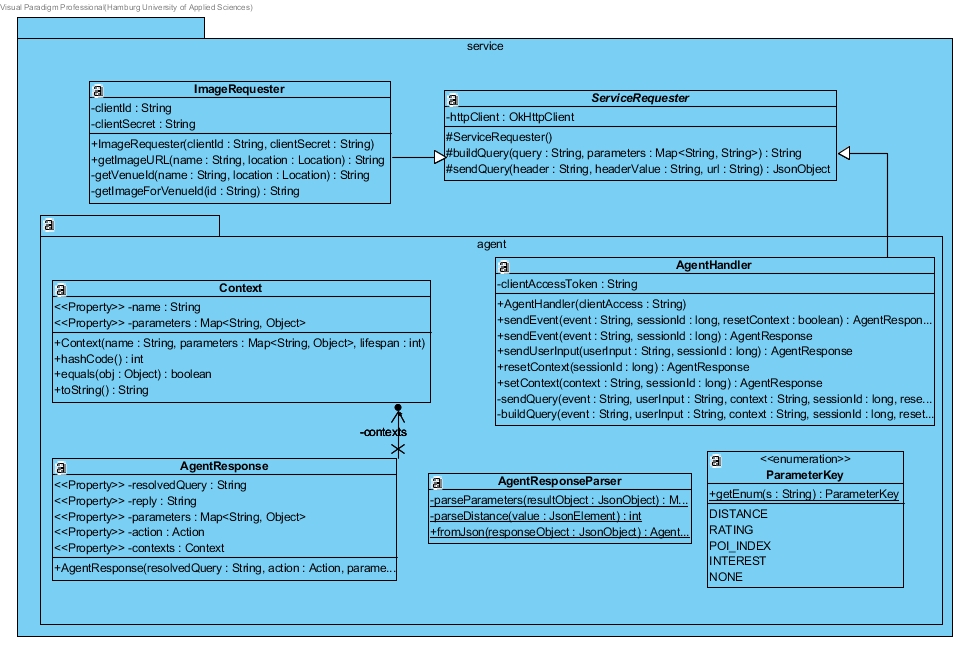
**Chatbot**:

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*.



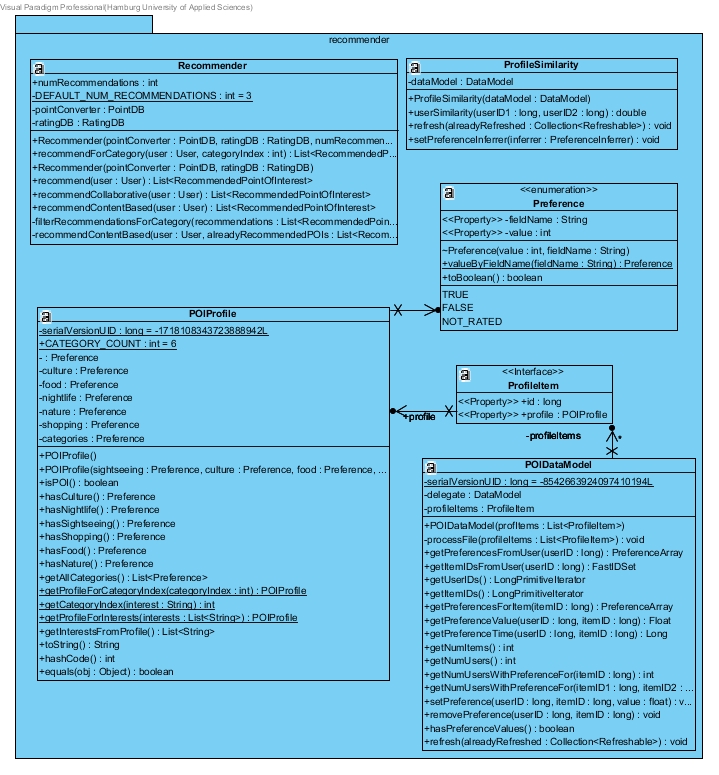
**Service:**

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.



