QA Plan – Twitteranalyser

Contents

[Overview of the document 1](#_Toc390695319)

[Requirements 1](#_Toc390695320)

[1: Event management 1](#_Toc390695321)

[2: Analysis 2](#_Toc390695322)

[3: Tweet-filtering 2](#_Toc390695323)

[4: Export 2](#_Toc390695324)

[5: Further Requirements 2](#_Toc390695325)

[Test Cases 2](#_Toc390695326)

[Test Results 3](#_Toc390695327)

[Defect Table 3](#_Toc390695328)

[Metrics 3](#_Toc390695329)

# Overview of the document

This document describes the quality assurance plan applied to the development of the application ‘Twitteranalyser’. The document consists of the following sections:

* Requirements: The identified requirements which define the basis of the test cases
* Test cases: The test cases defined to evaluate whether the requirements are implemented with a certain level of quality
* Test results: The result tables of the manual test runs
* Defect table: The table of identified defects, and their status on release

# Requirements

The requirements for the application ‘Twitteranalyser’ are split into four main points that have to be fulfilled in order to achieve the full desired functionality.

## 1: Event management

1: Events can be created and saved.

2: Existing events can be edited and the changes can be saved.

3: Existing events can be viewed in a table.

4: Tweets can be fetched for a configured event, according to the properties defined in the event (tweet-tags and date frame).

## 2: Analysis

1: A positive or negative sentiment can be assigned to each tweet for which the analysis is invoked.

2: A sentiment analysis can be invoked for every filter of an event (which filters the tweets of this event).

3: A pre-configured list of sentiment words and their weights can be used for the sentiment analysis.

4: Own sentiment words and their weights can be used for the sentiment analysis.

5: Analysis results can be displayed in a graphical representation.

## 3: Tweet-filtering

1: Fetched tweets can be viewed for each configured event.

2: The list of fetched tweets can be filtered by applying filter objects.

3: Filter objects for the tweets can be saved, loaded and edited.

4: Single tweets of an event can manually be set to ‘ignore’ status for a filter object, so that they are ignored in upcoming sentiment analysis’.

5: Single tweets of an event can manually be deleted so that they are not analyzed for any filter object.

## 4: Export

1: Tweets and their assigned sentiment value can be exported in a .csv format.

## 5: Further Requirements

Beyond these basic requirements, additional requirements global for the implementation of the application were identified.

1: Data processes by the application shall be stored in a persistent way. Data from previous sessions can be retrieved and used in subsequent ones.

2: The application shall be realized as a web application, and shall support the version of Mozilla Firefox which is current at release.

3: The user interface shall be supported in multiple languages.. Supported languages shall be German and English.

# Test Cases

The test cases for manual test runs were identified based on the requirements and features implemented. The full table of test cases is not printed for space reasons, but is included in the CD version of the application.

# Manual Test Results

The test cases were manually executed three full times, over the course of the development of the application. On each manual test run, new features were included in the tests, as well as a regression test for already working features. The encountered anomalies were collected and written in a defect table file. Again, because of space reasons, the full results of the test runs are not included in printed form, but can be found in the CD version of the application.

# Automated Test Cases

For automatic testing of the application, different approaches were investigated. Regarding the execution of automatic regression results on the user interface, the free web tool Selenium was examined. It was shown that the architecture of the application in some cases hindered the automatic test case execution of Selenium, therefore the tool was dropped. At unit test level, the respective frameworks of PHPUnit and JUnit were used to develop automatic unit tests.

# Defect Table

The anomalies of the test case executions (as well as problems discovered through exemplary use of team members) were collected in a defect table document in order to keep track of the defects that have to be addressed to enhance the quality of the application.

The following workflow was used to work on entries of the defect table.

* When a problem was encountered, it was added to the defect table along with estimated severity, date of occurrence a problem description and, if found through a manual test run, the text link to the test report where further information can be obtained
* When changes were made to the application, the problem description of enlisted defects could also be updated
* Open defects were assigned to team members when they did fit the sprint. Their status was then changed to ‘assigned’.
* After working on the defect and committing a solution, the status of the defect was changed to ‘resolved’, indicating that the problem solution has to be verified.
* After verification, the defect could either be re-opened (when the problem still persists), or closed.
* Also, if the same problem was found later again, the defect entry could be re-opened.

The full defect table is not included in printed form, but can be found in the CD version of the application.

# Metrics

To keep track of the development process with more formal measures, the SonarQube metrics server was used. This tool was chosen because it supports Java as well as JavaScript and PHP and can therefore be applied to all parts of the developed application. Some important key parameters along with the analysis results are summarized as following.

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Javascript | PHP | Java |
| Lines | 745 | 5022 | 1131 |
| Lines of code | 547 | 2304 | 832 |
| Comments % | 11,1% | 18,5% | 13,8% |
| Duplicated lines % | 21,6% | 15,1% | 0,0% |
| Complexity / file | 61,0 | 13,0 | 10,9 |
| Public documented API |  |  | 70,3% |

## PHP

* 2304 lines of code
* 43 analyzed source files, with an average complexity of 13,0
* 247 functions, with an average complexity of 2,1

## JavaScript

Some JavaScript files could not be analyzed with SonarQube.

* 574 lines of code
* 2 analyzed source files, with an average complexity of 61,0
* 72 functions, with an average complexity of 1,7

## Java

* 832 lines of code
* 9 analyzed source files, with an average complexity of 10,9
* 35 functions, with an average complexity of 2,8