# Annex – 4

Introduction

This section contains the technical programming documentation. In the following, the directory structure of the presented CD and an installation guide to set up the project is described as well as a documentation of the realized tests to measure the code quality.

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 to the necessary tools used 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
  + Agent: Contains the exported api.ai agent as a .zip
  + Javadoc: The source code’s documentation in Javadoc format.
  + Geographic data: Contains the applied geographic data and dumps of the initial database

Developer Manual

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

## 4.1 Database Setup

### 4.1.1 Set up PostgreSQL and PostGis

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

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.

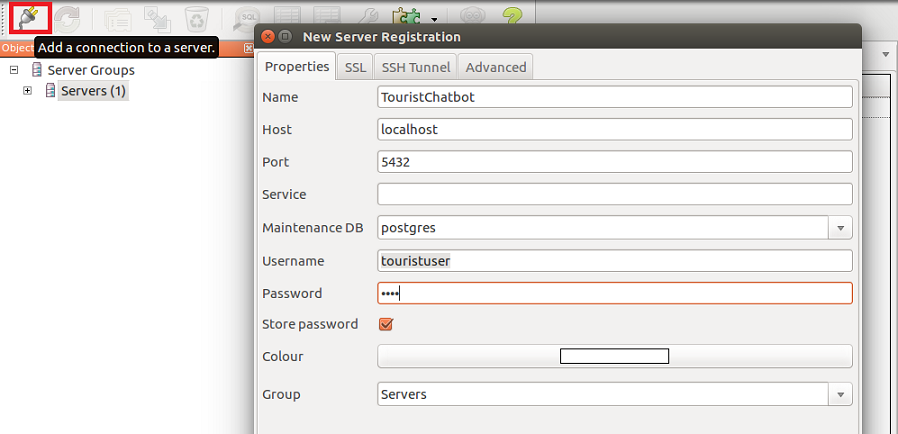
sudo apt-get install -y postgis postgresql-9.5-postgis-2.2

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.



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

### 4.1.2 Import Data into Database

Now that we have set up the database, it needs to be filled with geospatial test 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/. Then, 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.

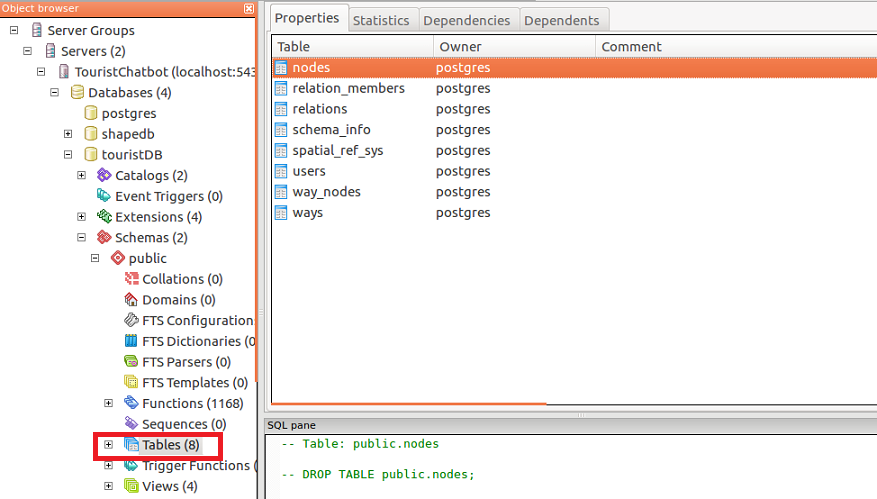
sudo -u postgres psql -c "CREATE EXTENSION hstore;" touristDB

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, the database tables can be viewed accessing TouristChatbot -> databases -> touristDB-> Schemas -> public -> Tables.



If you open the context menu on one of the tables (e.g. nodes) by clicking right, the option *view data* is available which shows the previously imported data.

