



SZAKDOLGOZAT-FELADAT

Wagner Attila

szigorló mérnök informatikus hallgató részére

Közösségi kottamenedzsment rendszer kialakítása

- Mutassa be, milyen funkciókat, szolgáltatásokat biztosíthat egy kottamenedzsment rendszer kisebb közösségek számára, s tekintse át a már létező megoldásokat!
- Tervezzen meg egy webes közösségi kottamenedzsment rendszert – lehetőség szerint létező technológiák felhasználásával! Az alkalmazás legyen moduláris és könnyen bővíthető meghatározott interfészeken keresztül!
- Tervezze meg és implementálja a dokumentumok hatékony tárolására szolgáló funkciót! Legyen meg a lehetőség újabb verzió feltöltésére és régebbi verziók visszaállítására is!
- Implementáljon egy, a dokumentumok kategorizálására és keresésére szolgáló funkciót azok metaadatai alapján!
- Alkalmazzon jogosultságkezelést a rendszerben, hogy csak bizonyos felhasználók érjék el és tudják módosítani az adatokat!
- Értékelje a létrehozott rendszert, és tegyen javaslatot a továbbfejlesztési lehetőségekre!

Tanszéki konzulens: Horváth Zoltán, egyetemi tanársegéd

Külső konzulens: -

Budapest, 2012. október 5.

/ Dr. Imre Sándor /
egyetemi tanár
tanszékvezető



THESIS ASSIGNMENT

for

Attila Wagner

graduate student in software engineering

Developing Music Score Management System for Communities

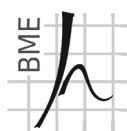
- Research already available services and gather features that are needed by smaller communities in a music score management system.
- Design the architecture of the web based system, using existing technologies where acceptable. The application should be modularized and easily extendable using well defined interfaces.
- Design and implement a solution for storing documents with the possibility to update to a newer revision or to restore an older version.
- Implement a function that makes possible to categorize and search the uploaded documents based on their meta descriptors.
- Implement user right management so as only specific users can access the system and make changes.
- Evaluate the solution, and highlight further areas and functions that could be implemented.

Consultant (from the department): Assistant Lecturer Zoltán Horváth
Consultant (from other association): -

October 5, 2012.
Budapest

/ Sándor Imre, DSc. /
Head of Department

Budapest University of Technology and
Economics
Faculty of Electrical Engineering and
Informatics
Department of Telecommunications



1117. Budapest, Magyar tudósok körútja 2. I. Ép. B.
121.
Phone: +36 1 463-3261 Fax: +36 1 463-3263
URL: <http://www.hit.bme.hu>

Hallgatói nyilatkozat

Alulírott Wagner Attila, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Százhalombatta, 2012. 12. 12.

.....
Wagner Attila



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications

Attila Wagner

**Developing Music Score Management System for
Communities**

Consultant:
Zoltán Horváth

2012.

Table of contents

Feladatkiírás	1
Hallgatói nyilatkozat	3
Table of contents	6
Kivonat	8
Abstract	9
1. Introduction	10
1.1. Personal motivations	10
1.2. Existing solutions	10
2. Specification	13
2.1. Target audience: choirs, musical bands	13
2.2. Document management	13
2.3. User right management	13
2.4. Version control	14
2.5. Error reporting	14
2.6. Requests and offers	14
2.7. Document format conversions	15
3. Design	16
3.1. Content management for the project	16
3.2. Use-cases	17
3.3. Flowchart	21
3.4. Storing metadata	23
3.5. Hosting strategy: with or without cloud?	24
3.6. Choosing a platform	26
3.6.1. Own code from the start	27
3.6.2. MediaWiki	27
3.6.3. WordPress	28
3.6.4. Decision	29
3.7. Database and file storage	29
3.7.1. Storing files in file system	29
3.7.2. Storing files in the database	29
3.7.3. Decision	31
3.8. Modularization	31
3.8.1. List of converter modules	32
4. Implementation	34
4.1. Splitting the work	34
4.2. WordPress plugin basics	35
4.2.1. Structure of the plugin	37
4.2.2. Plugin core file (kkm.php)	38
4.2.3. Installation script (install.php)	40
4.2.4. Bootstrap (bootstrap.php)	42

4.3.	Module handling.....	45
4.3.1.	Module loading	46
4.3.2.	Calling the modules.....	48
4.4.	Admin interface.....	48
4.4.1.	Tag categories	49
4.5.	User interface	50
4.5.1.	List of compositions.....	51
4.5.2.	Adding or editing a composition	52
4.5.3.	The tagging interface	52
4.5.4.	Composition details page	59
4.5.5.	Deleting a composition	60
4.5.6.	Uploading a document.....	60
4.5.7.	Document details page.....	66
4.5.8.	Version control in work	66
4.5.9.	Converting a document	68
4.5.10.	Searching the database	71
4.5.11.	Tag list	73
4.6.	Localization.....	74
5.	Evaluation and future plans	76
6.	List of figures	77
7.	Bibliography and references.....	79
8.	Appendix: Function library documentation (API).....	81
8.1.	File and upload related functions (kkm_files).....	81
8.2.	Module related functions (kkm_modules).....	86
8.3.	Module options processing related functions (kkm_options).....	92
8.4.	Search related functions (kkm_search)	93
8.5.	Tag management related functions (kkm_tags)	97
8.6.	User related functions (kkm_user)	106
9.	Appendix: Module interface documentation (API).....	107
9.1.	Converter module interface.....	107
9.2.	Diff module interface.....	110
9.3.	Parser module interface	111
10.	Appendix: Content of the attached CD	115

Kivonat

Szakdolgozatom célja egy olyan internetes alkalmazás fejlesztése, amely kórusok és zenekarok számára lehetővé teszi, hogy az általuk használt zenei dokumentumokat (például kották, hangfelvételek, dalszövegek) tagjaik megosszák egymással. A rendszer célja a központosított dokumentumkezelésen kívül a fájlokkal végzett munka segítése is automatizált funkciókkal.

A rendszer részét képezi a felhasználói jogosultságok kezelése, így annak ellenére, hogy a felhasználók bárholnan elérhetik dokumentumaikat, külső személyek részére a rendszerben tárolt adatok hozzáférhetetlenek.

A felhasználói felületen lehetőség van zeneművek szerint böngészni a dokumentumok között, dokumentumok korábbi verzióit elérni és visszaállítani verziókezeléssel, és természetesen az egész adatbázis kereshető címek, fájlnevek, és tetszőleges meta adatok alapján.

A rendszer automatikus szolgáltatásai lehetővé teszik a fájlok feltöltésekor azok helyességének ellenőrzését, meta adatok kinyerését, módosítását és azok fájlba történő visszaírását. A feltöltött dokumentumokat lehetőség van más formátumra alakítani a rendszerhez tartozó konvertereken keresztül. Ezen szolgáltatások tudása a rendszerhez írt beépülő modulokkal bővíthetők.

Abstract

The goal of my thesis is to develop an internet application that provides a way for choirs and musical bands to share their music related documents (eg. sheet music, audio record, lyrics) between their members. Alongside the centralized document management, assisting the members' work with the files stored in the system is also part of the goal.

The software supports user right management, so no unauthorized user can access any information stored in the system, yet the members of the band can access their documents anywhere with a modern web browser.

On the user interface one can navigate between documents based on the musical composition they belong to, access or restore older versions of the documents, and search between them based on titles, filenames, and any metadata assigned to them.

The automatized services of the software validate the file when a user uploads them, loads any stored metadata stored in the document itself, and write back any modification in these attributes the user wishes to make. Uploaded documents can be converted from one format to another with converters that are part of the system. These services can be further expanded with easy-to-develop add-on modules.

1. Introduction

There are several portals on the internet that aim to collect and share free sheet music and to provide printed or electronic documents for a composition. One thing that these solutions have in common is the availability for everyone, that they provide a single global collection. However this aspect makes them unsuitable for smaller musical ensembles to share their own documents between the members including audio records, lyrics, notations and other documents. For them, an application that supports working on the documents and helps planning program for events.

The topic of this thesis is the development of an online solution that targets these groups, starting from collecting the needed functions through the planning of the architecture to the final software.

1.1. *Personal motivations*

During the years I have attended elementary school, I have learnt playing the flute. As I have started attending a high school in the capital, my free time was reduced to such extent that I had to abandon my musical studies. Years later however I have bought my own instrument, and started my self-education in hope that one day I could be part of a group playing Irish music or an alternative rock band.

I have choose this topic for my thesis in hope that – even if it is an indirect way – I can learn about music and help the work of others who create or play music.

1.2. *Existing solutions*

Currently existing solutions – according to my consultant who participates in a choir for years – do not fulfill the needs of a choir. These services focus mostly on collecting documents and making them available in some

categorized and searchable form for the whole world. Managing documents that contain the songs in a version as they play them is not possible.

They also miss functions that support the editing of these documents, and they do not assist the users' workflow with document format conversions that could be automatically done using existing software. The usages of these external converters are not always straightforward, so a lot of users cannot work with them.

The screenshot shows the IMSLP/Petrucci Music Library website. The header includes the IMSLP logo and the text 'Petrucci Music Library'. Below the header, there are statistics: 53,926 works, 190,737 scores, 15,255 recordings, 7,205 composers, and 192 performers. A sidebar on the left contains navigation links like 'Main Page', 'Recent changes', 'Random page', 'Browse scores', 'Composers', 'All people', 'Nationality', 'Time period', 'Instrumentation/Genre by Melody', 'WIMA collection', 'Browse recordings', 'Participate', 'Other', 'Donate', 'For iPhone & iPad', 'Classical Scores', 'Toolbox', and 'Associated with Amazon.com, Amazon.ca, Amazon.co.uk, Amazon.de, Amazon.fr, Peachnote.com'. The main content area includes a search bar, a 'How to Contribute Works' section, a 'News' section with recent updates, an 'About us' section explaining the project, and a 'Copyright' section. The footer contains a modification date, license information, and various icons including GNU FDL, Creative Commons, and MediaWiki.

