D5.2
WP5
Application development and functionality Report 1

R - Report, RE - Restricted

The UniteEurope Consortium:

Participant no.	Participant organisation name	Short name	Country
1 (Coordinator)	INSET Research and Advisory	INSET	Austria
2	Erasmus University Rotterdam - Department of Public Administration	EUR	Netherlands
3	SYNYO Innovation	SYNYO	Austria
4	Imooty Lab	IMOOTY	Germany
5	Malmö University - Institute for Studies of Migration, Diversity and Welfare	MHU	Sweden
6	ZARA, Zivilcourage & Antirassismusarbeit	ZARA	Austria
7	City of Rotterdam	CITYROT	Netherlands
8	City of Malmö	CITYMAL	Sweden
9	University of Potsdam, Department for Public Management	UP	Germany

UniteEurope is co-funded by the European Commission under the FP7 Programme

Document Information

Contract Number:	288308	
Lead Beneficiary:	IMOOTY	
Deliverable Name:	Application development and functionality Report 1	
Deliverable Number:	5.2	
Dissemination Level:	RE	
Contractual Date of Delivery:	August 31, 2012	
Delivery Date:	August 31, 2012	
Authors:	Dr. Peter Leitner	SYNYO
	Blaise Bourgeois	IMOOTY
	Johannes Neubarth	IMOOTY
Checked by:	Dr. Verena Grubmüller	INSET
	Dr. Katharina Götsch	INSET
	Dr. Peter Scholten	EUR
	Kristoffer J. Lassen	IMOOTY

Table of contents

Figures

Tables

1 Introduction

The aim of this deliverable is to document the current development status of the project. The report will focus on all parts of the architecture: the Data Management System (DMS), the crawler index, and the website[1].

As presented in deliverable D5.1, the first component of the UniteEurope application is the Data Management System (DMS). It serves as the configuration tool for the whole project, and is targeted at the UniteEurope team members. Its main focus is to manage the sources, the keywords, the annotations, the integration measures, the organisations and the user accounts. It also ensures that every web source is augmented with a predefined set of annotations. In addition, a monitoring page displays several performance indicators for the tool, such as the server traffic load, the status of the crawler process or a list of sources that are disabled and need to be revised.

Section 2 of this document will demonstrate the development of the DMS inside the Zend Framework and illustrate how the Model-View-Controller pattern was realised. The subsequent section will then describe the database where the DMS data are stored.

In section 4, we give a short overview of the UniteEurope website structure as well as the modules we have developed and which will be integrated into the Pimcore CMS later.

Finally, section 5 describes the UniteEurope web crawler. We document the crawler's main workflow (downloading of online news and posts, linguistic and geographic analysis of these data including keyword matching, and database storage) in a detailed but understandable way.

[1] For this report, the term "UniteEurope website" exclusively refers to the tool platform for the end users.

2 DMS development inside the Zend Framework

Zend Framework 1.11 is installed on the UniteEurope development server and is available at the subdomain http://dms.uniteeurope.org/. The whole DMS is developed within this framework.

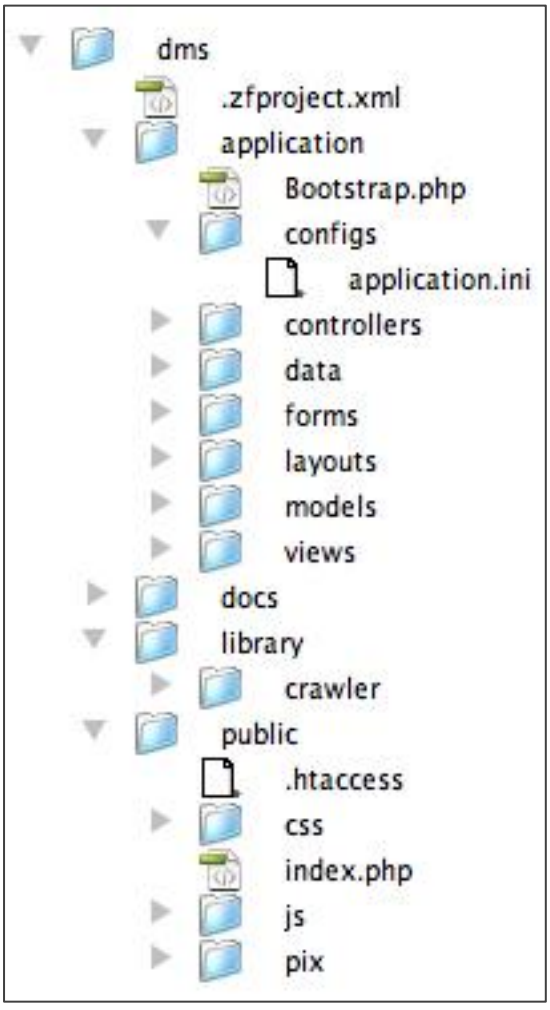

Figure 2.1: Structure of the DMS file hierarchy on the server

.

The Zend Framework, being a typical Model-View-Controller framework, is comprised of the corresponding folders (*configs*, *controllers*, *models*, *views* …), visible in Figure 2.1. Two main folders are relevant regarding the functionality of the DMS: *application* and *library*.

2.1 Folder <application>

The *application* folder organises the back end of the framework, and is composed of several PHP classes, configuration parameters and HTML templates. We easily recognise the MVC structure that builds on PHP scripts for the models, views and controllers. These scripts are saved in folders with the respective names. The folder *configs* contains the file `application.ini` which specifies the main configuration parameters, such as the database access credentials. The *data*, *layouts* and *forms* folders are used to save parts of the HTML.

2.1.1 File <Bootstrap.php>

This file defines the class `Bootstrap`, representing the top of the MVC framework. Everything that is initialised in this file is usable by every other file of the website.

Function _initDoctype ()	Define Doctype	Definition of the HTML Doctype
	Get User Info	Get the right of the user

Table 2.1: bootstrap.php

2.1.2 Folder <configs>

The *configs* folder contains the file `application.ini`. In this file, the default variables and configuration parameters are defined. For example, the path to library files is specified by `includePaths.library`, while other parameters define PHP settings (`phpSettings.display_startup_errors`) and also default values (`default.page`). Furthermore, the credentials for MySQL connections are defined in this file. The *production* parameters are always available, and will be updated when the application environment of the project is switched to *staging* or *development* in the `.htaccess` file of the website.

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0

includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
appnamespace = "Application"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
resources.frontController.params.displayExceptions = 1
resources.view[] =

resources.db.adapter = PDO_MYSQL
resources.db.params.charset = utf8
resources.db.params.host = localhost
resources.db.params.username = XXXXXXXXX
resources.db.params.password = XXXXXXXXX
resources.db.params.dbname = uniteeurope

resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts/"

default.page = 0

[staging : production]

