RENDER
FP7-ICT-2009-5
Contract no.: 257790
www.render-project.eu

RENDER

RENDER Technical Architecture

Detailed Component Description

Editor:	Delia Rusu, IJS
Author(s):	RENDER Consortium
Deliverable Nature:	Report (R)
Dissemination Level: (Confidentiality)	Public (PU)
Actual Delivery Date:	March 2013
Suggested Readers:	RENDER Consortium
Version:	3.0
Keywords:	Technical architecture, software components, interfaces, UML use case diagrams, UML component diagrams

Disclaimer

Full Project Title:	RENDER – Reflecting Knowledge Diversity
Short Project Title:	RENDER
Document Title:	RENDER Technical Architecture
Editor (Name, Affiliation)	Delia Rusu, IJS

Executive Summary

This document complements the RENDER technical architecture documentation by focusing on a detailed component description. We start with an overview of the general three-tier architecture, followed by an in-depth description of each of the components.

Each component is described from the user point of view, via UML use case diagrams, as well as from the software implementation point of view, via UML component diagrams.

We distinguish between a) **the data layer components**, which include the RDF repositories OWLIM and FactForge; b) **the application layer components**: the Fact and Opinion Mining toolkits included in Enrycher – the Diversity Mining Services, the Diversity-aware Ranking toolkit and Newsfeed, a Web Crawler; and c) **the presentation layer components**: the use case applications and other supporting tools.

Table of Contents

List of Figures

List of Tables

Abbreviations

ASQM	Article Statistics and Quality Monitor
TLG	Task List Generator
KDO	Knowledge Diversity Ontology
RDF	Resource Description Framework
RKS	Reference Knowledge Stack
DMOZ	Open Directory Project
UMBEL	Upper Mapping and Binding Exchange Layer
SIOC	Semantically-Interlinked Online Communities
SKOS	Simple Knowledge Organization System
FOAF	Friend of a Friend
RSS	Really Simple Syndication
REST	REpresentational State Transfer
SPARQL	SPARQL Protocol and RDF Query Language
UML	Unified Modelling Language

Definitions

Three-tier architecture	a software architecture in which presentation, application processing and data management functions are logically separated.
RESTful web service	(also called a RESTful web API) is a web service implemented using HTTP and the principles of REST.
OWLIM	is a family of semantic repositories, or RDF database management systems.
FactForge	aims to allow users to find resources and facts based on the semantics of the data.
Diversity Mining Services (Enrycher)	comprise the fact and opinion mining toolkits developed in WP2.

1 Introduction

In this version of the RENDER technical architecture document we focus on an in-depth description of the RENDER software components. We describe the software components from two different points of view: that of the **user** interacting with the software components, and that of the **software developer** who would want to re-implement or extend the existing RENDER software components.

If the previous version of the technical architecture document was presenting a general overview of the RENDER software components, in this document we aim at providing a more detailed description of the components, from the software implementation point of view.

The method of choice for describing the software components is via UML diagrams, which represent a standardized general-purpose modelling language used in software engineering.

As in the previous version of the document, we start with the RENDER three-tier technical architecture. The RENDER components are organized in a classical layered architecture, which includes a data layer, application layer and finally, a presentation layer. In the previous version of this document we motivated our choice for this type of architecture. Here we remind the reader the main components of the RENDER architecture, which are going to be described in the remainder of this document:

The **data layer components** include:

- the Technology and Data Infrastructure: FactForge and the OWLIM nested RDF repositories

The data layer also contains use-case specific datasets and the KDO – the Knowledge Diversity Ontology. These have been described in-depth in the previous version of this document.

The **application layer components** include:

- the Fact and Opinion Mining toolkits, with their functionality exposed through the Diversity Mining Services (Enrycher)
- the Diversity-aware Ranking toolkit
- Newsfeed, a Web Crawler for the acquisition of multilingual news articles and blogs

The **presentation layer components** include:

- the use case tools: Google, Wikipedia and Telefonica
- other supporting tools: Drupal extension, TwiDiViz – a web-based application that enables the analysis and visualization of diversity in Twitter data and the Interactive Modelling Tool supporting the Telefonica case study.

In what follows, we start by presenting the three-tier architecture overview, and continue with describing each component separately.

2 RENDER General Component Architecture

The RENDER three-tier general architecture showing the RENDER software components, is depicted in Figure 1. We make a logical distinction between the data layer components, the application layer components and the presentation layer components.

In the next sections of this document we are going to describe each of these components, using the UML use case diagrams and component diagrams.

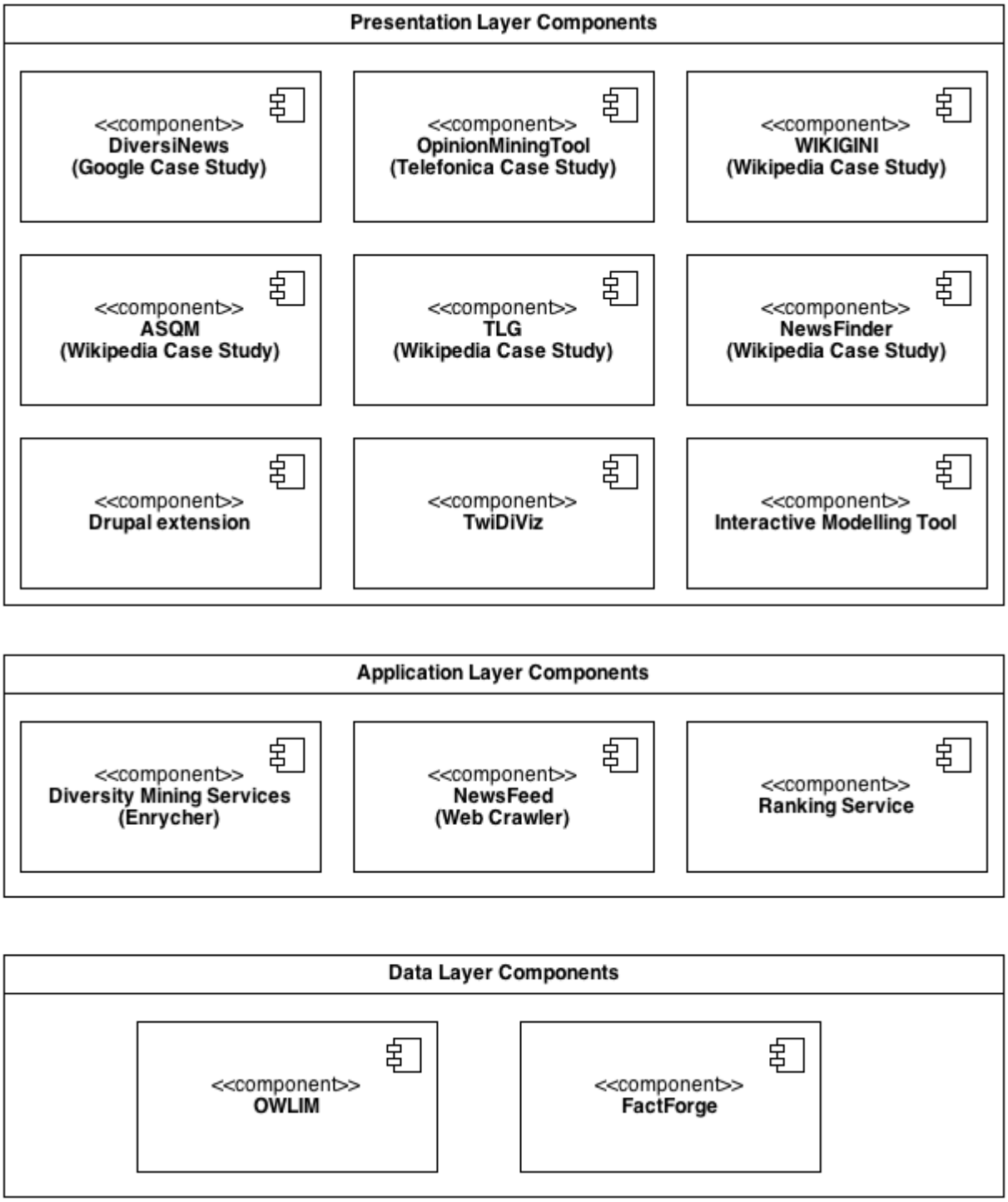

Figure 1. The RENDER three-tier general architecture, showing the software components.

3 Data Layer Software Components

Render data infrastructure, hosted in OWLIM semantic repository, is intended to support the different use case scenarios of the project. It consists of a data layer (basic – a reason-able view with a segment from LOD, secondary – nested repositories with processed additional data, e.g. news and tweets), and access APIs.

Render Data Infrastructure in Telefonica Use Case

Telefonica use case makes use of RENDER data infrastructure to retrieve information based on specified in the user interface parameters. These parameters fill specific spots in a predefined SPARQL query which is run through the SPARQL API. This use case uses the secondary data layer with twitter data, in a nested repository on top of the basic data layer, accessible at rendertweets.ontotext.com.

Render Data Infrastructure in Google Use Case

Google use case makes use of RENDER data infrastructure to retrieve related entities based on topics identified by the Google news processing. The identified topics by the document processor are being supplied to a component that launches a predefined SPARQL query that dynamically includes them. The query is run through the SPARQL API. This use case uses the basic data layer, the reason-able view of a segment of LOD, accessible at render.ontotext.com.

Render Data Infrastructure in Drupal Use Case/Drupal Extensions

Drupal extensions make use of RENDER data infrastructure to retrieve data for importing relevant information via the SPARQL API from rendernews.ontotext.com. This is done by giving the end user the opportunity of defining the credentials and restrictions on what articles should be imported to the Drupal system.

TwiDiViz is used to visualize data retrieved from the RENDER data infrastructure. The Tweets are retrieved by predefined SPARQL queries from rendertweets.ontotext.com and a subset of the data in the RENDER data infrastructure is selected by defining a valid time period. The used dataset is available at rendertweets.ontotext.com.

3.1 Use Case Diagrams

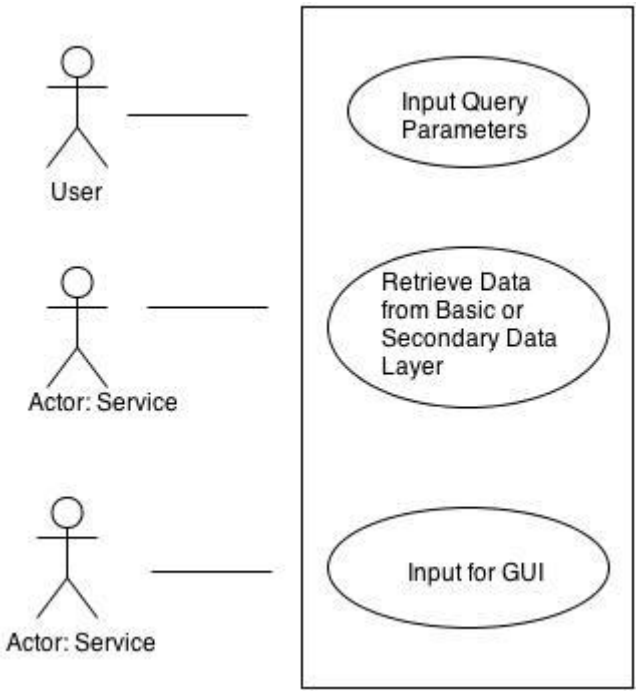

Figure 2. User-based use case diagram.

Table 1. The "RENDER Data Layer in Applications" use case scenario.

Use case name	**RENDER Data Layer in Applications**
Actor	RESTFul Service
Preconditions	RENDER Data Layer set up
Main scenario	1. Gets user inserted search parameters 2. Executes search with adjusted SPARQL query 3. Returns results in a format to be input for the GUI
Alternative scenario	3. Returns results to be further processed before becoming input for the GUI

Table 2. The "Use of RENDER Data Layer by Humans" use case scenario.

Use case name	**Use of RENDER Data Layer by Humans**
Actor	User
Preconditions	RENDER Data Layer set up
Main scenario	1. Formulates SPARQL query 2. Gets results
Alternative scenario	1. Formulate keyword term query 2. Gets results 3. Navigates through the results
Alternative scenario	1. Inserts two URIs in RelFinder 2. Gets results 3. Navigates through the results

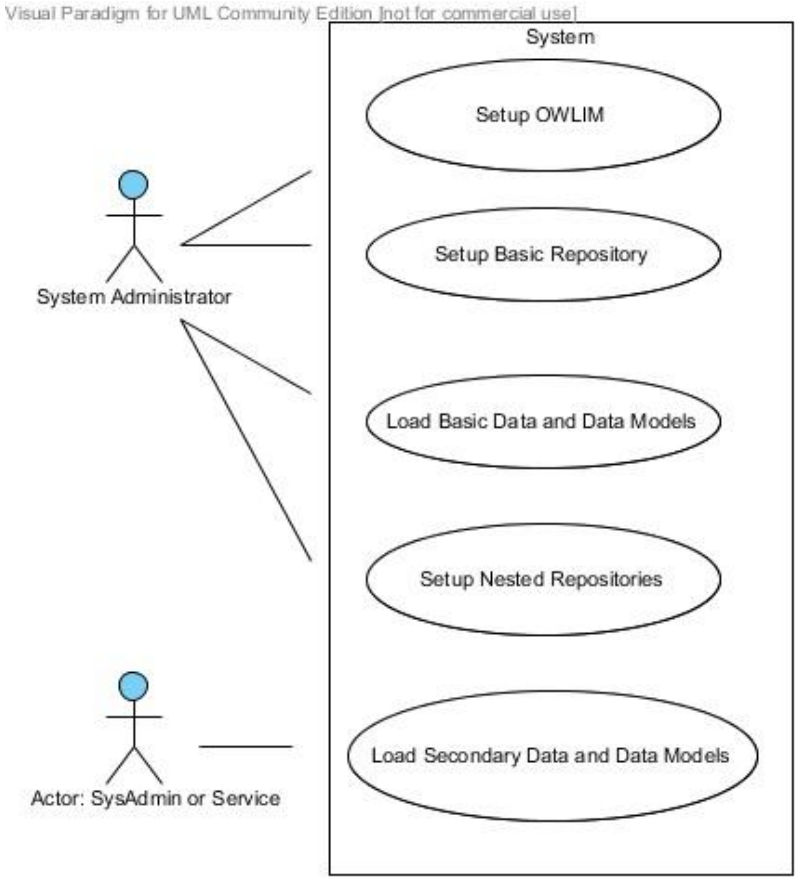

Figure 3. Data infrastructure setup use case diagram.

Table 3. The "Basic data layer setup" use case scenario.

Use case name	**Basic Data Layer Setup**
Actor	System Administrator
Preconditions	Hardware; OWLIM-SE license
Main scenario	1. System Administrator installs OWLIM 2. System Administrator creates a repository with OWL-Horst rule set, and additional rules for the Reference Knowledge Stack 3. System Administrator sets up Forest interface 4. System Administrator loads the data models, the Reference Knowledge Stack 5. System Administrator loads the datasets of the basic data layer 6. System Administrator places exemplary SPARQL queries
Alternative scenario	4. RESTFul service loads the data models, the Reference Knowledge Stack 5. RESTFul service loads the datasets of the basic data layer
Alternative scenario	4. System Administrator loads some models through the Forest Interface 5. System Administrator loads some data through the Forest Interface

Table 4. The "Secondary data layer setup" use case scenario.

Use case name	**Secondary Data Layer Setup**
Actor	System Administrator
Preconditions	Basic data layer set up
Main scenario	1. System Administrator sets up nested repositories, one per each specific dataset 2. System Administrator sets up Forest interface 3. System Administrator loads the data models for the secondary data layer, per dataset and per repository 4. System Administrator loads data in the secondary data layer, each dataset in the corresponding nested repository 5. System Administrator places exemplary SPARQL queries
Alternative scenario	3. RESTFul service loads the data models for the secondary data layer, per dataset and per repository 4. RESTFul service loads data in the secondary data layer, each dataset in the corresponding nested repository
Alternative scenario	3. System Administrator loads some models through the Forest Interface 4. System Administrator loads some data through the Forest Interface

3.2 Component Diagrams

The RENDER Data Layer components are presented in Figure 4. In what follows we detail the main software components of the Data Layer.

