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 — June 10, 2017
Tutor: Bruno Baruque Zanón

Contents

List of Figures

List of Tables

Appendix A

Plan de Proyecto Software

A.1 Introducción

A.2 Planificación temporal

A.3 Estudio de viabilidad

Viabilidad económica

Viabilidad legal

Appendix B

Especificación de Requisitos

B.1 Introducción

B.2 Objetivos generales

B.3 Catalogo de requisitos

B.4 Especificación de requisitos

Appendix C

Especificación de diseño

C.1 Introducción

C.2 Diseño de datos

C.3 Diseño procedimental

C.4 Diseño arquitectónico

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
```

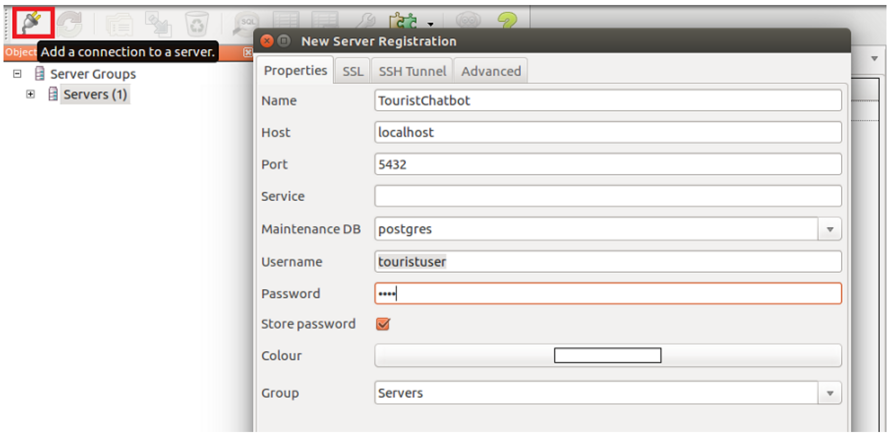
Now we have set up the database, the PostGIS extension is installed and added to prepare the database for geospatial data.

```
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.



Name	TouristChatbot
Host	localhost
Port	5432
Service	
Maintenance DB	postgres
Username	touristuser
Password	••••
Store password	✓
Colour	
Group	Servers

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:

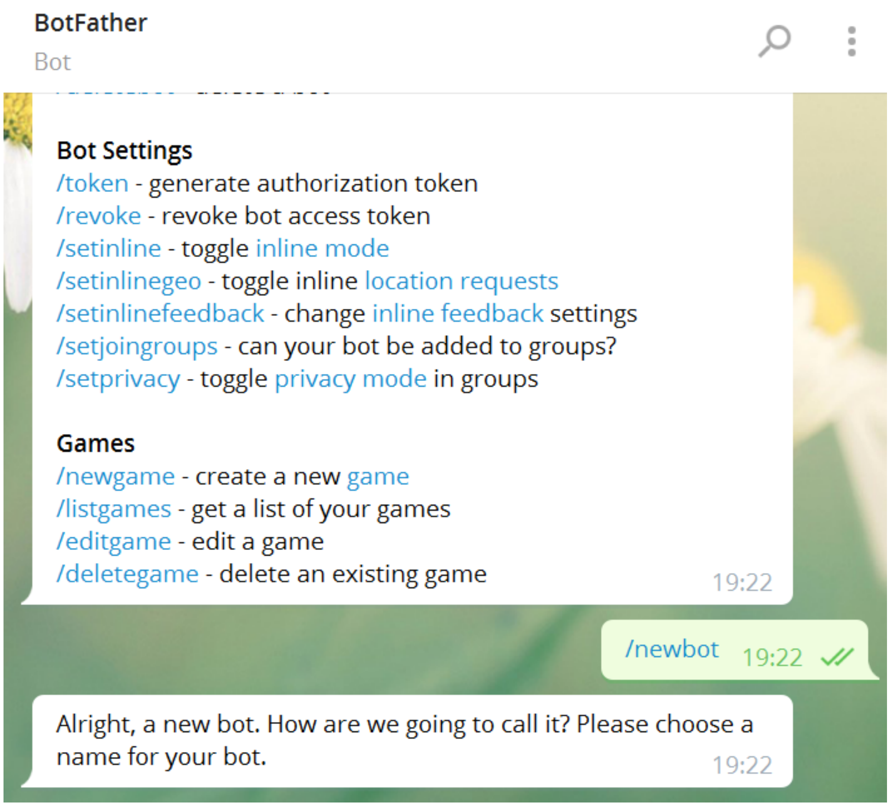
```
create table ratings(
userId bigint,
pointId bigint,
ratings integer,
PRIMARY KEY(userId, pointId))
```

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.