Set up user tables

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,

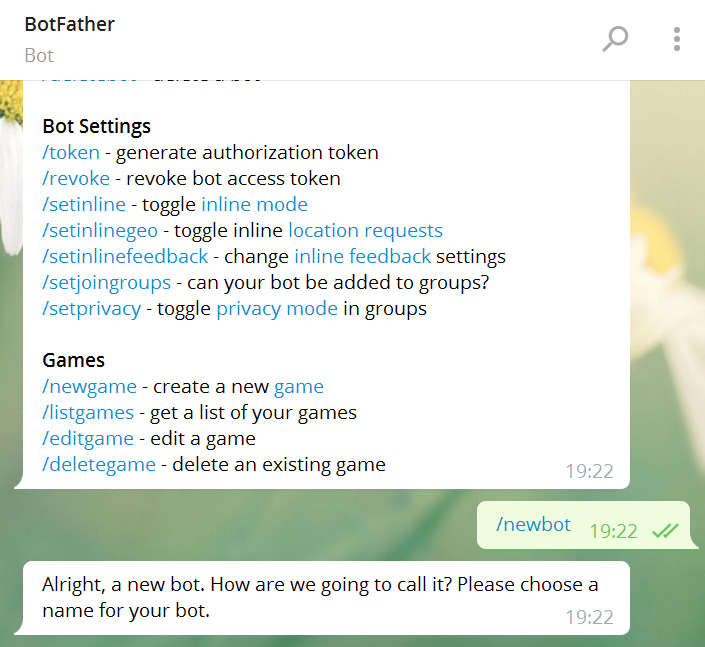
PRIMARY KEY(userId, pointId))

# Access to External Services

To access the conversational interface, the messenger Telegram as well as our natural language parsing platform have to be set up first.

### 4.2.3 Telegram Bot

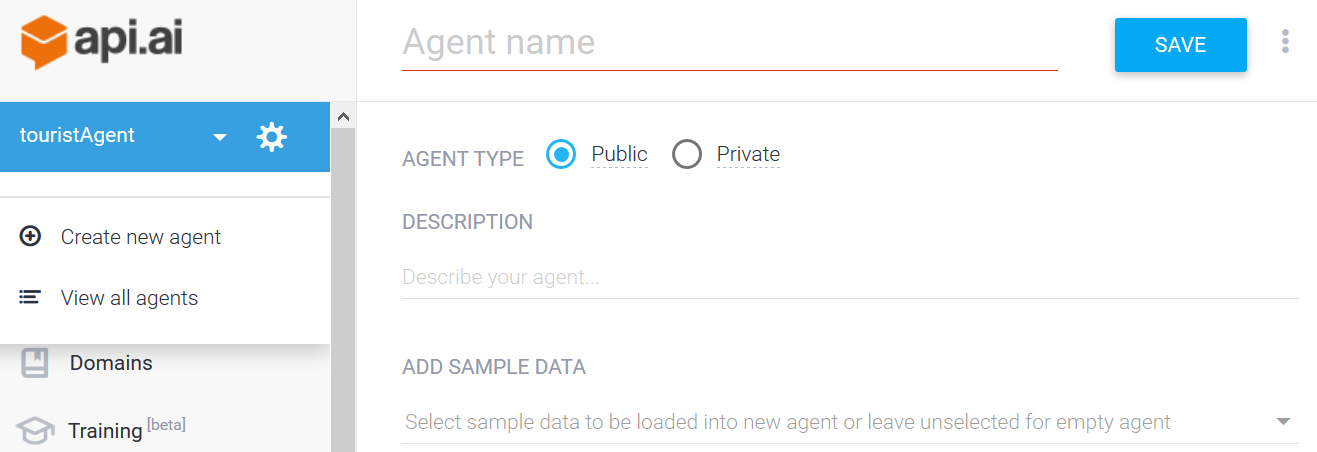
The messenger *Telegram* is used to provide an interface to our tourist bot. After installing *Telegram* on a mobile device and setting up an account, the bot can be created using *Telegram’s* *BotFather*. Using the messenger’s search function, the *BotFather* is accessed and the creation of the bot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



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. Therefore, we need to introduce the token as an environment variable as a value for the parameter *telegramToken*. After introducing this token, our web service is able to receive updates from and send messages to the Telegram bot via a webhook.