RENDER Data Infrastructure – Basic Data Layer

A segment of LOD is loaded as a reason-able view in OWLIM-SE using OWL-Horst inference rule set. The LOD segment comprises: DBpedia, Freebase, Geonames, MusicBrainz, NYTimes, CIA Factbook, Wordnet, Linvoj, Lexvo. The reason-able view is supplied by a Reference Knowledge Stack which contains mappings of DBpedia, Freebase and Geonames ontologies to the Upper Level ontology PROTON, and makes all data accessible through it.

RENDER Data Infrastructure – Nested Repositories

Nested repositories are an experimental OWLIM feature which allows for data sharing across repositories. It is set on top of the repository containing RENDER basic data layer, one nested repository per secondary dataset, e.g. one for the Tweeter data, one for the Google news data.

Render Data Infrastructure – Secondary Data Layer

Processed by Enrycher Tweeter data and Google news into RDF, based on the RENDER data models, which include KDO, SIOC, DC, PROTON and some RENDER specific properties and entities, are loaded in the corresponding nested repositories, which produces inferred statements as a result of the interaction between the data in the basic data layer and the secondary data loaded.

Render Data Infrastructure – Access APIs

RENDER Data Infrastructure is accessible programmatically via standard SPARQL end point RESTFul service for querying, loading and deleting, the later requiring security credentials. A human user friendly Forest interface gives the possibility to query the data with keywords, with SPARQL 1.1 queries, and relation discovery between two given entities. Three access points are provided, one for RENDER basic data layer, render.ontotext.com, and two for the two nested repositories of the RENDER secondary data layer, e.g. rendernews.ontotext.com and rendertweets.ontotext.com.

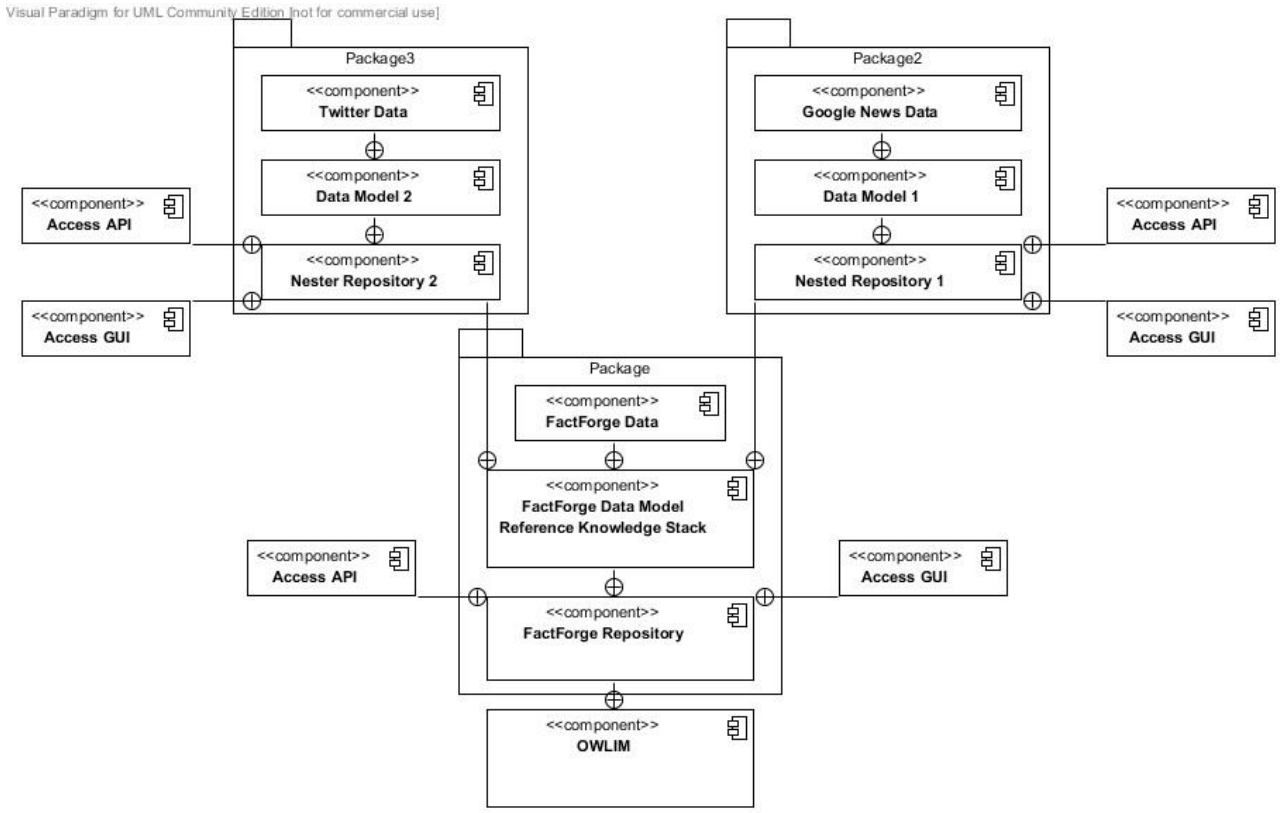

Figure 4. RENDER Data Layer Components.

4 Application Layer Software Components

The Application Layer software components comprise the Diversity Mining Services (Enrycher), The news crawler (Newsfeed) and the Ranking service. The following sub-sections further detail these components.

4.1 Diversity Mining Services (Enrycher)

The Diversity Mining Services (Enrycher) expose the functionality of the Fact and Opinion Mining toolkits via a RESTful API. Enrycher is a service-oriented system, providing shallow as well as deep text processing functionality at the text document level. The shallow text processing functionality includes topic and keyword detection and named entity extraction while the deep text processing functionality provides named entity resolution with respect to existing Linked datasets, named entity merging, word sense disambiguation into WordNet and assertion extraction, by identifying subject – predicate – object sentence elements together with their modifiers (adjectives, adverbs) and negations.

Enrycher comprises a set of components which can be combined in different ways, forming a customizable processing pipeline; each component is implemented as a service (in Java or C++) which can be called independently.

The Web interface of Enrycher is designed as a demo, showing some of the service functionality. Given a document as input, the user can obtain, via the demo interface, an RDF and XML representation of the enriched document, as well as a visual representation of the document via the semantic graph.

4.1.1 Use Case Diagram

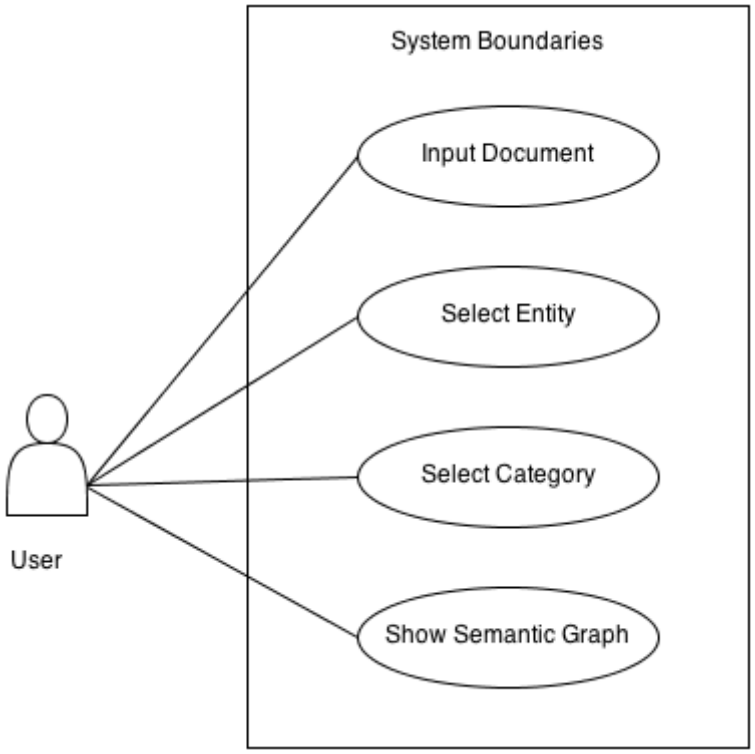

Figure 5. Enrycher (Web UI) use case diagram.

Table 5. The "Input document" use case scenario.

Use case name	**Input document**
Actors	User.
Preconditions	-

Postconditions	User visualizes extracted annotations.
Main scenario	1. User opens Enrycher web front-end in a web browser. 2. User inputs a document (plain-text) 3. User selects “Enrich”, “XML” or “RDF” output formats. 4. The system sends this information to the back-end server. 5. The back-end returns results.

Table 6. The “Select entity” use case scenario.

Use case name	**Select entity**
Actors	User.
Preconditions	User gave a document as input to the system.
Postconditions	User visualizes additional information about the selected entity.
Main scenario	1. User selects one of the entities from the list, for which more information should be provided. 2. The system sends this information to the back-end server. 3. The back-end returns results.
Alternative scenario	1. No entities are retrieved by the system.

Table 7. The “Select category” use case scenario.

Use case name	**Select category**
Actors	User.
Preconditions	User gave a document as input to the system.
Postconditions	User visualizes additional information about the selected category.
Main scenario	1. User selects one of the categories from the list, for which more information should be provided. 2. The system sends this information to the back-end server. 3. The back-end returns results.
Alternative scenario	1. No categories are retrieved by the system.

Table 8. The “Show semantic graph” use case scenario.

Use case name	**Show semantic graph**
Actors	User.
Preconditions	User gave a document as input to the system.
Postconditions	User visualizes the document semantic graph.
Main scenario	1. User clicks on “Show semantic graph”. 2. The system sends this information to the back-end server. 3. The back-end returns the semantic graph for the document.
Alternative scenario	3. The semantic graph cannot be retrieved, in which case the system shows an error message.

4.1.2 Component Diagram

All Enrycher components are using the Enrycher object model, as defined in D2.2.1 [5]. The Enrycher object model defines the main elements of an Enrycher document. Each component manipulates the Enrycher document: it receives an input version of the Enrycher document which it extends (or enriches) and provides an enriched document as output.

Text pre-processing consists of detecting sentence boundaries, splitting sentences into tokens, and assigning a part of speech tag to each token. The output of this component is a pre-processed version of the document, where each sentence has associated its tokens and their part of speech.

Entity extraction is performed via the OpenNLP[1] natural processing toolkit. As named entities we consider names of people, locations and organizations, dates, percentages and money amounts occurring in the text. Entities are provided as annotations to the document.

The **Entity Merging** component has two sub-components: co-reference and anaphora resolution one. Co-reference resolution s used to merge named entities with different representations. Anaphora resolution t identifies corresponding entities for a set of predefined pronouns.

The purpose of the **Entity Resolution** component provides integration with existing knowledge bases with the goal of using existing knowledge to enrich the set of features that we are able to extract from text.

The **Disambiguation** component identifies for each word/n-gram in text a corresponding WordNet (VUA) resource, if available.

The **Categorization** component provides DMOZ categories for the input document.

The **Article Template Discovery** component operates on a collection of documents received as input, and extracts common templates.

The **Topic Detection** and **Sentiment Analysis** components rely on the topic and sentiment models provided by the Interactive Modelling Tool. These components provide topic and sentiment annotations for the input document.

The **Enrycher client(s)** interact with the Diversity Mining Services via HTTP calls.

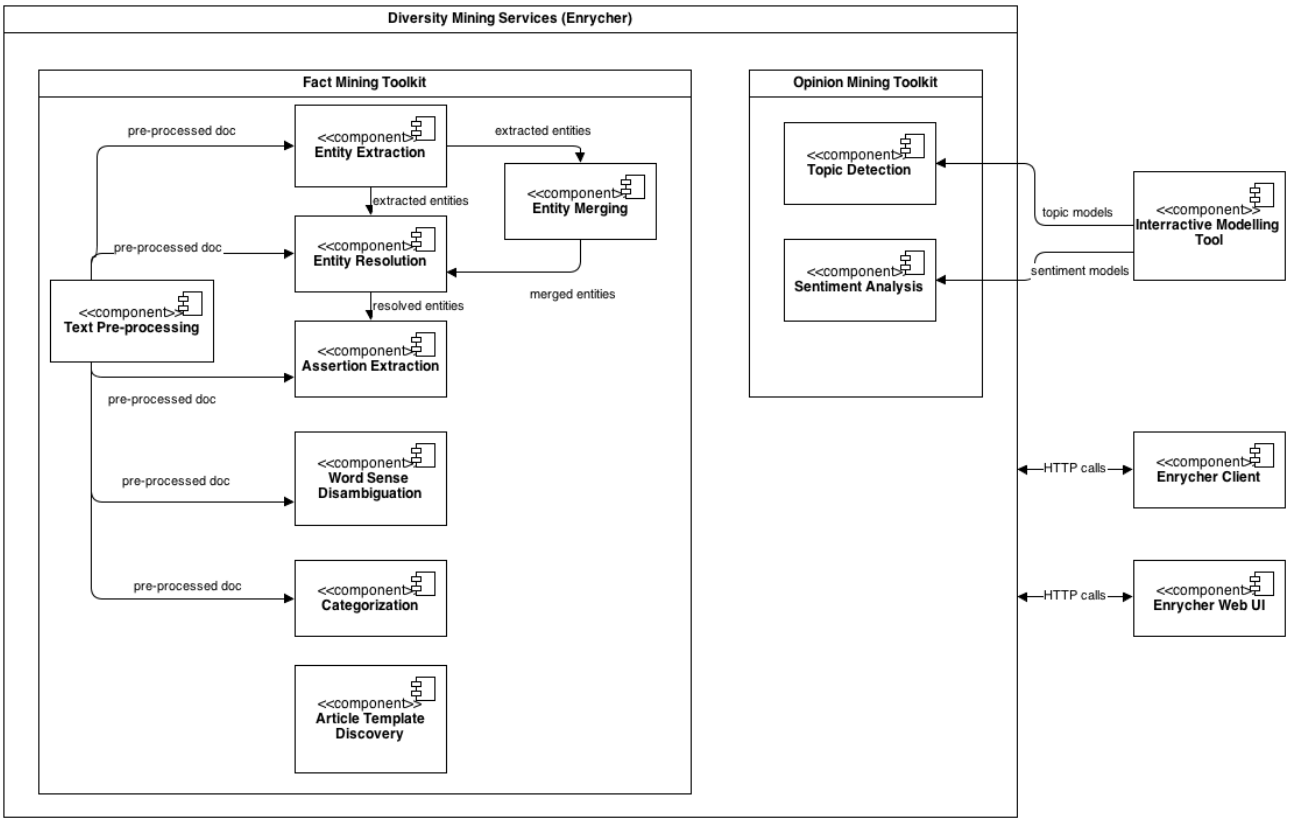

Figure 6. Diversity Mining Services component diagram.

[1] http://incubator.apache.org/opennlp/, last accessed on 30.03.2013.

4.1.3 Enrycher Performance

We analysed the performance of the Diversity Mining Services (Enrycher) in two settings:

- For a subset of the Twitter data collection provided by Telefonica, which contains 22 GB of uncompressed data, collected from April – August 2012
- For a subset of news articles from IJS Newsfeed, which includes 319 K news articles

The data processing step was parallelized over 2 machines with a total of 36 cores.

The performance results are summarized in Table 9, showing a comparative view of the two datasets.

Table 9. Enrycher performance on social media and news data.

	Social media data	**News data**
Enrycher Pipeline components	○ Entity extraction ○ Entity linking and classification to Linked Data Datasets ○ Sentiment analysis ○ Reverse geo-coding to Linked Data ○ RDF export	○ Entity extraction ○ Entity linking and classification to Linked Data Datasets ○ Sentiment analysis ○ RDF Export
Throughput	approx. 80 tweets per second (7M per day) Bottleneck on reverse geo-coding Dataset has approx. 1M tweets per day	approx. 10 news articles per second (approx. 1M per day) Current bottleneck on linking and classification Dataset throughput is approx. 150k per day

Figure 7 presents the performance (in articles per hour) of Enrycher over a time window of a week. The performance results show that the Diversity Mining Services (Enrycher) can process data streams (news and social media data). We also identify the bottlenecks which can further improve its performance.