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.frontController.params.displayExceptions = 1
```

Table 2.2: application.ini

2.1.3 Folder <controllers>

In this section, we present the **controllers** of the website, which represent its internal logic. The controller classes organise the repartition of the website's tasks. Whenever the website is called by the user, the controller functions organise the different tasks to perform the query, for example by examining the **models**. The models and their functionalities will be presented in section 2.1.7.

A Zend controller is built so that each of its functions is connected to an action and page. For example, the function `indexAction` of the `AccountController` will be called by the default account page (http://dms.uniteeurope.org/account/) whereas the `userAction` of this same controller will be called by the page http://dms.uniteeurope.org/account/user.

According to the current structure of the DMS, the different controllers are:

- AccountController
- AnnotationController
- FaqController
- IndexController
- KeywordController
- MeasureController
- MonitoringController
- OrganisationController
- SourceController

2.1.3.1 AccountController

The `AccountController` controls and manages the different account parameters of the DMS and of the UniteEurope website. The functions of the controller are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller. The actions are built around the following logic:

Test login / user right	Control if the user is currently logged in and has the right to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the following models, which are needed by the controller to perform the actions: • `Application_Model_DmsuserMapper` • `Application_Model_AccountMapper` • `Application_Model_OrganisationMapper`
Get default values from application.ini	Get the default values defined in the application.ini file.
Clean URL	Clean the URL to keep a part of the query string.
Get page	Get the current page number or set it equal to the default page value.
Define action and controller name	Define variables for the action and controller name for easier usage in the HTML script

Table 2.3: AccountController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/account/). This page manages the DMS users, i.e. users can be added, updated or deleted. The different functions called in `indexAction()` are defined in the models `Application_Model_Dmsuser` and `Application_Model_DmsuserMapper`, respectively, which are described later in this document. The actions are built around the following logic:

Get Users	Get the list of all DMS users. The function and variable used from the `Application_Model_DmsuserMapper` are `getUsers()` and `$listUsers`.
Set User	Set or update the user. Applies only if the fields *Lastname*, *Firstname*, *Email* and *Password* are correctly filled. The function used from the `Application_Model_DmsuserMapper` is *`setUser()`*.
Delete User	Delete the user corresponding to the DmsUserID sent as GET value *deluser*. The function and variable used from the `Application_Model_DmsuserMapper` is `deleteDmsUser()`.

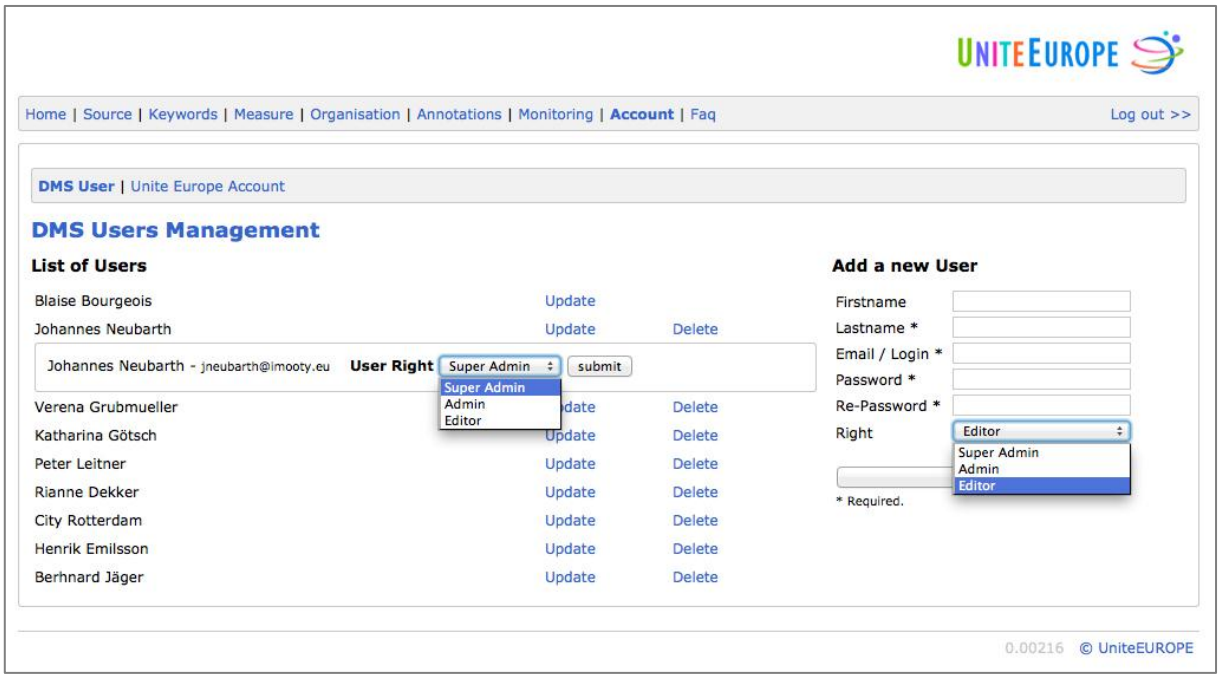

Figure 2.2: Screenshot DMS Account

Table 2.4: AccountController.indexAction()

Function userAction()

The function `userAction()` controls the user page of the controller (http://dms.uniteeurope.org/account/user). This page manages the users of UniteEurope and allows team members to add, update, delete and define authorisation details for the UniteEurope users. The functions called in `userAction()` are defined in the models `Application_Model_User`, `Application_Model_Account` and `Application_Model_AccountMapper`, respectively, and are described later in this document. The actions are built around the following logic:

Get account	Get the list of all DMS users. The function and variables used from the `Application_Model_AccountMapper` are `getAccount()`, `$currentListAccount` and `$currentUser`.
Get number of pages	Get the number of pages to display. The variable used from the `Application_Model_AccountMapper` is `$pages`.
List Product	Definition of the list of products.
Get organisation	Get the list of organisations, to allow connections between an account and its organisation. The function and variable used from the `Application_Model_OrganisationMapper` are `getOrganisation()` and `$currentListOrganisation`.
Set account	After having verified that the required field of the form is correctly filled, the subscription is processed. *Set account* is structured as follows: • define the account itself (*name*, *address*, *description*) • define the account access right • create a user with the role "*manager*" that inherits the access rights of the account • create an organisation from the account or connect an organisation to this account The function used from the `Application_Model_OrganisationMapper` is `setOrganisation()`. The functions used from the `Application_Model_AccountMapper` are `getAccessRightID()`, `setAccount()` and `setUser()`.
Update account	Update the address, the description, or the permissions of the account. The name of the account cannot be changed. The functions used from the `Application_Model_AccountMapper` are `getAccessRightID()`, `setAccount()`, `setUser()` and `setUserAccessRightID()`.
Delete account	Delete the account corresponding to the *AccountID* sent as GET value `delaccount`. The function used from the `Application_Model_AccountMapper` is `deleteAccount()`.

Table 2.5: AccountController.userAction()

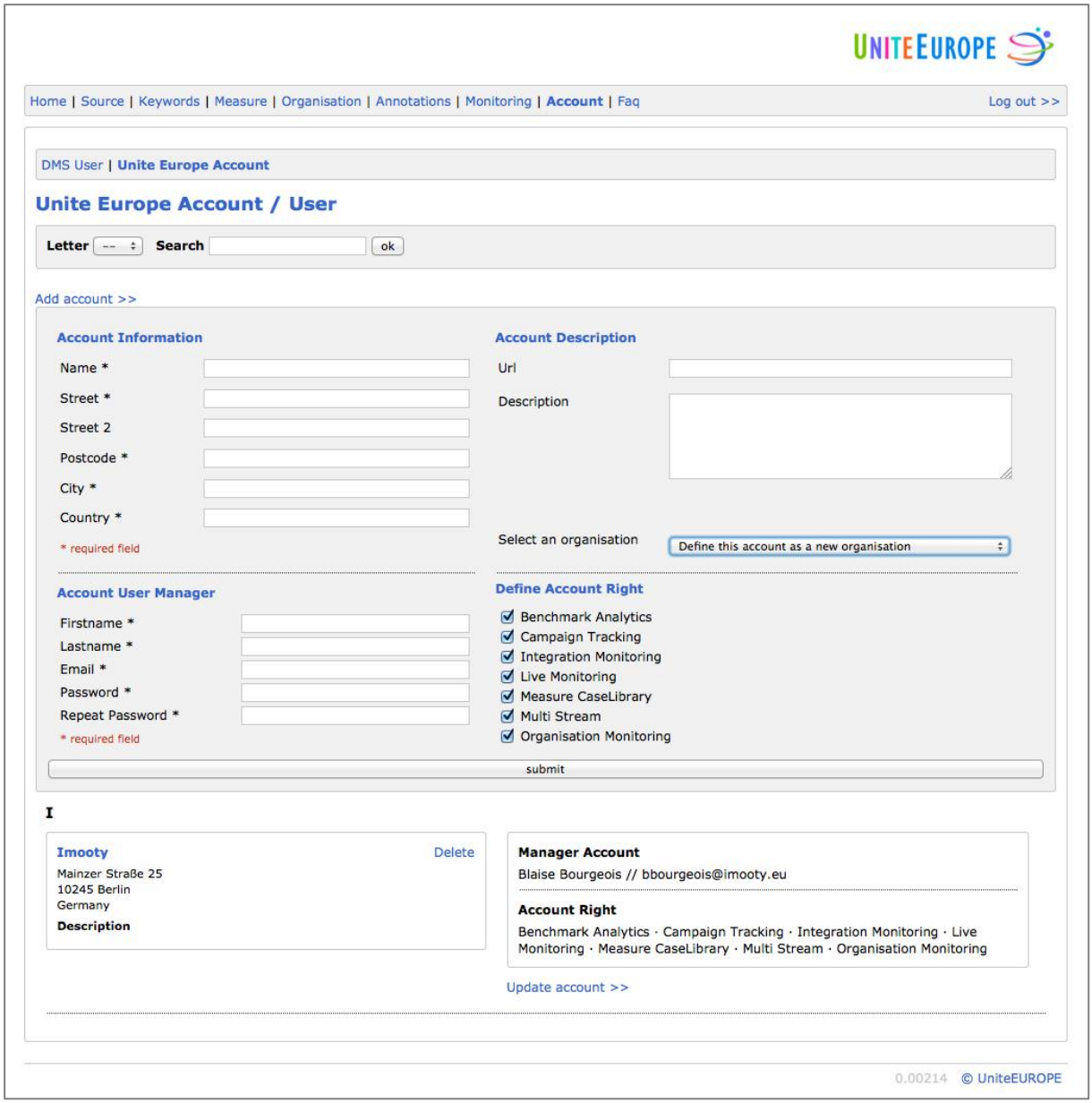

Figure 2.3: Screenshot DMS Account User

2.1.3.2 AnnotationController

The AnnotationController controls and manages the *Annotation*, *ItemAnnotation* and *ItemAnnotation Relation* parameters. The functions of the AnnotationController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller. The actions are built around the following logic:

Test login / user right	Verify if the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the models needed by the controller: • Application_Model_AnnotationMapper • Application_Model_ItemAnnotationMapper • Application_Model_AnnotationToTableMapper
Get default valued from application.ini	Get the default values defined in the application.ini file.
Define action and controller name	Define variables for the action and controller name for easier usage in the HTML script

Table 2.6: AnnotationController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/annotation/). This page manages *Annotations* and *ItemAnnotations*. It allows the user to add, update or delete *Annotations* and *ItemAnnotations*. The functions called in `indexAction()` are defined in the models `Application_Model_AnnotationMapper`, `Application_Model_ItemAnnotation Mapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document. The actions are built around the following logic:

Get Annotation	Get the list of all annotations. The function and variable used from the `Application_Model_AnnotationMapper` are `getAnnotation()` and `$currentAnnotationID`.
Set Annotation	Create a new annotation or update an old one. The function used from the `Application_Model_AnnotationMapper` is `setAnnotation()`.
Delete Annotation	Delete the Annotation corresponding to the AnnotationID sent as GET value `delannotation`. The function used from the *Application_Model_Annotation-Mapper* is `deleteAnnotation()`.
Delete ItemAnnotation	Delete the ItemAnnotation corresponding to the ItemAnnotationID send as GET value `delitem`. The function used from the `Application_Model_Item-AnnotationMapper` is `deleteItemAnnotation()`.
Get current ItemAnnotation	Get the ItemAnnotation of the selected Annotation. The function used from the `Application_Model_Item-AnnotationMapper` is `getItemAnnotation()`.

Get enabled ItemAnnotation Annotation	Get the enabled Annotation for each selected ItemAnnotation. The function used from the `Application_Model_AnnotationMapper` is `getListAnnotation()`. The functions used from the `Application_Model_ItemAnnotationMapper` are `getItemAnnotationMenu()` and `getSelectedItemAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getListAnnotation()`.
Set ItemAnnotation	Create or update an ItemAnnotation by defining the different ItemAnnotations enabled and by defining the value of the ItemAnnotation. The functions used from the `Application_Model_ItemAnnotationMapper` are `setItemAnnotation()` and `setItemAnnotationRelation()`.

Table 2.7: AnnotationController.indexAction()

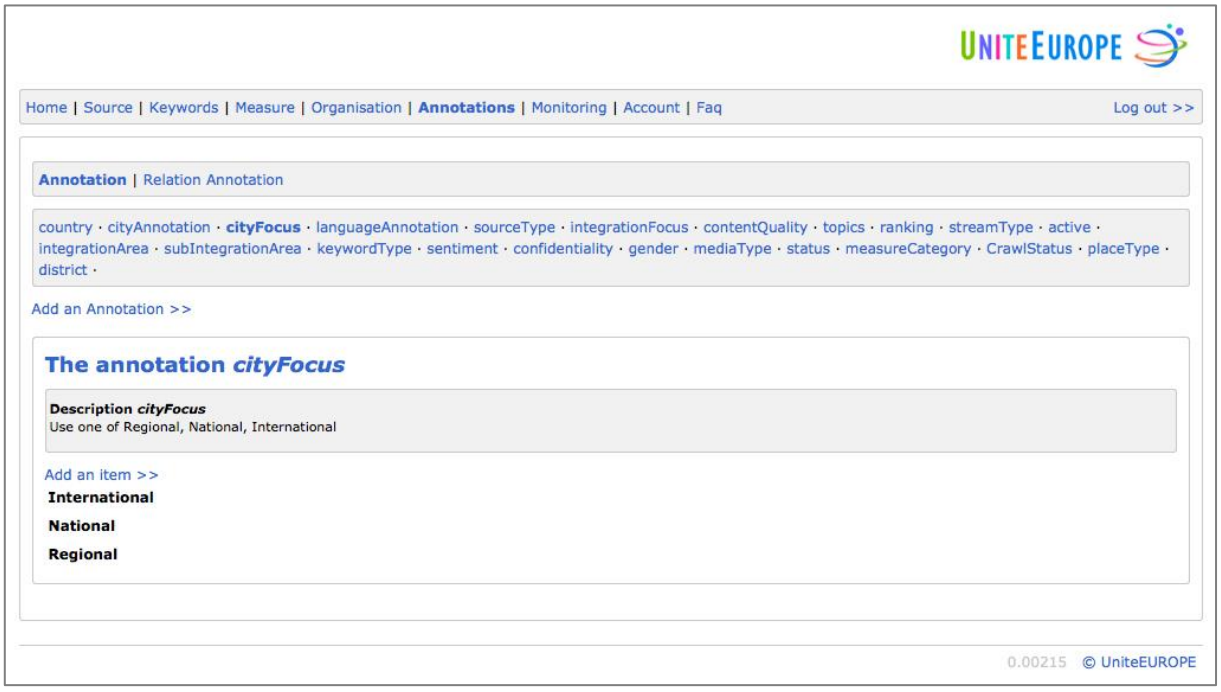

Figure 2.4: Screenshot DMS Annotation

Function relationAction()

The function `relationAction()` controls the *relation* page of the controller (http://dms.uniteeurope.org/annotation/relation). This page manages the relations between

Annotations and Elements, defining which Annotations can be applied to each Element. This relation also determines which Annotations will be enabled as filter options for the UniteEurope website dashboard.

The different functions called in `relationAction()` are defined in the models `Application_Model_AnnotationMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document. The actions are built around the following logic:

Get Annotation	Get the list of all Annotations. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Source Form	Get the list of Annotations for the element *Source*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Feed Form	Get the list of Annotations for the element *Feed*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Keyword Form	Get the list of Annotations for the element *Keyword*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Organisation Form	Get the list of Annotations for the element *Organisation*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Item Annotation Form	Get the list of Annotations for the element *ItemAnnotation*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Measure case Form	Get the list of Annotations for the element *MeasureCase*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Measure Form	Get the list of Annotations for the element *Measure*. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Get Dashboard Form	Get the list of Annotations for the UniteEurope website menu. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.
Set Relation	Set the list of Annotations for the selected Annotation. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`.

Table 2.8: AnnotationController.relationAction()

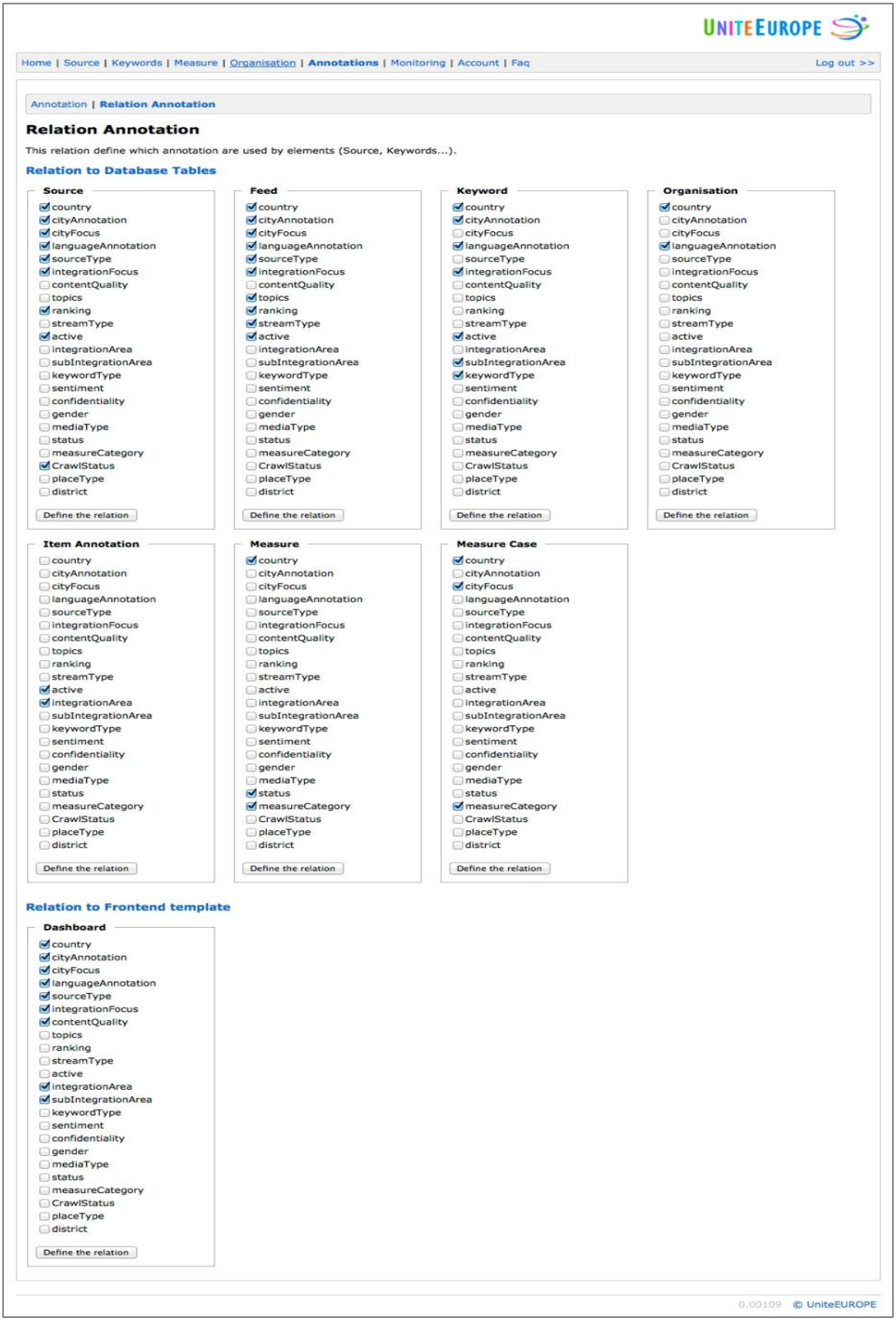

Figure 2.5: Screenshot DMS Annotation relation

2.1.3.3 FaqController

The FaqController does not manage anything, it simply displays the content of the FAQ (frequently asked questions). The different functions of the FaqController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Test login / user right	Verify that the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Define action and controller name	Define variables for the action and controller name to simplify their usage in the HTML script.

Table 2.9: FaqController.init()

Function indexAction()

The function `indexAction()` is currently empty and simply calls the default page (http://dms.uniteeurope.org/faq). The content is a static HTML file that contains the FAQ.

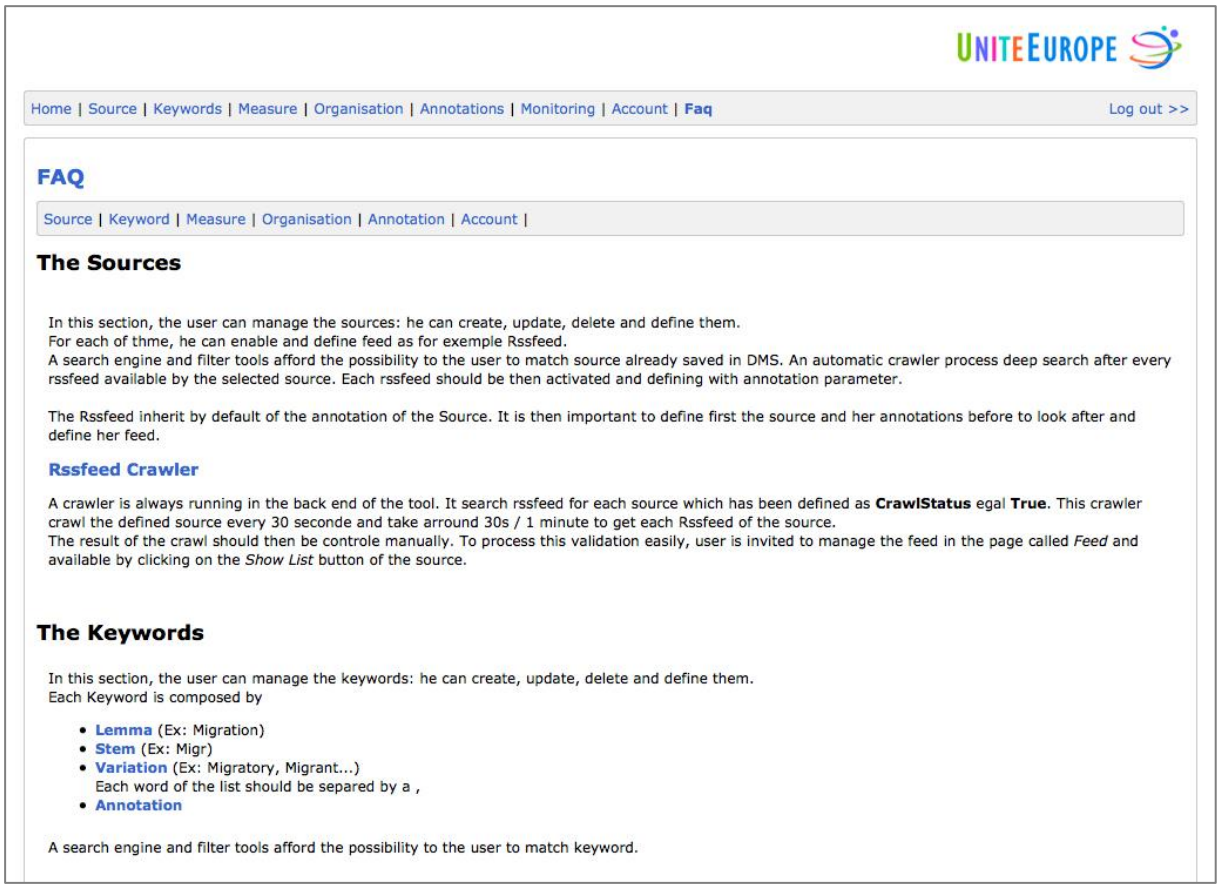

Figure 2.6: Screenshot DMS FAQ

2.1.3.4 IndexController

The IndexController controls and manages the authentication process. The functions of the IndexController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Initialisation of the models needed	Initialisation of the `Application_Model_Dmsuser-Mapper` models to process the login action.

Table 2.10: IndexController.init()

Function indexAction()

The function `indexAction()` controls the default page of the DMS website (http://dms.uniteeurope.org/) The authentication process is controlled here. The model used is `Application_Model_DmsuserMapper`, it is described later in this document.

Get Login Form	Process the authentication of the user. The function used from the `Application_Model_Dms-userMapper` is `logUser()`.

Table 2.11: IndexController.indexAction()

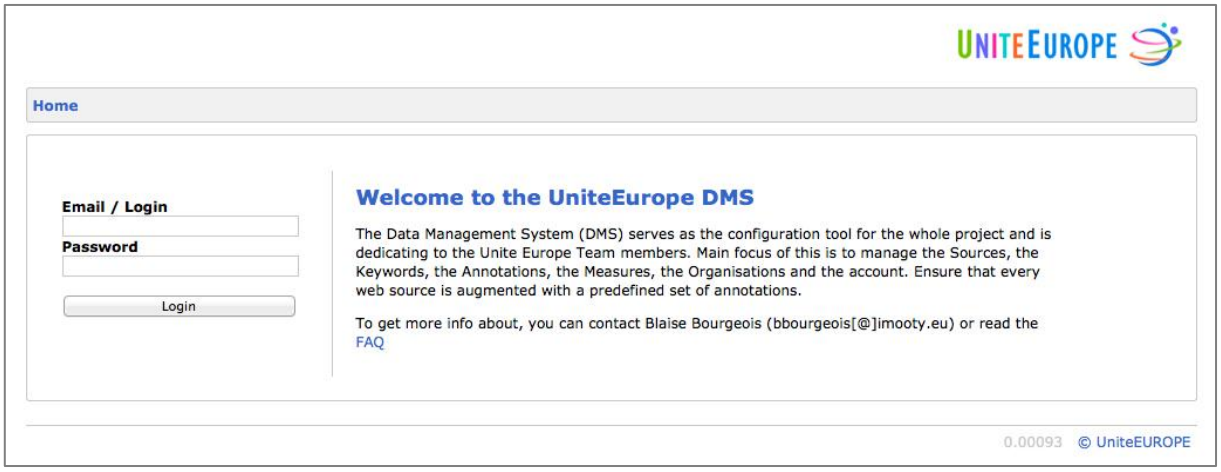

Figure 2.7: Screenshot DMS Homepage

Function logoutAction()

The function `logoutAction()` controls the logout page of the DMS website (http://dms.uniteeurope.org/index/logout). By calling this page, the authentication of the user is reinitialised.

Logging out clears the identity of the user with *Zend_Auth*. The user is then redirected to the homepage.

2.1.3.5 KeywordController

The KeywordController controls and manages the Keyword parameters. The functions of the KeywordController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Test login / user right	Verify that the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the different models needed by the controller to process the actions: • `Application_Model_KeywordMapper` • `Application_Model_ItemAnnotationMapper` • `Application_Model_AnnotationToTableMapper`
Get default values from application.ini	Get the default values defined in the application.ini file.
Clean URL	Clean the URL to keep a part of the query string.
Get page	Get the current page number or set it to the default page value.
Define action and controller name	Define variables for the action and controller name to simplify their usage in the HTML script.

Table 2.12: KeywordController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/keyword/). This page manages the keywords. It allows the user to add, update or delete keywords. The different functions called in `indexAction()` are defined in the models `Application_Model_KeywordMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Keyword	Get the list of keywords according to the filter parameters sent. The function and variable used from the `Application_Model_KeywordMapper` are `getKeyword()` and `$listKeyword`.
Number of Page	Get the total amount of pages. The variable used from the `Application_Model_KeywordMapper` is `$pages`.
Delete Keyword	Delete the keyword corresponding to the KeywordID send as GET value `delk`. The function used from the `Application_Model_KeywordMapper` is `deleteKeyword()`.
Get Annotation Menu	Get the list of annotations enabled for the keyword, as well as their respective ItemAnnotations. The function used from the `Application_Model_KeywordMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Set Keyword	Create or update a keyword by defining the different

	ItemAnnotations and the value of the keyword and its parameters. The function used from the `Application_Model_KeywordMapper` is `setKeyword()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.

Table 2.13: KeywordController.indexAction()

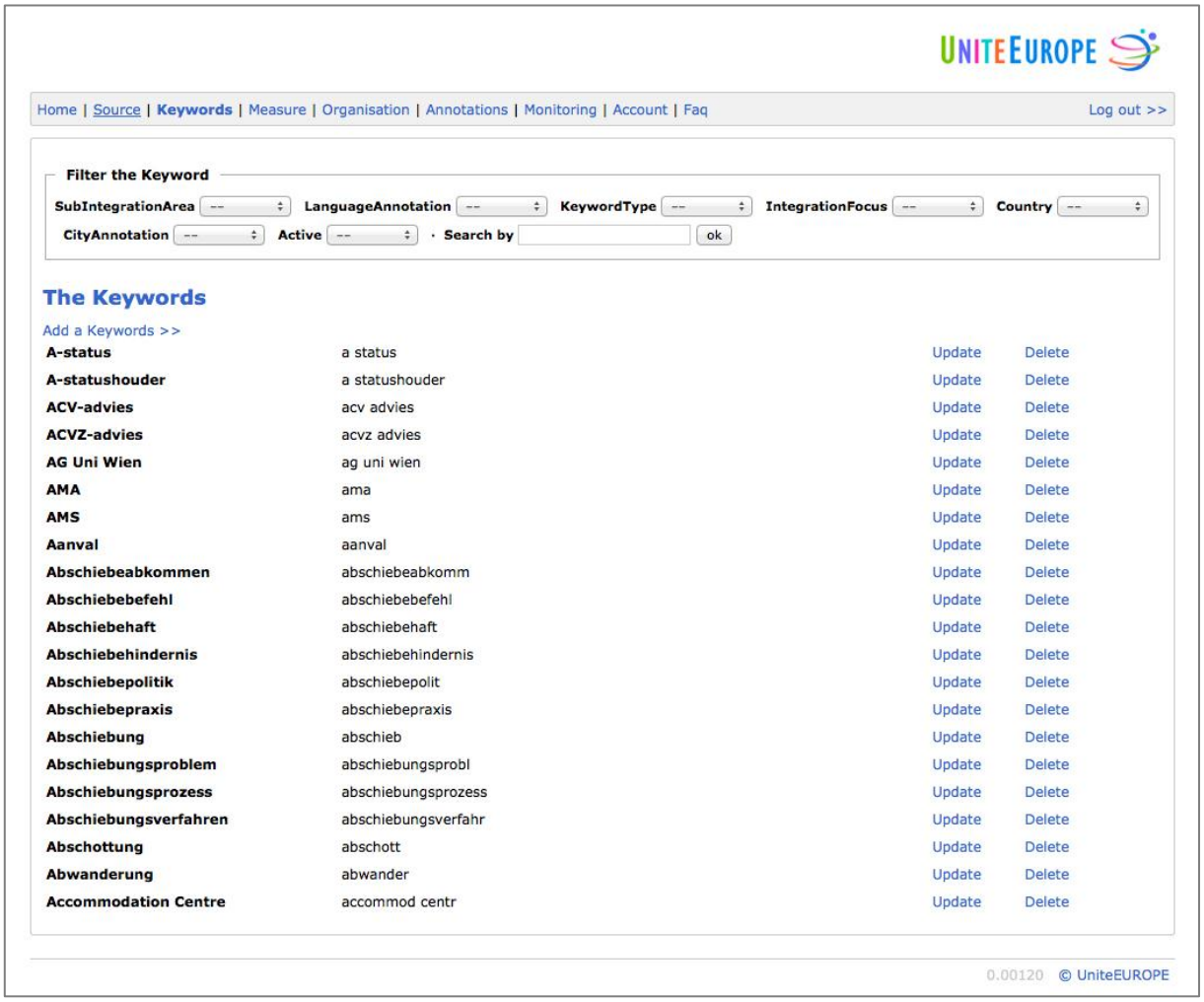

Figure 2.8: Screenshot DMS Keyword

2.1.3.6 MeasureController

The MeasureController controls and manages the different integration measure and measure case parameters. The functions of the MeasureController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Test login / user right	Verifies that the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the models needed by the controller to process the different actions: • `Application_Model_MeasureMapper` • `Application_Model_MeasureCaseMapper` • `Application_Model_OrganisationMapper` • `Application_Model_ItemAnnotationMapper` • `Application_Model_AnnotationToTableMapper`
Get default values from application.ini	Get the default values defined in the application.ini file.
Get the organisation	Get the name of the organisation. The function used from the `Application_Model_Organi-sationMapper` is `getOrganisationLabel()`.