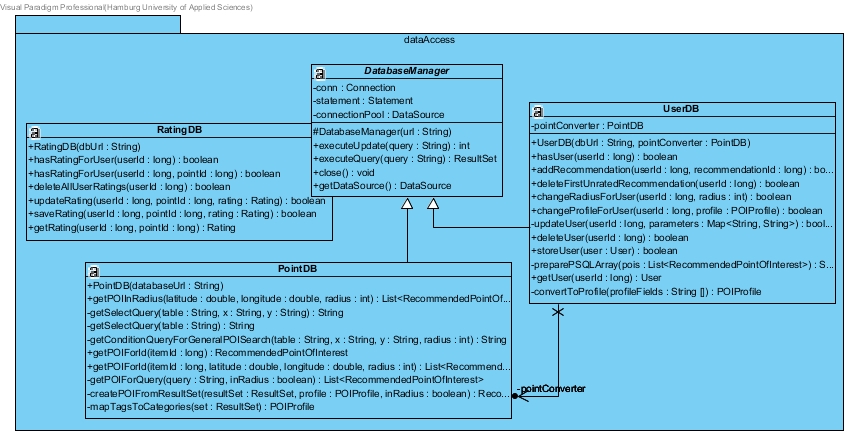
**Recommender:**

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 Procedural Design.



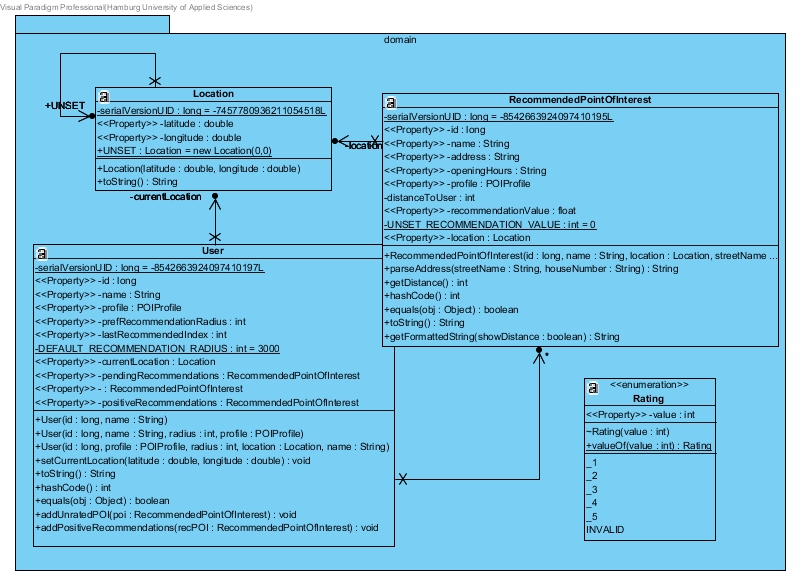
**Data Access Package:**

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*.



**Domain:**

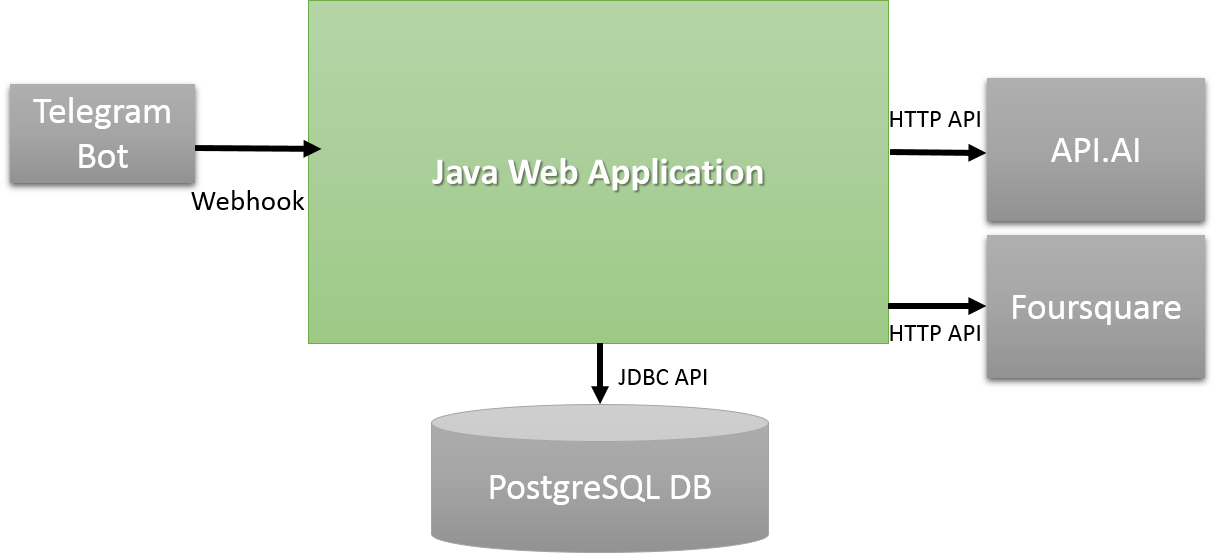
The domain package contains the business objects of the project, meaning natural objects that have a representation in the real world. They do not have any logic or provide any services. The domain objects are used system-wide and are shared between the components of the data model.