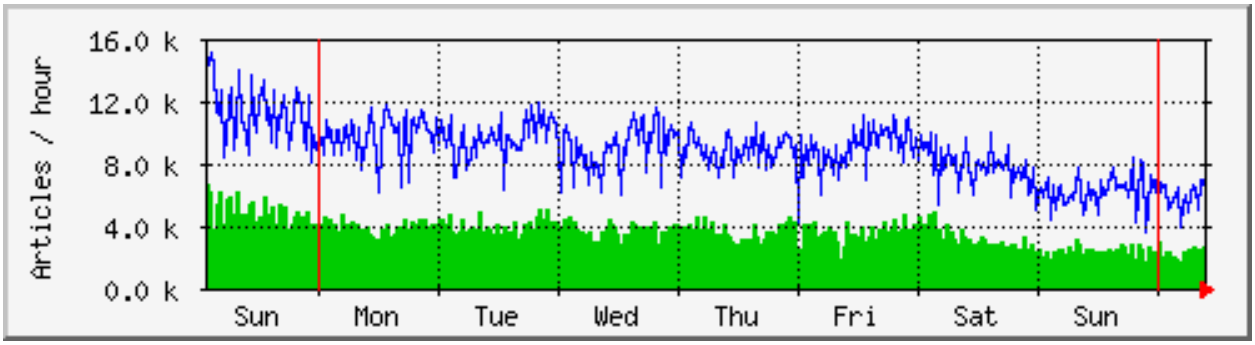

Figure 7. News articles processed by Enrycher per hour, during a time window of a week.

4.2 Newsfeed (Web Crawler)

Newsfeed offers a clean, continuous, real-time aggregated stream of semantically enriched news articles from RSS-enabled sites across the world.

The tool has associated a visual demo, whose use case diagram is presented in what follows.

4.2.1 Use Case Diagram

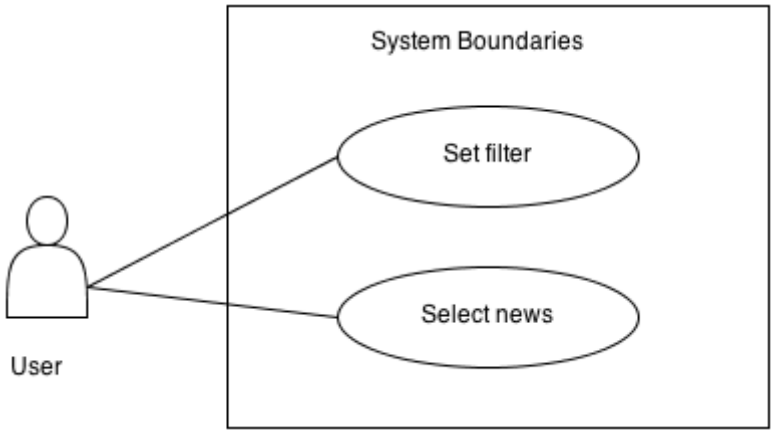

Figure 8. The Newsfeed use case diagram.

Table 10. The "Set filter" use case scenario.

Use case name	**Set filter**
Actors	User.
Preconditions	-
Postconditions	User visualizes Newsfeed results, filtered according to the specified keywords.
Main scenario	1. User opens Newsfeed web front-end in a web browser. 2. User sets a filter for news (by specifying keywords) in the main page. 3. The system sends this information to the back-end server. 4. The back-end returns results.

Table 11. The "Select news" use case scenario.

Use case name	**Select news**
Actors	User.
Preconditions	-
Postconditions	User visualizes Newsfeed results, for the selected news items.
Main scenario	1. User opens Newsfeed web front-end in a web browser. 2. User selects news items from the real-time list of news. 3. The system sends this information to the back-end server. 4. The back-end returns results for the selected news items – an image associated to the news item, and first paragraphs of the news.

4.2.2 Component Diagram

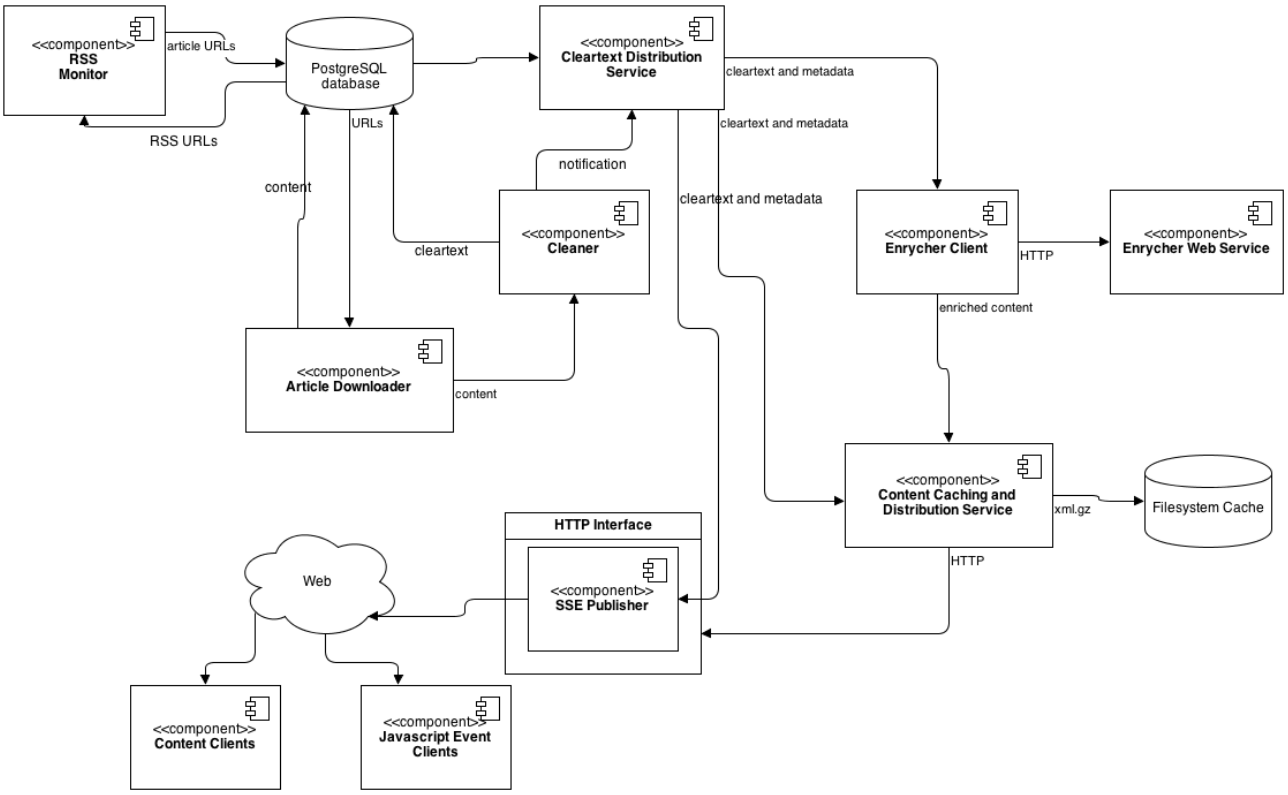

Figure 9. The Newsfeed component diagram.

The **PostgreSQL** database is running on a Linux server. The database contains a list of RSS feeds, which are periodically downloaded by the RSS monitoring component (**RSS Monitor**). RSS feeds contain a list of news article URLs and some associated metadata, such as tags, publication date, etc.

A separate component (**Article Downloader**) periodically retrieves the list of new articles and fetches them from the web. The complete HTML is stored in the database, and simultaneously sent to a set of cleaning processes (**Cleaner**) over a *0mq* message queue.

The cleaning process converts the HTML into UTF8 encoding and updates the database with it. The cleaned version of the text is stored back in the database, and sent over a message queue to consumers.

English language documents are sent to the **Enrycher Web Service**, where named entities are extracted and resolved, and the entire document is categorized into a DMOZ topic hierarchy.

Both the clear text and the enriched versions of documents are fed to a file system cache (**Content Caching and Distribution Service**), which stores a sequence of compressed xml files. The caching service exposes an **HTTP interface** (containing **SSE Publisher**) to the world through an **Apache transparent proxy**, serving those compressed xml files on user request. The Apache server also hosts a CGI process capable of generating HTML5 server side events, which contains the article metadata and clear text as payload. These events can be consumed using Javascript's EventSource object in a web browser.

4.3 Ranking Service

The ranking component provides diversity-aware ranking with respect to the two dimensions "topic" and "sentiment". The output is formed of document clusters each with a denoted representative.

4.3.1 Use Case Diagram

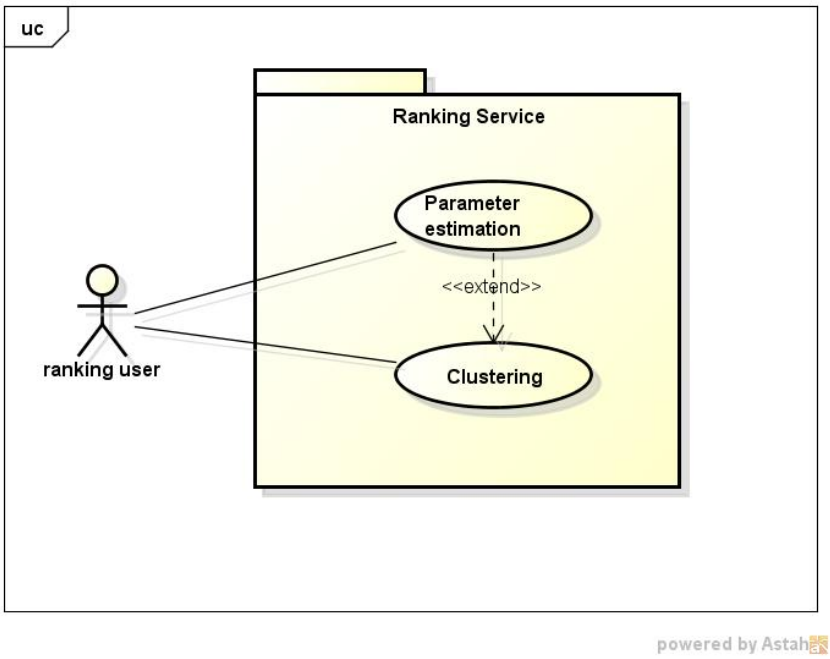

Figure 10. The Ranking service use case diagram.

Table 12. The "Clustering" use case scenario.

Use case name	**Clustering**
Actors	Ranking User.
Preconditions	Parameter estimation.
Postconditions	Document cluster output
Main scenario	The ranking user receives diversity-aware ranked data with respect to the two dimensions "topic" and "sentiment".

4.3.2 Component Diagram

The folding and maximum algorithms are both cluster algorithms that were both introduced in [1]. The service is not RESTful as per definition but the interaction is via HTTP and URL parameters. The only HTTP function in use is GET. The service has four main parameters, which are: endpoint, restrictions, orderBy, rank. The functionality is as follows:

- **[endpoint]** The Sesame/OWLIM SPARQL endpoint where KDO-conform data is contained.
- **[restrictions]** Restrictions can be put on the document/statement ?s.
- **[orderBy]** One can specify additional parameters in accordance to which should be ordered. If this parameter is set, the FOLDING algorithm is applied by default.
- **[rank]** This parameter defines an emphasis on a certain property (either kdo:hasSentiment or sioc:topic).

There exist futher parameters which are detailed in D3.3.1 [2] and D3.3.2. [3].

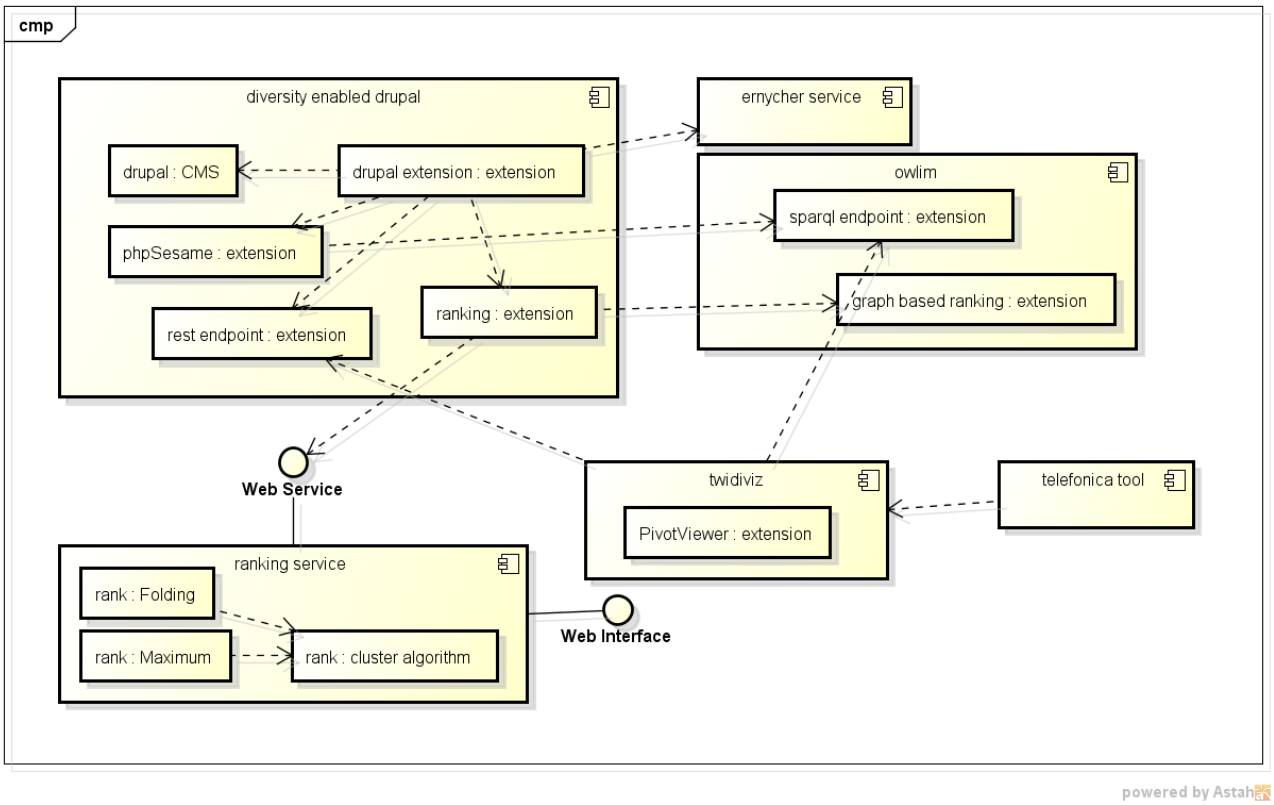

Figure 11. The Ranking Service, Drupal extension and TwiDiViz component diagram.

5 Presentation Layer Software Components

RENDER has a rich Presentation Layer, consisting of the three case study components, as well as supporting tools. The remainder of the section presents each of these components in greater detail.

5.1 Telefonica Case Study Software Components

Telefónica Opinion Mining Tool (OMT) is intended for business and market products analysis, based on users published opinions.

5.1.1 Use Case Diagram

Use case context: A new product has been launched in the market or a new release of any existing product goes into the market. Responsible Business Units need customer's feedback about the acceptance of the new product or release.

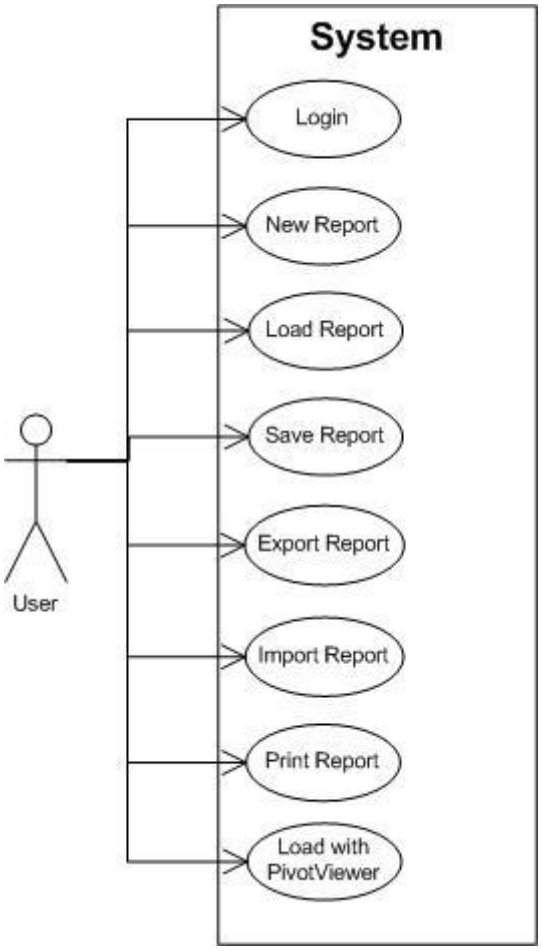

Figure 12. The Telefonica OMT use case diagram.