Clean URL	Clean the URL to keep a part of the query string.
Get page	Get the current page number or set it to the default page value.
Define action and controller name	Define variables for the action and controller name to simplify their usage in the HTML script.

Table 2.14: MeasureController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/measure/). This page manages the integration measures. It allows the user to add, update or delete measures. The different functions called in `indexAction()` are defined in the models `Application_Model_MeasureMapper`, `Application_Model_MeasureCaseMapper`, `Application_Model_ItemAnnotation Mapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Measure	Get the list of measures according to the given filter parameters. The function and variable used from the `Application_Model_MeasureMapper` are `getMeasures()` and *$*`listMeasure`.
Number of Page	Get the total amount of pages. The variable used from the `Application_Model_MeasureMapper` is `$pages`.
Get Annotation Menu	Get the list of annotations enabled for the measure, as well as their respective ItemAnnotations. The function used from the `Application_Model_MeasureMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Get the Measure Case	Get the labels of an integration measure case. The function used from the `Application_Model_MeasureCaseMapper` is `getMeasureCaseLabels()`.
Set Measure	Create or update a measure by defining all necessary ItemAnnotations and the parameters of the measure. The functions used from the `Application_Model_MeasureMapper` are `setMeasure()` and `setMeasureHasMeasureCase()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.
Delete Measure	Delete the measure corresponding to the MeasureID send as GET value `delmeasure`. The function used from the `Application_Model_MeasureMapper` is `deleteMeasure()`.

Table 2.15: MeasureController.indexAction()

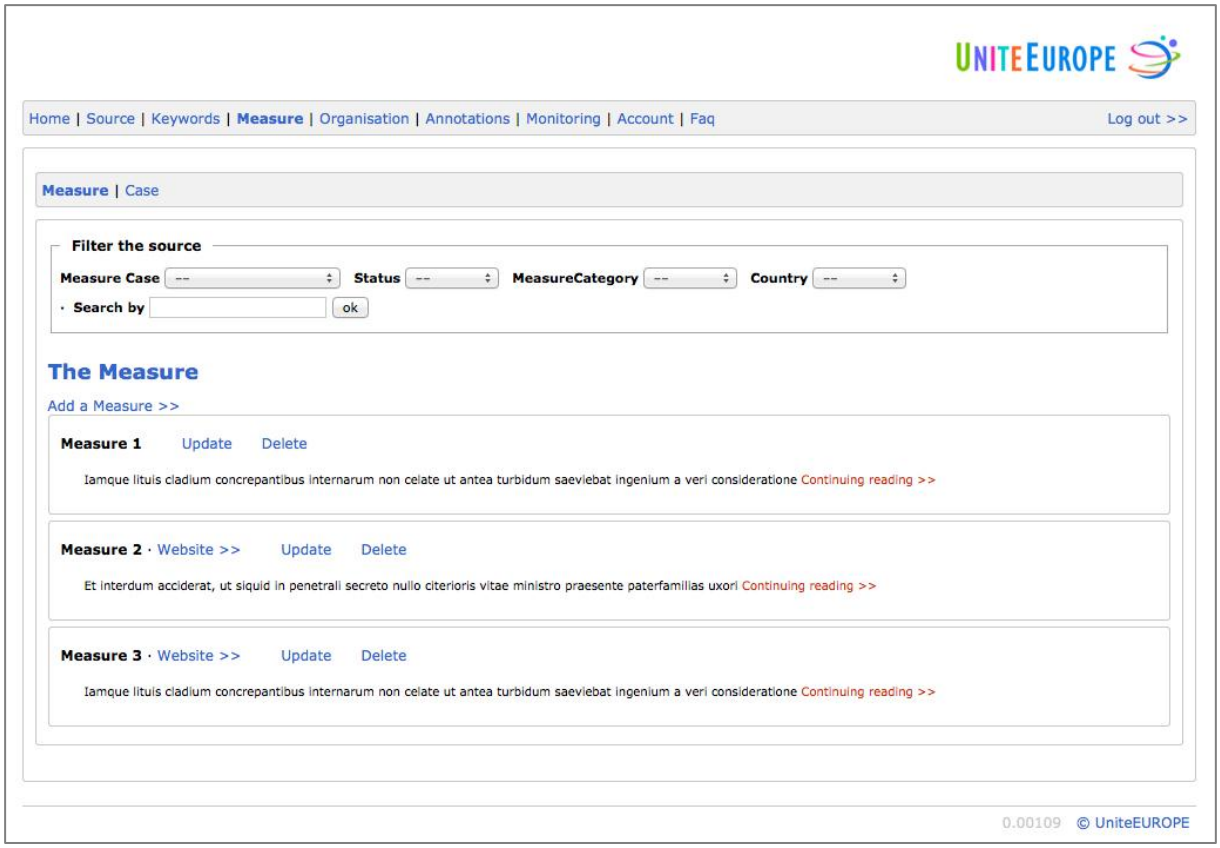

Figure 2.9: Screenshot DMS Measure (currently with dummy content)

Function caseAction()

The function `caseAction()` controls the integration measure case page of the controller (http://dms.uniteeurope.org/measure/case). This page manages the integration measure cases. It allows the user to add, update or delete a case. The functions called in `caseAction()` are defined in the models `Application_Model_MeasureMapper`, `Application_Model_MeasureCaseMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Measure Case	Get the list of measure cases according to the given filter parameters. The function and variable used from the `Application_Model_MeasureCaseMapper` are `getMeasureCases()` and `$listMeasureCase`.
Number of Pages	Get the total amount of pages. The variable used from the `Application_Model_MeasureCaseMapper` is `$pages`.
Get Annotation Menu	Get the list of annotations enabled for the case, as well as their respective ItemAnnotations. The function used from the `Application_Model_MeasureCaseMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Set Measure Case	Create or update a case by defining all necessary ItemAnnotations and the parameters of the measure. The function used from the `Application_Model_MeasureCaseMapper` is `setMeasureCase()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.
Delete Measure	Delete the case corresponding to the MeasureCaseID sent as GET value `delmeasurecase`. The function used from the `Application_Model_MeasureCaseMapper` is `deleteMeasureCase()`.

Table 2.16: MeasureController.caseAction()

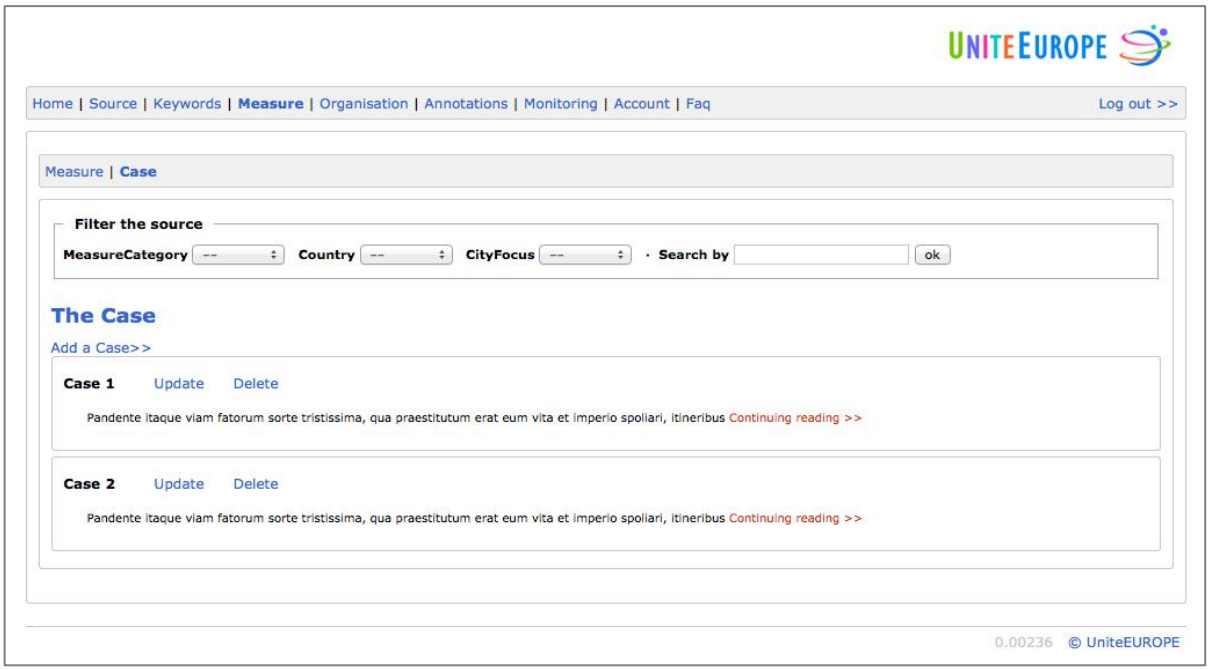

Figure 2.10: Screenshot DMS Measure Case (currently with dummy content)

2.1.3.7 MonitoringController

This part of the DMS is not yet implemented.

2.1.3.8 OrganisationController

The OrganisationController controls and manages the different organisation parameters. The functions of the OrganisationController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Test login / user right	Verify that the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the models needed by the controller to process the different actions: • `Application_Model_OrganisationMapper` • `Application_Model_ItemAnnotationMapper` • `Application_Model_AnnotationToTableMapper`
Get default values from application.ini	Get the default values defined in the application.ini file.
Clean URL	Clean the URL to keep a part of the query string.
Get page	Get the current page number or set it to the default page value.
Define action and controller name	Define variables for the action and controller name to simplify their usage in the HTML script.

Table 2.17: OrganisationController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/organisation/). This page manages the organisations. It allows the user to add, update or delete an organisation. The different functions called in `indexAction()` are defined in the models `Application_Model_OrganisationMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Organisation	Get the list of organisations according to the given filter parameters. The function and variable used from the `Application_Model_OrganisationMapper` are `getOrganisation()` and `$currentListOrganisation`.
Number of Pages	Get the total amount of pages. The variable used from the `Application_Model_OrganisationMapper` is `$pages`.
Get Annotation Menu	Get the list of annotations enabled for the organisation, as well as their respective ItemAnnotations. The function used from the `Application_Model_OrganisationMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Set Organisation	Create or update an organisation by defining all necessary ItemAnnotations and the parameters of the organisation. The function used from the `Application_Model_OrganisationMapper` is `setOrganisation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.
Delete Organisation	Delete the organisation corresponding to the OrganisationID sent as GET value `delorganisation`. The function used from the `Application_Model_OrganisationMapper` is `deleteOrganisation()`.

Table 2.18: OrganisationController.indexAction()

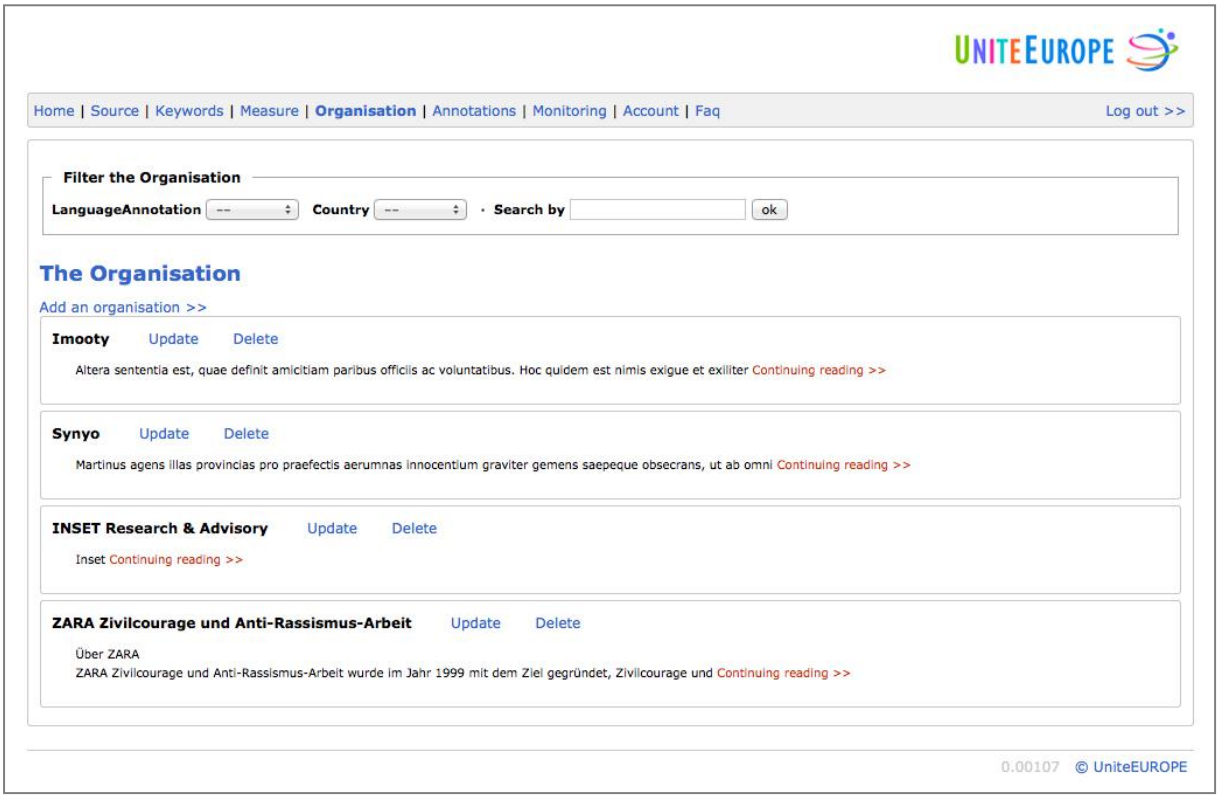

Figure 2.11: Screenshot DMS Organisation

2.1.3.9 SourceController

The SourceController controls and manages the web source and web feed parameters. The functions of the SourceController are described below.

Function init()

The function `init()` initialises the controller and defines variables that will be active in every page of the controller.

Test login / user right	Verify that the user is currently logged in and has the permission to access the page. If not, he is redirected to the home page.
Initialisation of the models needed	Initialisation of the models needed by the controller to process the different actions: • `Application_Model_SourceMapper` • `Application_Model_FeedMapper` • `Application_Model_ItemAnnotationMapper` • `Application_Model_AnnotationToTableMapper`
Get default values from application.ini	Get the default values defined in the application.ini file.
Clean URL	Clean the URL to keep a part of the query string.
Get page	Get the current page number or set it to the default page value.
Define action and controller name	Define variables for the action and controller name to simplify their usage in the HTML script.

Table 2.19: SourceController.init()

Function indexAction()

The function `indexAction()` controls the default page of the controller (http://dms.uniteeurope.org/source/). This page manages the web sources and their feeds. It allows the user to add, update or delete a source. Furthermore, feeds can be added, updated or deleted to/from each source. The functions called in `indexAction()` are defined in the models `Application_Model_SourceMapper`, `Application_Model_FeedMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Sources	Get the list of sources according to the given filter parameters. The function and variable used from the `Application_Model_SourceMapper` are `getSources()` and `$entries`.
Number of Pages	Get the total amount of pages. The variable used from the `Application_Model_SourceMapper` is `$pages`.
Get Feeds	Get all feeds for the selected source. The function and variable used from the `Application_Model_SourceMapper` are `getSourcesFeeds()` and `$listFeed`.

Get Annotation Menu	Get the list of annotations enabled for the source, as well as their respective ItemAnnotations. The function used from the `Application_Model_SourceMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_Item-AnnotationMapper` is `getItemAnnotationMenu()`.
Delete Source	Delete the source corresponding to the SourceID sent as GET value `del`. The function used from the `Application_Model_SourceMapper` is `deleteSource()`.
Set Source	Create or update a source by defining all necessary ItemAnnotations and the parameters of the source. The function used from the `Application_Model_SourceMapper` is `setSource()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.
Get Annotation Feed	Get the list of annotations enabled for the feed of each source, as well as their respective ItemAnnotations. The function used from the `Application_Model_FeedMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_Item-AnnotationMapper` is `getItemAnnotationMenu()`.
Set Feed	Create or update a feed by defining all necessary ItemAnnotations and the parameters of the feed. The function used from the `Application_Model_FeedMapper` is `setFeed()`. The function used from the `Application_Model_Item-AnnotationMapper` is `setItemAnnotation-Relation()`.
Delete Feed	Delete the feed corresponding to the FeedID sent as GET value `delfeed`. The function used from the `Application_Model_Feed-Mapper` is `deleteFeed()`.

Table 2.20: SourceController.indexAction()

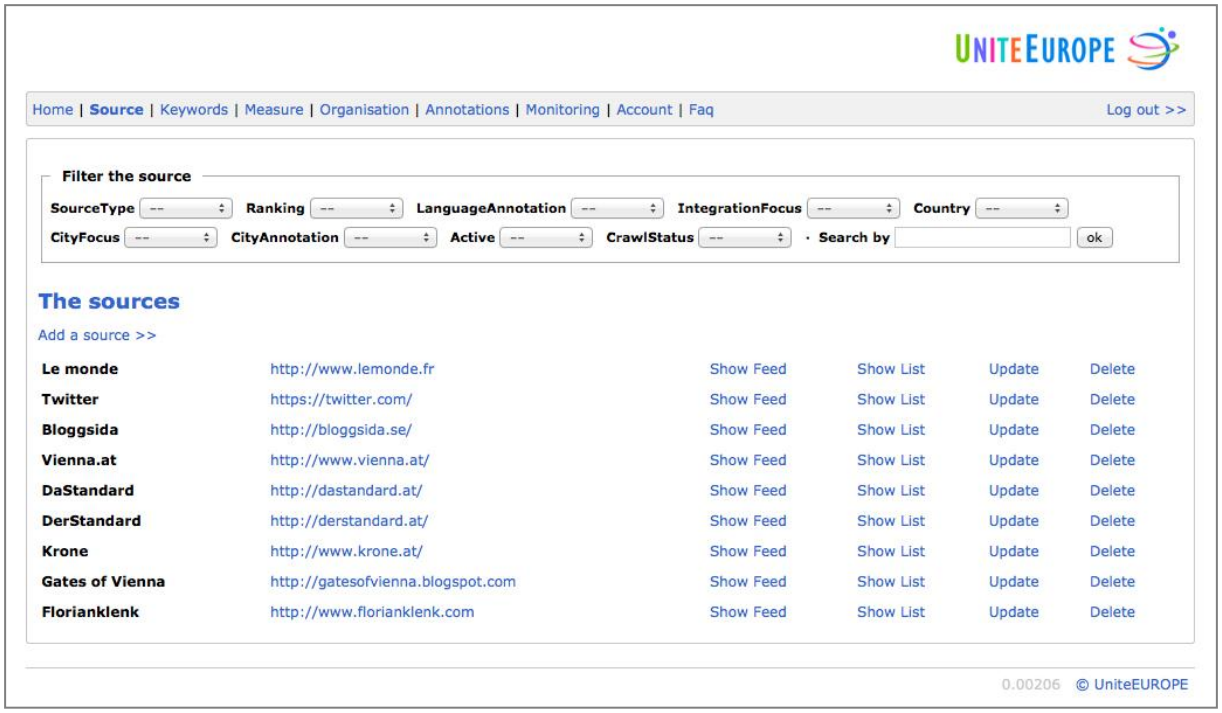

Figure 2.12: Screenshot DMS Source

Function feedAction()

The function `feedAction()` controls the *web feed* page of the controller (http://dms.uniteeurope.org/source/feed). This page manages all feeds for a certain source. It allows the user to add, update or delete feeds of the selected source. The different functions called in `feedAction()` are defined in the models `Application_Model_SourceMapper`, `Application_Model_FeedMapper`, `Application_Model_ItemAnnotationMapper` and `Application_Model_AnnotationToTableMapper`, respectively, which are described later in this document.

Get Source	Get the parameters of a source. The function and variable used from the `Application_Model_SourceMapper` are `getSource()` and `$currentSourceID`.
Get Annotation Menu	Get the list of annotations enabled for the source, as well as their respective ItemAnnotations. The function used from the `Application_Model_SourceMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Get Feeds	Get the feeds of the selected source.

	The function and variable used from the `Application_Model_SourceMapper` are `getSourcesFeeds()` and `$listFeed`
Get Annotation Feed	Get the list of annotations enabled for the feed of each source, as well as their respective ItemAnnotations. The function used from the `Application_Model_FeedMapper` is `getSelectedItemAnnotation()`. The function used from the `Application_Model_AnnotationToTableMapper` is `getListAnnotation()`. The function used from the `Application_Model_ItemAnnotationMapper` is `getItemAnnotationMenu()`.
Set Feed	Create or update a feed by defining all necessary ItemAnnotations and the parameters of the feed. The function used from the `Application_Model_FeedMapper is setFeed()`. The function used from the `Application_Model_ItemAnnotationMapper` is `setItemAnnotationRelation()`.
Delete Feed	Delete the feed corresponding to the FeedID sent as GET value `delfeed`. The function used from the `Application_Model_FeedMapper` is `deleteFeed()`

Table 2.21: SourceController.feedAction()

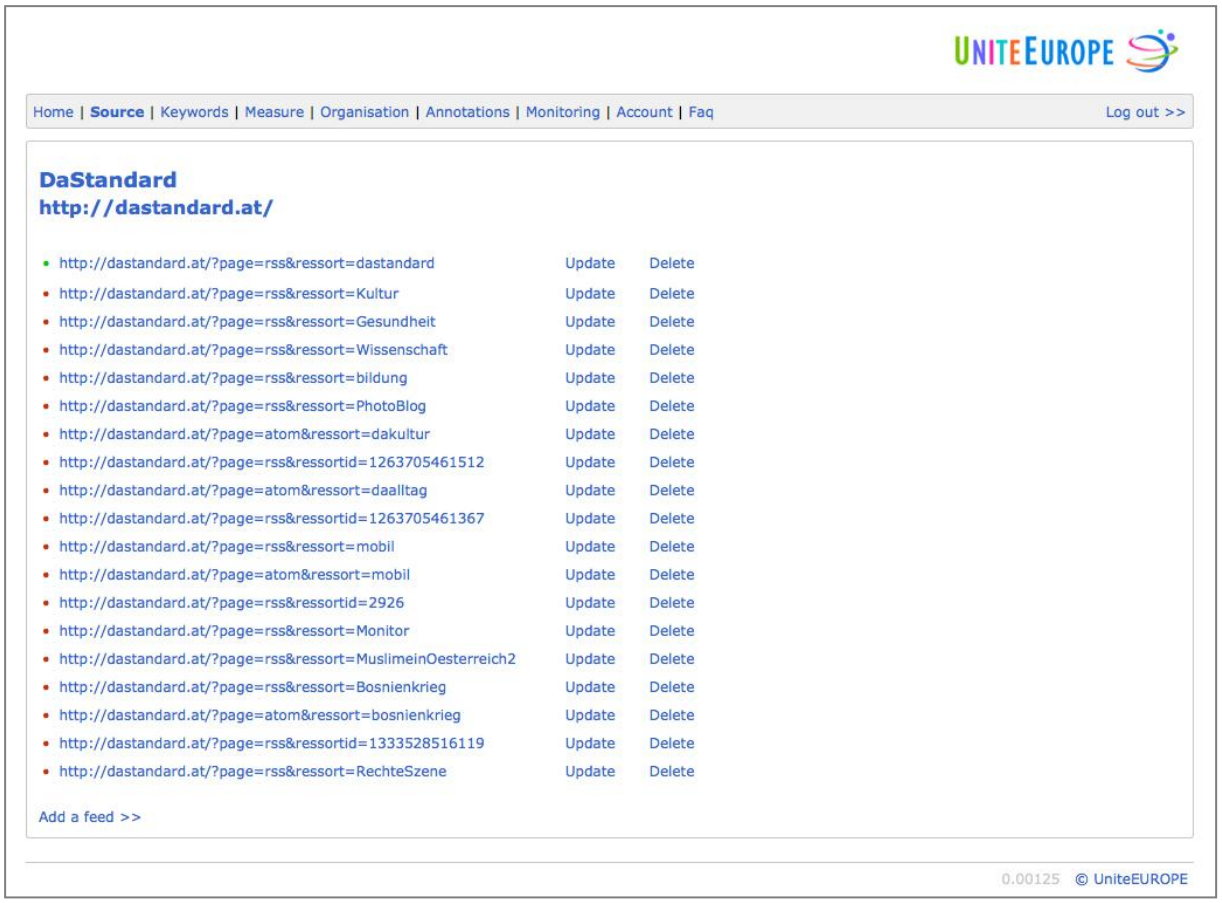

Figure 2.13: Screenshot DMS Source Feed

2.1.4 Folder <data>

The Folder *data* contains files used by the Zend Framework, such as log files or session files.

2.1.5 Folder <forms>

The Folder *forms* contains PHP classes created with *Zend_form*. Two classes have been created for our project: `Login` and `Standard`.

2.1.6 Folder <layouts>

The folder *layouts* contains HTML templates used for the DMS. These represent the main structure of the HTML page, which is then filled by each specific controller views script.

2.1.7 Folder <models>

The folder *models* contains every object class and mapper class of the website. The models consist of application data and business rules. Regarding the aim of this deliverable, we are going into detail for each of them.

Two kinds of models have been programmed, which are either processing object definitions or mapper processes. The object classes define every component of an object, whereas the mapper classes define rules.

The following classes have been implemented:

- `Application_Model_Account`
- `Application_Model_AccountMapper`
- `Application_Model_Annotation`
- `Application_Model_AnnotationMapper`
- `Application_Model_AnnotationToTable`
- `Application_Model_AnnotationToTableMapper`
- `Application_Model_Dmsuser`
- `Application_Model_DmsuserMapper`
- `Application_Model_Feed`
- `Application_Model_FeedMapper`
- `Application_Model_ItemAnnotation`
- `Application_Model_ItemAnnotationMapper`
- `Application_Model_Keyword`
- `Application_Model_KeywordMapper`
- `Application_Model_Measure`
- `Application_Model_MeasureMapper`
- `Application_Model_MeasureCase`
- `Application_Model_MeasureCaseMapper`
- `Application_Model_Organisation`
- `Application_Model_OrganisationMapper`
- `Application_Model_Source`
- `Application_Model_SourceMapper`
- `Application_Model_User`

2.1.7.1 Application_Model_Account

`Application_Model_Account` allows us to create the account object and define it by assigning a name, an address, and some third parameter, such as URL or description. Each account has also a role, an access right definition and a relation to an organisation.

Parameters of the object

- AccountID: MySQL primary key of the account object
- Name
- Description
- Street
- Street2
- Postcode
- City
- Country
- URL
- AccessRight: MySQL foreign key relation to the *DmsAccessRight* table
- OrganisationID: foreign key relation to the *DmsOrganisation* table
- Role

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
AccountID	`setAccountID($accountID)`	`getAccountID()`
Name	`setName($name)`	`getName()`
Description	`setDescription($description)`	`getDescription()`
…	…	…

Table 2.22: Application_Model_Account

2.1.7.2 Application_Model_AccountMapper

`Application_Model_AccountMapper` contains several methods for managing accounts.

Parameters of the object

- currentAccountIDs: List of the selected AccountIDs
- currentListAccount: List of the selected accounts
- pages: Total amount of pages, based on the amount of available results
- amountResult: Amount of results returned by getAccount; defaults to 10

Methods of the object

Function name	Description
`getAccount()`	Get the list of accounts filtered by characters or search terms. Results are sorted alphabetically by the account name. The function returns the full account information in the variable `$currentListAccount`, and every AccountID

	in the variable `$currentAccountIDs`.
`setAccessRight()`	Define a specific access permission composed of a role name and a list of available pages. A role name is either a “manager” or “user”. The permission is a list of one or multiple parts of our product: • Benchmark Analytics • Campaign Tracking • Integration Monitoring • Live Monitoring • Measure • Multi Stream • Organisation Monitoring AccessRight is a unique combination of role name and permission.