## Architecture Design

**System Overview**

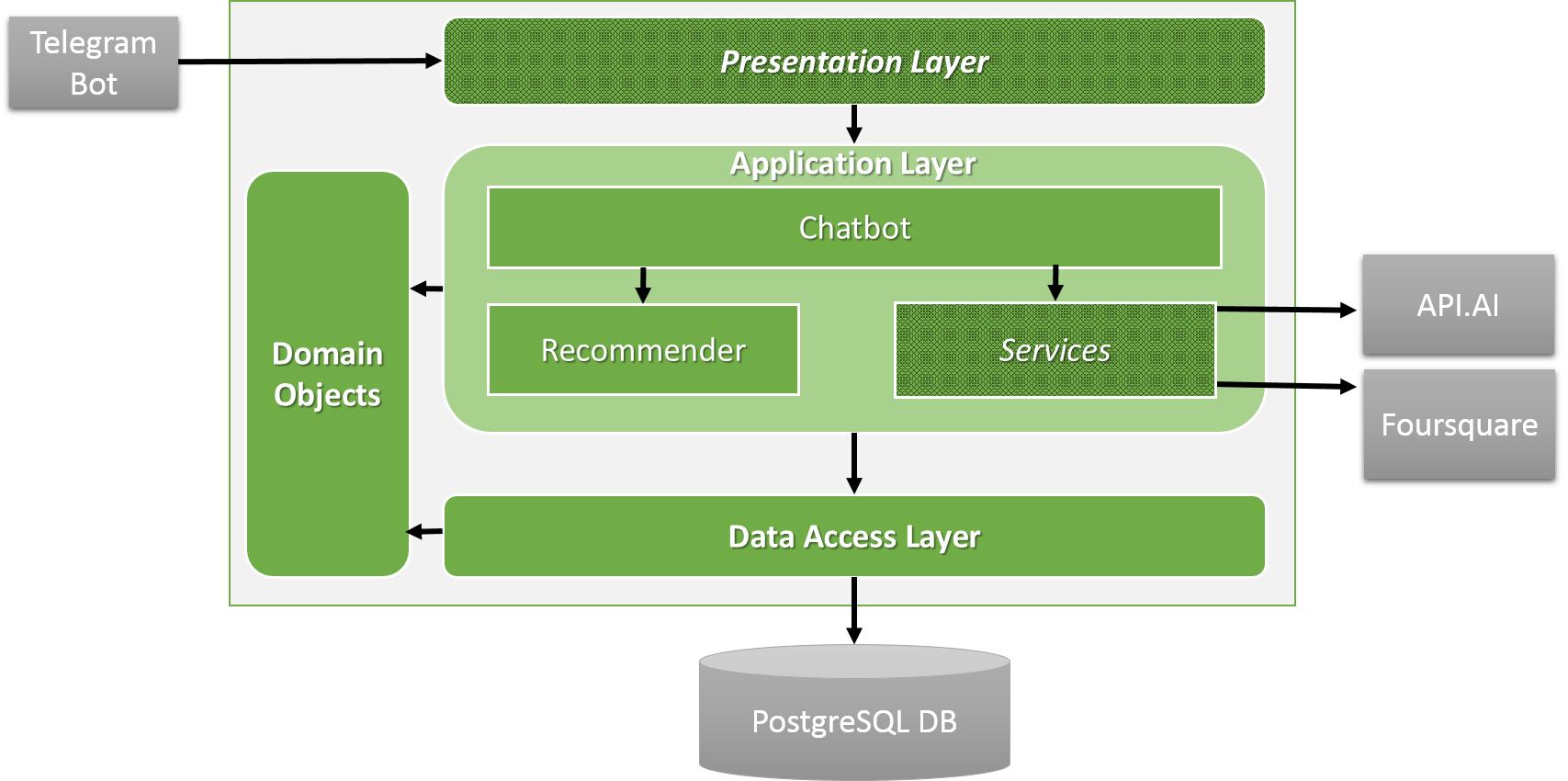
In this section, the application’s architecture is examined in depth. To get a first overview of the system’s topography, figure 1 is shown which gives an impression of the main components.