Table 13. The "Login" use case scenario.

Use case name	**Login.**
Actors	User.
Preconditions	User must have signed in the system.
Postconditions	User has access to the reports generator and to its old saved reports feature.

Main scenario	1. User opens the T-OMT web front-end on a web browser. 2. User tips its 'user name' and 'password' in the login page. 3. The system sends these values to the back-end server. 4. The back-end verifies that the input values match. The access is granted. 5. The user web page is automatically redirected to the tool options page. 6. The server provides to this page the list of saved reports for the logged user.
Alternative scenario	4. The back-end detects that the input values do not match the stored user profile values. 5. The front-end notifies a login error.
Alternative scenario	6. There is an error loading the user saved reports. 7. The front-end notifies there was an error loading its saved reports.

Table 14. The "News report" use case scenario.

Use case name	**New report.**
Actors	User.
Preconditions	The user is logged.
Postconditions	A properly formed report will be provided to the user.
Main scenario	1. The user opens the 'new report' form. 2. The user selects the 'data source' from the list of available sources. 3. The user selects the desired 'topics' from the list. 4. The user selects the 'initial date' to sort out the data to be considered (optional) 5. The user selects the 'final date' to sort out the data to be considered (optional) 6. The user clicks on the 'new report' button invoking the report generation. 7. A new report is generated according to the input filtering options. 8. The web page opens a new internal tab for the report and assigns it a default name. 9. At this new internal tab, the user will find the report charts, the information and options.
Alternative scenario	4. If the user lefts this field empty, the system will use all the data from the beginning to the 'final date' selected. 5. The user selects the 'final date' to sort out the data to be considered (optional) 6. The user clicks on the 'new report' button invoking the report generation. 7. A new report is generated according to the input filtering options. 8. The web page opens a new internal tab for the new report and assigns it a default name. 9. Report charts, information and options will be added to this report tab.
Alternative scenario	5. If user left this field empty, the system will use all the data from 'initial date' selected to the current date. 6. User clicks on 'new report' button invoking the report generation. 7. A new report is generated according to the input filtering options. 8. Web page opened a new internal tab for the new report an assign a default name for it. 9. At this new internal tab, user will find, the report charts, information and options.

Table 15. The "Load report" use case scenario.

Use case name	**Load report.**
Actors	User.
Preconditions	The user must be logged in. The user must have any saved report to load.
Postconditions	The saved report will be shown again to the user without needing further data analysis.
Main scenario	1. The user opens the 'new report' form. 2. The user selects a saved report by its name. 3. The user clicks on the 'load report' button. 4. The report is loaded from the storage. A new internal tab is opened with the name of the saved report. 5. Here the use case follows the same process of the 'new report' use case at step '9'.
Alternative scenario	4. If there were any error downloading the report file, an error message will be shown to the user.

Table 16. The "Import report" use case scenario.

Use case name	**Import report.**
Actors	User.
Preconditions	The user must be logged in. The user must have any report file properly formed.
Postconditions	The report will be shown again to the user without further data analysis.
Main scenario	1. The user opens the 'new report' form. 2. The user clicks on the 'import report' button. 3. A pop-up explorer window asks the user to select the report to be imported from its local file system. 4. Here the use case follows the same process of the 'new report' use case at step '9'.
Alternative scenario	4. If there were any error uploading the report file, an error message will be shown to the user.
Alternative scenario	4. If there were any error in the report file (malformed), an error message will be shown to the user.

Table 17. The "Save report" use case scenario.

Use case name	**Save report.**
Actors	User.
Preconditions	The user must be logged in. The report to be saved must be generated on an internal web tab of the T-OMT.
Postconditions	The report will be storage in a backend.
Main scenario	1. The user opens the 'report options' panel. 2. The user clicks on the 'save report' button. 3. A pop -up explorer window asks the user a name to identify the report. 4. The report will be uploaded to the storage and a confirmation message will appear in the web interface.
Alternative scenario	4. If there were any error uploading the report file, an error message will be shown to the user.

Table 18. The "Export report" use case scenario.

Use case name	**Export report.**
Actors	User.
Preconditions	The user must be logged in. The report to be saved must be generated on an internal web tab of the T-OMT.
Postconditions	The user will download the report file to its local file system.
Main scenario	1. The user opens the 'report options' panel. 2. The user clicks on the 'export report' button. 3. A pop-up explorer window asks the user location and the name for the report file.

Table 19. The "Load report with Pivot Viewer (Twidiviz)" use case scenario.

Use case name	**Load report with Pivot Viewer (Twidiviz)**.
Actors	User.
Preconditions	The user must be logged in. The report must be generated on an internal web tab of the T-OMT.
Postconditions	The Pivot Viewer will be shown to generate the interface with the same inputs for the report.
Main scenario	1. The user opens the 'report options' panel. 2. The user clicks on the 'Pivot Viewer' button. 3. T-OMT interface will be replaced by the PivotViewer's UI. 4. PivotViewer's tool will show its own search using the same input values of the original report.

5.1.2 Component Diagram

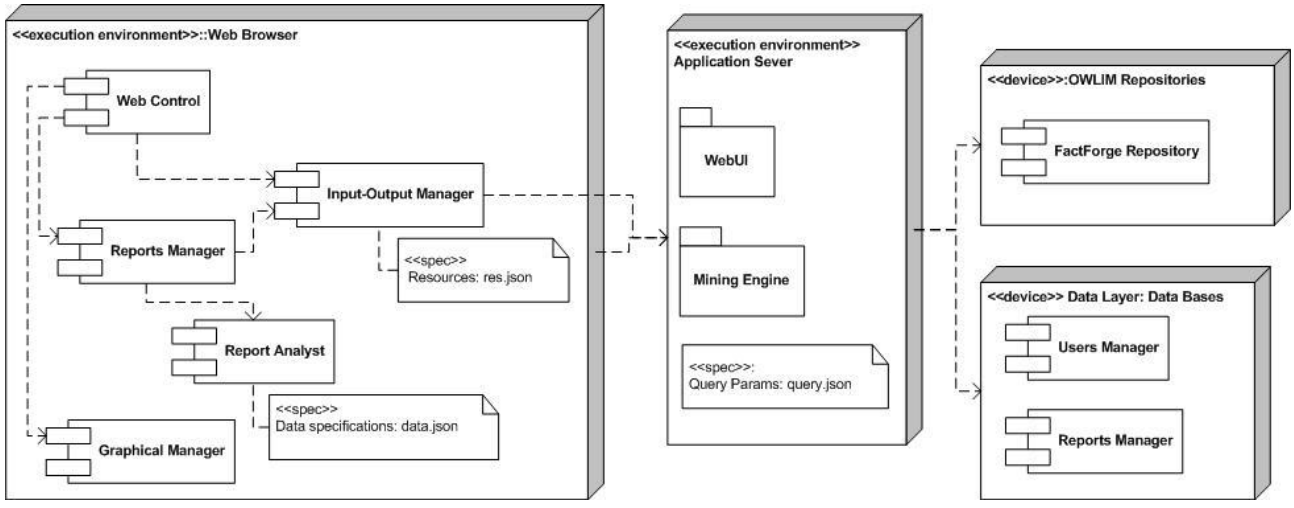

Figure 13. The Telefonica OMT components and deployment diagram.

Web Application (web browser side)

This is the front-end of the tool. It graphically shows the reports to the user. It is aimed at the user for report handling. However, its main functionality is being the interface between the user and the report generation engine to compose new reports.

- **Web Control**: To manage the coordination of front-end components and handing every event (from users or backend callbacks).

- **Input-Output Manager**: This component will handle the communications with the backend, composing the AJAX messages and formatting the data received or to be sent.
    - Resources file: This file describes the resources required for input-output operations like external URLs, back-end API functions or input-output expected formats.
- **Reports Manager**: It will handle all the generated reports. It is an API to dispatch the needs from the 'graphical manager' and the 'input-output manager' to set or to take data from the reports.
- **Report Analyst**: This component will receive analyzed data and it will adapt them to be used by the 'graphical manager' to generate the graphs, charts and all the graphical representations of the report statistics.
    - Data specifications File: This file describes the format and the nature of the received data to compose reports from preprocessed data.
- **Graphical Manager**: This component will handle the user interface showing the reports, managing the input forms and plotting report charts and results.

Web Application (web server side)

This is the back-end of the tool. It receives the user inputs, composes the queries and processes the data coming from the repository. Once it has all the data, it takes out the analysis task and sends the results to the front-end for their graphical visualization.

- **WebUI**: This component will collaborate with the front-end application. It will receive the request from the user side (queries, save and load reports, login…)
    - Query parameters file: This file describes some data needed to compose and send queries to FactForge or to the data layer like URLs, templates, language descriptions…
- **Mining Engine**: This component will receive all the raw data from the repository when a new report is ordered. It will take out the statistical analysis and the data processing to answer a front-end request.

Data Layer (Databases)

This component will support some minor functionalities of the tool like the users' login profiles or manage the reports to be saved.

- **Users Manager**: It will handle and store the user profiles.
- **Reports Manager**: It will manage and store the saved reports.

OWLIM repositories (Massive data storage)

This component will store all the data available to compose the reports. It will resolve the upcoming queries providing the data filtered by the user inputs. Its responsible is Ontotext.

5.2 Wikipedia Case Study Software Components

The Wikipedia case study contains four main software components, which we describe in what follows. We start with the Task List Generator and the Article Statistics and Quality Monitor, which are two supporting tools for the Wikipedia community. The next components that we describe are WIKIGINI and NewsFinder.

5.2.1 Task List Generator

The Task List Generator is a tool which is aimed at supporting in particular established Wikipedia authors. With this tool users can generate lists of articles that need to be improved. The users can specify a category or fields of interest and select different filters concerning problems in the content. The results are provided in html or in wiki text.

5.2.1.1 Use Case Diagram

On the one hand an established Wikipedia author generates/requests list of articles which needs to be improved with help of the task list generator. The user can choose his/her preferred category (intersection of categories and/or exclusion of categories) and select specific filters to generate the result list. This list of filters can be extended according to the requests of the users and the availability of further analysis approaches. Currently, the user can find:

- outdated articles
- non-neutral articles
- articles containing certain flaws (templates)
- articles deviating in size
- articles with a majority of negative reader feedback
- articles without images

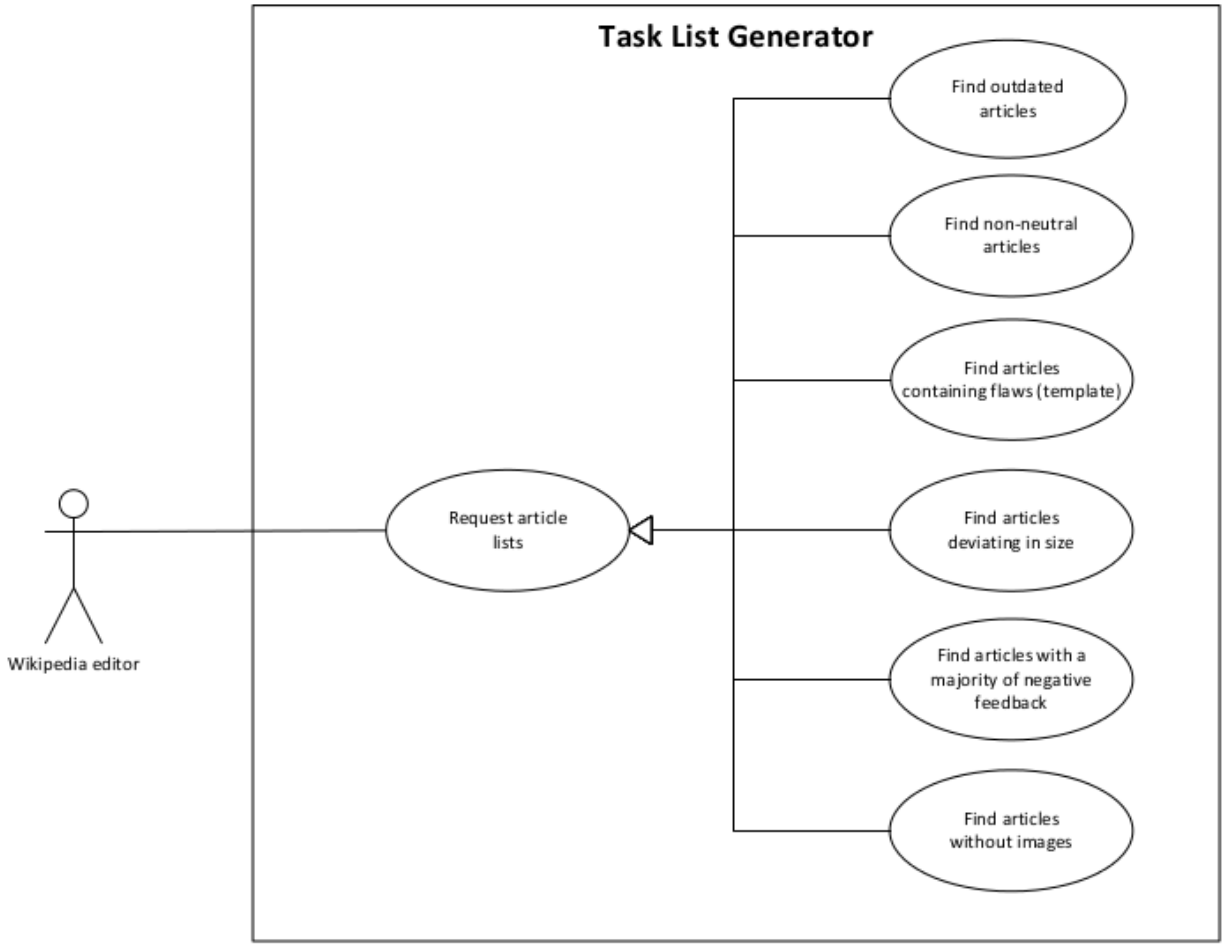

Figure 14. The Task List Generator use case diagram

Table 20. The "Request article list" use case scenario.

Use case name	**Request article list**
Actors	Wikipedia editors
Preconditions	Open TLG form

Main scenario	1. User selects a language. 2. User selects a search set (category, intersection of categories or exclusion). 3. User selects the search depth within the category tree. 4. User selects the output format HTML. 5. User chooses results on demand. 6. User specifies the filters for searching • outdated articles • non-neutral articles • articles containing certain flaws (templates) • articles deviating in size • articles with a majority of negative reader feedback • articles without images 7. The backend calculates the article list for these conditions. 8. The results are presented as table below the search form. 9. User analyse the results and can check for necessary actions to improve the articles.
Alternative scenarios	4. User selects the output format WikiText. 5. User chooses results via Email. 8. User gets an email with the results.