Fig. 1. IMSLP/Petrucci Music Library – a MediaWiki based collection [1]

Developing Music Score Management System for Communities

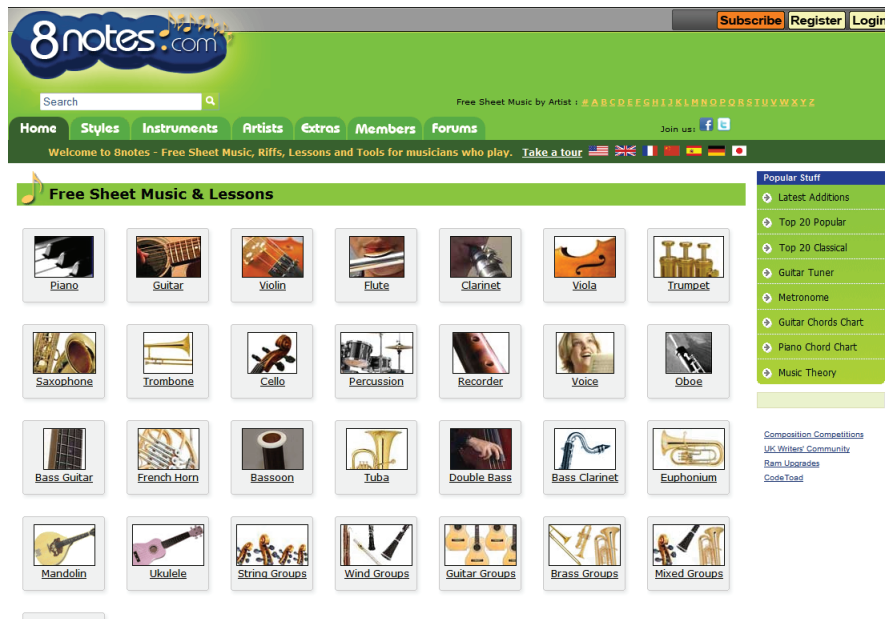


Fig. 2. 8notes.com homepage – well structured and easy to navigate [2]

Fig. 1 and Fig. 2 contain the screenshots of the homepage of two existing global collection [1][2]. These two use different software and while are unable to fulfill those requirements we aim to do with this project, display good categorization and browsing functions. The 8notes.com page also shows good search functionality, something our implementation could follow later on.

2. Specification

After the brief review of the existing solutions, I would like to iterate over the points that distinguish the planned CMS from those already available.

2.1. Target audience: choirs, musical bands

While currently available services provide global collections, in this project I want to develop software that could be freely downloaded and installed by a choir or band much like wiki or blog engines. I aim to provide an easy way for completing tasks members face in they everyday work in the band or choir.

2.2. Document management

One of the primary functions of this CMS is to store and manage musical compositions and their related documents. These documents show great inhomogeneity as an audio record stored in an MP3 file can be uploaded next to the MIDI recorded from a synthetizer, or photos of printed or hand written sheet music as well as project files of music notating software.

Parts of the document management functions are the upload, browsing and search interfaces. Documents need to be searchable based on their metadata and the composition they belong to.

2.3. User right management

As the system under design is an online service, user authentication and authorization functions naturally must be part of the software. There needs to be a way to allow only specific users to browse the documents and to limit any content editing actions to a separate group of users.

2.4. Version control

Documents stored in the system may change with time. Mistyped lyrics may get updated to the correct version, or someone replace a partially notated song with the full document. In cases like this, it is important that other members always get the most recent version of the documents.

Retrieving and restoring older versions of documents is also important. The system should provide a way to download any version of a document and to mark that the current version.

2.5. Error reporting

The software should provide a way for its users to report an error in a document. Users with editing rights should be able to list these reports, correct the documents, and upload the newer version. When users upload a version of a document that contains fixes, they should be able to select the report ticket that they worked on. Users who reported fixed errors should receive notifications about the new version.

There may be reports that aren't considered errors by others. Users with editing rights should be able to close these requests and write an explanation on their decision.

2.6. Requests and offers

Users may seek documents that are not uploaded into the database. When they need such a document, they should be able to file a request ticket into the database.

Users may also have documents in their possession that they haven't uploaded yet, but if someone needs them, they plan to do so. Uploading paper based content requires a lot more time than digitally available ones due to the scanning and retouching process involved. In case they have such documents

the users should be able to file an offer ticket into the database that contains what documents are they able to provide on request.

After a user saves an offer or a request, the system should try to find matches between the stored tickets and if a match found, it should notify the concerned users.

2.7. Document format conversions

The main purpose of the system is to hold many documents in its database that come from different sources in different file formats. Some of these document types are source formats that could be used to generate other documents using them as input with specific conversion software.

Some users may need documents in a specific format (for example: a PDF that could be viewed and printed on any platform), but the sheet music for a song is only available as a document for a notation software. In cases like this, the user should be able to request a format conversion that will be executed by the system, and if it succeeds users can get the document in a format that they can use with their devices.

3. Design

3.1. Content management for the project

The timeframe for this project was two semesters, and involved three people since the first semester (two students and our consultant). To manage design notes, to-do lists, and later the source code, we needed a service.

We chose Assembla [3], as it provides a free all-in-one solution including a wiki, an error reporting system, provides SVN and Git repositories, and we had previous experience with this service. For source code management we went with SVN solely for the reason that our group had more experience with it than with Git, so learning the usage of a new environment did not set back the works on the project.

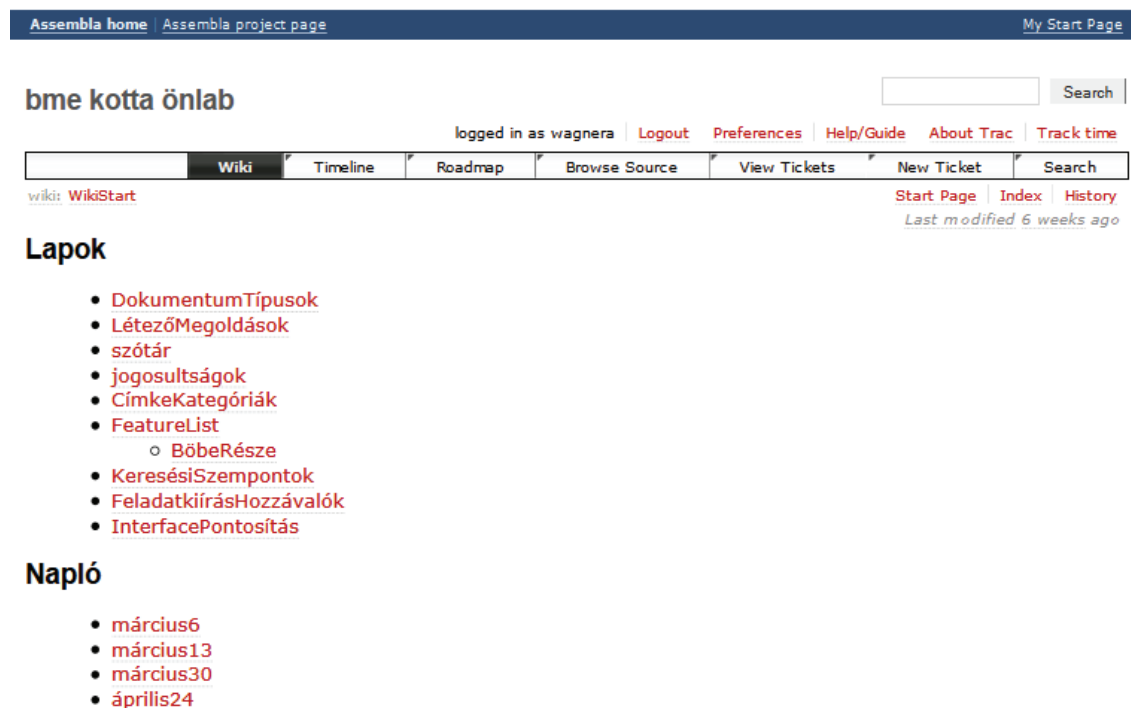


Fig. 3. The project page on Assembla [4]

3.2. Use-cases

I was entrusted with the task to write a list of use-cases that the final product should support. I based this list on the original flowchart made by Zoltán Horváth [5]. The system of course should realize some other use-cases not present in this list which describe trivial actions in an online multiuser software solution. These omitted actions include user registration, login, logout, assigning roles to users, etc.

Use-case name	Adding a composition
Description	A user adds a composition to the database.
Actors	USER (editor)
Basic flow	The user fills out a form.

Use-case name	Editing a composition
Description	A user changes the data associated with a composition.
Actors	USER (editor)
Basic flow	The user fills out a form.

Use-case name	Deleting a composition
Description	A user deletes a composition.
Actors	USER (editor)
Basic flow	The user clicks on a delete button on the composition page, and confirms the deletion. Every associated document gets deleted, and the composition gets removed from the database.

Use-case name	Sending request for a document
Description	A user requests a specific document that belongs to a given composition.
Actors	USER
Basic flow	The user fills out a form.

Developing Music Score Management System for Communities

Use-case name	Sending request for a composition
Description	A user requests documents that belong to a given composition.
Actors	USER
Basic flow	The user fills out a form.

Use-case name	Sending offer for a document
Description	A user offers the community to upload a specific document if someone needs them.
Actors	USER
Basic flow	The user fills out a form.

Use-case name	Uploading a document
Description	A user uploads a digitally available document to an already existing composition.
Actors	USER (editor)
Basic flow	The user fills out a basic form containing mandatory fields (including the file filed), and if the file was sent successfully, fills out another form regarding the metadata of the document.

Use-case name	Uploading a modified document
Description	A user uploads a modified version of a document already stored in the system.
Actors	USER (editor)
Basic flow	Same as in 'Uploading a document', but the previous versions of the document get saved in a way that it could be restored later.

Use-case name	Deleting a document
Description	A user removes a document from the system.
Actors	USER (editor)
Basic flow	On the document editing interface the user chooses to delete the document, and confirms it. The file gets deleted.

Use-case name	Downloading a document
Description	A user requests a file stored in the system for saving a local copy.
Actors	USER
Basic flow	The user clicks on a download link.

Use-case name	Automatically adding metadata to a document
Description	When a user uploads a document, the system loads the metadata from the file, and suggests them for saving.
Actors	SYSTEM, USER (editor)
Basic flow	After the file arrives at the server, the system loads metadata from it. If there was data the system could load, it will suggest the user to save the descriptors into the database. The user may change the values presented by the system. The new metadata gets written back into the file once the user saves the uploaded document.

Use-case name	Editing document metadata
Description	A user modifies the associated meta descriptors of a document.
Actors	USER (editor)
Basic flow	The user fills out a form.

Use-case name	Searching for a composition
Description	A user searches for compositions with a specific string.
Actors	USER, SYSTEM
Basic flow	The user gives a search query string. Any composition that matches the terms in the query string (either in its title or in its meta descriptors) will be listed to the user by the system.

Developing Music Score Management System for Communities

Use-case name	Searching for a document
Description	A user searches for documents with a specific string.
Actors	USER, SYSTEM
Basic flow	The user gives a search query string. Any document that matches the terms in the query string (either in its filename or in its meta descriptors) will be listed to the user by the system.