`getAccessRightID()`	Given a role name and permission, return the AccessRightID of the combination.
`setAccount()`	Create the account parameters in the table *DmsAccount,* or update them if the given AccountID already exists.
`setUser()`	Create the user parameters in the table *DmsUser*, or update them if the given UserID already exists.
`getUser()`	Get the list of user objects whose role equals “manager” and return it as `$currentListUser`.
`setUserAccessRightID()`	Create or update the AccessRightID for the user.
`deleteAccount()`	Delete an account, given its AccountID.

Table 2.23: Application_Model_AccountMapper

2.1.7.3 Application_Model_Annotation

`Application_Model_Annotation` allows us to create annotation objects and define them by giving them a label and a description.

Parameters of the object

- AnnotationID: MySQL primary key of the table *DmsAnnotation*
- Label
- Description

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
AnnotationID	`setAnnotationID($AnnotationID)`	`getAnnotationID()`
Label	`setLabel($label)`	`getLabel()`
Description	`setDescription($description)`	`getDescription()`

Table 2.24: Application_Model_Annotation

2.1.7.4 Application_Model_AnnotationMapper

`Application_Model_AnnotationMapper` contains several methods for managing annotations.

Methods of the object

Function name	Description
`getAnnotation()`	Get an annotation from *DmsAnnotation* and return it as an object.
`getAnnotationByID ()`	Get an annotation from *DmsAnnotation* and return it as a map where the key corresponds to the label of the annotation.
`setAnnotation ()`	Create the annotation in the table *DmsAnnotation*, or update it if the given AnnotationID already exists.
`deleteAccount()`	Delete an annotation given its AnnotationID.

Table 2.25: Application_Model_AnnotationMapper

2.1.7.5 Application_Model_AnnotationToTable

`Application_Model_AnnotationToTable` allows us to create AnnotationToTable objects and define their parameters.

Parameters of the object

- AnnotationToTableID: MySQL primary key of the table *Dms AnnotationToTable*
- TableName
- ListAnnotation

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
AnnotationToTableID	`setAnnotationToTableID( $AnnotationToTableID)`	`getAnnotationToTableID()`
TableName	`setTableName($tableName)`	`getTableName()`
ListAnnotation	`setListAnnotation( $ListAnnotation)`	`getListAnnotation()`

Table 2.26: Application_Model_AnnotationToTable

2.1.7.6 Application_Model_AnnotationToTableMapper

`Application_Model_AnnotationToTableMapper` contains several methods for managing AnnotationToTable objects.

Methods of the object

Function name	Description
`getListAnnotation()`	Get the list of annotations related to a given table.
`setListAnnotation()`	Create a list of annotations.

Table 2.27: Application_Model_AnnotationToTableMapper

2.1.7.7 Application_Model_Dmsuser

`Application_Model_Dmsuser` allows us to create a user object in the DMS and to define its parameters.

Parameters of the object

- DMSUserID: MySQL primary key of the table *DmsDMSUser*
- Firstname
- Lastname
- Email
- Password
- Right

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
DmsUserID	`setDmsUserID($dmsUserID)`	`getDmsUserID()`
Firstname	`setFirstname($firstname)`	`getFirstname()`
…	…	…

Table 2.28: Application_Model_Dmsuser

2.1.7.8 Application_Model_DmsuserMapper

`Application_Model_DmsuserMapper` contains several methods for managing DmsUser objects.

Parameters of the object

- currentUsers: List of the user objects selected
- pages: Total amount of pages based on the amount of available results
- amountResult: Amount of results returned by `getAccount`. Defaults to 10.

Methods of the object

Function name	Description
`setUser()`	Create the user object in the table *DmsDMSUser*, or update it if the given DmsUserID already exists. If the user is the last remaining "superadmin", the permissions of the user cannot be changed. Return the DmsUserID.
`logUser()`	Check the authentication of a user based on the user's email and password. Return true if authentication succeeded, or false otherwise.
`getAuthAdapter()`	Get authentication parameters and return them.
`getUser()`	Get the user or a list of users and return it as user object(s).
`getSuperAdmin()`	Get the list of "superadmin" users.
`deleteDmsUser()`	Delete a user given its DMSUserID, unless it is the last remaining user.

Table 2.29: Application_Model_DmsuserMapper

2.1.7.9 Application_Model_Feed

`Application_Model_Feed` allows us to create web feed objects and define their parameters.

Parameters of the object

- FeedID: MySQL primary key in the table *DmsFeed*
- SourceID
- Url

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
FeedID	`setFeedID($feedID)`	`getFeedID()`
SourceID	`setSourceID($sourceID)`	`getSourceID()`
…	…	…

Table 2.30: Application_Model_Feed

2.1.7.10 Application_Model_FeedMapper

`Application_Model_FeedMapper` contains several methods for managing web feed objects.

Methods of the object

Function name	Description
`getFeed()`	Get every feed of a source.
`getSelectedItemAnnotation`	Retrieve all ItemAnnotations for each selected feed. Return a map where the key corresponds to the FeedID and the value is an array that contains every ItemAnnotationID of the feed.
`setFeed()`	Create the feed object in the table *DmsFeed*, or update it if the given FeedID already exists.
`deleteFeed()`	Delete a feed given its FeedID.

Table 2.31: Application_Model_FeedMapper

2.1.7.11 Application_Model_ItemAnnotation

`Application_Model_ItemAnnotation` allows us to create ItemAnnotation objects and define their parameters.

Parameters of the object

- ItemAnnotationID: MySQL primary key of the table *DmsItemAnnotation*
- AnnotationID
- Label
- Value

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
ItemAnnotationID	`setItemAnnotationID ($itemAnnotationID)`	`getItemAnnotationID()`
AnnotationID	`setAnnotationID($AnnotationID)`	`getAnnotationID()`
…	…	…

Table 2.32: Application_Model_ItemAnnotation

2.1.7.12 Application_Model_ItemAnnotationMapper

`Application_Model_ItemAnnotationMapper` contains several methods for managing DmsUser objects.

Parameters of the object

- currentItemAnnotationIDs: List of the current ItemAnnotationIDs

Methods of the object

Function name	Description
`setItemAnnotation()`	Create the ItemAnnotation in the table *DmsItemAnnotation*, or update it if the given ItemAnnotationID already exists.
`setItemAnnotationRelation()`	Create a relation between an item (Source, Keyword, Organisation, Measure …) and an ItemAnnotation in the MySQL table [*ITEM]HasItemAnnotation,* where *ITEM* is the type of the item. For example, the table *DmsSourceHasItemAnnotation* contains every relation between a source and its ItemAnnotations.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each ItemAnnotation.
`getItemAnnotation()`	Get all ItemAnnotations per Annotation. Return `$currentItemAnnotationIDs`, containing all selected ItemAnnotationIDs.
`getItemAnnotationMenu()`	Get the list of Annotations and ItemAnnotations, with the primary key as the AnnotationID and the secondary key being the ItemAnnotation label.
`deleteItemAnnotation()`	Delete an ItemAnnotation given its ItemAnnotationID.

Table 2.33: Application_Model_ItemAnnotationMapper

2.1.7.13 Application_Model_Keyword

`Application_Model_Keyword` allows us to create keyword objects and define their parameters.

Parameters of the object

- KeywordID: MySQL primary key of the table *DmsKeyword*
- Lemma
- Stem
- WordVariation

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
KeywordID	`setKeywordID($keywordID)`	`getKeywordID()`
…	…	…

Table 2.34: Application_Model_Keyword

2.1.7.14 Application_Model_KeywordMapper

`Application_Model_KeywordMapper` contains several methods for managing DmsKeyword objects.

Parameters of the object

- listKeyword: List of selected keyword objects
- currentKeywordIDs: List of selected KeywordIDs
- pages: Total amount of pages based on the amount of available results
- amountResult: Amount of results returned by `getAccount`. Defaults to 20.

Methods of the object

Function name	Description
`getKeyword()`	Get a list of keywords according to the given filter parameters. Return `$listKeyword`, `$currentKeywordIDs` and `$pages`.
`setKeyword()`	Create the keyword in the table *DmsKeyword*, or update it if the given KeywordID already exists.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each keyword.
`deleteKeyword()`	Delete a keyword given its KeywordID.

Table 2.35: Application_Model_KeywordMapper

2.1.7.15 Application_Model_Measure

`Application_Model_Measure` allows us to create measure objects and define their parameters.

Parameters of the object

- MeasureID: MySQL primary key of the table *DmsMeasure*
- Label
- Description
- URL
- DateBegin
- DateEnd
- OrganisationID

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
MeasureID	`setMeasureID($measureID)`	`getMeasureID()`
…	…	…

Table 2.36: Application_Model_Measure

2.1.7.16 Application_Model_MeasureMapper

`Application_Model_MeasureMapper` contains several methods to manage DmsMeasure objects.

Parameters of the object

- listMeasure: List of selected measure objects
- listMeasureID: List of selected MeasureIDs
- pages: Total amount of pages based on the amount of available results
- amountResult: Amount of results returned by `getAccount`. Defaults to 20.

Methods of the object

Function name	Description
`getMeasure()`	Get a measure object given its MeasureID.
`getMeasureHasMeasureCase()`	Get the MeasureCaseIDs for each measure.
`setMeasure()`	Create the measure in the table *DmsMeasure*, or update it if the given MeasureID already exists.
`setMeasureHasMeasureCase()`	In the table *DmsMeasureHasMeasureCase*, create a MeasureCaseID for every measure.
`getMeasures()`	Get the list of measures according to the given filter parameters. Return `$listMeasure`, `$listMeasureID` and `$pages`.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each measure.
`deleteMeasure()`	Delete a measure given its MeasureID.

Table 2.37: Application_Model_MeasureMapper

2.1.7.17 Application_Model_MeasureCase

`Application_Model_MeasureCase` allows us to create MeasureCase objects and define their parameters.

Parameters of the object

- MeasureCaseID: MySQL primary key of the table *DmsMeasureCase*
- Label

- Description
- URL
- DateBegin
- DateEnd
- OrganisationID

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
MeasureCaseID	`setMeasureCaseID($measureCaseID)`	`getMeasureCaseID()`
…	…	…

Table 2.38: Application_Model_MeasureCase

2.1.7.18 Application_Model_MeasureCaseMapper

`Application_Model_MeasureCaseMapper` contains several methods for managing DmsMeasureCase objects.

Parameters of the object

- listMeasureCase: List of selected MeasureCase objects
- listMeasureCaseID: List of selected MeasureCaseIDs
- pages: Total amount of pages based on the amount of available result
- amountResult: Amount of results returned by `getAccount`. Defaults to 20.

Methods of the object

Function name	Description
`getMeasureCase()`	Get a MeasureCase object given its MeasureCaseID.
`setMeasureCase()`	Create the MeasureCase in the table *DmsMeasureCase*, or update it if the given MeasureCaseID already exists.
`getMeasureCaseLabels()`	Get all MeasureCase labels.
`getMeasureCases()`	Get the list of MeasureCase objects according to the given filter parameters. Return `$listMeasureCase`, `$listMeasureCaseID` and `$pages`.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each MeasureCase.
`deleteMeasure()`	Delete a MeasureCase given its MeasureCaseID.

Table 2.39: Application_Model_MeasureCaseMapper

2.1.7.19 Application_Model_Organisation

`Application_Model_Organisation` allows us to create organisation objects and define their parameters.

Parameters of the object

- OrganisationID: MySQL primary key of the table *DmsOrganisation*
- Name
- Description
- URL

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	***Set* function**	***Get* function**
OrganisationID	`setOrganisationID($OrganisationID)`	`getOrganisationID()`
…	…	…

Table 2.40: Application_Model_Organisation

2.1.7.20 Application_Model_OrganisationMapper

`Application_Model_OrganisationMapper` contains several methods for managing DmsOrganisation objects.

Parameters of the object

- currentListOrganisation: List of selected Organisation objects
- currentOrganisationIDs: List of selected OrganisationIDs
- pages: Total amount of pages based on the amount of available results
- amountResult: Amount of results returned by `getAccount`. Defaults to 10.

Methods of the object

Function name	Description
`setOrganisation()`	Create the organisation in the table *DmsOrganisation*, or update it if the given OrganisationID already exists.
`getOrganisationLabel()`	Get all organisation labels.
`getOrganisation()`	Get the list of organisations according to the given filter parameters. Return `$currentListOrganisation`, `$currentOrganisationIDs` and `$pages`.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each organisation.
`deleteOrganisation()`	Delete an organisation given its OrganisationID.

Table 2.41: Application_Model_OrganisationMapper

2.1.7.21 Application_Model_Source

`Application_Model_Source` allows us to create source objects and define their parameters.

Parameters of the object

- SourceID: MySQL primary key of the table *DmsSource*
- Label
- URL

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
SourceID	`setSourceID($sourceID)`	`getSourceID()`
…	…	…

Table 2.42: Application_Model_Source

2.1.7.22 Application_Model_SourceMapper

`Application_Model_SourceMapper` contains several methods for managing DmsSource objects.

Parameters of the object

- entries: List of selected sources
- currentSourceIDs: List of selected SourceIDs
- listFeed: List of feeds of the selected source

- currentFeedIDs: List of FeedIDs of the selected source
- pages: Total amount of pages based on the amount of available results
- amountResult: Amount of results returned by `getAccount`. Defaults to 20.

Methods of the object

Function name	Description
`setSource()`	Create the source in the table *DmsSource*, or update it if the given SourceID already exists.
`getSource()`	Get a source object given its SourceID.
`getSources()`	Get the list of sources according to the given filter parameters. Return `$entries`, `$currentSourceIDs` and `$pages`.
`getSourcesFeeds()`	Get the feeds with all their parameters (URL, ItemAnnotations) for every selected source. Return `$listFeed`, `$currentFeedIDs`.
`getSelectedItemAnnotation()`	Get the ItemAnnotations enabled for each source.
`deleteSource ()`	Delete a source given its SourceID.

Table 2.43: Application_Model_SourceMapper

2.1.7.23 Application_Model_User

`Application_Model_User` allows us to create user objects and define their parameters.

Parameters of the object

- UserID: MySQL primary key of the table *DmsUser*
- Firstname
- Lastname
- Email
- Right
- AccessRightID
- Role
- Password

Methods of the object

These methods are composed of a `set` function to define the object attribute value and a `get` function to get the object attribute.

Name attribute	*Set* function	*Get* function
UserID	`setUserID($userID)`	`getUserID()`
…	…	…

Table 2.44: Application_Model_User

2.2 Folder <library>

Special utility applications, such as the DMS Crawler, are collected in the *library* folder of the Zend Framework.

2.2.1 Source Crawler

The Source Crawler is a powerful way of getting all RSS, Atom or XML feeds for a web source. The aim of this tool is to retrieve all these source feeds as quick as possible, and store them as feed objects in the DMS.