5.2.1.2 Components Diagram

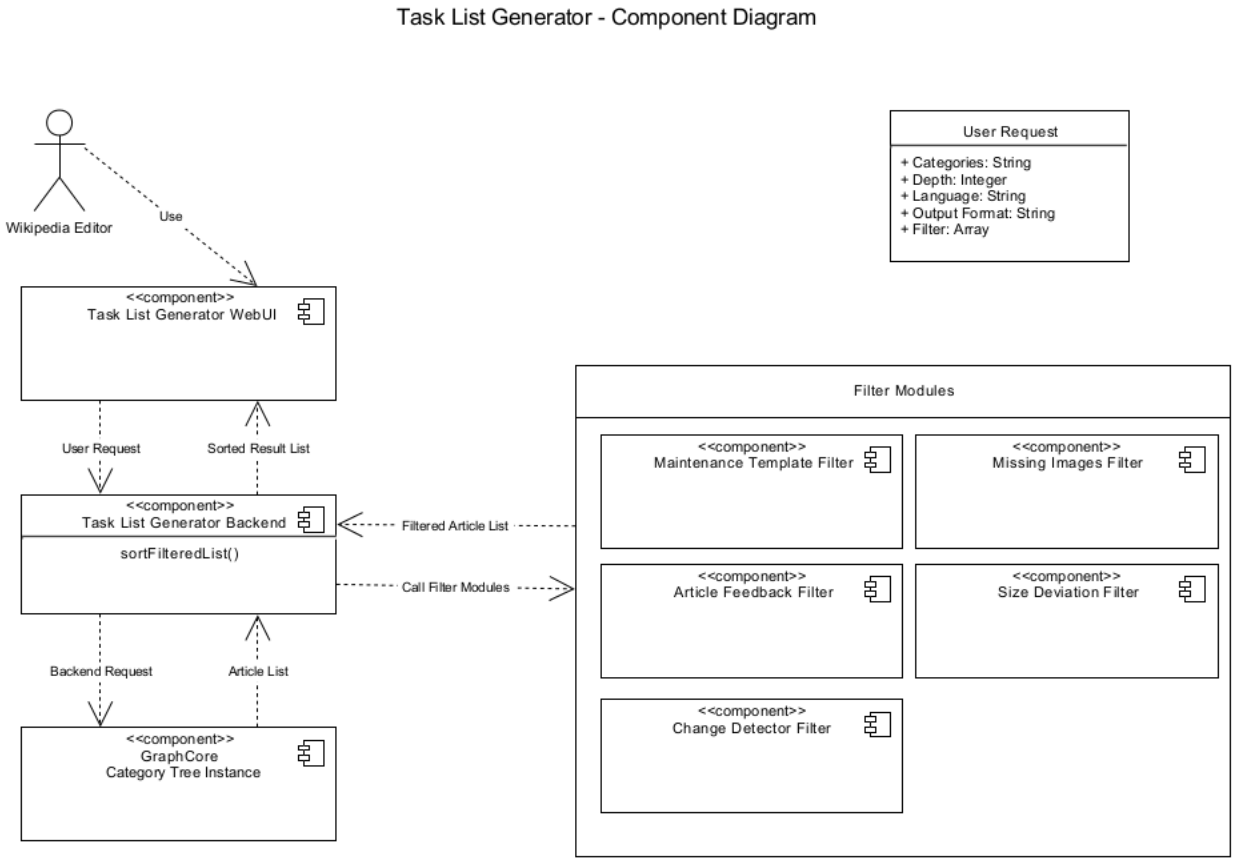

Figure 15 The Task List Generator component diagram.

Task List Generator WebUI: The users can access the TLG via an online form. There they can specify categories, the search depth in the category tree and the language version. Additionally, they can choose the output format (HTML or WikiText) and select certain filters to generate a article list.

Task List Generator Backend: requests the GraphCore Category Tree Instance for articles according to the language version, the category selection and the search depth. The resulting article list will be analysed by the selected filter modules. The final filtered article list will be sorted and presented in a table in the chosen output format.

Filter Modules create a unsorted list of articles according to the requested filters (filter list can be extended):

- The Maintenance Template Filter checks which articles contain maintenance templates.
- The Article Feedback Filter checks for which articles an article feedback is available and collects the percentage of negative ratings and the number of overall ratings.
- The Change Detector Filter searches for articles which were found by the Change Detector analysis.
- The Missing Images Filter checks which articles do not contain an image.
- The Size Deviation Filter checks which articles deviate in the article size compared to other articles in the search article set.

5.2.2 Article Statistics and Quality Monitor

With help of ASQM a user can get a brief overview about the statistics and the status of a Wikipedia article. A user can install a gadget within its user page in Wikipedia.

5.2.2.1 Use Case Diagram

On the other hand a Wikipedia reader can requests additional information about an article with help of the Article Statistics and Quality Monitor. Thia information can be used to understand the status and the quality, to improve or to extend an article, and also to find further information about a topic. For this the user can use the following provided parameters and information:

- General statistics
- Results of fact coverage analysis
- Results of currentness analysis
- Results of neutrality analysis
- Results of authorship analysis
- Results of further assessment tools, like rating from the Article Feedback Tool

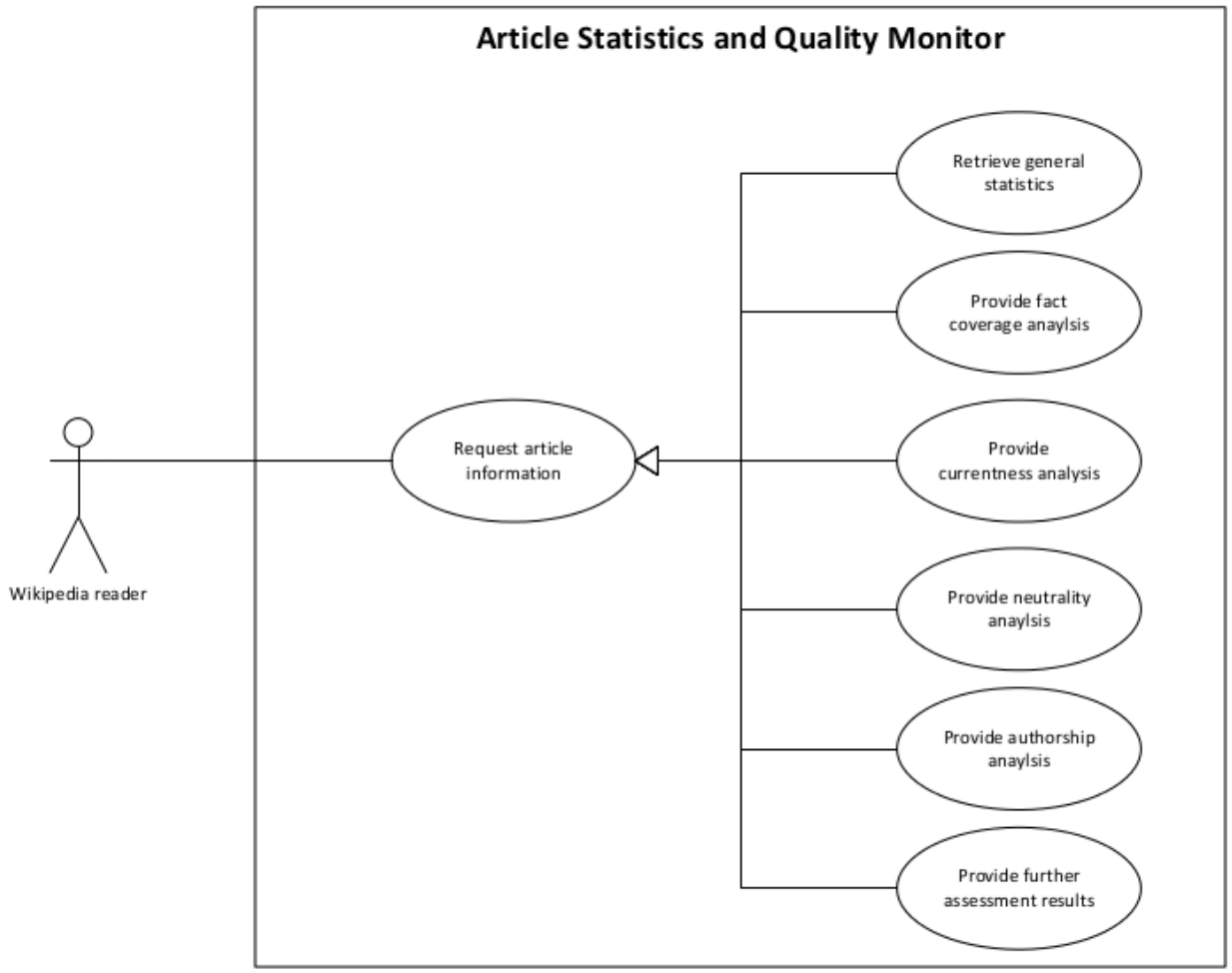

Figure 16. The ASQM use case diagram.

Table 21. The "Request article information" use case description.

Use case name	**Request article information**
Actors	Wikipedia readers
Preconditions	1. User has to be logged in 2. ASQM gadget has to be installed in his account settings
Main scenario	1. User clicks on the ASQM tab. 2. ASQM script requests statistics, analysis scores and generates specific links. 3. All available results are presented in a box. 4. User checks the statistics for the requested articles. 5. User follows the link to the LEA analysis and can check if links are missing in the requested article.
Alternative scenarios (if these information are available for the requested article)	6. User follows link to Change Detector result table. 7. User realises a number of published news articles and opens a link to the list of news articles. 8. User realises a WikiGini Score and follows a link to the whole analysis of this article. 9. User realises the percentage of the feedback rating and follows the link to all feedbacks about this article.

5.2.2.2 Components Diagram

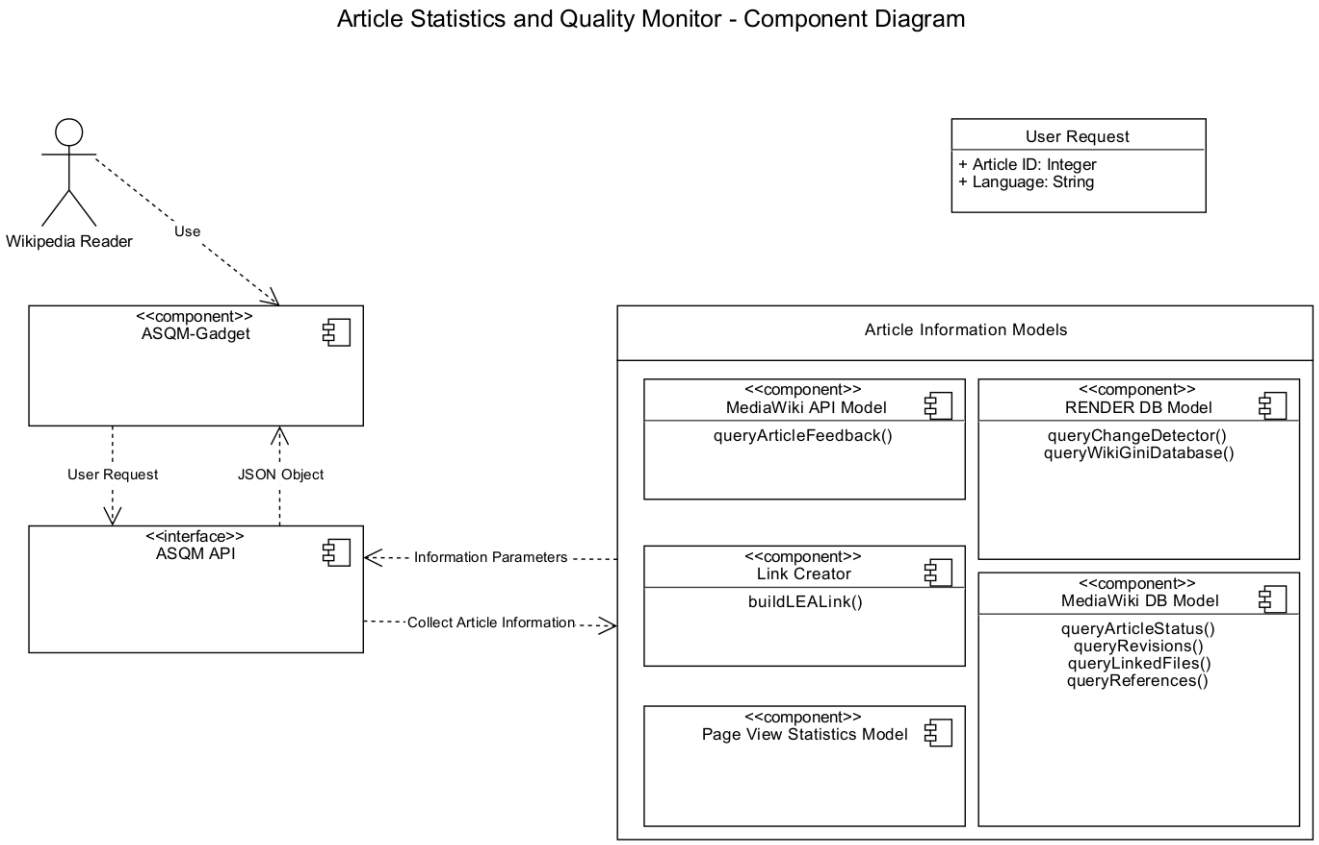

Figure 17. The ASQM component diagram.

ASQM Gadget: To use ASQM a user has to install a gadget. Further information on http://toolserver.org/~render/stools/asqm.

ASQM API: After a successful installation a new tab "ASQM" will be visible in the tab bar for Wikipedia articles. By pressing this tab, the information about this current article will be collected (by using the article information models). The calculated results will be presented in an output box to the users.

Article Information models:

- The MediaWiki API Model requests the article feedback information for a given article from the MediaWiki API.
- The Link Creator builds up the query link to the Link ExtrActor Analysis.
- The Page View Statistics Model requests http://stats.grok.se/ for a certain article.
- The RENDER DB Model requests for analysis results which are stored in the RENDER DB on the Wikimedia Toolserver. If the article was detected by the Change Detector as out of date the link to the result visualisation page will be provided. If a WikiGini result is available, the score of the latest revision will be extracted and the link to the overall visualisation of all revisions of this article will be build up.
- The MediaWiki DB Model requests the status of the article, the revision information, the information of the linked files and the number of references.

5.2.3 WIKIGINI

WIKIGINI visualizes the inequality of authorship in a Wikipedia article: it accurately determines the authors of all the words in every revision of an article and calculates a so-called “Gini-Index” as a measure of inequality. It can thus show if the “ownership” of words in an article shifted from being quite equal (e.g. every editor wrote about the same amount of words) to being unequal (e.g. 2% of editors wrote 90% of the words).

5.2.3.1 Use Case Diagram

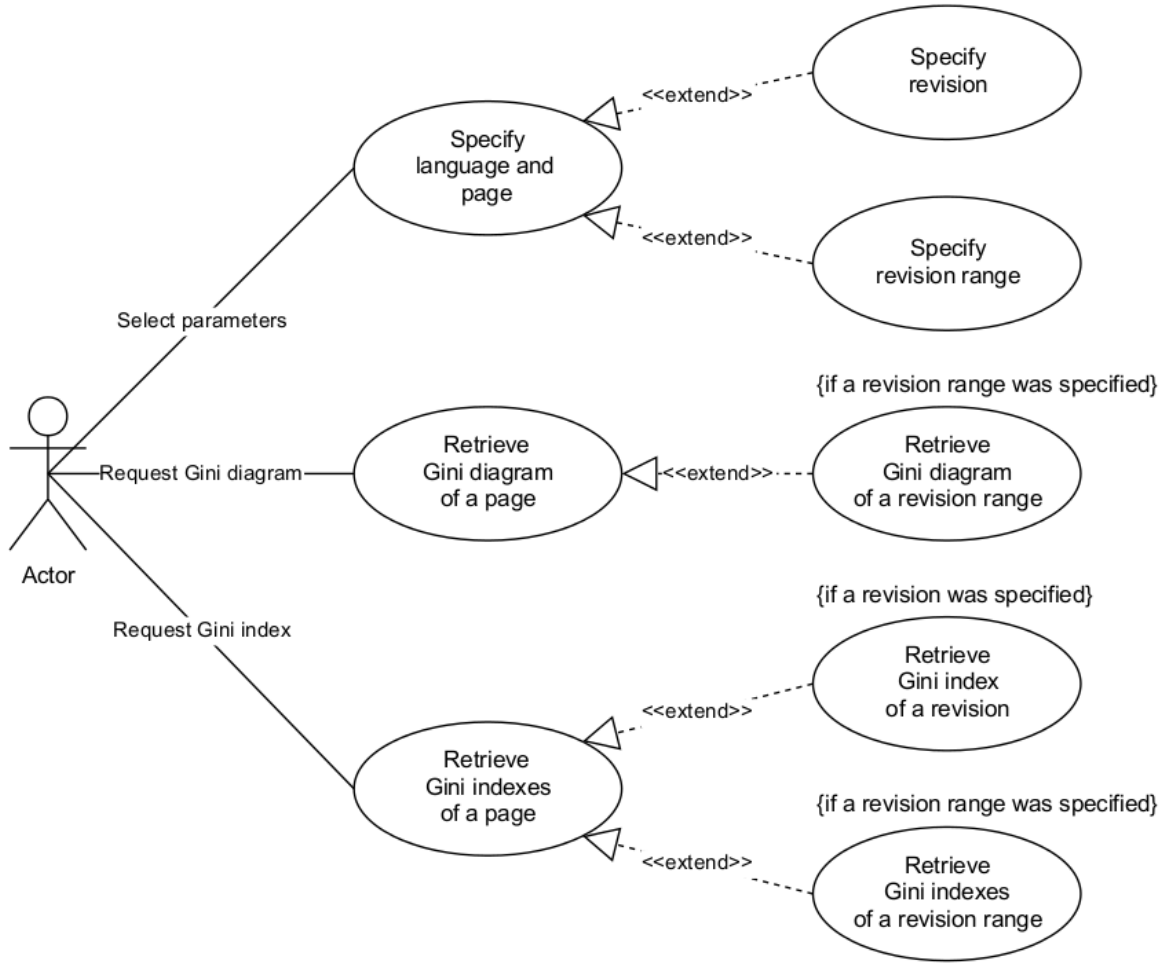

Figure 18. The WIKIGINI use case diagram.