Use-case name	Browsing compositions
Description	A user browses the list of available compositions.
Actors	USER
Basic flow	The user views the list of saved compositions.

Use-case name	Browsing documents
Description	A user browses the available documents for a composition.
Actors	USER, SYSTEM
Basic flow	The user views the list of documents that belong to a given composition. Old versions of the documents are hidden, but may be viewed by the user on request.

Use-case name	Restoring a document version
Description	A user chooses to restore a previous version of a specific document.
Actors	USER (editor)
Basic flow	The user selects a version of the document which is not the current version, and marks it as the current version.

3.3. Flowchart

Based on the use-case list provided in the previous point and using my consultant's original flowchart [5], I drew a new diagram showing the workflow of the system along with two different user groups' workflow.

In the diagram I used colors to differentiate the actors:

- Blue arrows follow the actions of a user with read-only access to the system. These actions do not involve editing, uploading, or modifying the database in any other way.
- Green arrows represent the workflow of an editor. Editors of course can start any action that a normal user has capability for, but only they can modify the content of the database.
- Purple arrows mark the flow of actions the system automatically takes. These mostly belong to the conversion functions.

The diagram also differentiates between paths that could be considered as constant steps in the workflow (solid straight lines), and paths that may be optionally taken if the actor decides to do so (dashed lines). These optional paths are mostly branch along the capabilities of the actor, and on the decisions of the user if acceptable. For example, if a composition does not exist in the database, any user could add it, but it is not their obligation to do so. This means, that even if a user found a missing composition, it could not be expected by any further steps that it gets created.

Developing Music Score Management System for Communities

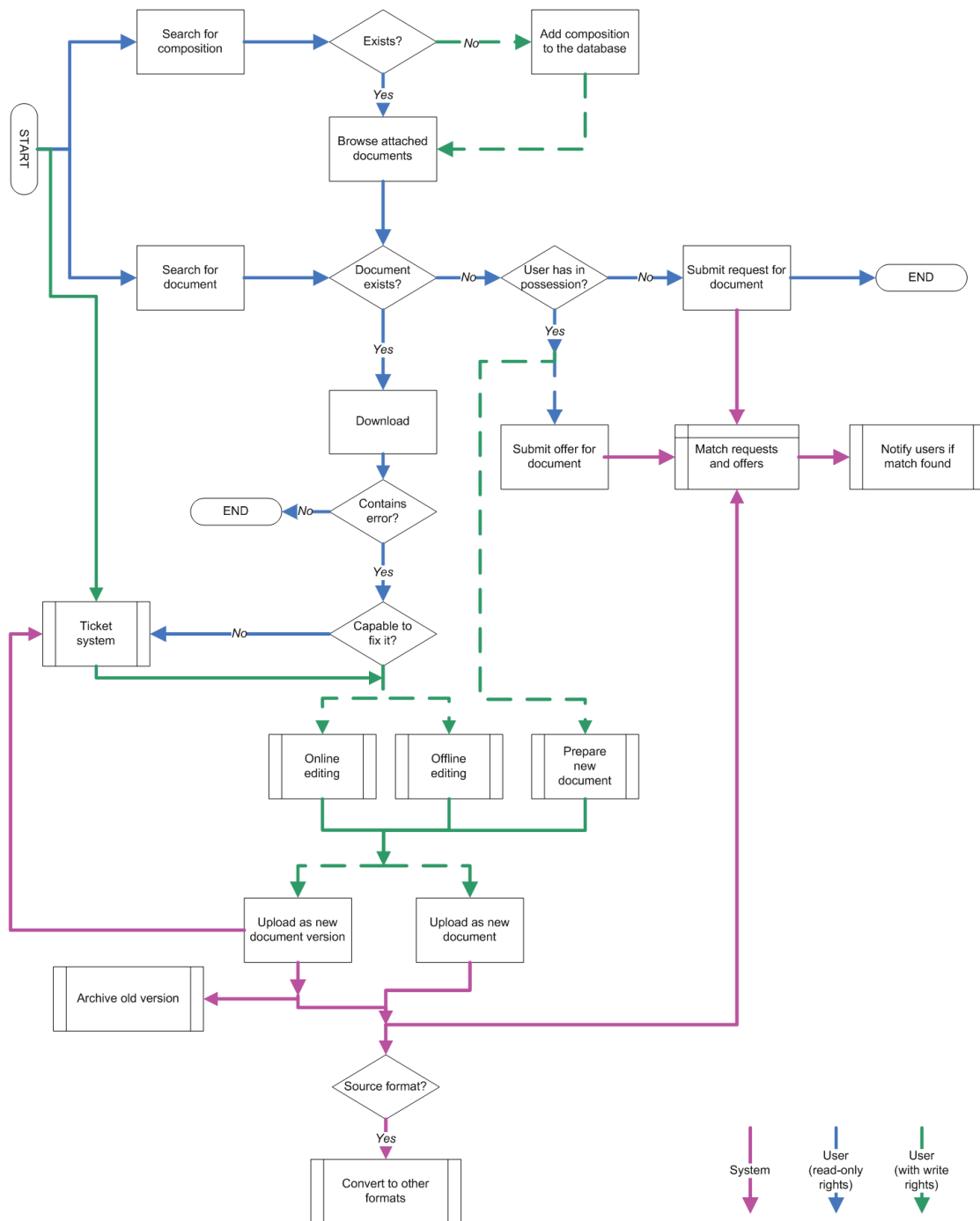


Fig. 4. Flowchart (overview)

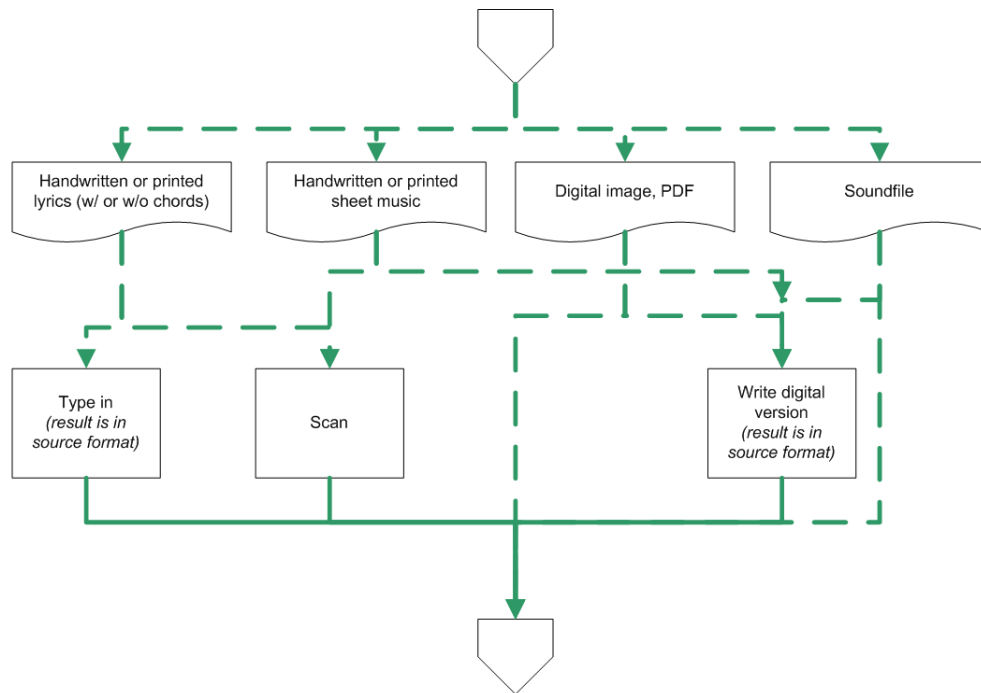


Fig. 5. Flowchart (prepare new document)

3.4. Storing metadata

There are several different approaches to handle the metadata, ranging from identifying possible properties and storing them in a tabular format to using a generic method that can handle new descriptor types without modification. I have suggested the usage of a hierarchical tag system, and we agreed to go along my suggestion.

In this system every tag belongs to a single tag category, and optionally can be a sub-tag of a single other tag. For example, there is a category named *Quality* which has three tags: *Low quality*, *High quality*, *Very high quality*. The *Low quality* tag can be assigned to a scanned document of poor resolution, and a studio recording can be marked with *Very high quality*. Let us assume that the *Very high quality* tag is a sub-tag of the *High quality* tag. In this case, if the user searches for high quality documents, every document having been tagged as either *High quality* or *Very high quality* will be returned.

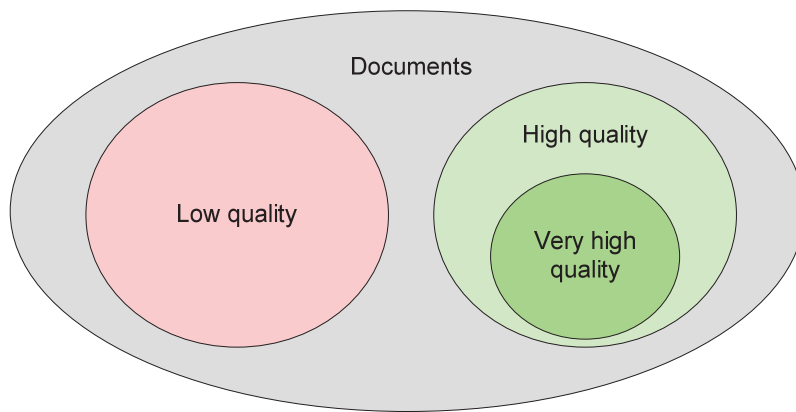


Fig. 6. Illustration to the hierarchical tags example

3.5. Hosting strategy: with or without cloud?

Nowadays 'cloud hosting' is a term that often used to catch the eyes and ears of customers. There are several types of cloud architecture available, ranging from IaaS (Infrastructure as a Service) to SaaS (Software as a Service), and each step on the scale represents different management requirements. [6]

Some level of virtualization can be done without modifying the application that was built to run on a physical hardware to run on the virtualized platform. In the case of our web application this mostly means the PaaS (Platform as a Service) solutions, as using a shared web hosting provider instead of an own web server.