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* [s]. 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 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’s input. The Foursquare API is used to provide images to the recommended points of interest.

**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.

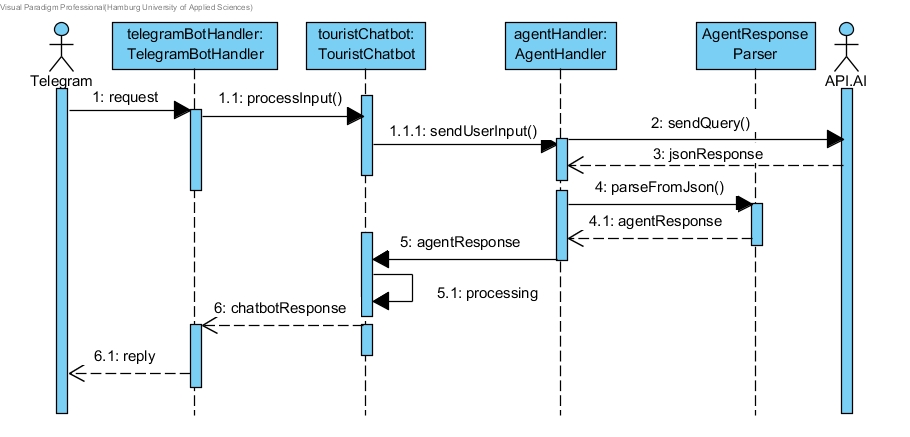
Certainly, the application layer can be considered as the heart of the software. It contains the two major components *Chatbot* and *Recommender* as well as the helper component *Services.* In contrast to the vertical 3-tier layers, a horizontal layer is introduced that contains the domain objects of 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 the figure. 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 it is decided to use a different messenger or include another 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 it is decided to use another natural language parsing platform or access a different API to retrieve images.