Actors

WIKIGINI does not require any additional actors but the requester, who may be

- a human person using the web user interface (or manually the web API)
- a program using the web API

Both ways to send request to the application are accessible via standard HTTP request.

WIKIGINI supports two main use cases, one supporting use case as well as a number of derived special use cases.

Specify language and page (supporting use case)

In order to be able to retrieve results from WIKIGINI an actor must specify the parameters language as well as page. The language corresponds to one of the supported Wikipedia language versions; the page must be a page within the corresponding Wikipedia language version. Both parameters specify the results to be delivered in the two main use cases and are extended by the following two special use cases.

Specify revision (supporting use case)

The "Specify revision" use case extends the "Specify language and page" use case. In addition to a language and a page the actor may specify a specific revision. The revision must correspond to a revision of the requested page. The results returned will only include the results for the revision.

Specify revision range (supporting use case)

The "Specify revision range" use case extends the "Specify language and page" use case. In addition to a language and a page the actor may specify a specific revision range. The revision range must correspond to a revision range of the requested page. The results returned will only include the results for the revisions with the given range.

Retrieve Gini diagram of a page

The first main use case is the retrieval of a Gini diagram visualizing the development of the Gini index in the history of a Wikipedia page. Preferable the diagram is requested by a human person using the web user interface. The result presented to the user is a dynamic chart enabling the user to retrieve more detailed information about specific revisions by directly in the diagram. An actor may also specify a range of revisions directly in the diagram and thus trigger the following specialized use case.

Retrieve Gini diagram of a revision range

The "Retrieve Gini diagram of a revision range" extends the "Retrieve Gini diagram of a page" use case. A user may select a range of revisions directly in the diagram. The diagram will be automatically adjusted to show the specific range of revisions in detail.

Retrieve Gini indexes of a page

The second main use case is the retrieval of Gini data preferable from third party applications for further processing of the results. The diagram may be requested by an application or manual by a human person using the web based applications programing interface. The result delivered to the actor includes the Gini indexes of all revisions of a page. An actor may also specify a range or a specific version an thus trigger one of the following specialized use cases.

Retrieve Gini index of a revision

The "Retrieve Gini index of a revision" extends the "Retrieve Gini indexes of a page" use case. The result will include the Gini index for a specific revision if the actor specifies a specific revision and this revision is part of the revision history of the page.

Retrieve Gini indexes of a revision range

The "Retrieve Gini indexes of a revision range" extends the "Retrieve Gini indexes of a page" use case. The result will include the Gini index for a range of revision if the actor specifies a valid revision range and this revision range is part of the revision history of the page.

5.2.3.2 Component Diagram

WIKIGINI WebUI

The WIKIGINI WebUI is accessible via standard web browsers and intended for human users. It renders the available results of the analysis of a particular Wikipedia page, the Gini indexes, in a user-friendly manner.

WIKIGINI WebAPI

The WIKIGINI WebAPI is accessible via standard web browsers and intended for programmatic access. It return the available results of the analysis of a particular Wikipedia page, the Gini indexes, in an application-friendly manner.

WIKIGINI Database

The WIKIGINI Database is the central storage for results of the analysis as well as for analysis request. The database is deployed in a standard database system, e.g. MySQL, and thus accessible via standard SQL mechanisms. However, human as well as programmatic access should be handled via the WebUI and WebAPI components.

WIKIGINI Analyser

The WIKIGINI Analyser encapsulates the logic to handle new requests, load article dumps and calculate authorship of text as well as Gini indexes. This component runs independent from the WebUI and WebAPI as a background process and is triggered asynchronously via the Database component. New requests for the analysis of articles are stored in a queue within the WIKIGINI database and processed as soon as resources for the analysis are available.

Authorship Calculator

Calculation of Gini indexes is based on authorship information of the words in the corresponding page. The build-in Authorship Calculater component calculates incremental the authoship of each word in every revision of a page. Its input is a full dump of a page, including all revisions since creation of the article. As a build-in component it is not accessible via public interfaces.

Gini Calculator

The build-in Gini Calculator component calculates the Gini indexes for revisions of a page. Its input is the authorship information calculated by the Authorship Calculator component for all revisions since creation of the article. The result, a list of Gini Indexes with additional metadata is stored in the WIKIGINI Database for further processing. As a build-in component it is not accessible via public interfaces.

Page Dump Loader

To be able to analyse the different revsisions of a page, the dump of the corresponding page has to be loaded. This dump can be either extraced from the last full dump of the corresponding Wikipeda language version, or via the export functionality of Wikipedia. The Page Dump Loader component is able to load the dumps of a requested article on demand via the export functions of Wikipedia. The full dumps are handed over to the Authorship Calculator component for further processing. As a build-in component it is not accessible via public interfaces.

Client Request Handler

The Client Request Handler component comprises the indirect interface to the WIKIGINI Analyser component. Request for the analysis of pages are stored in the WIKIGINI Database and asynchronously processed by the Client Request Handler component. For each new request the component triggers the download of new revisions of the corresponding page, the analysis of the authorship of words and the calculation of the Gini indexes. As a build-in component it is not accessible via public interfaces.

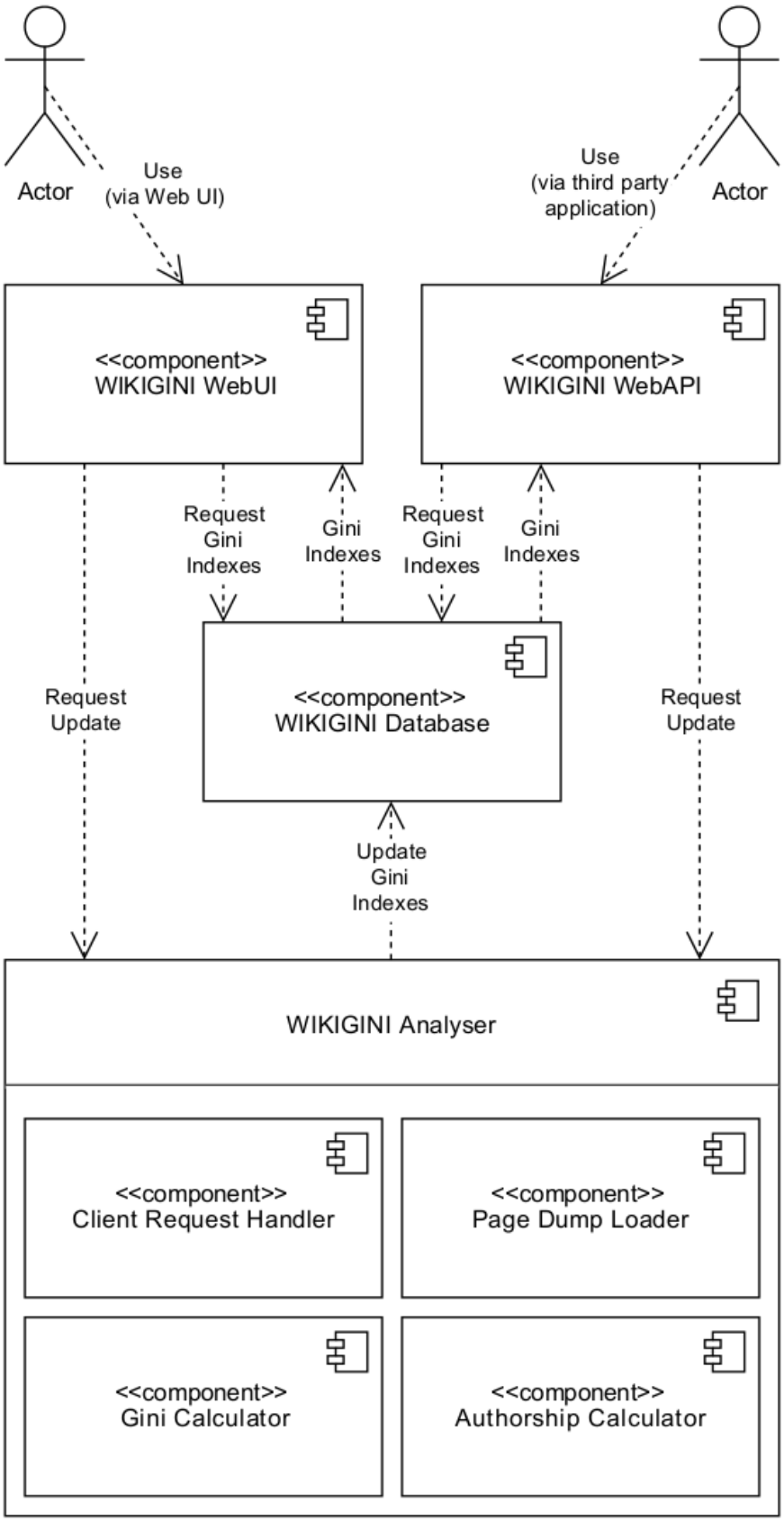

Figure 19. The WIKIGINI component diagram.

5.2.4 NewsFinder

NewsFinder provides news articles relevant for Wikipedia articles. Given a Wikipedia article title and the language version, NewsFinder determines news articles ranked by relevance.

5.2.4.1 Use Case Diagram

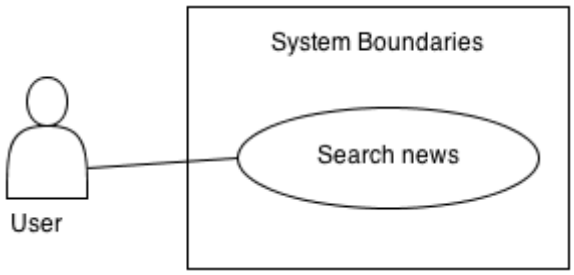

Figure 20. NewsFinder use case diagram

Table 22. The "Search news" use case scenario.

Use case name	**Search news**
Actors	User.
Preconditions	-
Postconditions	User visualizes the list of related news articles.
Main scenario	1. User opens NewsFinder web front-end in a web browser. 2. User searches for a Wikipedia article title (and its language ID). 3. The system sends this information to the back-end server. 4. The back-end returns results.

5.2.4.2 Component Diagram

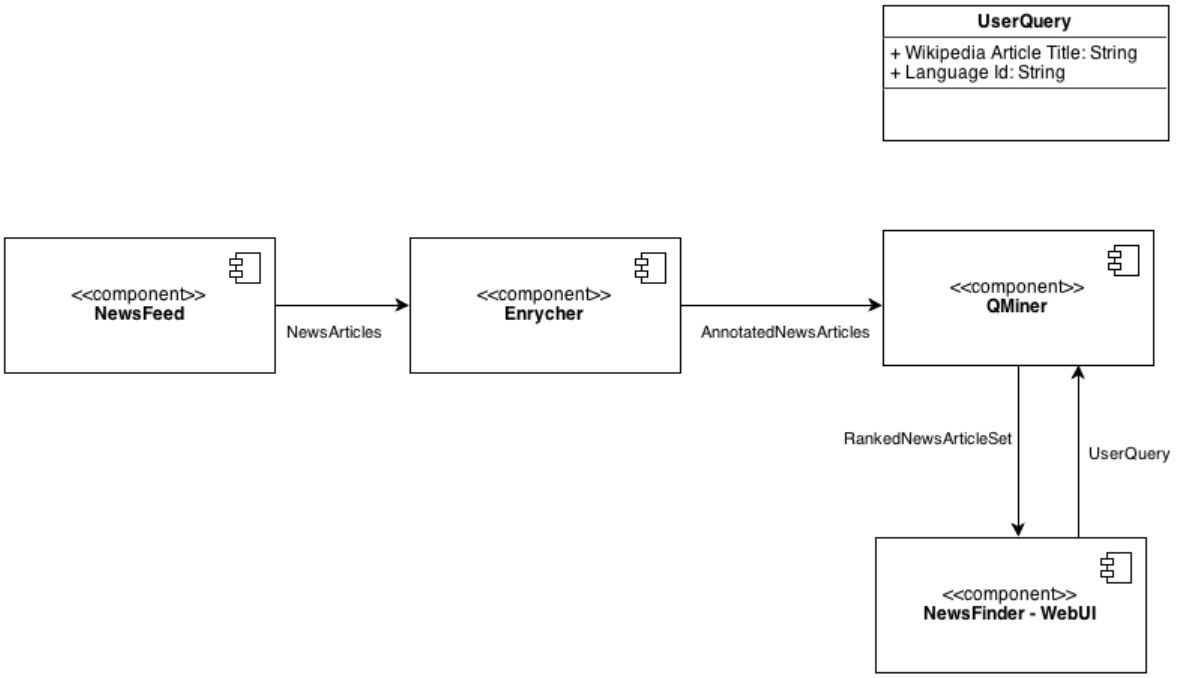

Figure 21. NewsFinder component diagram.

Newsfeed provides the feed of news articles crawled from RSS-enabled sites. These news articles are annotated using the **Enrycher** service pipeline. The annotated news are stored in **QMiner**.

The **QMiner** component is NoSQL database that enables fast querying of the database based on user input (only the search term / Wikipedia article and language Id are considered at this point) and returns the matching news articles. A ranking algorithm is used to order the news based on how related they are to the Wikipedia article.

5.3 Google Case Study Software Components

DiversiNews is an interactive tool which allows users to browse and summarize news articles from different perspectives. Given a certain topic of interest, the tool provides a succinct summary of the latest news, as well as the individual news items that have been summarized, sorted by relevance. The user can further specify which perspective on the news should be emphasized: articles containing certain predominant keywords, articles that belong to a certain geographical region, or articles with a positive or negative outlook. Depending on the user preferences, the summary is updated accordingly and the articles get reordered.

5.3.1 Use Case Diagram

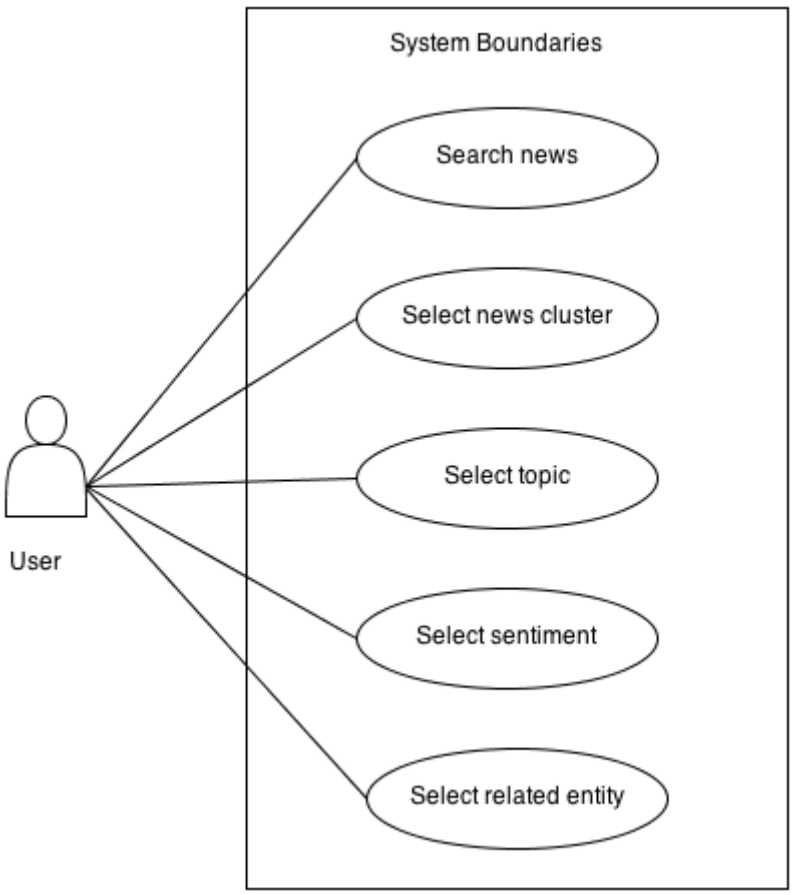

Figure 22. DiversiNews use case diagram.