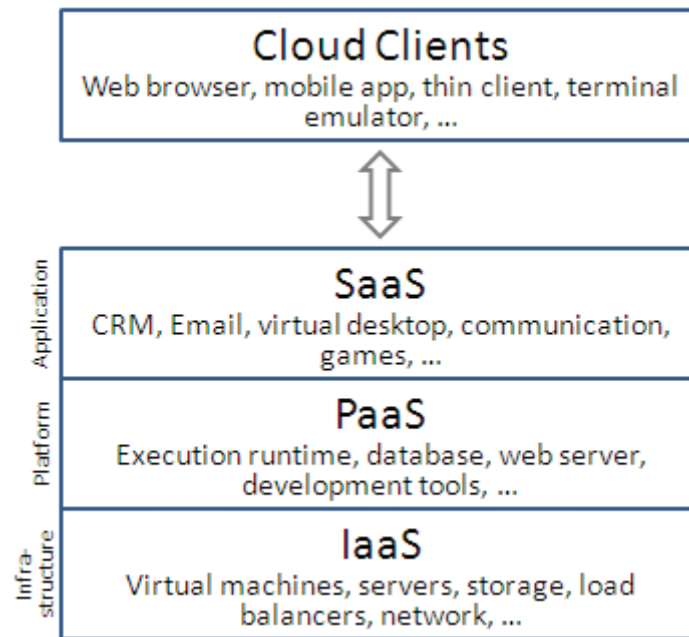


Fig. 7. Cloud computing layers [7]

To utilize the true power of cloud computing, the application must be aware of its hosting environment. Services that are hosted on infrastructure level cloud platforms do have advantages compared to applications that run on a single server:

- Better availability: another instance can handle the tasks of a failed component.
- Scalability: peaks or sudden increase in requests can be handled by adding new instances of components (servers, databases, etc.)
- Lower response time: distributed hosting means that end users can transparently access the servers with lesser workload and servers that are closer to them on the network.

However these advantages do not influence the current project as:

- The service provided by the software isn't critical to the end users, as downtime should not have any effect on them in health, security, or financial aspects.

Developing Music Score Management System for Communities

- Number of users will be low in terms of an online service, and peaks in workload aren't anticipated.
- This project is designed for small groups of people, typically residing in close proximity of each other. This combined with the previously mentioned arguments eliminate the need for load balancing.

Based on the above explanation I came to the conclusion that for this project any development work that relates to cloud computing can be safely omitted.

3.6. Choosing a platform

For the realization of this project, a software environment and possibly a framework should be chosen. The goal was to implement the most functionality that could be done in the given timeframe, so choosing a language and an environment that I have experience with is crucial to reduce the overhead that learning a new set of development tools imply.

I would like to enumerate possible solutions that are all based on the following (or better) server software environment:

- Apache 2.2 HTTP server with mod_rewrite and mod_headers enabled [8]
- PHP 5.2 with GD2 library for possible image processing [9]
- MySQL 5 [10]
- If the project utilizes existing applications (for example LilyPond for PDF generation from its own document format), the server must be configured to allow the execution of native applications from within the PHP script.

3.6.1. Own code from the start

My original idea was to develop the software without using existing frameworks. This way we get the greatest possible freedom in terms of architecture and user interface design. Going without the usage of other software dependencies we further eliminate the problems that updating and maintaining these packages mean.

3.6.2. MediaWiki

Based on their experience working with MediaWiki [11] based systems as editors, my consultant and the other student on this project suggested basing our solution on this software. Along their suggestion, it was my task to examine the main functions, the development API, and the extensibility of MediaWiki and to check what functionality it provides that helps the implementation of the project.

I had an objection against the user interface of the wiki, as it shows way more links and buttons than an average user needs to use, which makes it cluttered. As our main intention is using this software in a different way it was designed to work, a serious redesign would have been necessary.

MediaWiki includes functions that make it possible to use skins to change the user interface, but it is very limited and assumes a wiki-like usage. It would have been impossible to customize the interface to meet our needs in such a short period of time that could be afforded for this task.

As a workaround I suggested using the MediaWiki engine in the background only, calling its web service API [12], and implementing our own user interface. However, as it later turned out, most of the extensions we would have used did not support this method, and as a result the only functions we could gain from this method was the user management and some limited version controlling.

3.6.3. WordPress

While I worked on a different project at the same time as we were designing this software, I got to know the WordPress blog system [13] and I have gained knowledge in its plugin API. [14]

WordPress showed a far greater extensibility and customizability than MediaWiki, and I found its development API both better structured and better documented than the API of the other system.

I mentioned in section 3.6.1 that using external software means that they must be updated to fix security flaws in them, and that this process could be considered an overhead while running an instance of our software. In WordPress, there is a one-click way to update the core system – a function MediaWiki lacks – and there is a similar one-click method to update all of the installed plugins at once. [15] The latter means that our final software would have an auto-update feature without any development work done on our end.

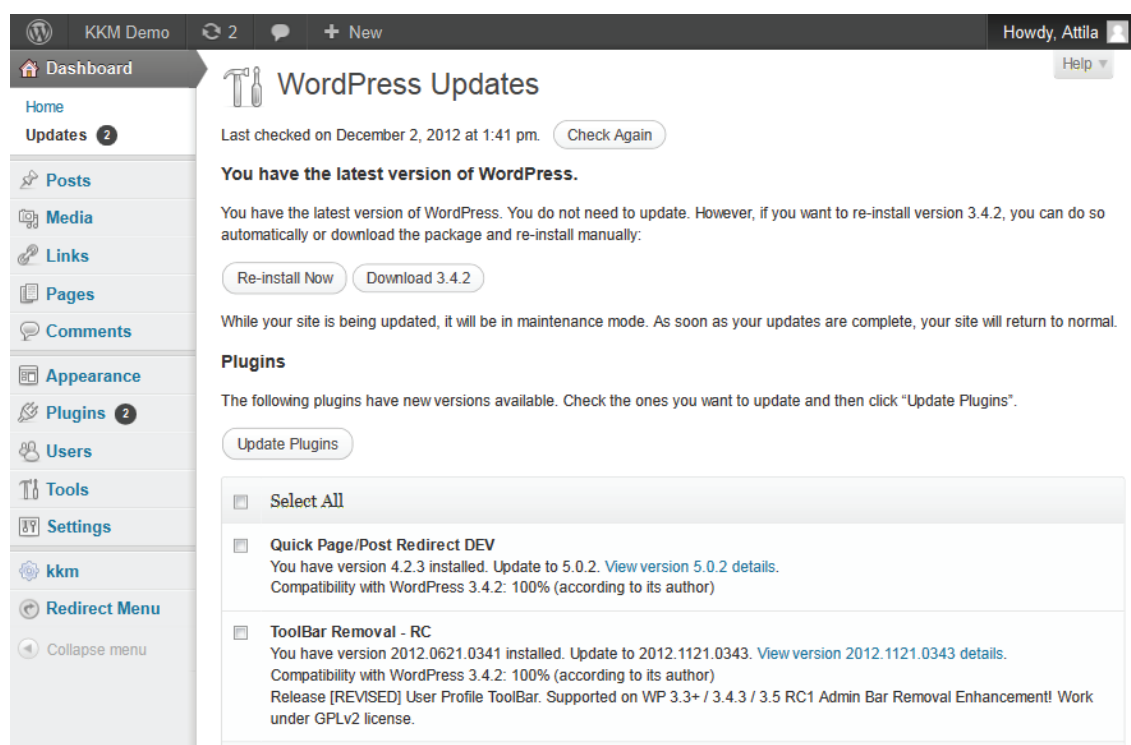


Fig. 8. Update page in the admin section of a WordPress installation

3.6.4. Decision

After I presented my findings, our group agreed to continue the design and implementation process focused on the solution using WordPress. This decision was also backed by the fact that for every extension we would have possibly used later with the MediaWiki implementation, we found a plugin with the same functionality for WordPress.

3.7. Database and file storage

There are two generic ways of storing uploaded content on the server side, and both have its own advantages and drawbacks.

3.7.1. Storing files in file system

The trivial method is storing files in the file system of the underlying OS. Benefits of this method are include the possibility to access files outside the software, caching by the operating system, and optimized methods of delivering them to the user over HTTP by PHP, Apache and proxies (for example: `http_send_file()` PHP method [16], X-Sendfile HTTP header [17]).

There are two main drawbacks of this method. One is the requirement of using different backup solutions for the files and their metadata (assuming they are stored in database), and the other is the consistency issue between files and their metadata. For example there is a possibility that an updated version of the file fails to be written to the storage, while its metadata in the database already belongs to the newer version. Issues like this however can mostly be circumvented by using transactions in the database for the metadata update with the commit point after the verified file system modifications.

3.7.2. Storing files in the database

In most database server software there is a method for storing large amount of binary data in a database table. MySQL provides the BLOB (Binary Large Object) data type [18].

While storing the data and metadata in one place addresses the issues described above, it cannot be viewed as a universal ‘best practice’ solution. Accessing the database is considered an expensive (if not the most expensive) step in generating and delivering content to the user. The PHP module must connect to the database server, run the queries, and process the results before sending it over HTTP to the user’s browser. DB servers usually cache query results in memory, and for large content it means large memory consumption. The data is streamed from the database to the user by the PHP server, which means both the PHP interpreter along with its hosting HTTP server and the database server are occupied while the user downloads a single file.

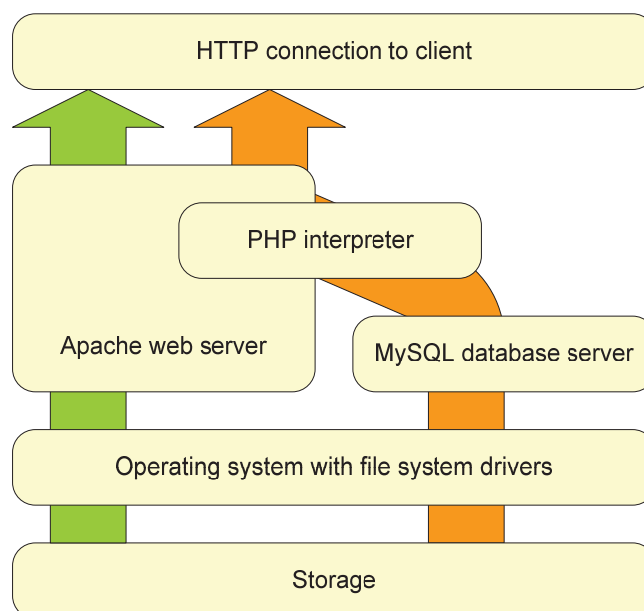


Fig. 9. Serving downloads from file system (green) and from database (orange) using the same host for every service application

Most web host providers use a single database server for multiple user accounts and enforce strict limits on database size which impacts the amount of uploaded content that could be stored by the software. There is also an option for limiting concurrent connections to the database server, which makes the previously described case a more severe issue. Using a private dedicated server may liberate these limits at the cost of a more expensive hardware.