The Source Crawler is running every 30 seconds. A special source annotation called *CrawlStatus* allows the crawler to determine which sources should be crawled. The value of this annotation is *True* when a crawling iteration is desired, and is manually set by the UniteEurope team members in the DMS.

The crawler implements the following methodology, illustrated in Figure 2.14:

1. Get the URL of the source saved in the DMS.
2. Get the annotations of the source, which are also defined in the DMS.
3. Crawl the first page of the source to collect its URL (Level 1) and create a sitemap.
4. Invoke parallel threads that crawl simultaneously every URL found on the first page, and identify all links pointing to RSS, Atom or Xml feeds.
5. Save the feeds, including their URLs, in the DMS. Each feed inherits the annotations of its source.
6. Finally, crawling is finished and the feeds are reviewed and activated manually in the CMS in the source feed template.

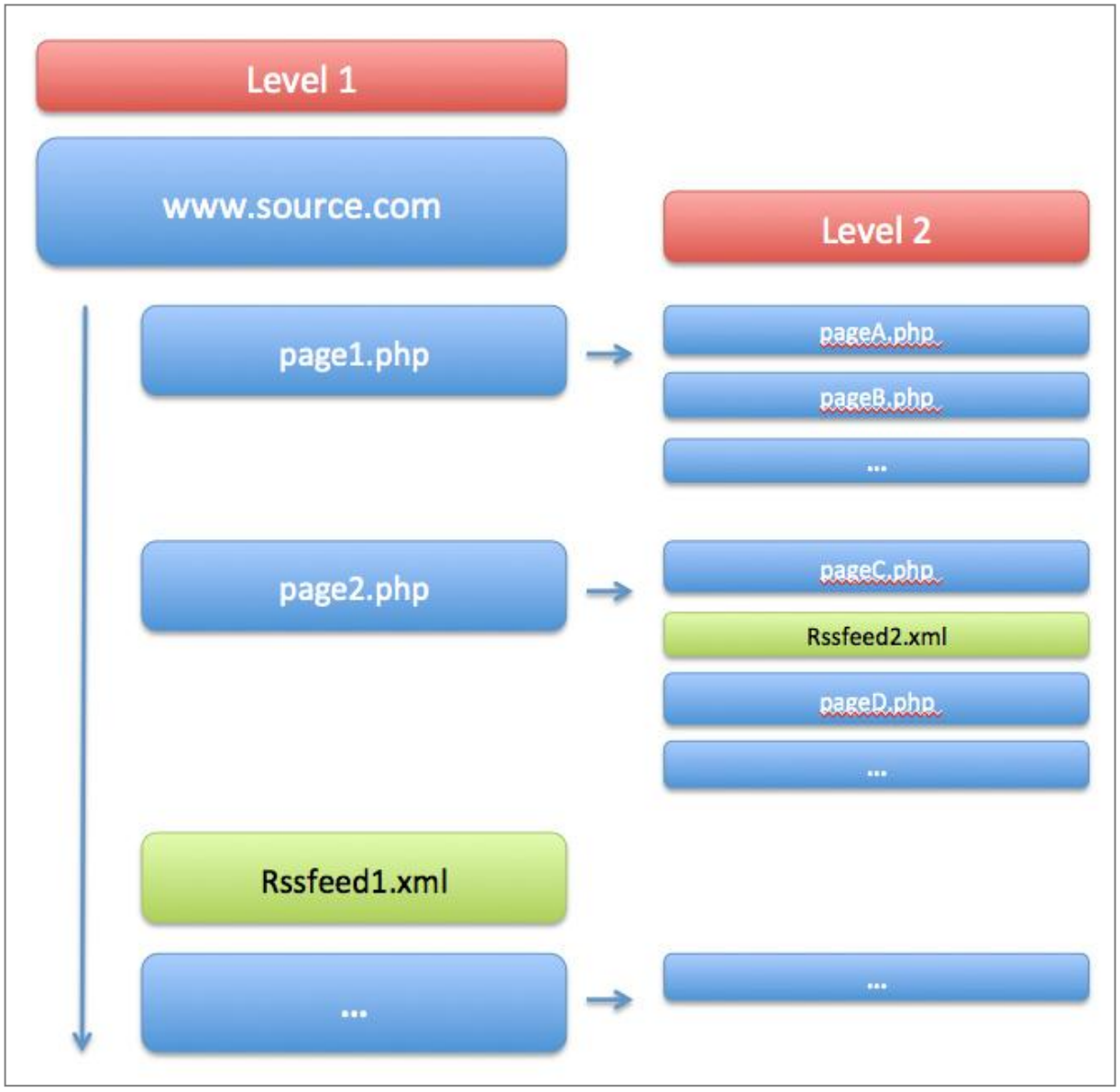

Figure 2.14: Source Crawler methodology

3 DMS Database

The database is designed to optimise the flexibility of the tool regarding its development. The tables in the MySQL database holding DMS data are prefixed with the string "*Dms"* in order to dissociate them from the tables holding Pimcore data.

For a better understanding of the structure of the database and the logic of our matrix, it is important to understand the distinction between Annotation and ItemAnnotation. An ItemAnnotation is an instance (i.e. value) of one Annotation. For example, the ItemAnnotation "*Germany*" is an instance of the Annotation "*Country*". As explained in an earlier deliverable, the distinction between Annotations and ItemAnnotations allows us to create a multi-layer logic pattern that efficiently structures the collected content.

The tables from the DMS are:

- *DmsAccessRight*
- *DmsAccount*
- *DmsAccountHasFeed*
- *DmsAnnotation*
- *DmsAnnotationHasFeed*
- *DmsAnnotationToTable*
- *DmsComment*
- *DmsCommentHasItemAnnotation*
- *DmsDMSUser*
- *DmsFeed*
- *DmsFeedHasItemAnnotation*
- *DmsItemAnnotation*
- *DmsItemAnnotationHasItemAnnotation*
- *DmsKeyword*
- *DmsKeywordHasAccount*
- *DmsKeywordHasItemAnnotation*
- *DmsMeasure*
- *DmsMeasureCase*
- *DmsMeasureCaseHasItemAnnotation*
- *DmsMeasureHasItemAnnotation*
- *DmsMeasureHasMeasureCase*
- *DmsOrganisation*
- *DmsOrganisationHasItemAnnotation*
- *DmsSource*
- *DmsSourceCrawler*
- *DmsSourceHasItemAnnotation*
- *DmsUser*

A diagram describing the relations between the different MySQL tables is available in Appendix 1.

3.1 DmsAccessRight

Table *DmsAccessRight* holds all role definitions for the UniteEurope DMS, including their associated permissions. The table uses the InnoDB engine and the charset *utf8_bin*.

AccessRightID	Primary key of the table
Role	Name of the role **Ex:** “manager”
Right	Serialised array with the list of permissions. The content of this array is used to authorise access to the UniteEurope tools. **Ex: “**Benchmark Analytics, Campaign Tracking, Integration Monitoring, Live Monitoring, Measure Case Library”

Table 3.1: DmsAccessRight

3.2 DmsAccount

Table *DmsAccount* holds all account information. The table uses the InnoDB engine and the charset *utf8_bin*.

AccountID	Primary key of the table
Name	Name of the account member. **Ex:** “Imooty”
Description	Short description of the account member
Url	Website of the account member
Street	Street of the account member’s address
Street2	Second street of the account member’s address
Postcode	Postcode of the account member’s address
City	City of the account member’s address
Country	Country of the account member’s address
AccessRightID	AccessRightID pointing to the permission definition of the account member
OrganisationID	OrganisationID pointing to an organisation. **Ex:** “City Berlin” is treated as an organisation in the UniteEurope tools, and also has an account.

Table 3.2: DmsAccount

3.3 DmsAccountHasFeed

Table *DmsAccountHasFeed* holds the relations between account members and web feeds. This optimises the integration of the information and the customisation of the tools. The table uses the InnoDB engine and the charset *utf8_bin*.

AccountID	Primary key of the table
FeedID	FeedID of the related feed

Table 3.3: DmsAccountHasFeed

3.4 DmsAnnotation

Table *DmsAnnotation* holds the list of available annotations. The table uses the InnoDB engine and the charset *utf8_bin*.

AnnotationID	Primary key of the table
Label	Name of the annotation
Description	Description of the annotation

Table 3.4: DmsAnnotation

3.5 DmsAnnotationToTable

Table *DmsAnnotationToTable* allows us to easily attach annotations to the elements of the website. The aim is to define which annotation is enabled for which element. The table uses the InnoDB engine and the charset *utf8_bin*.

AnnotationToTableID	Primary key of the table
TableName	Name of the table (corresponding to an element of the website)
ListAnnotation	List of annotations available for the element.

Table 3.5: DmsAnnotationToTable

3.6 DmsComment

Table *DmsComment* holds the comments for measures and MeasureCases. The table uses the InnoDB engine and the charset *utf8_bin*. This table is not currently used.

CommentID	Primary key of the table
Title	Title of the comment
Content	Content of comment
Date	Date of the comment
MeasureCaseID	MeasureCaseID of the related MeasureCase
MeasureID	MeasureID of the related measure

Table 3.6: DmsComment

3.7 DmsCommentHasItemAnnotation

Table *DmsCommentHasItemAnnotation* holds the relations between comments and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*. This table is not currently used.

CommentID	ID of the comment
ItemAnnotationID	ID of the related ItemAnnotation
StringItemAnnotationID	String value of every ItemAnnotation for optimising MySQL queries. ItemAnnotations are separated by minus signs. **Ex:** -ItemAnnotation1--ItemAnnotation12--ItemAnnotation21-

Table 3.7: DmsCommentHasItemAnnotation

3.8 DmsDMSUser

Table *DmsDMSUser* holds the DMS user profiles. The table uses the InnoDB engine and the charset *utf8_bin*.

DMSUserID	Primary key of the table
Lastname	Last name of the user
Firstname	First name of the user
Email	Email address of the user
Password	Password of the user
Right	Permission of the user (one of “editor”, “admin” or “superadmin”)

Table 3.8: DmsDMSUser

3.9 DmsFeed

Table *DmsFeed* holds the definition of each web feed. The table uses the InnoDB engine and the charset *utf8_bin*.

FeedID	Primary key of the table
Url	URL of the feed
CrawlTime	Date and time of the last crawl (see section 5.3.2.4)
CrawlEveryMinutes	A frequency defining how often the feed shall be crawled
LastEntryDownload	Date and time of the last time a feed entry was downloaded and determined to be integration-related. This value is used for the DMS monitoring function.
SourceID	SourceID of the related source

Table 3.9: DmsFeed

3.10 DmsFeedHasItemAnnotation

Table *DmsFeedHasItemAnnotation* holds the relations between feeds and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

FeedID	ID of the feed
ItemAnnotationID	ID of the related ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.10: DmsFeedHasItemAnnotation

3.11 DmsItemAnnotation

Table *DmsItemAnnotation* defines the ItemAnnotations for all annotations. The table uses the InnoDB engine and the charset *utf8_bin*.

ItemAnnotation	Primary key of the table
Label	String value of the ItemAnnotation
Value	Float value of the ItemAnnotation (if string value is NULL)
AnnotationID	ID of the corresponding Annotation, holding the label for the string or float value

Table 3.11: DmsItemAnnotation

3.12 DmsItemAnnotationHasItemAnnotation

Table *DmsItemAnnotationHasItemAnnotation* holds relations between ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

ParentItemAnnotationID	ID of the parent ItemAnnotation
ChildItemAnnotationID	ID of the child ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.12: DmsItemAnnotationHasItemAnnotation

3.13 DmsKeyword

Table *DmsKeyword* holds the keywords. The table uses the InnoDB engine and the charset *utf8_bin*.

KeywordID	Primary key of the table
Lemme	Lemma of the keyword
Stem	Stem of the keyword
WordVariation	List of words (surface forms) that are grammatical variations of the keyword (e.g. different number, case, gender or tense)

Table 3.13: DmsKeyword

3.14 DmsKeywordAccount

Table *DmsKeywordAccount* holds relations between keywords and accounts. This relation may become important for the customisation of accounts. The table uses the InnoDB engine and the charset *utf8_bin*. It is currently not used by the DMS and may change during the further development of the tools.

KeywordID	Primary key of the table
AccountID	ID of the related account

Table 3.14: DmsKeywordAccount

3.15 DmsKeywordHasItemAnnotation

Table *DmsKeywordHasItemAnnotation* holds the relations between keyword and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

KeywordID	ID of the keyword
ItemAnnotationID	ID of the ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.15: DmsKeywordHasItemAnnotation

3.16 DmsMeasure

Table *DmsMeasure* holds the integration measures. The table uses the InnoDB engine and the charset *utf8_bin*.

MeasureID	Primary key of the table
Label	Label of the measure
Description	Description of the measure
DateBegin	Date of the begin of the measure
DateEnd	Date of the end of the measure
Url	URL of the measure
OrganisationID	ID of the related organisation

Table 3.16: DmsMeasure

3.17 DmsMeasureCase

Table *DmsMeasureCase* holds the integration measure cases. The table uses the InnoDB engine and the charset *utf8_bin*.

MeasureCaseID	Primary key of the table
Label	Label of the case
Description	Description of the case
DateBegin	Date of the begin of the case
DateEnd	Date of the end of the case
Url	Url of the case
OrganisationID	ID of the related Organisation

Table 3.17: DmsMeasureCase

3.18 DmsMeasureCaseHasItemAnnotation

Table *DmsMeasureCaseHasItemAnnotation* holds the relations between MeasureCases and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

MeasureCaseID	ID of the MeasureCase
ItemAnnotationID	ID of the ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.18: DmsMeasureCaseHasItemAnnotation

3.19 DmsMeasureHasItemAnnotation

Table *DmsMeasureHasItemAnnotation* holds the relations between integration measures and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

MeasureID	ID of the measure
ItemAnnotationID	ID of the ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.19: DmsMeasureHasItemAnnotation

3.20 DmsMeasureHasMeasureCase

Table *DmsMeasureHasMeasureCase* holds the relations between integration measures and their MeasureCases. The table uses the InnoDB engine and the charset *utf8_bin*.

MeasureID	ID of the measure
MeasureCaseID	ID of the MeasureCase

Table 3.20: DmsMeasureHasMeasureCase

3.21 DmsOrganisation

Table *DmsOrganisation* holds information about organisations (municipalities, NGOs, companies, institutions). The table uses the InnoDB engine and the charset *utf8_bin*.

OrganisationID	Primary key of the table
Name	Name of the organisation
Description	Description of the organisation
Url	URL of the organisation

Table 3.21: DmsOrganisation

3.22 DmsOrganisationHasItemAnnotation

Table *DmsOrganisationHasItemAnnotation* holds the relations between organisations and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

OrganisationID	ID of the organisation
ItemAnnotationID	ID of the ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.22: DmsOrganisationHasItemAnnotation

3.23 DmsSource

Table *DmsSource* holds the web sources. The table uses the InnoDB engine and the charset *utf8_bin*.

SourceID	Primary key of the table
Label	Name of the source
Url	URL of the source

Table 3.23: DmsSource

3.24 DmsSourceCrawler

Table *DmsSourceCrawler* is used by the Source Crawler for managing the pages of the currently crawled website. The table uses the InnoDB engine and the charset *utf8_bin*.

SourceCrawlerID	Primary key of the table
SourceID	ID of the source
Url	URL of the link
Level	Level of the link (1 or 2)
Status	Status of the link (either *feed link* or *other type of link*)

Table 3.24: DmsSourceCrawler

3.25 DmsSourceHasItemAnnotation

Table *DmsSourceHasItemAnnotation* holds the relations between sources and their ItemAnnotations. The table uses the InnoDB engine and the charset *utf8_bin*.

SourceID	ID of the source
ItemAnnotationID	ID of the ItemAnnotation
StringItemAnnotationID	String representation of all ItemAnnotations for optimising MySQL queries

Table 3.25: DmsSourceHasItemAnnotation

3.26 DmsUser

Table *DmsUser* holds the users of UniteEurope. The table uses the InnoDB engine and the charset *utf8_bin*.

UserID	Primary key of the table
Lastname	Last name of the user
Firstname	First name of the user
Email	Email of the user
Password	Hashed password of the user
AccountID	ID of the related account
AccessRightID	ID of the related AccessRight (Role and permission definition for the user)

Table 3.26: DmsUser

4 UniteEurope Website

4.1 Pimcore integration

Pimcore is a Content Management System (CMS), meaning that it allows us to create articles and manage the editorial content of the website, for example the pages *About* and *Help*. The UniteEurope tools are then fully integrated into Pimcore, but are managed from the DMS, except for the editorial content of the website.

At this point, we are skipping a definition of what a CMS is, since this is a standard editorial process. You are invited to read the Pimcore documentation[2] for getting more information about the Pimcore CMS functionality. Instead, we will focus on the presentation of the integration of our tools into Pimcore, and on the architecture of this integration.

There are currently four key modules which constitute the UniteEurope functionality:

1. Solr / Lucene – Get and filter the content of the UniteEurope index
2. Twitter – Get content from Twitter
3. RSS Feed – Get content from a RSS, Atom or XML feeds
4. Facebook – Get content from Facebook

Each of these modules represents a basis for the whole application. They are the first step of every media monitoring task and allow us to retrieve and quantify content. Based on these results, we are then able to perform benchmarking processes or statistical analyses or search optimisations.

For example, the Live Stream Monitoring component uses the Twitter, Facebook and RSS feed modules by displaying live results from these three news providers. On the other hand, the benchmarking process will do statistical analysis on the results of targeted queries to Solr/Lucene.

[2] http://www.pimcore.org/wiki/display/PIMCORE/Pimcore+Documentation

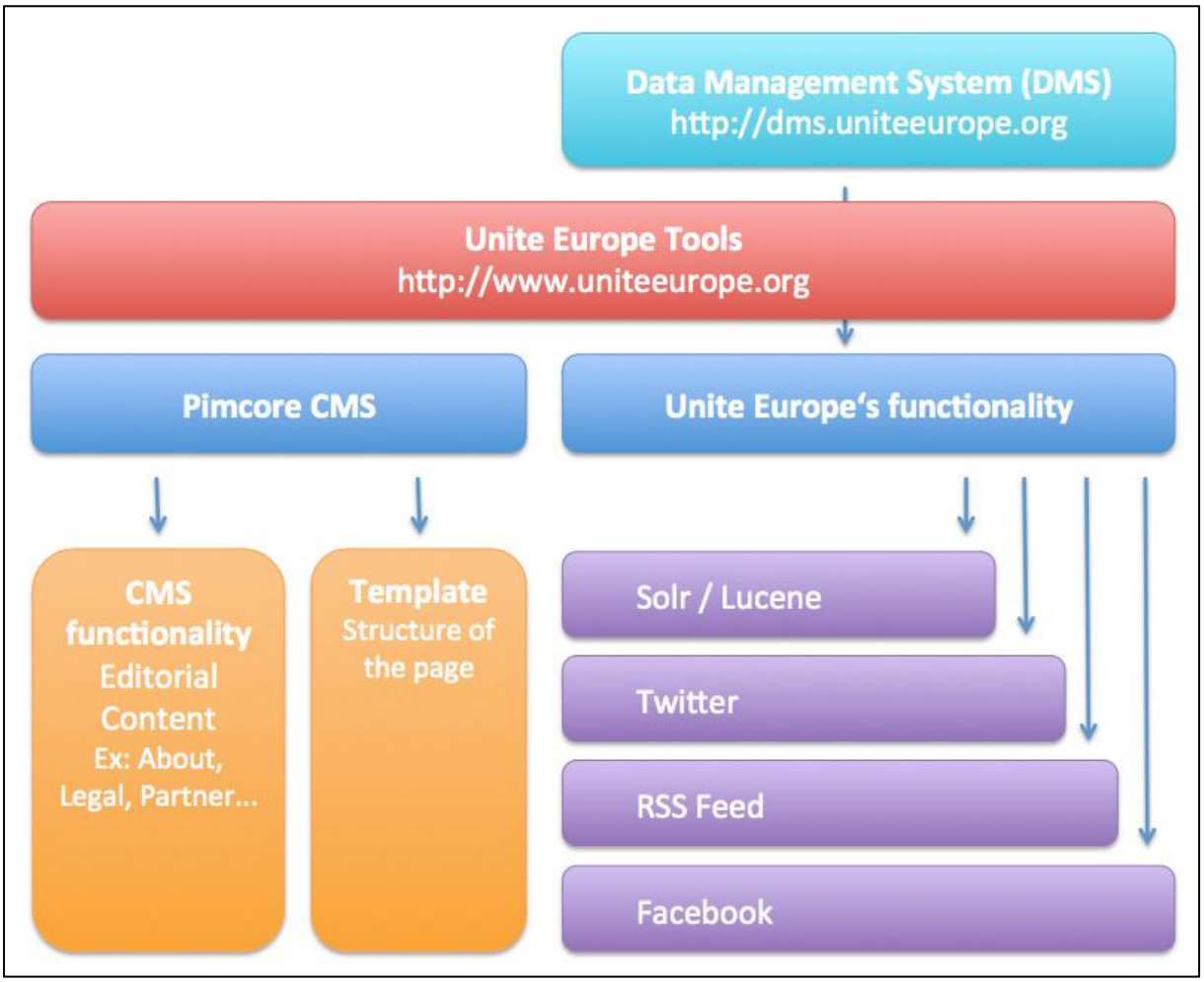

Figure 4.1: Pimcore integration

4.2 Modules

Since the various components of the UniteEurope website (Live Stream and Integration Monitoring, Benchmarking, Campaign Tracking etc.) rely on the modules listed in the previous section, we concentrated first on the development of these modules. When this step is finished, we can proceed by putting the modules into action and thereby structuring the components.

4.2.1 Twitter

The integration of live Twitter feeds (*tweets*) is provided by the **Zend_Service_Twitter**[3] and **Zend_Service_Twitter_Search** [4] class of the Zend Framework. Zend_Service_Twitter

[3] http://framework.zend.com/manual/1.11/en/zend.service.twitter.html
[4] http://framework.zend.com/manual/1.11/en/zend.service.twitter.html#zend.service.twitter.search