Table 23. "Search News" use case scenario.

Use case name	**Search news**
Actors	User.
Preconditions	-
Postconditions	User visualizes the summarization results, extracted and related entities, news source location, sentiment.
Main scenario	1. User opens DiversiNews web front-end in a web browser. 2. User enters search keywords in the main page. 3. The system sends these values to the back-end server. 4. The back-end returns results.
Alternative scenario	4. The back-end returns no results for the provided keywords.

Table 24. "Select news cluster" use case scenario.

Use case name	**Select news cluster**
Actors	User.
Preconditions	-
Postconditions	User visualizes the summarization results, extracted and related entities, news source location, sentiment.
Main scenario	1. User opens DiversiNews web front-end in a web browser. 2. User selects a news cluster in the main page. 3. The system sends this information to the back-end server. 4. The back-end returns results.

Table 25. "Select topic" use case scenario.

Use case name	**Select topic**
Actors	User.
Preconditions	User searched for news via the keyword search or selected a news cluster
Postconditions	The summarization results, extracted and related entities, news source location, sentiment change according to the selected topic.
Main scenario	1. User moves the search point on one of the possible displayed topics. 2. The system sends this value to the back-end server. 3. The back-end updates the results.

Table 26. "Select sentiment" use case scenario.

Use case name	**Select sentiment**
Actors	User.
Preconditions	User searched for news via the keyword search or selected a news cluster
Postconditions	The summarization results, extracted and related entities, news source location, sentiment change according to the selected sentiment.
Main scenario	1. User moves the sentiment scroll bar to either positive or negative sentiment. 2. The system sends this value to the back-end server. 3. The back-end updates the results.

Table 27. "Select related entity" use case scenario.

Use case name	**Select related entity**
Actors	User.
Preconditions	User searched for news via the keyword search or selected a news cluster
Postconditions	The summarization results, extracted and related entities, news source location, sentiment change according to the selected topic.
Main scenario	1. User selects one of the related entities from the list. 2. The system sends this value to the back-end server. 3. The back-end updates the results.

5.3.2 Component Diagram

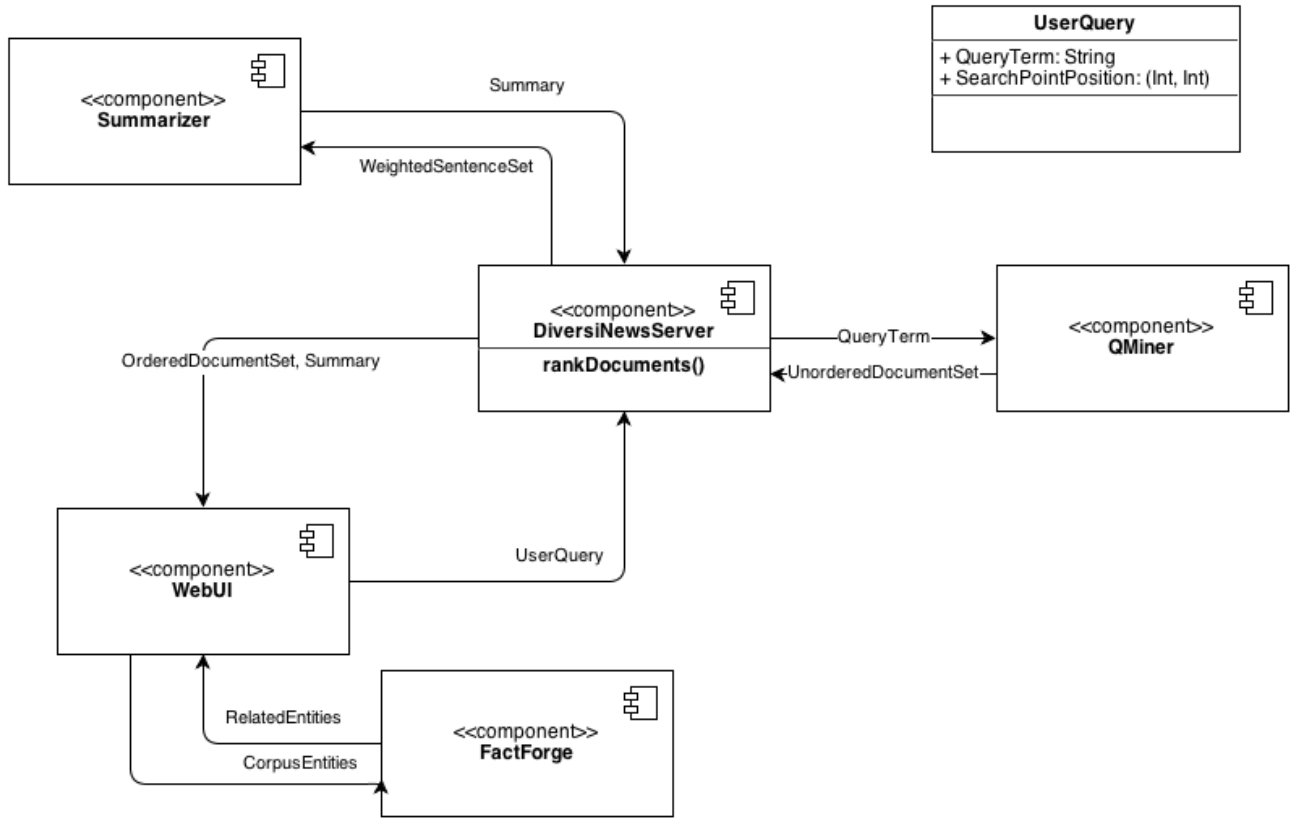

Figure 23. DiversiNews component diagram.

DiversiNews is implemented as a classic server-client architecture. The DiversiNewsServer component is central both visually in the diagram and logically as far as dataflow is concerned. It delegates work to other components and enables communication between them. In particular:

The **QMiner** component is NoSQL database that enables fast querying of the database based on user input (only the search term / cluster ID are considered at this point) and returns the matching documents.

The **Server** component itself performs ranking of the results based on the remainder of the user input, i.e. the position of diversity controls (topical map, world map, sentiment bias)

The ranked documents are relayed to the **Summarizer** component which returns a summary.

The filtered, ranked and summarized results are sent to the **Client** component, written in JavaScript and running in the user's browser. This is a "thin client" that does no processing: it relays the user inputs to the Server and receives full results in response. Its only responsibility is formatting and display. The one

exception to this is the FactForge component which, due to greater ease of integration, is currently contacted from the Client rather than the Server.

The **FactForge** component receives as input the documents currently being displayed to the user. It queries the semantic database to find related entities and returns those to the Client to be displayed as further browsing suggestions, enabling exploratory analysis of the dataset.

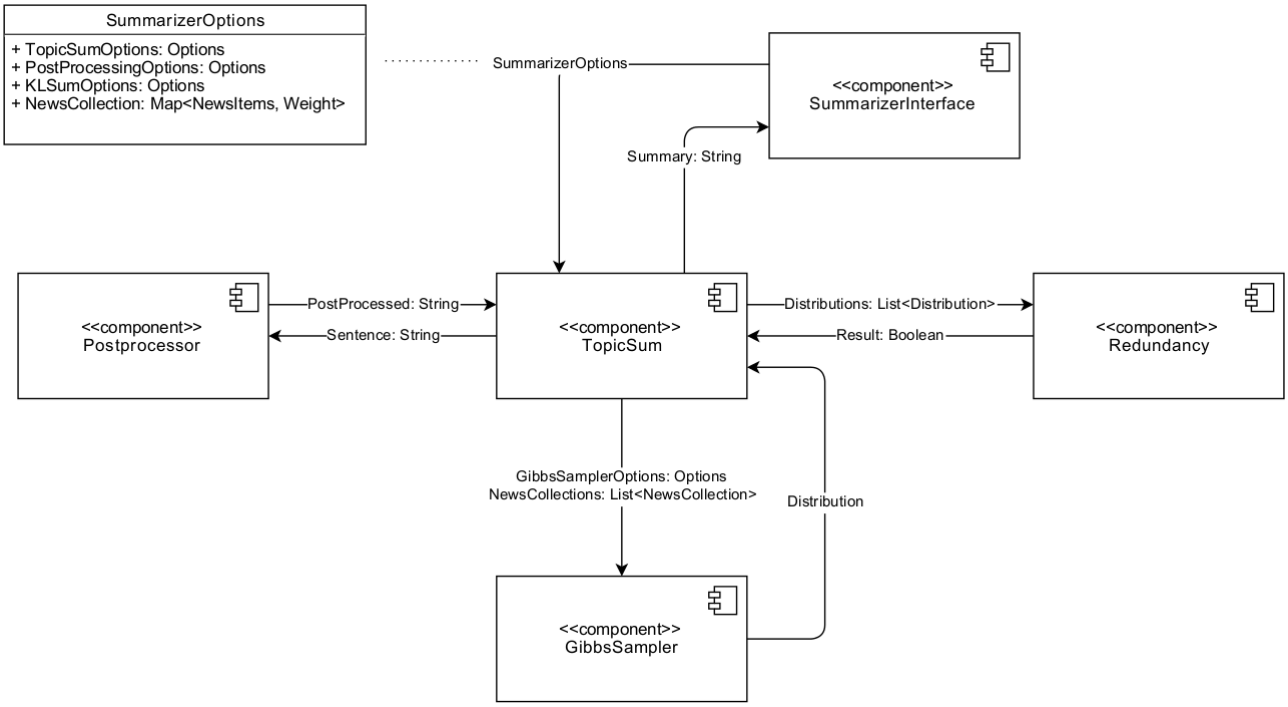

Figure 24. The Google Summarizer component diagram.

TopicSum is a C++ library that is invoked by its clients to generate summaries of news collections. The bundle with the summarizer source code is available for download from:

https://code.google.com/p/summarizer/source/checkout.

Initially, the output from the DiversiNewsServer (provided in XML) needs to be parsed and transformed into the summarizer's input format. The data structured used for input and output in the summarizer are all based on protocol buffers, which allow for very efficient serialization, extensibility, and are language and platform neutral, facilitating portability across applications.

TopicSum: the main component of the summarization tool. It stores the set of sentences from the original documents. Using these, an extractive summary will be constructed. It initializes sentence candidate scorers by analyzing the document collection, from which statistics are extracted, e.g. identifying terms that represent the central topic which the news are reporting. This step needs to be run just once, when the user selects a news collection of interest.

GibbsSampler: implements the Gibbs sampling algorithm; this component is responsible for sampling from the distribution used by TopicSum.

SummarizerInterface: provides an interface for the summarization tool, which the DiversiNewsServer accesses. The weighted sentence set is received as input from the DiversiNewsServer, and the output sent back to the DiversiNewsServer is the generated summary.

Postprocessor: removes unnecessary fragments of the selected sentences, may attempt to reorder them, and returns them to the client.

Redundancy: For every summarization request, the collection sentences are scored and a non –redundant summary is generated. Particular summarization options, e.g. whether the summary needs to reflect more positive or more negative opinions, are taken into account during sentence selection.

The **Postprocessor** and **Redundancy** steps are performed whenever the user requests a new summary, e.g. by changing the options about polarity or the entities of interest.

5.4 Drupal Extension

The RENDER Drupal extension provides a diversity-aware view on posts provided with Drupal. It is able to show extracted topics and named entities and provides links for these.

5.4.1 Use Case Diagram

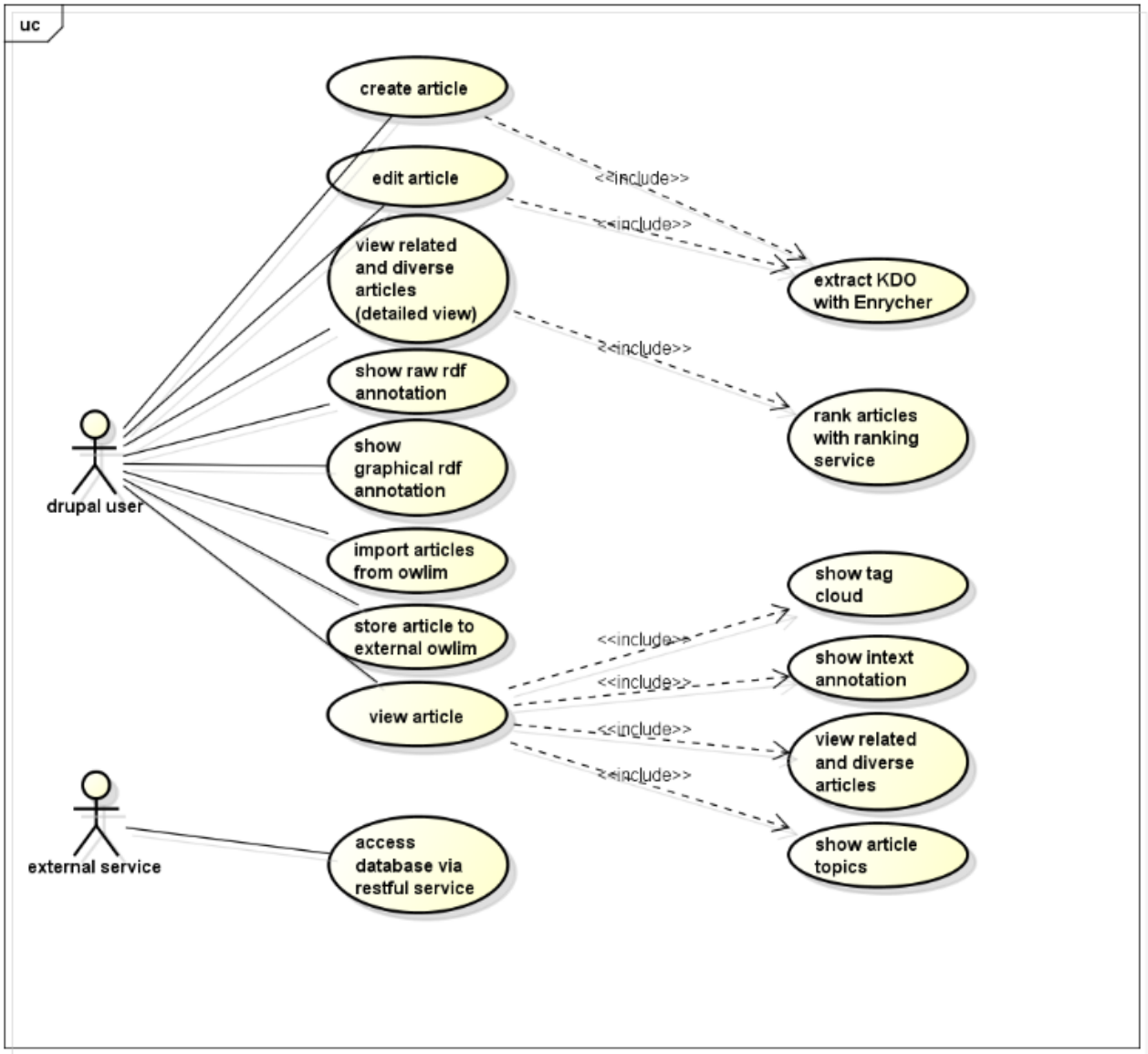

Figure 25. The Drupal extension use case diagram.

Table 28. The "Create article" use case scenario.

Use case name	**Create article**
Actors	Drupal User.
Preconditions	-
Description	• creates article in drupal database • extracts diversified information using the enricher service • stores the metadata in a local owlim store • stores mapping between drupal article and metadata in drupal database

Table 29. The "View related and diverse articles (detailed view)" use case scenario.