Economy	Deluxe	Ultimate
\$1.99 /month	\$8.99 /month	\$14.99 /month
<ul style="list-style-type: none"> • 10 GB Space • Unlimited Bandwidth • 100 Email Accounts² • 10 MySQL Databases (1 GB ea.) 	<ul style="list-style-type: none"> • 150 GB Space • Unlimited Websites^{oo} & Bandwidth • 500 Email Accounts² • 25 MySQL Databases (1 GB ea.) 	<ul style="list-style-type: none"> • Unlimited Websites^{oo}, Space⁺ & Bandwidth • 1000 Email Accounts² • Unlimited MySQL Databases (1 GB ea.) • FREE^{tt} Premium DNS, Site Scanner, SSL Certificate with Fixed IP address
3 mo - \$1.99/mo Save 67%+ SALE	1 mo - \$8.99/mo	1 mo - \$14.99/mo

Fig. 10. Pricing of GoDaddy, one of the most popular host providers [19]

3.7.3. Decision

I choose the file system storage for this project for the following reasons:

- We use external applications that depend on files stored in the file system for conversions.
- WordPress uses this convention.
- Easier implementation.

If the project was intended to be used in cloud environments, the manageability itself would weight the scale on the latter method.

3.8. Modularization

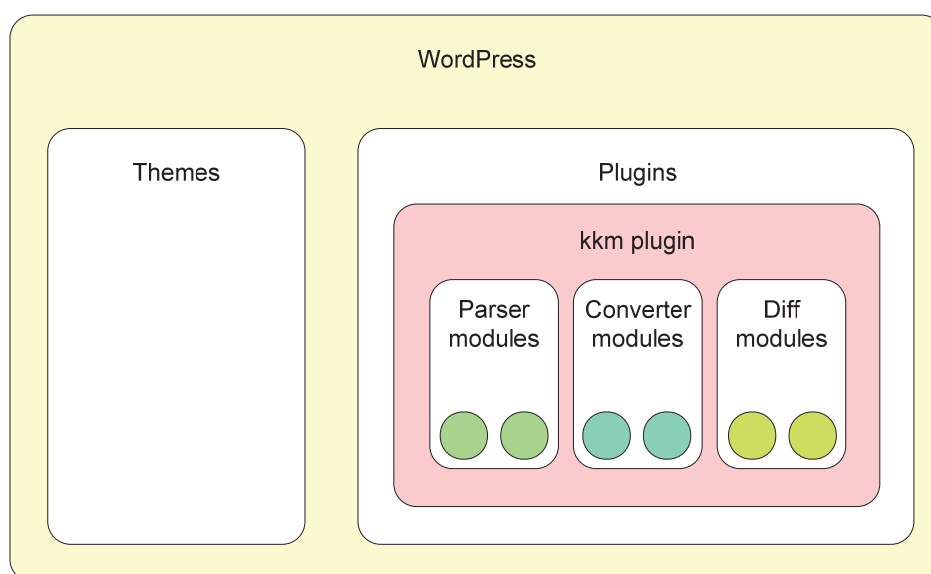


Fig. 11. Overview of the modularized software architecture

Developing Music Score Management System for Communities

Our group decided together what types of modules we should use to separate logical blocks with distinct functionality. We agreed that the following three types would be sufficient:

- Parser modules to handle file validation, meta information extraction and addition as well as preview generation.
- Converter modules that are used when there is need to convert from one document format into another (for example: from LilyPond to PDF), or when some modification is needed on the document (transposing a song).
- Diff modules to show the difference between versions of the same document.

Every module must have a PHP class that implements an interface specific to the module type, but there won't be any further restrictions on its structure or inner workings. If calling external applications is allowed, and there is a working utility that implements the functionality of a module, only writing a thin wrapper around that external application is enough.

In the next chapter I will explain when and how these modules are used in greater detail.

3.8.1. List of converter modules

As part of the planning process, we assembled a list of format conversions that should be supported by the final software. Some of these could be easily done by calling an external application, while others need custom logic.

- LilyPond document to PDF sheet music (can be done by calling LilyPond)
- LilyPond document to MIDI (can be done by calling LilyPond)
- Lyrics in plain text to LilyPond document

- Lyrics in plain text to PowerPoint presentation
- Lyrics with chords in plain text to LilyPond document
- MIDI to LilyPond document (can be done by calling external application)
- Guitar tablature in text file to LilyPond document
- Guitar tablature in text file to plain text containing lyrics
- Text formats to PDF document

4. Implementation

In this chapter I will present the parts of the final software I developed by describing its user interface and showing the code that works in the background.

4.1. *Splitting the work*

At the beginning of the implementation process, we decided to split the work between the two students participating in this project. As both of us had our own time schedule, the best solution was to make the split at a point that made it possible to work independently of each other.

My half of the project was to implement the plugin itself, provide the framework that the converters can use, as well as to build the user interface.

The other half consisted of writing the modules that the system could call, including the converters, file format validators and the difference calculator modules. As some of these would require custom logic, this could be considered as an equal to the first half in workload.

This allocation meant that the common point between our work was the API the modules should fulfill. As I had more experience with WordPress development, and the other student had more extensive knowledge on the file formats system should support, we both agreed to this apportionment.

4.2. WordPress plugin basics

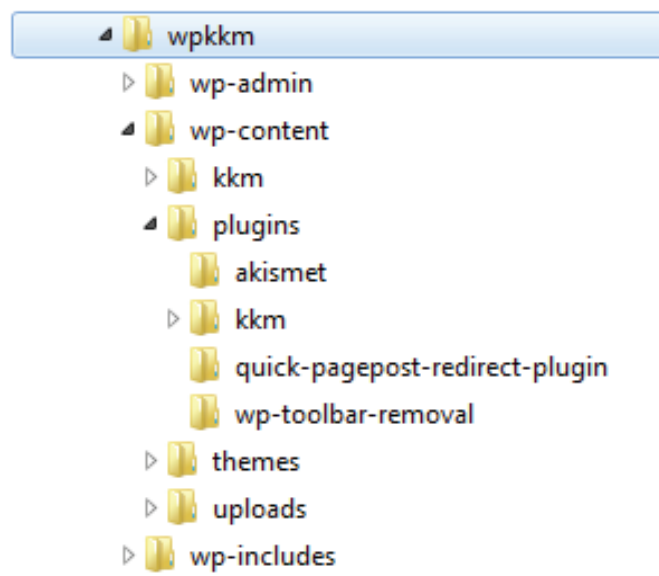


Fig. 12. Directory structure of a WordPress installation

In WordPress there are two types of plugins: *normal plugins* that can be turned on or off on the admin interface and *must use plugins* that are loaded in a different state of the page rendering process, and cannot be turned off without removing them from the file system. The latter ones while are ideal in some cases, lack the automatic updating feature WordPress provides for normal plugins, and can only use a restricted API from the WordPress core.

Normal plugins are loaded from the `wp-content/plugins` directory after WordPress loaded the must use plugins and done some basic initialization. They have a core file with a header comment in a strict format, and this is the file loaded by WordPress, and the content of the header will be displayed on the admin interface. [20] Of course this header can contain a lot more detail than what is present in our plugin:

```
/*
Plugin Name: kkm
Description: Community based music score management system.
Version: 1.0
Author: Attila Wagner, Erzsébet Frigó
*/
```

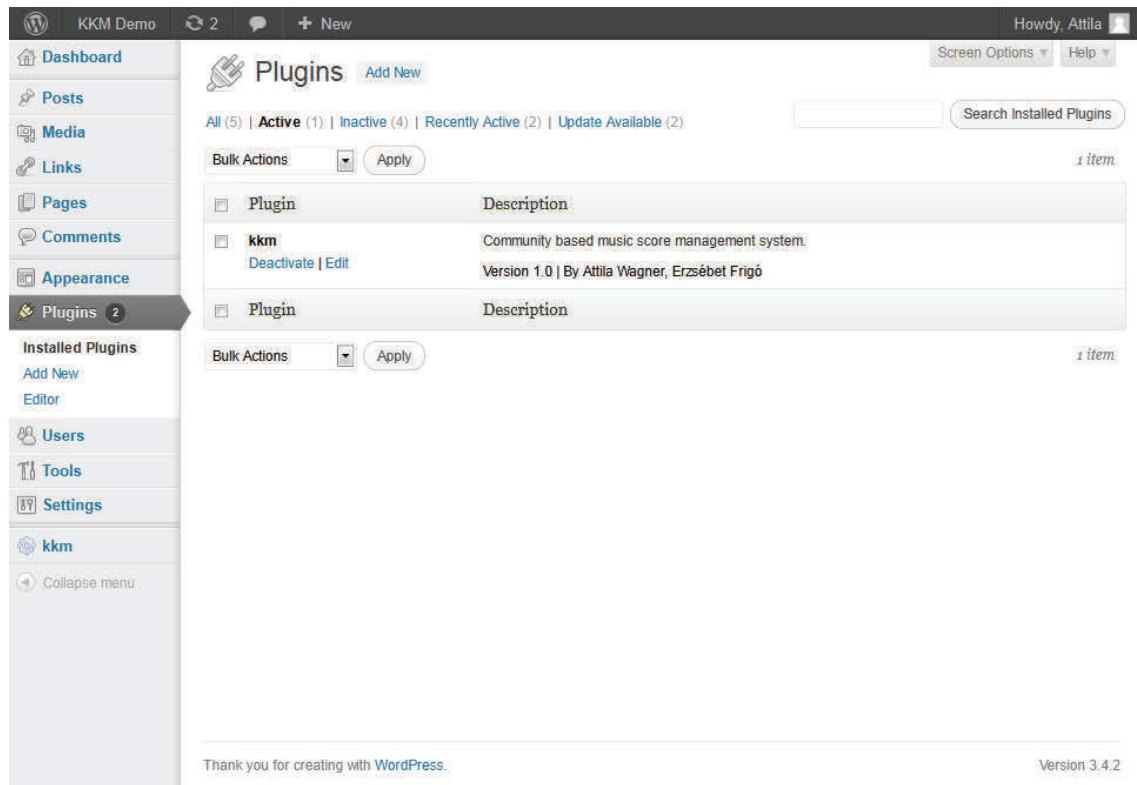


Fig. 13. The activated plugin on the admin interface showing its description

A plugin can subscribe to *Action hooks* and *Filter hooks*. [21] The difference between the two hook types is that while *actions* mostly could be considered as notifications for the plugin (someone posted a comment, the plugin can now take action), the *filters* are functions to modify the result of a processing function (the user name would be rendered as 'Anonymous', the plugin may change it to another string).