## Procedural Design

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

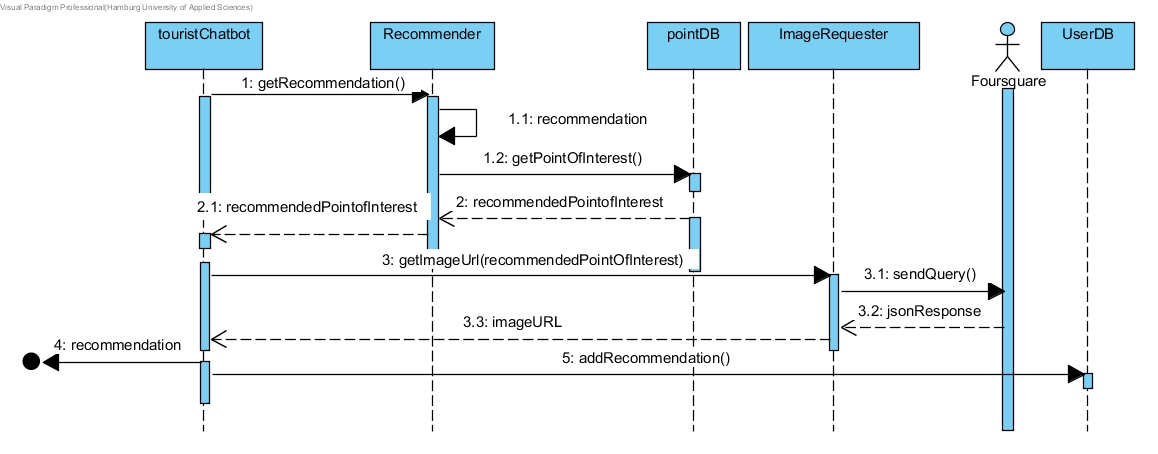
The principal message flow is shown in the sequence diagram. It includes all objects that are involved in forwarding and processing user input. This flow is always the same for any user input. As already explained previously, the input is received from the *TelegramBotHandler* and forwarded to API.AI 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 on which action needs to be taken and processes the action accordingly. The computed chatbot answer is transmitted back to the messenger.



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 examined in the following paragraph.

**Recommendation**

The recommendation is shown in detail exemplary 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.



The actual recommendation mechanism itself is represented by step 1.1. As 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.

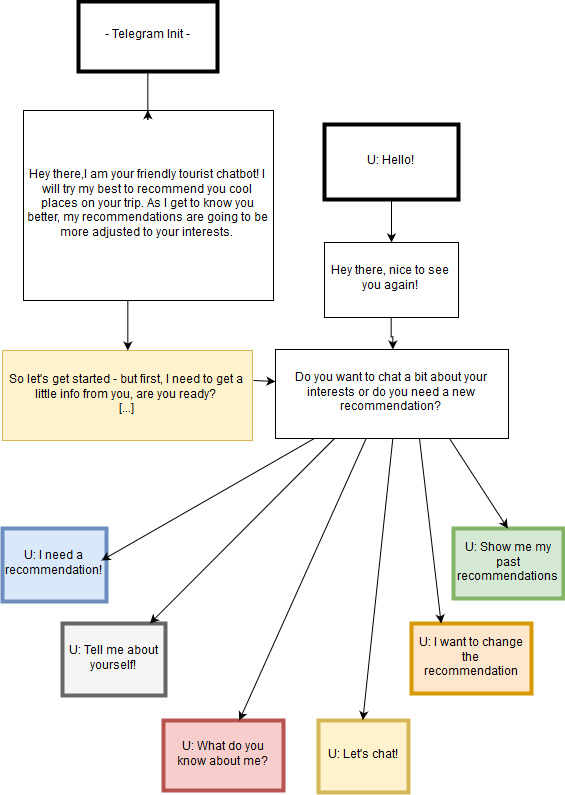
In order to apply the content-based mechanism, some of Mahout’s functionality was overridden to fit the project’s needs: The classes *ProfileSimilarity* and *POIDataModel* are created that are derived from Mahout interfaces. The *POIDataModel* is used to formalize the relevant data for the recommender, including both the user and all points of interest in the specified recommendation radius. More precisely, the *POIDataModel* introduces *ProfileItem*s into the recommender. The *ProfileItem* interface is created to standardize objects that contain a POIProfile. The POIProfile defines the existence of the different tourist categories. The classes User and RecommendedPointOfInterest both implement the interface as they either show interest in the tourist categories or are categorized.