provides a powerful client for the Twitter API whereas Zend_Service_Twitter_Search provides convenient search functionality.

4.2.1.1 Zend_Service_Twitter_Search

This API provides a convenient way for searching tweets. Queries return data in Atom (XML) or JSON format.

The API provides access to the Twitter trends, the top ten queries that are currently trending on Twitter, and also allow for direct search in the Twitter index, given specific queries.

Several search parameters are available:

- **Lang:** Match a specific language
- **Rpp**: Amount of tweets to return per page
- **Page:** Amount of pages to return
- **Since_id:** Tweets with IDs that are greater (i.e. newer) than the given ID
- **Show_user:** Display user names at the begin of the tweet
- **Geocode:** Filter tweets by user-defined geographical information

4.2.2 Facebook

The Facebook Class is not completed yet. It will provide a client for the Facebook API and will allow searching for information in Facebook, as well as getting users statuses. This function will be documented in a future deliverable or in the final documentation of the project.

4.2.3 RSS Feed

The integration of live RSS feeds is provided by the **Zend_Feed_Rss**[5] class of the Zend Framework.

The class offers the following data:

- Required channel elements
  - **title**: the name of the channel
  - **link**: the URL of the website corresponding to the channel
  - **description**: one or more sentences describing the channel
- Common optional channel elements
  - **pubDate**: the publication date of this set of content, in RFC 822 date format
  - **language**: the language the channel is written in
  - **category**: one or more (specified by multiple tags) categories the channel belongs to

[5] http://framework.zend.com/manual/1.11/en/zend.feed.consuming-rss.html

- **RSS** <*item*> elements do not have any strictly required elements. However, either title or description must be present.

- Common item elements
  - **title**: the title of the item
  - **link**: the URL of the item
  - **description**: a synopsis of the item
  - **author**: the author's email address
  - **category**: one or more categories that the item belongs to
  - **comments**: URLs of comments relating to this item
  - **pubDate**: the date the item was published, in RFC 822 date format

4.2.4 Solr / Lucene

A special class called **Website_Model_SolrMapper** has been implemented and contains several methods for querying items from the UniteEurope Solr/Lucene index. This class uses the PHP Solr extension, which allows communicating effectively with the Apache Solr server in PHP 5[6].

Methods of the object

Function name	Description
`getContent()`	Filter the content from the Solr index by these parameters: • **searchTerms:** one or several terms formatted following the Lucene query parser syntax[7]. • **listItemAnnotationIDs:** List of ItemAnnotationIDs for filtering the content. For example, the ItemAnnotationID of "country=Germany" is 179 (see also section 5.3.3.3). • **page:** the page number to retrieve • **nbResult:** amount of item matches per page Returns an array with the items matched by the query.

Table 4.1: Website_Model_SolrMapper

[6] http://www.php.net/manual/en/book.solr.php
[7] http://lucene.apache.org/core/3_6_0/queryparsersyntax.html

5 Web Crawler

The UniteEurope web crawler is responsible for reading the web source definitions from the Data Management System, as well as for downloading, analysing and storing the corresponding online posts. The last task on this list, the storing part, was documented in Deliverable 5.1; it comprises the crawler's database technology (Hadoop + HBase), its search engine (Solr), and the software that connects these two components (Lily). We will now describe in more detail the current state of implementation of the download and analysis tasks.

5.1 Software projects

There are currently two different software projects under development, one for utility classes and one for the crawler. Both projects are written in Java and are configured as Maven[8] projects. They are both children of a third parent project, which only contains a Maven configuration file (`pom.xml`) and which defines all Maven parameters that are shared by every project in our development environment. This approach avoids redundancy, making every project inherit the following parameters:

- Source encoding: All source code files are encoded in UTF-8.
- Maven repository: The Sonatype Nexus[9] repository where Maven artefacts are stored.
- Maven dependencies: Every project depends on JUnit[10].
- Build excludes: Property files (**.properties*) are excluded from the built JAR files, so that projects must override inherited properties.
- Maven Eclipse plugin: Employ the Maven plugin for Eclipse[11] in order to generate configuration files for the Eclipse IDE[12].
- Maven compiler plugin: All source code complies to Java[13] SE 6.

5.2 Utilities project

The utility classes are kept separate in order to make them usable by other, more specific projects. There is only one other project at the moment, but this might change in the future.

Below, we document the most important classes from the utility project:

- `MySQLConnection`: Manage pools of connections to a MySQL database, using the Apache library for database connection pooling[14]. Every pool is defined by a unique

[8] http://maven.apache.org/
[9] http://www.sonatype.org/nexus/
[10] http://www.junit.org/
[11] http://maven.apache.org/plugins/maven-eclipse-plugin/
[12] http://www.eclipse.org/
[13] http://www.java.com

ID of the form *URL–Schema–User*, e.g. *jdbc:mysql://192.168.1.25:3306/–uniteeurope–crawlerUser*, and can hold up to eight connections. The connection credentials are expected to be defined in a file called `uniteeurope.properties`.

- `DatabaseUtilities`: Open and close connections to a MySQL database, perform create/read/update/delete operations, and validate input strings in order to avoid SQL injections.
- `DateUtilities`: Simplify the conversion between strings (e.g. "*2012-05-07 09:51:45*") and date objects.
- `PropertyLoader`: Read properties from a Java properties file which must exist somewhere on the class path. Furthermore, if the file is named `uniteeurope.properties`, for example, all properties defined in `uniteeurope_private.properties` will have precedence. This allows different users to define customised parameter values, while sharing the same properties file via Subversion[15].
- `PropertyUtils`: Query the properties read by the `PropertyLoader`, returning a string, a boolean or a list of strings.
- `UELogger`: Create a logger from the Java Logging API. Configuration parameters are expected to be defined in a file called `logging.properties`.

For all classes, unit tests are defined in order to ensure higher quality of code. For example, the test case `MySQLConnectionTest.testGetConnection()` fetches and closes multiple connections from the pool and validates the number of active and idle connections. The current code coverage (percentage of instructions covered by unit tests) is 59.1%.

5.3 Crawler project

5.3.1 Maven configuration

Just like the utilities project, the crawler inherits some universal Maven properties from the parent project, see section 5.1. In addition, the following dependencies are defined:

- The UniteEurope utilities project (see above)
- The Lily client and repository libraries
- The ROME[16] library for syndication feed parsing
- The Apache API for sending emails[17]
- The Solr Java client API[18]

[14] http://commons.apache.org/dbcp/
[15] http://subversion.apache.org/
[16] https://rometools.jira.com/wiki/display/ROME/Home
[17] http://commons.apache.org/email/
[18] http://wiki.apache.org/solr/Solrj

- Three different Lucene[19] libraries (core functionalities, language analysers, and a stemmer for Polish)
- The Twitter4J[20] library for accessing the Twitter API
- The Tika[21] library for language detection

Many of these libraries depend on other libraries themselves, resulting in the dependency network depicted in Figure 5.1. Furthermore, the Maven assembly plugin[22] is used for building JAR files from the project which contain all of these libraries. This will enable us later to deploy the project in a standalone way, without having to install the dependencies on the target machine.

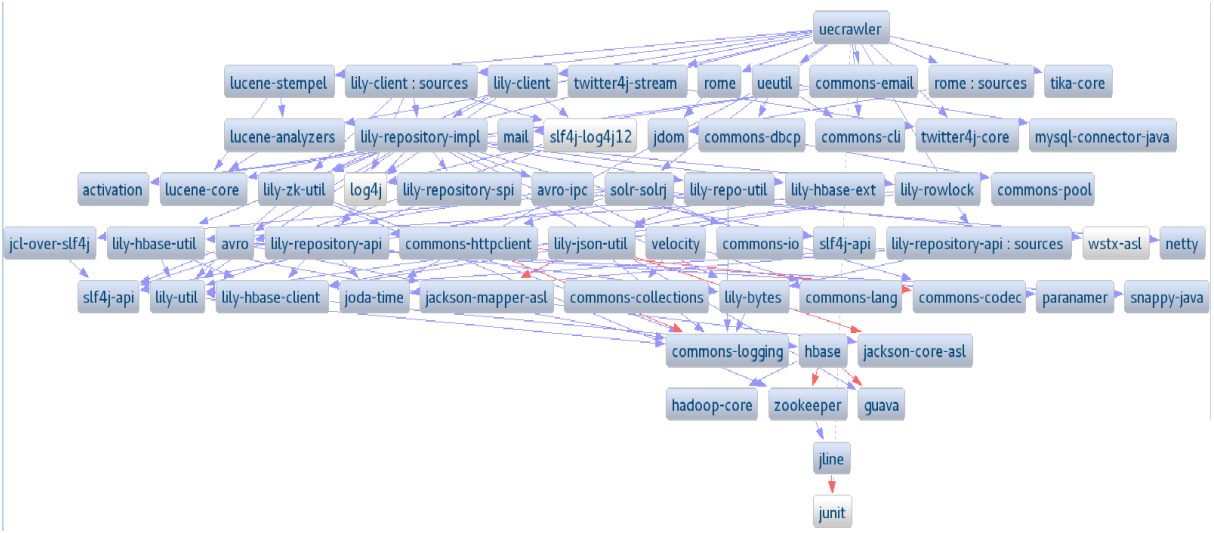

Figure 5.1: Maven dependency graph

[19] http://lucene.apache.org/
[20] http://twitter4j.org/
[21] http://tika.apache.org/
[22] http://maven.apache.org/plugins/maven-assembly-plugin/

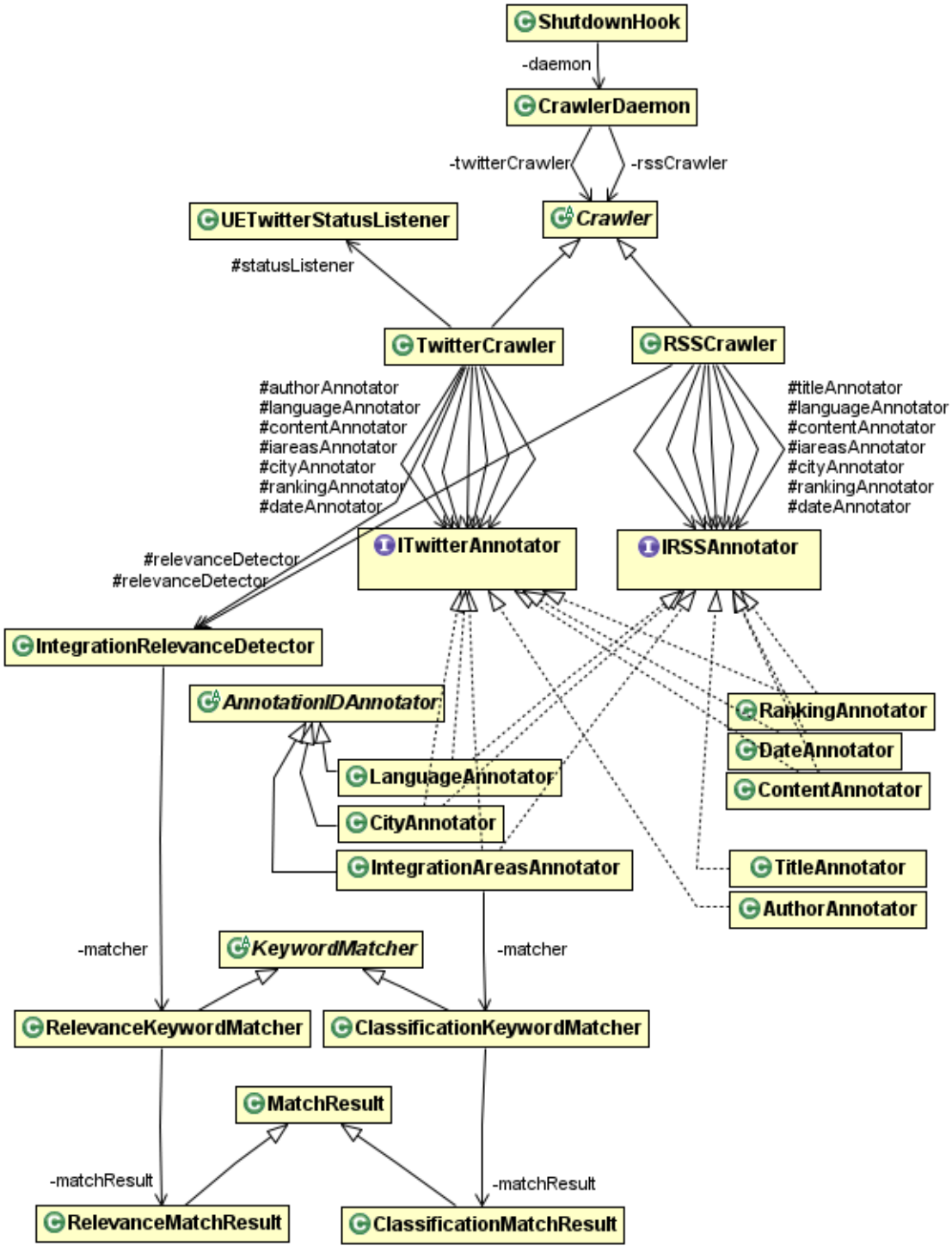

Figure 5.2: UML diagram for the crawler project

5.3.2 Downloading of online data

In this section, we give an overview of how online data is downloaded, while in the next section, we describe how this data is analysed. Figure 5.2 shows a UML[23] diagram for the most important classes of both tasks.

5.3.2.1 Class CrawlerDaemon

The `CrawlerDaemon` represents the starting point of the crawler project. It contains the `main()` method which is called to start the crawling process. This process will not terminate until it is stopped by the user or a severe error occurs.

As soon as the `CrawlerDaemon` is started, it performs two actions. First, the `rssCrawler` instance is scheduled to run recurrently after one minute of waiting, like this:

1. Run the `rssCrawler`
2. Wait one minute
3. GOTO 1.

In the beginning of the project, there will be only a few RSS feeds defined in the DMS, so the frequency of how often the feed information is read from the database will equal 1/minute. As soon as we add more feeds, the runtime of step 1 will increase, so that the feed definitions will be read less often.

All exceptions that occur during the RSS crawling are logged, and an email will be sent to the administrators with a stack trace of the first exception. Only in the case of fatal errors will all Java processes be stopped, e.g. when the Lily repository is permanently unavailable.

The second process which is started is the `twitterCrawler`, a continuously running task which does not have to be rescheduled. Similar processes for other APIs, such as Google+ or Facebook, will soon be implemented.

5.3.2.2 Class ShutdownHook

During initialisation, the `CrawlerDaemon` will register a shutdown hook with the Java virtual machine. This class will stop the `rssCrawler` and `twitterCrawler`, as well as close connections to Lily and MySQL, in two cases:

1. a fatal error occurs, see section 5.3.2.1, or
2. the user kills the Java process.

5.3.2.3 Class Crawler

This abstract class defines functionality that is shared by the `RSSCrawler`, `TwitterCrawler` and other future crawler classes. Among these functions, the following are most important:

[23] http://www.uml.org/

- `SINGLE_VALUE_STRING_ANNOTATIONS`: This constant defines all feed annotations that are read from the DMS and whose value is exactly one string:
  - Annotation 1: **country**
  - Annotation 2: **cityAnnotation**
  - Annotation 3: **cityFocus**
  - Annotation 4: **sourceType**
  - Annotation 5: **integrationFocus**
  - Annotation 6: **streamType**
- `MULTI_VALUE_STRING_ANNOTATIONS`: This constant defines all feed annotations that are read from the DMS and that can have multiple string values:
  - Annotation 7: **languageAnnotation**
  - Annotation 8: **topics**
- `INTEGER_ANNOTATIONS`: This constant defines all feed annotations that are read from the DMS and whose value is exactly one integer:
  - Annotation 9: **ranking**
- `initializeAnnotators()`: This abstract function must be implemented by child classes; it is abstract because the set of annotators can be different for every crawler.
- `getFeedIDsByStreamType(String streamType)`: Returns all feed IDs from the DMS with the given stream type.
- `getActiveFeedIDs()`: Returns all feed IDs from the DMS which are annotated as active, and whose source is also annotated as active.

5.3.2.4 Class RSSCrawler

This class extends the `Crawler` class; it downloads, analyses and persists RSS feeds[24]. During initialisation, it creates instances of the following classes:

- `LanguageAnnotator`
- `ContentAnnotator`
- `DateAnnotator`
- `TitleAnnotator`
- `CityAnnotator`
- `IntegrationAreasAnnotator`
- `RankingAnnotator`
- `IntegrationRelevanceDetector`

The `AuthorAnnotator` is not included, because only APIs such as Twitter provide a unique author ID.

[24] Wherever we talk about RSS feeds, we are implicitly including Atom feeds. The term “syndication feed” would be the appropriate hypernym.

When the `RSSCrawler` is started by the `CrawlerDaemon`, it gets from the DMS all IDs of feeds which are active, and whose stream type is *Rss*. Furthermore, every feed definition contains information about how often it shall be crawled (e.g. every 360 minutes), and about the last date it was crawled. These two specifications determine if the feed is processed any further. Note that the waiting time of one minute in the `CrawlerDaemon` only determines how often feed information is read from the DMS, but the `RSSCrawler` determines if a feed is actually crawled.

Afterwards, we extract the annotations for every feed in the list that results from the previous step. This creates a map from strings (the feed URLs) to a list of attribute-value pairs. In addition to the single- and multi-value string annotations and integer annotations mentioned at the beginning of section 5.3.2.3, three additional annotations are included in this map:

- Annotation 10: **sourceID**
- Annotation 11: **feedID**
- Annotation 12: **annotationIDs** (the database IDs of annotations 1–9)

Feeds with missing or unknown annotations are discarded. New annotations have to be agreed on by the DMS and web crawler developers, and entail changes in the Lily and Solr configuration.

Valid feeds are now downloaded and parsed with the ROME library. The feed supplies us with a list of feed entries (i.e. posts). For every entry, a copy of the feed's attribute-value pairs is created, and the following annotations are added:

- Annotation 13: Extract the **title** of the post (see section 5.3.3.11).
- Annotation 14: Extract the **content** of the post (see section 5.3.3.10).
- Annotation 15: Concatenate the title and content into one *document text*, and automatically determine the most probable document **language** (see section 5.3.3.4).

The concatenation of title and content is now normalised and stemmed. This includes the removal of punctuation, the transformation of words to lower case and their subsequent replacement by word stems. Given this stemmed document text, and the extracted language, we determine if the document is integration-related (see section 5.3.3.7). If the document is not integration-related, it is discarded. Otherwise, more annotations are added:

- Annotation 16: Determine if the stemmed document text is related to a certain **city** (see section 5.3.3.5).
- Annotation 17 + 18: Determine the most probable **integration subarea** covered by the stemmed document text, and the probability itself (**rankingAuto**), see sections 5.3.3.6 and 5.3.3.8.
- Annotation 19: Extract the **date** of the post (see section 5.3.3.9).

- Annotation 20: Determine the database IDs of annotations 15–17 (**annotationIDsAuto**), see section 5.3.3.3.
- Annotation 21: Generate a unique **document ID** from the feed entry's URL, performing the following replacements in order to comply to Lily's requirements:

Character:	Replaced by:
.	\\
=	%3D
,	%2C
;	%3B

Table 5.1: Character replacements in document IDs

An example for the resulting list of annotations would be:

```
[country:              "AT",
cityAnnotation:        "VIE",
cityFocus:             "National",
sourceType:            "Newspaper",
integrationFocus:      false[25],
streamType:            "Rss",
languageAnnotation:    ["DE"],
topics:                ["Various"],
ranking:               2,
sourceID:              4,
feedID:                4,
annotationIDs:         [1,2,4,5,9,11,13,14,27]},
title:                 "Fremdenrecht – Ungarn Rückschiebungen
                       erschwert",
content:               "Luxemburger Gericht soll prüfen – Wien:
                       Kampf gegen Ausweisung dreier Teenager"
languageAuto:          "DE",
cityAuto:              "VIE",
iareas:                ["Citizenship"],
rankingAuto:           1.0,
date:                  "2012-07-13 16:21:14",
annotationIDsAuto:     [5,1,151],
lily.id:               "USER.http://derstandard\\at/
                       1342139043692/Ungarn-Rueckschiebungen-
                       weiter-erschwert"]
```

[25] The integration focus is stored as a string in the DMS, but will be converted to a boolean.

The 21 annotations are then passed to the Lily client library, which will persist it to HBase and Solr[26]. The original XML data from the RSS or Atom feed will not be stored. If the ID already exists, no change is made[27].

Finally, the DMS is updated. For every feed, we change the time of the last crawl to the current system time. In addition, we update the `LastEntryDownload` attribute if there was at least one feed entry which was not in HBase yet.

5.3.2.5 Class TwitterCrawler

This class extends the `Crawler` class; it downloads, analyses and persists Twitter posts (*tweets)*. During initialisation, it creates instances of the following classes:

- `LanguageAnnotator`
- `ContentAnnotator`
- `DateAnnotator`
- `AuthorAnnotator`
- `CityAnnotator`
- `IntegrationAreasAnnotator`
- `RankingAnnotator`
- `IntegrationRelevanceDetector`

The `TitleAnnotator` is not included, because tweets have no title field.

The class constructor will also read the Twitter account information from `uniteeurope.properties`. Two different accounts have been created for the crawler, one for testing and one for production use. This was necessary because two simultaneous logins from one account are not possible.

When the `TwitterCrawler` is started by the `CrawlerDaemon`, it connects to Twitter's streaming API[28] via the Twitter4J library. Twitter allows us to define (up to 400) keywords to track. Tweets containing these keywords are continuously sent to our Java client, as long as the thread is not stopped by the user or a severe exception occurs.

The keywords to track are a subset of those defined in the DMS, for two reasons:

[26] However, Solr may decide to queue the document in main memory before actually persisting it to the index on the hard disk, because committing several documents at a time improves performance.

[27] Actually, a `RecordExistsException` is thrown, which we catch silently. A more consistent approach would query if the ID already exists in HBase, before performing the analysis. However, hard disk access is at least 10,000 times slower than main memory access, so we prefer to access the hard disk only once, during creation of the (possibly existing) record. An improvement for a future version of the crawler is to check the feed's last crawl time and analyse only feed entries that are younger than this date. This will require that the entries' publish times are correct, and time zone information is always available.

[28] https://dev.twitter.com/docs/streaming-apis

- Twitter allows only 400 keywords per streaming connection, but there are 2430 active, integration-related keywords defined in the DMS. There are two options to overcome this issue:
    1. Create more Twitter accounts, and use each to track a subset of the keywords.
    2. Remove keywords that occur so rarely that the Twitter search returns no results for them (e.g. *guest worker generation, Anti-Ausländer-Kampagne, feitelijke terugkeer).*
- Many keywords will yield a large number of results that are not integration-related. For example, the word *ghetto* is used in many different contexts. The keyword list will therefore have to be manually adapted to Twitter, possibly using the following approach:
    1. Choose a subset of 400 keywords from the DMS.
    2. Start tracking tweets containing these keywords.
    3. Identify a tweet that is not integration-related; this will require a person that speaks the language under consideration.
    4. Identify which keyword caused the "wrong" tweet to show up. Remove the keyword from the current set, and mark it as non-integration related.
    5. Continue with steps 2–4 until most tweets returned from Twitter are relevant. In the DMS, add a new annotation `IntegrationFocusTwitter`. Set the annotation value to *false* for every keyword marked in step 4. This annotation can then be used by the crawler for determining Twitter keywords.
- Alternatively, every single keyword could be checked for relevance individually. This approach would be more consistent, but would also take more time than it takes to reach step 5. Either approach will be implemented as an additional user interface in the DMS.

Since the issues mentioned above have not been addressed yet, the current list of Twitter search terms is hard-coded into class `TwitterCrawler`. It consists of some English and German keywords and was created in a fashion similar to steps 1–5 as explained above. The tweets which are returned for this list are processed by a `UETwitterStatusListener`, analogously to section 5.3.2.4:

- Read the feed definition for Twitter from the DMS and store it as a list of annotations.
- Add annotations according to the current tweet:
    - content
    - automatically determined language
- If the detected language matches any keyword language (currently English or German), and if the stemmed tweet content is integration-related, add more annotations:
    - date
    - city
    - most probable integration subarea, and its probability

- the database IDs of some annotations
- a unique document ID[29]

An example for the resulting list of annotations would be:

```
[country:              "UNK", (unknown)
cityAnnotation:        "UNK",
cityFocus:             "International",
sourceType:            "Microblog",
integrationFocus:      false,
streamType:            "Twitter",
languageAnnotation:    ["UNK"],
topics:                ["Various"],
ranking:               5,
sourceID:              8,
feedID:                79,
annotationIDs:         [3,9,11,12,20,23,24,26,28]},
content:               "How can you disrespect someone's
                       religion and remove their burka? Nah this
                       isn't halal"
languageAuto:          "EN",
cityAuto:              "UNK",
iareas:                ["Cultural practices"],
rankingAuto:           1.0,
date:                  "2012-07-16 14:04:45"[30],
annotationIDsAuto:     [6,20,162],
lily.id:               "USER.https://twitter\\com/#!/kennizla/
                       status/224867167190007808",
authorID:              "kennizla",
authorName:            "kenzo ♥"]
```

The 22 annotations are then passed to the Lily client library, which will persist it to HBase and Solr. If the ID already exists, no change is made.

The streaming API will sometimes send out status deletion notices. This happens when a user deletes a tweet. All users of the API are encouraged to remove these tweets from their database, so we implemented this functionality.

All occurring exceptions are logged. If the Lily repository becomes permanently unavailable, the `TwitterCrawler` is stopped.

[29] For technical reasons, also the Twitter users' IDs and screen names will have to be captured with the tweet. However, in order to protect the authors' of the tweets privacy (in compliance with D2.6), we will integrate a function to render them anonymous. This will be elaborated in a more detailed manner at a later stage of the project.

[30] Publish times for tweets from the streaming API are given in GMT (Greenwich Mean Time).

5.3.3 Analysis of downloaded documents

5.3.3.1 Interface IRSSAnnotator

`IRSSAnnotator` is an interface for all annotators that analyse RSS feed entries. They are required to implement a function which creates new annotations, given a `com.sun.syndication.feed.synd.SyndEntry` object.

5.3.3.2 Interface ITwitterAnnotator

`ITwitterAnnotator` is an interface for all annotators that analyse tweets. They are required to implement a function which creates new annotations, given a `twitter4j.Status` object.

5.3.3.3 Class AnnotationIDAnnotator

When feed definitions are read from the DMS, we also extract the database IDs of these annotations; see section 5.3.2.4, Annotation 12. For example, the annotation

subIntegrationArea: "Citizenship"

corresponds to the ItemAnnotationID 151. This number is stored in a list of integers, resulting in a new annotation:

annotationIDs: […, 151, …]

While this information seems redundant, it may later be used to query the Solr index more efficiently, because we are searching for integers instead of strings. The differences in performance will be measured as soon as we have a significant amount of data.

In addition, it would be desirable to have an ID list also for the automatic annotations that are extracted from the individual posts. However, this is not possible for most of these annotations, because the *content* for example can take any arbitrary string value. Only three automatic annotations (*languageAuto, cityAuto* and *iareas*) can be matched to DMS annotations. The IDs of these corresponding annotations are stored in another list of integers:

annotationIDsAuto: [6, 20, 151]
≙languageAuto: EN, cityAuto: UNK, iareas: [Citizenship]

The abstract class `AnnotationIDAnnotator` implements a function for looking up corresponding IDs and adding them to *annotationIDsAuto*. The class is extended by `Language-Annotator, CityAnnotator` and `IntegrationAreasAnnotator`.

5.3.3.4 Class LanguageAnnotator

This class is an `AnnotationIDAnnotator`, and it implements `IRSSAnnotator` and `ITwitterAnnotator`. It concatenates the content and title of a post, if present, and uses the Tika language identifier to automatically detect the language. The identifier does not work very well for short texts like tweets, which leaves room for future improvement.

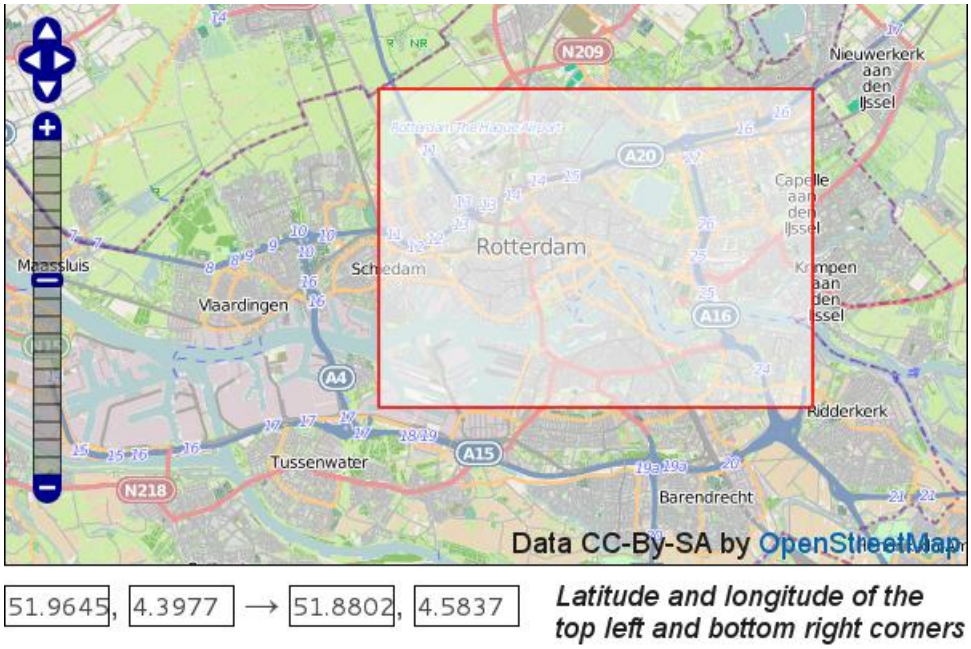

Figure 5.3: Example for geographic bounding box (Rotterdam)

Example: *languageAuto: EN*

5.3.3.5 Class CityAnnotator

This class is an `AnnotationIDAnnotator`, and it implements `IRSSAnnotator` and `ITwitterAnnotator`. In the future, it will annotate posts with a certain city if they contain keywords which are names of places in this city (e.g. districts, streets, parks or schools). This functionality has not been implemented yet, so we simply copy the city annotation from the DMS.

Example: *cityAuto: VIE*

In addition to looking for place keywords, tweets offer us to extract geographic information directly. Since the Twitter feed itself has no city defined in the DMS (*cityAnnotation: UNK → "unknown", country: UNK*), we allow the `CityAnnotator` to change these annotations now. This belated modification represents an exception among the annotators, but it allows us to distinguish between keyword-based and geolocation-based city detection without introducing a new annotation.

Geographic information is extracted by the following procedure:

- Some Twitter users add latitude and longitude values to their tweets via GPS. For every city under consideration (currently Berlin, Malmö, Rotterdam and Vienna), we have defined a bounding box as shown in Figure 5.3. If the tweet's position lies within a bounding box, the corresponding city is selected.

- Alternatively, some users tag their post with a certain *place*. The list of places, defined by Twitter, includes countries, cities, districts and restaurants, for example. All places, except for countries, come with bounding box information that is equivalent to the boxes we have defined. If the centre of a place's bounding box lies within one of our bounding boxes, the corresponding city is selected.
- If a city was selected, the city, country and the two annotation IDs are updated. However, most tweets (> 99.99%) are either not enriched with geographic information, or do not match one of our four cities.

Example: *cityAnnotation: ROT, country: NL, annotationIDs: [..., 186, 188, …]*

5.3.3.6 Class IntegrationAreasAnnotator

This class is an `AnnotationIDAnnotator`, and it implements `IRSSAnnotator` and `ITwitterAnnotator`. It uses a `ClassificationKeywordMatcher` to count the number of keyword matches for every integration subarea in the given post, see section 5.3.4.3. Since the number of keywords in the DMS can be different for every area, we normalise by this number. The area which maximises

$$\frac{\#\ keyword\ matches\ for\ this\ area}{\#\ keywords\ for\ this\ area}$$

will be selected as the most probable integration subarea. If no matches are found, the area is set to *UNK* (unknown). Of course, only keywords in the detected document language will be used for matching.

In the future, there is lots of room for improvement – we could assign a higher weight to compound term matches, or set a probability threshold so that multiple areas can be selected, or drop the keyword approach altogether and use standard text classification algorithms instead.

Example: *iareas: [Citizenship]*

5.3.3.7 Class IntegrationRelevanceDetector

This class determines if a given text is integration-related or not. It is therefore not an annotator, but will be used during the annotation process in order to analyse if a document is processed any further, or if it is discarded. Only integration-related posts will end up in the database.

While the `IntegrationAreasAnnotator` uses a `ClassificationKeywordMatcher` to classify a text into one or more integration areas, the `IntegrationRelevanceDetector` uses a `RelevanceKeywordMatcher` to identify relevant documents. This task is equivalent to a classification into the set *{relevant, irrelevant}*. For the given document language, the matcher will retrieve all integration-related keywords, and count the number of matches (see section 5.3.4.2). This count is then normalised by the length of the text:

$$\frac{\#\ keyword\ matches}{\#\ document\ tokens}$$

If this value exceeds a defined threshold of 0.005, i.e. at least one match per 200 tokens, the document is classified as relevant. This requirement is very permissive, and will be made stricter in the future.

Note that we currently assume all tweets to be relevant, because they were returned from the streaming API after querying for integration-related keywords.

5.3.3.8 Class RankingAnnotator

This class implements `IRSSAnnotator` and `ITwitterAnnotator`. Its purpose is to assign every document a ranking value greater than 0, based on the content. At least two different ranking concepts are conceivable:

1. The probability of the document being integration-related, as shown in section 5.3.3.7.
2. The matching value of the integration area assignment, as shown in section 5.3.3.6.

Since these values would be assigned by the `IntegrationAreasAnnotator` or the `IntegrationRelevanceDetector`, the `RankingAnnotator` will probably be discarded. At the moment, it assigns every post a ranking of 1.0.

Example: *rankingAuto: 1.0*

5.3.3.9 Class DateAnnotator

This class implements `IRSSAnnotator` and `ITwitterAnnotator`. It extracts the publishing time from a post.

Example: *date: 2012-07-16 14:04:45*

5.3.3.10 Class ContentAnnotator

This class implements `IRSSAnnotator` and `ITwitterAnnotator`. It extracts the text from a post.

Example: *content: "How can you disrespect someone's religion and remove their burka? Nah this isn't halal"*

5.3.3.11 Class TitleAnnotator

This class implements `IRSSAnnotator`, but not `ITwitterAnnotator` because tweets do not have a title.

Example: *title: "Fremdenrecht – Ungarn Rückschiebungen erschwert"*

5.3.3.12 Class AuthorAnnotator

This class implements `ITwitterAnnotator`, but not `IRSSAnnotator` because many syndication feeds do not offer a unique author ID.

While the *authorID* is unique and can have at most 15 characters (letters, numbers and '_'), the *authorName* is not unique and can contain up to 20 UTF-8 characters.

Example: *authorID: "kennizla", authorName: "kenzo ♥"*

5.3.4 Keyword Matching

5.3.4.1 Class KeywordMatcher

This abstract class implements the basic functionality for every keyword matcher. Given a list of tokens (in our case, word stems), we go through this list and process every token. The function `processToken(String token)` itself is abstract, i.e. it will be implemented by extending classes. The problem that we deal with in `KeywordMatcher` is that we do not only want to match single tokens, but also multiple ones, and that longer matches are preferred.

For example, the following sentence

Among the urban citizens, right-wing extremist backgrounds are very uncommon.

might correspond to the following word stems:

among the urban citizen right wing extremist background ar veri uncommon

The stems from the following keywords would match:

[citizen, citizen rights, right-wing extremist, right-wing extremist background, extremist]

In general, we prefer matches for compound words, because they are usually much better for distinguishing integration areas. Our algorithm will therefore match the longest keyword first: *right-wing extremist background.* This will invalidate all other keyword matches, except for *citizen*, resulting in the following matching:

Among the urban [citizens], [right-wing extremist backgrounds] are very uncommon.

Note that *citizen rights* in the list of possible matches represents a problem that results from the normalisation step where commas are removed. In a later phase of the project, we will have to estimate whether normalisation should be skipped to avoid these problems.

Finally, we have set the maximum size of compound words to 4 stems, which conforms to the list of keywords submitted by the project partners. Larger compound words will necessitate a change of this parameter.

5.3.4.2 Class RelevanceKeywordMatcher

This class extends `KeywordMatcher`. It is used by the `IntegrationRelevanceDetector` to match integration-related keywords in the document text. The function `match(List<String> tokens, String language)` will return a `RelevanceMatchResult`, specifying the number of matches in the given token list

and for the given language. The set of keywords is read from the DMS and updated every ten minutes.

5.3.4.3 Class ClassificationKeywordMatcher

This class extends `KeywordMatcher`. It is used by the `IntegrationAreasAnnotator` to match integration-related keywords in the document text. The function `match(List<String> tokens, String language)` will return a `ClassificationMatchResult`, specifying the number of matches in the given token list for every integration area. The formula for selecting the most probable area was given in section 5.3.3.6.

Just like for the `RelevanceKeywordMatcher`, the set of keywords and their assigned integration areas are read from the DMS and updated every ten minutes.

5.3.4.4 Class MatchResult

This is a base class for all match counters. At the moment, it only contains a variable for incrementing the token count.

5.3.4.5 Class RelevanceMatchResult

This class extends `MatchResult`. At the moment, it only contains a variable for incrementing the match count.

5.3.4.6 Class ClassificationMatchResult

This class extends `MatchResult`. It contains a map from strings to integers, counting the number of matches for every integration area.

5.4 Deployment

With the help of the Maven assembly plugin (see section 5.3.1), we generate a JAR file containing all the afore-mentioned classes and the libraries they depend on. The JAR file is then copied to our Hadoop development server:

/home/hduser/PROD/uecrawler-1.0.19-SNAPSHOT-jar-with-deps.jar

The property files (`logging.properties, uniteeurope.properties`) are also located in *PROD*; they were excluded from the JAR file. In `logging.properties`, we set the log level to `FINER`, and generate up to five log files with a maximum size of 10 MB. In `uniteeurope.properties`, we specify the MySQL and Twitter connection parameters.

The Java process is then started as follows:

java -XX:+HeapDumpOnOutOfMemoryError -Xms128M -Xmx256M
-Dfile.encoding=UTF-8 -cp .:uecrawler-1.0.19-SNAPSHOT-jar-with-deps.jar
org.uniteeurope.uecrawler.CrawlerDaemon &

6 Summary

This Deliverable 5.2 describes accurately the various developed features and therefore allows a better understanding of the technical implementation of the project. The structure is built around the four main parts of the project: the Data Management System (DMS), the Mysql database, the website and the web crawler.

The structure of the Zend Framework and all the details of its personalisation through the development of DMS are described in detail in Section 2 of this document. Each feature of the DMS is summarised and structured following the logic of the Model-View-Controller of the framework. The application controllers and models, the different Class of objects and methods, are detailed.

Section 3 presents the structure of the Mysql database, support for all meta-information such as the sources, keywords, annotations and data users of the tools. This section allows a better understanding of the structure of meta-information essential to the implementation of an effective multi-layer pattern to categorise the content. The MySQL database is used by all parts of the project.

Section 4 presents the implementation of the Pimcore CMS and its integration of the different applications of the website. It describes the four modules currently programmed to access the content through the public website: The *Lucene / Solr module* that provides access to the contents of the Unite Europe Index, the *Twitter module* that provides access to twitter content and Twitter account, *Rss module* that interprets the XML and RSS feed, and finally the *Facebook module*, which enables the access to the Facebook network and content.

Finally, Section 5 explains the details of the web crawler application programmed in Java, whose goals are to retrieve information from sources saved in the DMS and verify them according to specific criteria of location and suitability of content before saving them in a powerful Lucene/Solr Index. The Web Crawler organises the information capture and categorises the content.

The next steps are the implementation of the prototype and its improvement. This means the integration of the different templates and all components required like for example the place tags analysis, Facebook content connections, Twitter content optimisation and all other optimisation and statistical processes defined in WP3 and WP4.