4.2.1. Structure of the plugin

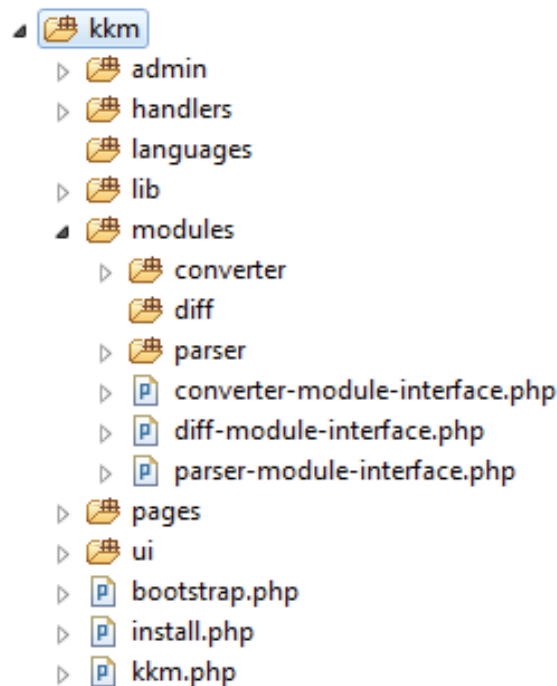


Fig. 14. Directory structure of the plugin

The plugin I wrote consists of 200,000 characters of code in more than 50 files. A codebase of this size needs to be well structured to be maintainable.

In the root directory of the plugin are placed the plugin core file (`kkm.php`), the script responsible for the installation and the removal of the plugin (`install.php`), and a bootstrap that starts the processing of the users' requests (`bootstrap.php`).

The configuration pages accessible from the admin section of WordPress are placed in the `admin` directory.

The pages visible for normal visitors are in the `pages` directory. These include the list of compositions, the search result page, etc.

In close relationship to the pages are the action handlers in the `handlers` directory. These files are responsible for processing user actions, such as saving an uploaded document or updating the metadata of a composition.

The `ui` directory contains JavaScript and CSS files used on the pages of the plugin.

The `modules` directory contains the interface definitions for the three module types, and a directory for each. I will describe the structure of these modules in greater detail in section 4.3.

Most of the logic in the plugin is contained in the files inside the `lib` directory. The static classes defined in here are collecting common functions that are called from multiple different locations in the plugin. The documentation of these files is in the Appendix: Function library documentation (API) at the end of my thesis.

Lastly, the `languages` directory is the one that contains the files needed to localize the user interface of the plugin.

4.2.2. Plugin core file (`kkm.php`)

As this is the file that WordPress includes when it loads the plugin, I placed the `require` directives here to include the function libraries. Action handlers are also defined here:

```
/*
 * Register WP hooks.
 */
register_activation_hook(__FILE__, array('kkm',
    'plugin_activation'));
register_deactivation_hook(__FILE__, array('kkm',
    'plugin_deactivation'));
add_action('init', array('kkm', 'init'), 99);
add_action('admin_menu', array('kkm', 'register_admin_menus'));
add_action('template_redirect', array('kkm', 'on_template_redirect'),
    1);
```

The first two lines are responsible for the registration of the handles for the plugin activation and deactivation processes. The functions references here

are only short snippets to include the installation script and call the proper function from it.

The function registered for the `init` hook determines whether the plugin is visible in the current request, and adds the files from the UI directory to the output:

```
//Files used by both the admin section and the frontend
wp_register_style('kkm', plugins_url('kkm/ui/main.css'));
wp_enqueue_style('kkm');
wp_register_script('kkm', plugins_url('kkm/ui/main.js'),
    array('jquery'));
wp_enqueue_script('kkm');
```

In the code above a JavaScript registration is shown. As my script file depends on jQuery (which is included by default in every WordPress installation), I stated that this dependency must be included before my script.

The `init` function is also checks whether the current request is directed at a normal page of the plugin, and calls the bootstrap to handle it if this is the case:

```
if (($pos = strpos($_SERVER['REQUEST_URI'], '/kkm/')) !== false) {
    $plugin_visible = true;

    add_action('wp_loaded', array('kkm', 'bootstrap'), 99, 2);
}
```

I will explain the hooks related to rendering the admin page in section 4.4.

4.2.3. Installation script (install.php)

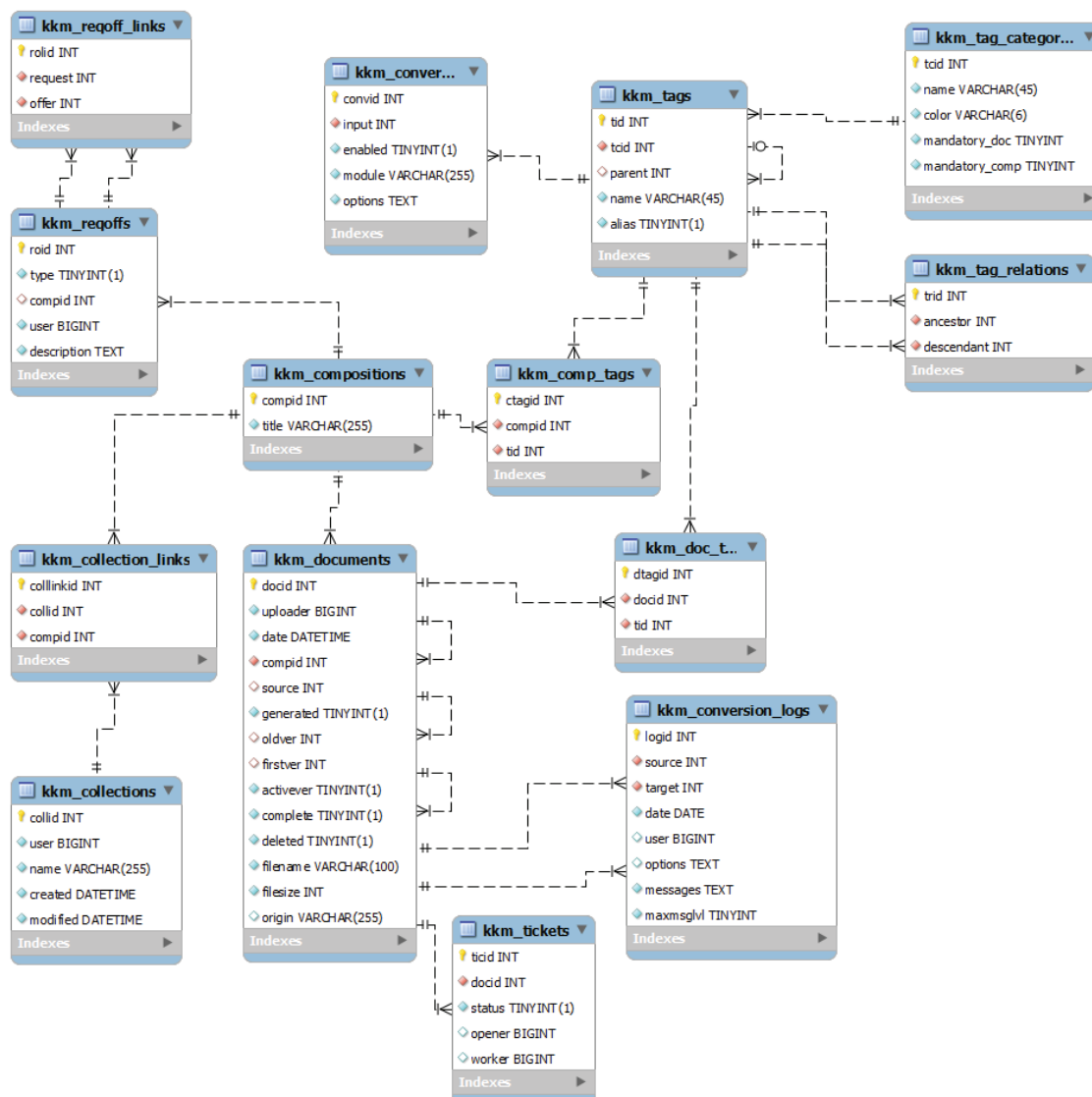


Fig. 15. Overview of the database structure

The installation script is responsible for creating the database and directory structure at installation, and removing the tables used by the plugin at the deactivation of the plugin.

I have designed the database structure using MySQL Workbench [22]. This tool makes it possible to define the tables on a graphical interface, and to export a MySQL script that could be used to create the structure on a server. However, in PHP there is no easy and universal method for running an SQL script, so the generated script must be manually processed.


```

/*
 * Remove comments from the script
 * and replace whitespaces with a single space character.
 * Semicolons inside column comments are escaped too.
 */
function kkm_install_sql_fix_callback($matches) {
    return preg_replace('/(?<!\|\|\|);/', '\\;', $matches[0]);
}
$install_sql = preg_replace_callback(
    '/comment\s*\'((?:[^\']|(?<=\\|\|\|)\')+)'\/im',
    'kkm_install_sql_fix_callback',
    $install_sql
);
$install_sql = preg_replace('%/\s*\.\s*/%', ' ', $install_sql);
$install_sql = preg_replace('/^--.*$/m', '', $install_sql);
$install_sql = preg_replace('/\s+/', ' ', $install_sql);

```

After this cleanup, the individual SQL commands can be executed using the `$wpdb->query($command)` function call.

As the plugin does not contain a separate SQL script for the removal of the database tables, the uninstall function processes the installation script, extracts the table and view names from it, and lists these names in drop statements:

```

//Tables
preg_match_all('/create +table +(?:if not exists +)?`(.*)`/i',
    $install_sql, $preg_res, PREG_SET_ORDER);
foreach ($preg_res as $res) {
    $tables[] = $res[1];
}
$tables = array_reverse($tables); //Tables should be dropped in
    reverse order due to foreign key constraints

if (!empty($tables)) {
    $wpdb->query('drop table if exists `'.implode('`,`',
        $tables).'`;');
}

```

After the creation of the database tables, the installation script adds the file formats used by the modules to the database, so the users can select the file format at upload from the supported types without further configuration.

4.2.4. Bootstrap (bootstrap.php)

The processing of user requests (HTTP GET and POST requests that arrive to the /kkm/ path or its children) start at the bootstrap. These requests generally can take two forms:

- GET requests: These requests do not involve the handling of user submitted content, only serving a page (or some special content described below) to the user.
- POST requests: Only these requests can change the content of the database and the file system based on user input. POST requests are results of forms being posted on the user interface, and every form has its own handler code in a separate file, identified by the action name stored as a hidden field on every form. No handler can produce any output, so at the end of the processing, the user gets redirected to a page where the result is visible.