BotFather
Bot

Bot Settings
/token - generate authorization token
/revoke - revoke bot access token
/setinline - toggle inline mode
/setinlinegeo - toggle inline location requests
/setinlinefeedback - change inline feedback settings
/setjoingroups - can your bot be added to groups?
/setprivacy - toggle privacy mode in groups

Games
/newgame - create a new game
/listgames - get a list of your games
/editgame - edit a game
/deletegame - delete an existing game
19:22

/newbot 19:22

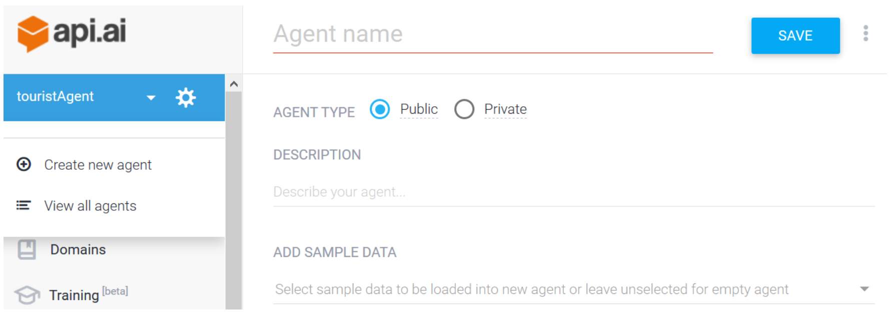
Alright, a new bot. How are we going to call it? Please choose a name for your bot.
19:22

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.



Agent name

AGENT TYPE [✓ Public] Private

DESCRIPTION

Describe your agent...

ADD SAMPLE DATA

Select sample data to be loaded into new agent or leave unselected for empty agent

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
```

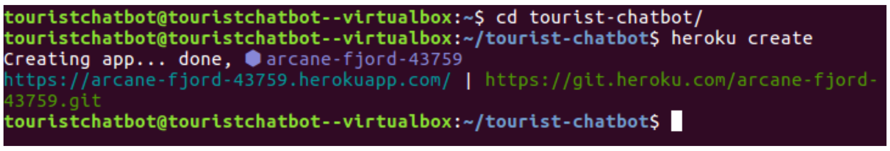
After installing the command line, execute the following command and enter the Heroku credentials when asked.

```
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.



```
touristchatbot@touristchatbot--virtualbox:~$ cd tourist-chatbot/
touristchatbot@touristchatbot--virtualbox:~/tourist-chatbot$ heroku create
Creating app... done, ⬢ arcane-fjord-43759
https://arcane-fjord-43759.herokuapp.com/ | https://git.heroku.com/arcane-fjord-
43759.git
touristchatbot@touristchatbot--virtualbox:~/tourist-chatbot$ 
```

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.

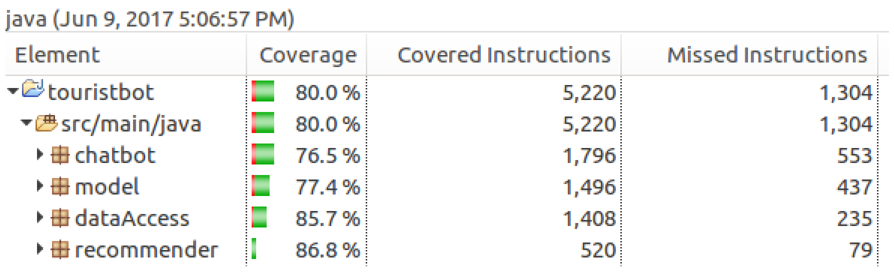
Automation Testing using JUnit

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 previously mentioned [1]

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

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

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

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

Documentación de usuario

E.1 Introducción

E.2 Requisitos de usuarios

E.3 Instalación

E.4 Manual del usuario

Bibliography

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