### 4.2.4 api.ai agent

In order to use the NLU platform *api.ai*, we need to set up an [account](https://console.api.ai/api-client/)*.* Creating a free account is sufficient for this application. 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.



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 (<https://de.foursquare.com/developers/register>). If the registration is successful, the API access token can be found in the application overview (https://de.foursquare.com/developers/apps). Again, these tokens are saved in the System Environment Variables.

## 4.2 Web Service Setup

### 4.2.1 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**](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html).

*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](https://github.com/join?source=header-home) 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

### 4.2.5 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 ./](https://cli-assets.heroku.com/branches/stable/apt%20./)”

curl –L <https://cli-assets.heroku.com/apt/release.key> | sudo apt-key add –

sudo apt-get update

sudo apt-get install heroku

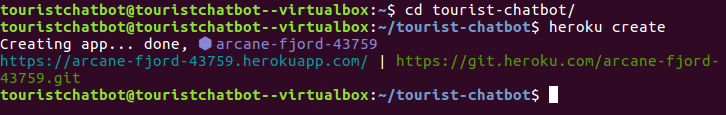
After installing the command line, execute the following command and enter the *Heroku* credentials when asked.

heroku login

Change into our 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 <https://arcane-fjord-43759.herokuapp.com/>.



To connect our web service with this application, we need to change the file *config.properties* and add the name to the parameter *serviceUrl*.

The source code can be now deployed using the command:

git push heroku master

### 4.2.5 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

Using Heroku’s psql command, the following commands are executed to enable the PostGis support in the PostgreSQL database.

*heroku pg:psql*

*CREATE EXTENSION postgis;*

*CREATE EXTENSION hstore;*

*CREATE EXTENSION postgis\_topology*

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

*PGUSER=postgres heroku pg:push touristdb DATABASE\_URL*

## 4.2.2 Integrated Development Environment

The project is developed using Eclipse Neon as an IDE. The 64-bit installer can be downloaded from the [**Eclipse**](https://www.eclipse.org/downloads) 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 folder source.

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.

# 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="https://safe-ridge-54902.herokuapp.com/"

TELEGRAM\_TOKEN=”<insert telegram token>”

API\_AI\_ACCESS\_TOKEN="<insert api.ai client access token>”"

DATABASE\_URL="postgres://touristuser:<password>@localhost:5432/touristdb"

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. For the productive runtime environment, the bash is used to setting the environment variables on Heroku using 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 DATABASE\_URL as this is an already predefined environment variable.

# Program Compilation, Installation and Execution

The presented application was designed for online usage and deployment in a web service. Therefore, no further compilation, installation or execution steps are needed as this is managed by the PaaS Heroku. Newly made changes to the program are published by using the Git workflow, meaning to commit changes and then push them to Heroku using the command:

git push heroku master

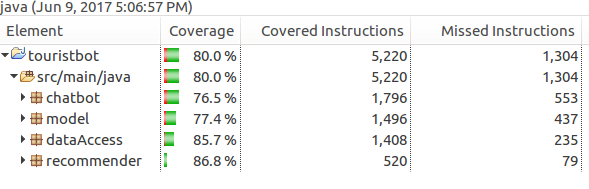
# Tests

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. 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. These tests are the majority of the written tests.
* 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 class TouristChatbotTest.

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

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).

Recommendation Testing

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 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 used as well as the f-measure which is a weighted average of the previously mentioned. (see rijsbergen79)

*Precision: the fraction of retrieved items that are relevant*

*Recall: the fraction of relevant items that are retrieved*

*F-Measure: 2\* precision\*recall/precision+call*

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. In the following table, the f-measures of the 16 investigated combinations can be seen.

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

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).