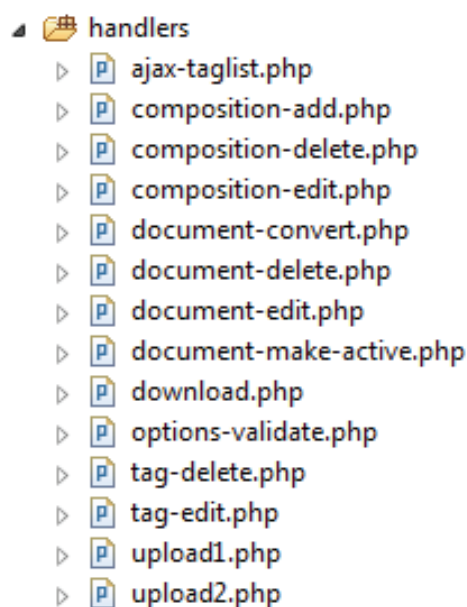


Fig. 16. Content of the handlers directory

Handlers are identified by their file names, so the form that triggers the deletion of a composition looks like:

```
echo('<form action="'.site_url('kkm/').'" method="post">');
echo('<input type="hidden" name="action" value="composition-delete"
    />');
echo('<input type="hidden" name="comp" value="'. $comp_id.'" />');
echo('<p>'.$message.'</p>');
echo('<input type="submit" name="delete_composition"
    value="'.__('Delete','kkm').'" />');
echo('<input type="submit" name="cancel"
    value="'.__('Cancel','kkm').'" />');
echo('</form>');
```

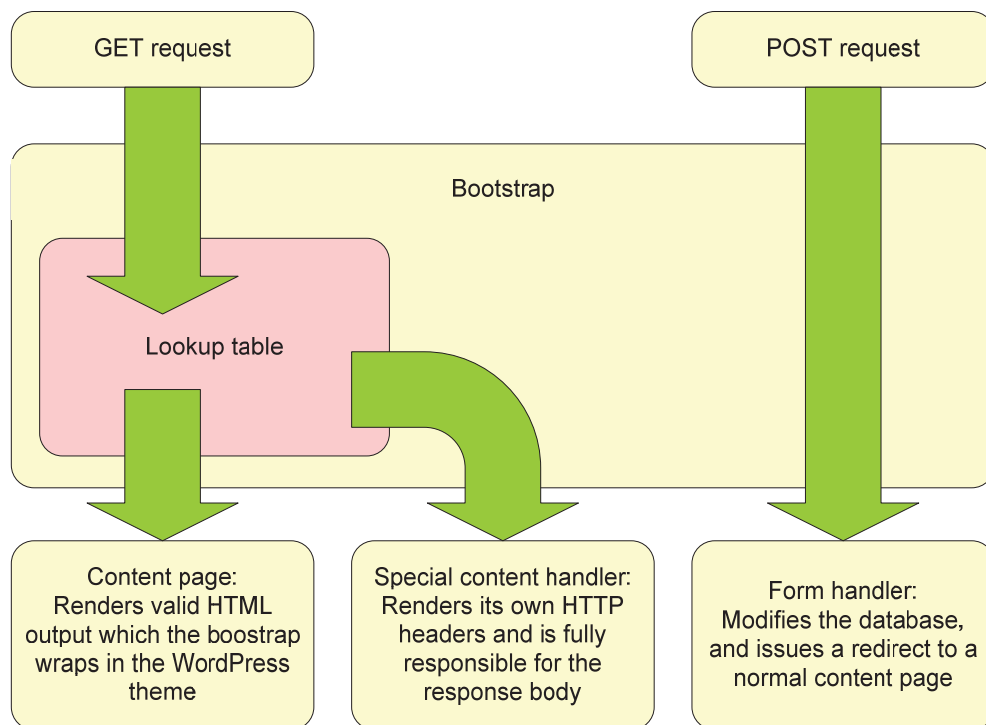


Fig. 17. Types of requests and their handling

Generally the GET requests are mapped to rendered content pages. For these, the bootstrap is responsible for rendering the WordPress frame around the real content. However, there are a few exceptions to this schema: in some special cases, the generic WordPress layout must be omitted, and only the raw content must be sent to the client. One of these cases is when the user

downloads a document: special HTTP headers must be sent to the browser, and the response body can only contain the file itself. These requests even though are of GET type need to be handled by special handlers located in the `handlers` directory.

The lookup table shown on Fig. 17 is an array that defines which file should be used for a specific request URI. This method makes possible to abstract the request URI from the physical file name, and this allows us to show more natural URLs in the browser and to group coherent files together: “delete-document” with its verb-noun structure is easier to understand, but from the developer’s point of view, a noun-verb structure in the filenames are better, as every page related to documents will be listed one below the other in the editor.

The lookup table lists its entries in a way that their keys are the URIs which will be queried, and their values are arrays containing the filename and a flag that tell apart special content handlers and normal pages. As every entry in this map is of the same structure, I show only a part of it here for illustration:

```
$global_sections = array(  
    //Special pages - no theme header and footer rendered  
    'taglist' =>          array('ajax-taglist', true),  
    'make-document-active'=>array('document-make-active', true),  
    'download' =>          array('download', true),  
  
    //Normal pages - wrapped inside the WP theme  
    'edit-document' =>     array('document-edit', false),  
    'delete-document' =>   array('document-delete', false),  
    'convert-document' =>  array('document-convert', false),  
    'search' =>            array('search', false),  
    'tags' =>              array('tag-list', false),  
    'edit-tag' =>          array('tag-edit', false),  
    'delete-tag' =>        array('tag-delete', false)  
)
```

Here I would like to point out that every page in the system has an URL that acts like a permalink – links that can be saved as a bookmark or sent to other users in email or by any other means. Furthermore, these URLs are *nice URLs* (also called *pretty URLs*): the structure of the file system on the server side is completely hidden from the eyes of the users, as only a virtual directory path consisting of sensible words and numerical identifiers is what points to a page in the system.

This is the reason the plugin needs a permalink structure other than the default to be enabled in WordPress, which utilizes the `mod_rewrite` module of Apache.

4.3. Module handling

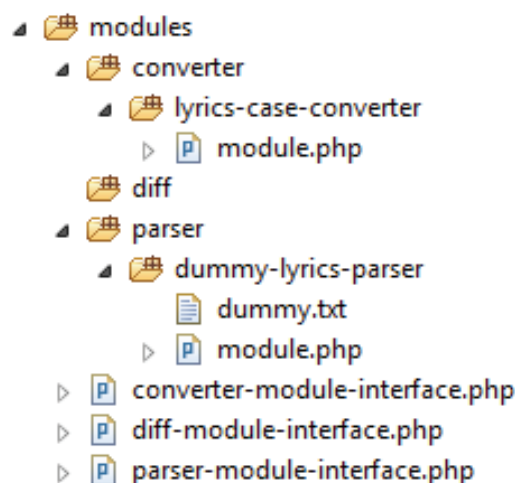


Fig. 18. The modules directory with two modules

I have described the basic design of the modularized system in section 3.8. Here I would like to explain how these modules get loaded and how they work.

Each of the three module types has an interface definition which for I included the documentation at the end of my thesis in Appendix: Module interface documentation (API). Every module has its own subdirectory inside the directory assigned to its type. The name of this subdirectory won't be used by the system, so there is no restriction on it as long as it doesn't block the PHP environment from loading the files stored inside.

For debugging purposes, I wrote the module loading in a way that it ignores any module that has a directory name starting with a period (.). This way a broken module can be easily and quickly disabled by renaming its directory.

Inside the own directory of the module, there must be a `module.php` file which contains a class that implements the interface of its type. The class name of the module must be unique and by convention should start with the `kkm_` prefix.

There are may be other files stored inside this directory, but they will be ignored by the system. Modules themselves can use these files of course, either by including them or by reading their content. Writing to these files is highly discouraged though, as it is not guaranteed that WordPress (and our plugin with its modules as a result) has write access to the `wp-content/plugins` directory tree.

I have created two modules, one converter and one parser, which I will use to demonstrate the workings of my plugin.

4.3.1. Module loading

Modules are only loaded when the system needs them. Code that loads the modules – along with functions to utilize them – is located in the `kkm_modules` function library which for I included the documentation in the appendix at the end of my thesis (section 8.2).

The loading of the modules starts by including the interface definitions. This is done by calling the `kkm_modules::load_interfaces()` method. Without these files, the PHP interpreter would throw errors when it would encounter a class that implements one of these interfaces.

When PHP includes a file, its content will be evaluated, and so the class defined in a module file will be available in the global namespace. This is the

principle behind the loading method, as the names of the declared classes can be queried using the `get_declared_classes()` function.

After the interfaces get included, the list of declared classes gets saved in a variable. This list contains every class currently available, and will be used as a reference later.

Modules get loaded separately, one type at a time. After the `module.php` files get included, the list of declared classes changes. By calculating the difference between the list stored previously, and the list returned at this stage, the list of module class names can be acquired.

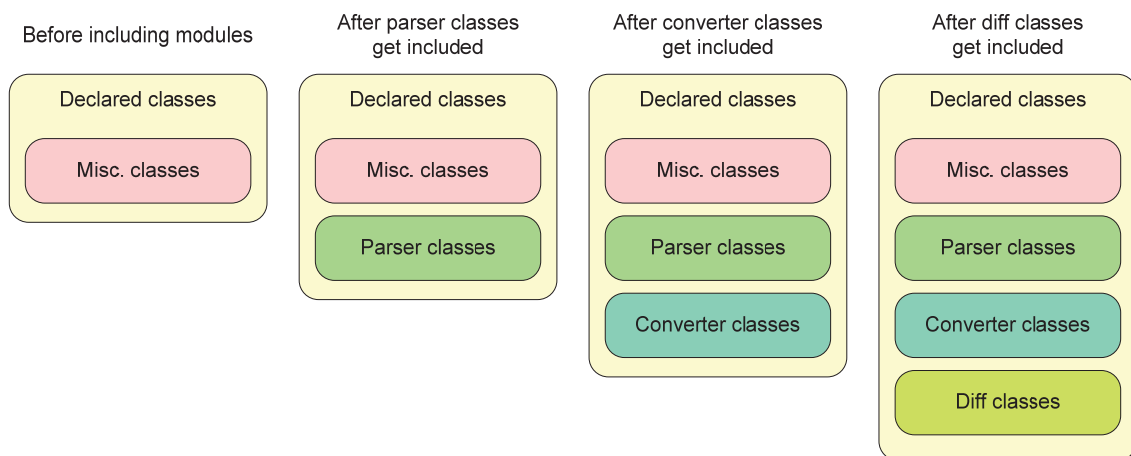


Fig. 19. List of declared classes at different stages of the module loading

For every module type there is a private static variable in the `kkm_modules` class that stores the lists of class names. The code responsible for loading the parser modules:

```
//Store already declared classes so we can filter them out later.
$unrelated = get_declared_classes();

//Load parsers
self::load_modules_from_dir(KKM_PLUGIN_DIR.'modules/parser/');
$loaded_classes = get_declared_classes();
self::$parser_modules = array_values(array_diff($loaded_classes,
    $unrelated));
```

4.3.2. Calling the modules

I wrote the `kkm_modules` function library in a way that it acts as a wrapper around the modules. For every function in the modules there is a method in this class that can handle the execution of the module functions based on the names of the modules. This way, other parts of the plugin doesn't need to be aware of the dynamically loaded modules, and modules can be accessed based on their functionality alone.