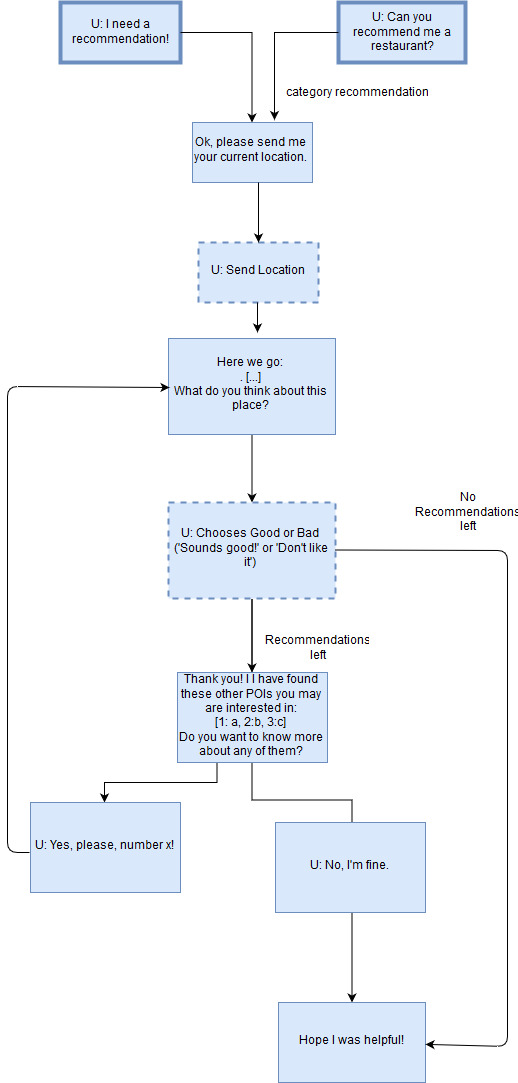
The similarity measure that is made to compute the points of interest that are most similar to user’s preferences is specified in the class *ProfileSimilarity.* In this class, the user interests are compared with the properties of the points of interest based on their *POIProfile*.

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.

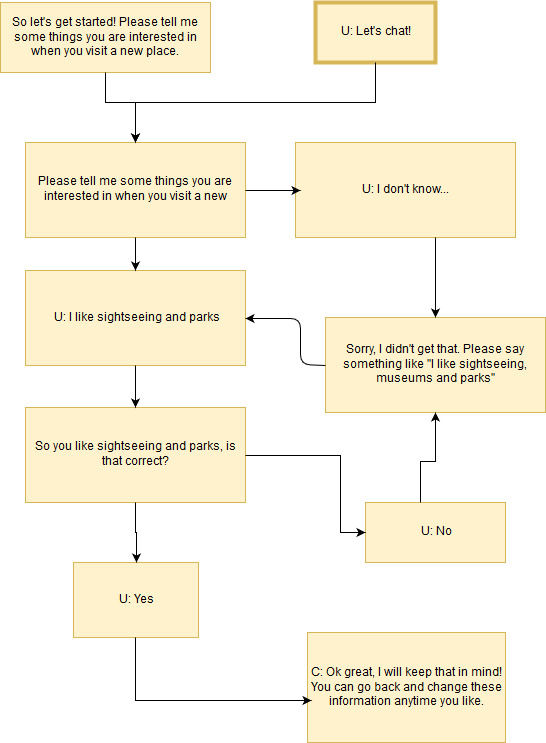
**Conversation Flow Design**

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.

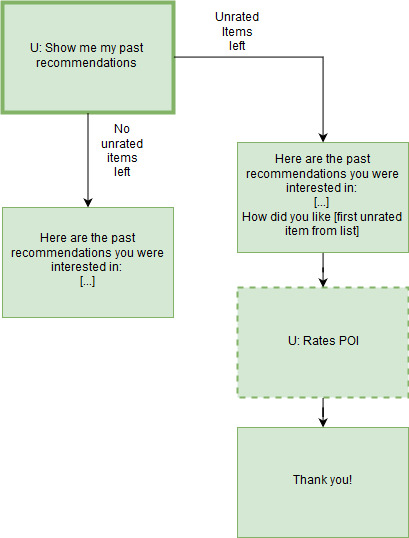




**Chat About Preferences**

****

**Show Past Recommendations**

****

**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 can be defined in the agent. Based on the action, different application functionalities are performed, such as information storage in the database or a recommendation. The agent’s functionality is included in the .zip file *api\_ai\_tourist\_agent.zip* and can be introduced into API.AI for further examination (see Technical Programming Documentation).

Bibliography:

[x]: <https://github.com/pengrad/java-telegram-bot-api>

[s]: http://sparkjava.com/