Use case name	**View related and diverse articles (detailed view)**
Actors	Drupal User.
Preconditions	clicked on article in the treeview on the righthand side
Description	• shows related articles (based on ranking) in new frame split up based on sentiment

Table 30. The "Edit article" use case scenario.

Use case name	**Edit article**
Actors	Drupal User.
Preconditions	Article shown
Description	• edits article in drupal database • extracts diversified information using the enrycher service for the new content • stores the metadata in a local owlim store • updates mapping between drupal article and metadata in drupal database

Table 31. The "Show raw rdf annotation" use case scenario.

Use case name	**Show raw rdf annotation**
Actors	Drupal User.
Preconditions	clicked on "show rdf" button
Description	• shows rdf representation of extracted kdo metadata

Table 32. The "Show graphical rdf annotation" use case scenario.

Use case name	**Show graphical rdf annotation**
Actors	Drupal User.
Preconditions	clicked on "show rdf structure" button
Description	• shows rdf representation in a graphical view (triples with different colors)

Table 33. The "Import articles from OWLIM" use case scenario.

Use case name	**Import articles from OWLIM**
Actors	Drupal User.
Preconditions	clicked on "owlim" button
Description	• loads articles stored in the owlim store into the drupal system

Table 34. The "Store article to external OWLIM" use case scenario.

Use case name	**Store article to external owlim**
Actors	Drupal User.
Preconditions	clicked on "load articles from owlim" button and user provided owlim credentials
Description	• submits metadata of currently shown article to external owlim store

Table 35. The "View article" use case scenario.

Use case name	**View article**
Actors	Drupal User.
Preconditions	Opened specific article
Description	• shows intext annotation • shows related articles on right hand side • shows article topics • shows tag cloud

Table 36. The "Access database via restful service" use case scenario.

Use case name	**Access database via restful service**
Actors	External service.
Preconditions	-
Description	• provides restful interface which returns a json representation of the most important kdo properties extracted for a specific article

Table 37. The "Extract KDO with Enrycher" use case scenario.

Use case name	**Extract KDO with Enrycher**
Actors	Drupal User.
Preconditions	-
Description	• see Enrycher description

Table 38. The "Rank articles with ranking service" use case scenario.

Use case name	**Rank articles with ranking service**
Actors	Drupal User.
Preconditions	-
Description	• see ranking service description

Table 39. The "Show tag cloud" use case scenario.

Use case name	**Show tag cloud**
Actors	Drupal User.
Preconditions	-
Description	• provides tag cloud containing the sioc:tag values. The most used sioc:tag values are displayed with bigger size

Table 40. The "Show intext annotation" use case scenario.

Use case name	**Show intext annotation**
Actors	Drupal User.
Preconditions	-
Description	• annotates each occurrence of a topic in the article text and highlights these annotations. A more detailed view can be shown by clicking on the annotation

Table 41. The "View related and diverse articles" use case scenario.

Use case name	**View related and diverse articles**
Actors	Drupal User.
Preconditions	-
Description	• shows a treeview with related articles (based on ranking) separated by sentiment

Table 42. The "Show article topics" use case scenario.

Use case name	**Show article topics**
Actors	Drupal User.
Preconditions	-
Description	• lists all topics the article talks about

5.4.2 Component Diagram

The component diagram of the Drupal extension was presented in Figure 11.

Diversity enabled Drupal extension: The diversity - enabled Drupal extension utilizes several RENDER technologies, such as OWLIM, Enrycher or the ranking service. These are used to extract diversity – aware information from articles. This information is displayed in several ways.

The Drupal extension uses Enrycher that extracts KDO diversity information from Drupal articles. The extension makes use of the Render Diversity-Aware Ranking Service that performs clustering on extracted data based on topics and sentiment. Articles in the same cluster have most topics in common with current article are therefore considered as related. The Drupal extension uses SPARQL queries on OWLIM that can be used to retrieve information that is displayed (e.g. tags, topics, article sentiment, ...).

5.5 TwiDiViz

TwiDiViz is a web-based application that enable the analysis and visualization of diversity in Twitter data. This tool uses core RENDER technologies (such as the Knowledge Diversity Ontology (KDO) ontology and the Enrycher service) to process Twitter datasets for the purpose of analyzing the impact of products on basis of sentiment and topic mining.

Using TwiDiViz, users can explore and get familiar with diversity aspects of the Twitter data in two different ways. We offer a static printable report summarizing the results of our diversity-oriented analysis. In addition, we provide an interactive visualizer based on Microsoft Pivot Viewer.

5.5.1 Use Case Diagram

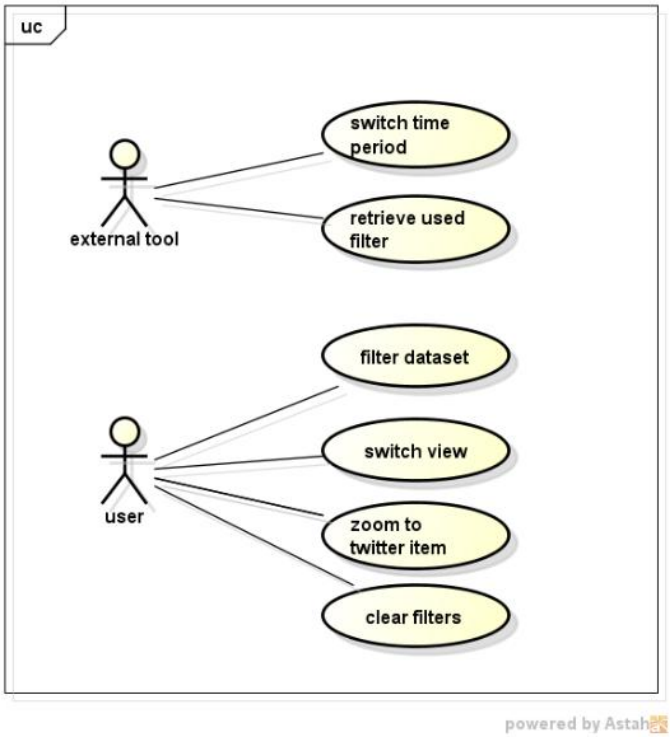

Figure 26. The TwiDiViz use case diagram.

Table 43. The "Switch time period" use case scenario.

Use case name	**Switch time period**
Actors	External tool.
Preconditions	External tool shown
Description	• external tool defines time period of the dataset • twidiviz loads the data • switch to pivot viewer

Table 44. The "Retrieve used filter" use case scenario.

Use case name	**Retrieve used filter**
Actors	External tool
Preconditions	pivot viewer loaded
Description	• applied filters of pivot viewer are submitted to external tool • switch to external tool

Table 45. The "Filter dataset" use case scenario.

Use case name	**Filter dataset**
Actors	User.
Preconditions	pivot viewer and data loaded
Description	• filter along several properties: ○ document type, language, topic, tag, sentiment

Table 46. The "Switch view" use case scenario.

Use case name	**Switch view**
Actors	User.
Preconditions	pivot viewer and data loaded

Description	• switch between different pivot viewer views:
	  ○ grid view
	  ○ graph view

Table 47. The "Zoom to twitter item" use case scenario.

Use case name	**Zoom to twitter item**
Actors	User.
Preconditions	pivot viewer and data loaded, clicked on specific element
Description	• twitter item is zoomed • preview is replaced by complete twitter view

Table 48. The "Clear filters" use case scenario.

Use case name	**Clear filers**
Actors	User.
Preconditions	pivot viewer and data loaded
Description	• all filters are removed

5.5.2 Component Diagram

TwiDiViz is a visualization tool for visualizing a high number of KDO annotated twitter posts. The Microsoft pivot viewer is used to enable to user to play with the data with different filter and sorting methods.

The component diagram of the TwiDiViz tool was presented in Figure 11.

Most technologies used to implement the TwiDiViz prototype are common sense web technologies (i.e. JSON, etc.). However, the TwiDiViz prototype has to deal with potentially big amounts of data in a web browser environment. This is a non-trivial issue and several visualization techniques have been evaluated for this purpose. To solve this, Microsoft Pivot Viewer is used. Also, TwiDiViz uses two core RENDER infrastructure services, namely Enrycher and the Data Layer based on OWLIM. Enrycher is a service-oriented system, providing shallow as well as deep text processing functionality at the text document level. The data layer efficiently stores the diversity knowledge. It offers a RESTfull interface to access and query the data, which is used in the implementation of TwiDiViz. As TwiDiViz is a visualization tool, it was designed for human users and does not offer a huge API. However, JSON encoded data filters can be sent and received by TwiDiViz. More details on that are provided in D4.2.1 [6].

5.6 Interactive Modelling Tool

The Interactive Modeling Tool provides interactive diversity analysis based on social media data. The tool segments posts and users into meaningful groups, based on different viewpoints, that depend on the topic, users' location and language, providing sentiment and opinion modeling, focusing on ad-hoc training of novel topics. It answers questions like "*What kinds of sentiment do people express on O2's iPhone offer?*", "*Which operator has the most social media talk about Android devices?*".

5.6.1 Use Case Diagram

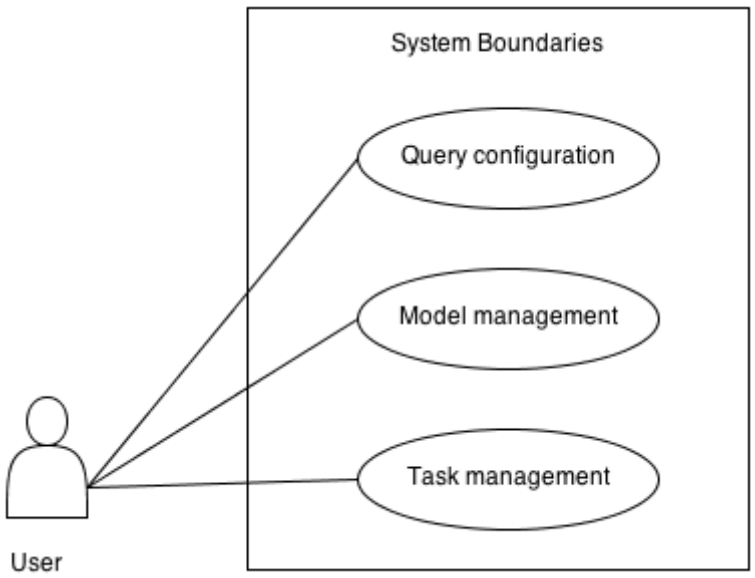

Figure 27. The Interactive Modelling Tool (HTML5 GUI) use case diagram.

Table 49. The "Query configuration" use case scenario.

Use case name	**Query configuration**
Actors	User.
Preconditions	-
Postconditions	User provided query configuration details.
Main scenario	1. User opens the Interactive Modelling Tool web front-end in a web browser. 2. User selects the task, model and query parameters. 3. The system sends this value to the back-end server. 4. The back-end updates the results.

Table 50. The "Model management" use case scenario.

Use case name	**Model management**
Actors	User.
Preconditions	-
Postconditions	User provided model management details.
Main scenario	1. User opens the Interactive Modelling Tool web front-end in a web browser. 2. User inputs the model name, and chooses to create a new model or update a current model. 3. The system sends this value to the back-end server. 4. The back-end updates the results.
Alternative scenario	4. The model cannot be created or updated, in which case the systems returns an error.

Table 51. The "Task management" use case scenario.

Use case name	**Task management**
Actors	User.
Preconditions	-
Postconditions	User provided task management details.
Main scenario	1. User opens the Interactive Modelling Tool web front-end in a web browser. 2. User inputs the task name and the classes for classification. 3. The system sends this value to the back-end server. 4. The back-end updates the results.
Alternative scenario	4. The task cannot be created, in which case the systems returns an error.

5.6.2 Component Diagram

The **embedded relational database** stores the data items and metadata, which need to be annotated. The search indexing and feature construction part is driven by IJS's **QMiner** infrastructure to ensure handling of datasets bigger than the available main memory.

Analysis Component. The learning models themselves are stored in-memory, since they need to be re-trained often. The system also utilized caching of generated feature vectors for items to enable efficient re-training of models when new labels arrive.

The **HTML5 client** (**GUI**) directly connects to the API provided by the Analysis Component. Via the GUI users can train the annotation models.

The API operates over the HTTP protocol, taking parameters as HTTP queries and returning responses as JSON documents. It is accessible on http://aidemo.ijs.si/render/[operation-name].

The following operations are supported (a detailed explanation can be found in D5.3.3 [4]).

- **addmodel**: create a new model for a given task using a particular algorithm
- **updatemodel**: manually trigger the model update
- **addtask**: define a new task with a given name and classes
- **addlabel**: add a new label to the task
- **search**: retrieve elements that match a free text query using full text search

- **searchandlabel**: retrieve elements that match a free text query and label them with a particular
- **getuncertain**: obtain the most uncertain elements given a particular active learning model
- **testmodel**: evaluate the performance of a given model using cross-validation.

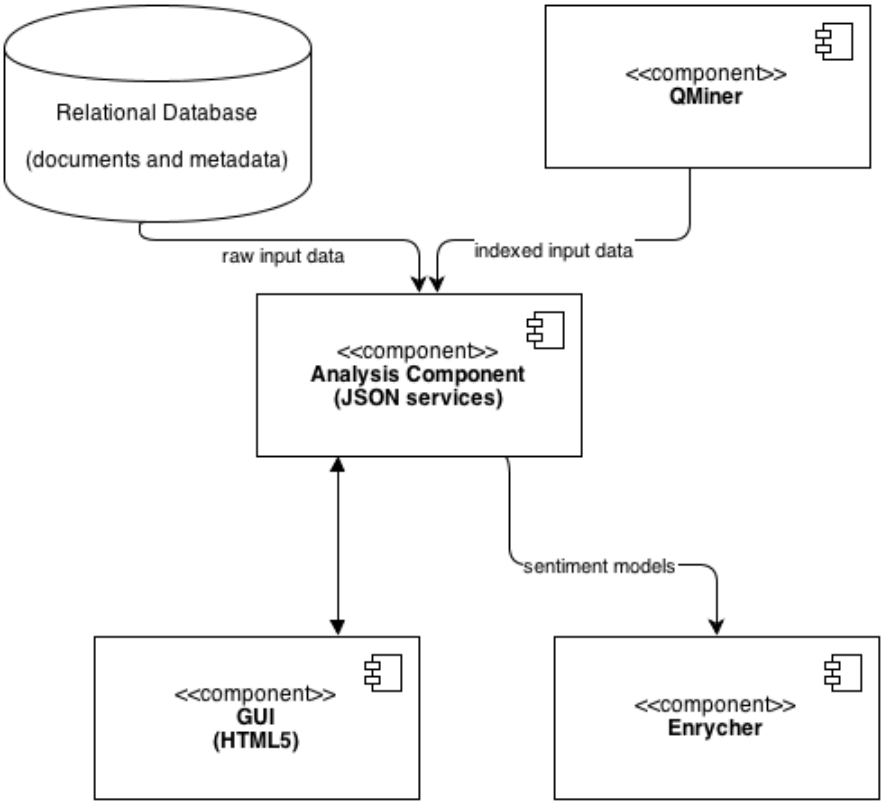

Figure 28. The Interactive Modelling Tool component diagram.

References

[1] Reinier H. van Leuken, Lluis Garcia, Ximena Olivares, and Roelof van Zwol. 2009. Visual diversification of image search results. In Proceedings of the 18th international conference on World wide web (WWW '09). ACM, New York, NY, USA, 341-350. DOI=10.1145/1526709.1526756 http://doi.acm.org/10.1145/1526709.1526756

[2] Thalhammer, A (Ed). 2012. Prototype of diversity-aware ranking. RENDER Deliverable D3.3.1.

[3] Grinberg, M. (Ed). 2013. Final version of the diversity-aware ranking. RENDER Deliverable D3.3.2.

[4] Caminero, J. (Ed). 2012. Initial version of diversity information extensions for Telefónica tools. RENDER Deliverable D5.3.3.

[5] Rusu, D (Ed). 2011. Prototype of the Fact Mining Toolkit. RENDER Deliverable D2.2.1.

[6] Floeck, F (Ed). 2012. Collecting best practices for diversity - aware collaboration. RENDER Deliverable D4.2.1.