I will illustrate the usage of the modules in the following sections where I describe the functions of the plugin that actually uses them.

4.4. Admin interface

In a WordPress plugin there is one action hook that need to be used to make the plugin visible on the admin interface. This is the `admin_menu` action, which executes the following function in my plugin:

```
public static function register_admin_menus() {
    add_menu_page('kkm', 'kkm', 'manage_options',
        'kkm/admin/index.php');
    add_submenu_page('kkm/admin/index.php', __('General settings
    &lsaquo; kkm', 'kkm'), __('General settings', 'kkm'),
        'manage_options', 'kkm/admin/index.php');
    add_submenu_page('kkm/admin/index.php', __('Tag categories
    &lsaquo; kkm', 'kkm'), __('Tag categories', 'kkm'),
        'manage_options', 'kkm/admin/tagcats.php');
}
```

The above code snippet is responsible for constructing the links in the toolbar displayed on the left side of the admin screen.

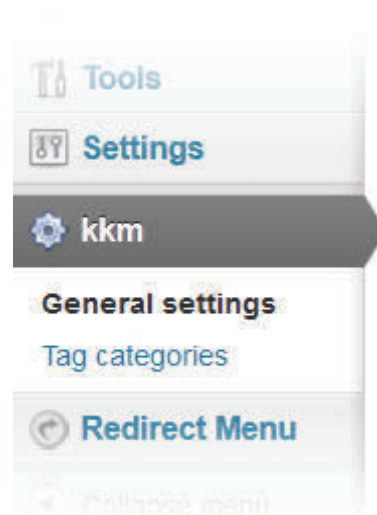


Fig. 20. The admin menu section that belongs to the plugin

The General settings page is currently empty, I have created it only for the reason that later some configuration options can be added when they deemed necessary.

4.4.1. Tag categories

Category name	Composition/Document	Tag count	Actions
Composer	Optional (visible)/-	0	
Format	-/Mandatory	4	
Instrument	Optional (visible)/Optional (visible)	0	Edit Delete
Performer	Optional (visible)/-	0	
Quality	-/Optional (hidden)	2	

[Delete](#)

Create new

Create new category

Category Name

For Compositions

For Documents

Color

[Create](#)

Fig. 21. Tag categories admin page

On the Tag categories page users with administration privileges can add, remove or edit tag categories. The system comes with some built in categories, which are created by the installation script. These cannot be modified or deleted, but tags can be added to them.

Categories are listed on the top of the page, showing the number of tags in each, along their mandatory status. For user created categories, there are Edit links in their rows. By clicking on one of these, the form at the bottom of the page gets populated with the properties of the category, and the Create button changes its functionality into saving the modified values instead of creating a new one.

User created categories can be deleted by selecting their Delete checkbox at the end of their rows, and clicking on the Delete button below the table.

4.5. User interface

Pages that belong to this plugin can be accessed by adding /kkm/ to the end of the site root URL (meaning that if the WordPress installation is accessible at <http://localhost/> then the main page of the plugins is located at <http://localhost/kkm/>).

The plugin pages can only be accessed by users who have at least Subscriber role. To modify the content stored in the system, users need to have at least Contributor role assigned to them. These roles are native to WordPress and can be set on the admin interface in the Users section.

4.5.1. List of compositions

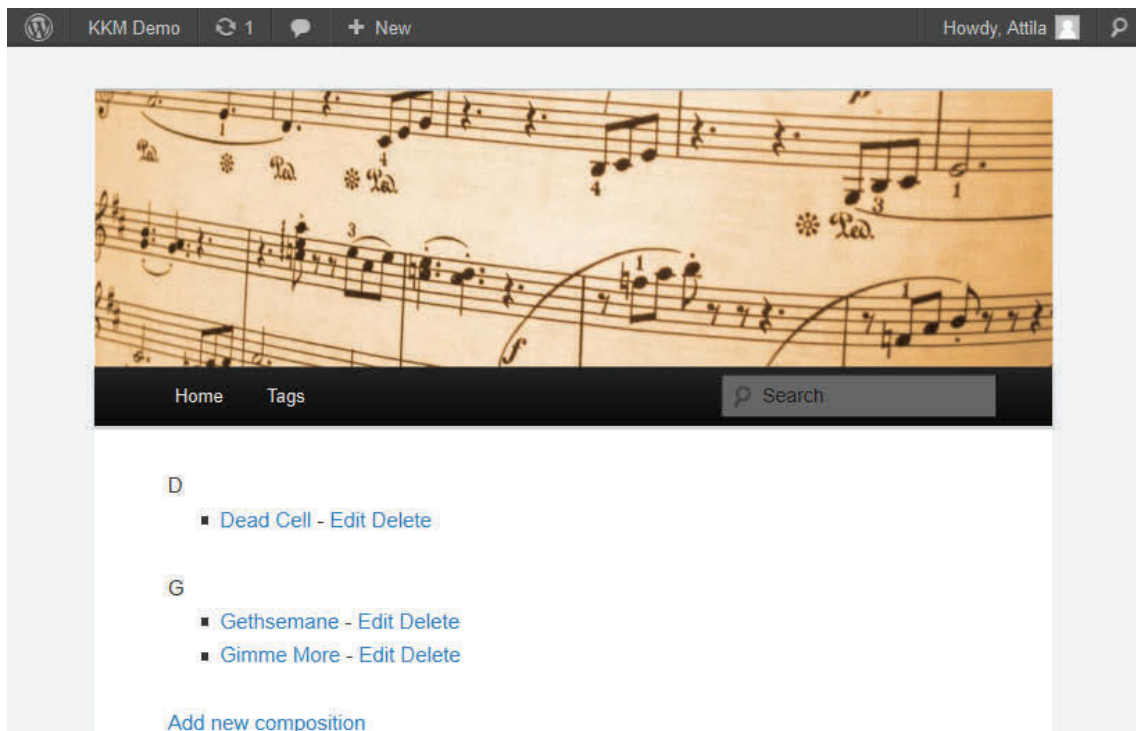


Fig. 22. Home page listing the compositions

Compositions are listed on the main page of the plugin located at the /kkm/ URL. The list is rendered in alphabetical order, and has a heading for each letter titles have as first character. These headings act as anchors, so they can be directly referenced or bookmarked. This way when the list is longer than what could be rendered on the user's screen at once, /kkm/#U URL can be used to jump directly to the titles that start with the letter U.

Next to the title of the compositions are links to edit or delete them. These functions are also accessible from the compositions' detail page.

4.5.2. Adding or editing a composition

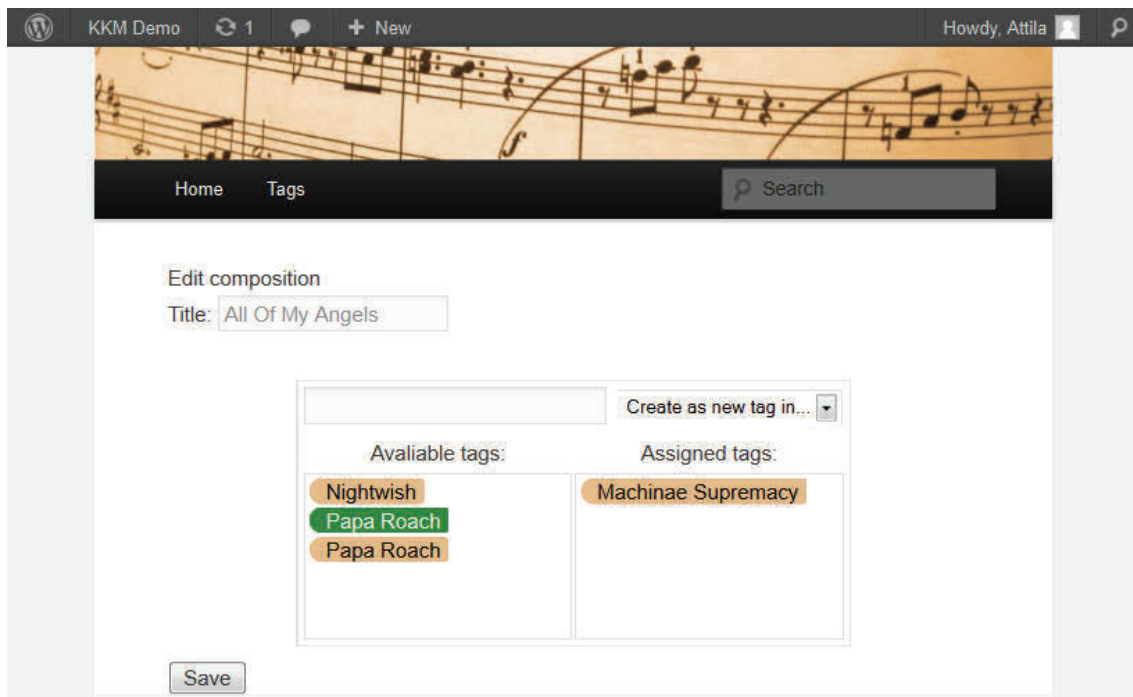


Fig. 23. Page for adding or editing compositions

The same page is used to add a new composition or edit an existing one. Only the title of the composition is requested separately, every other parameter may be assigned to the composition by the means of tagging it.

4.5.3. The tagging interface

Most of the information about documents and compositions are stored in tags. A common user interface is used for both documents and compositions to assign the tags to them. This interface extensively uses JavaScript, so a modern browser must be used with JavaScript enabled.

The two lists showing the available tags and the assigned tags could be considered as the main parts of the interface. Double-clicking a tag in a list moves it to the other side. When the page loads, the Assigned tags list shows the tags that are currently saved for the given composition or document. When the user presses the Save button, the content of the list will replace the content of the database, so tags can be added or removed freely before this confirmation of the changes.

Any character typed in the search field will trigger a function that filters the contents of the Available tags list. Filtering is done by taking each whitespace separated character chain and using these as individual search terms. If every term is part of a word in the name of the tag, then it will be included in the results and will be shown on the list. This means that the user can search for a tag that has multiple words in its name by typing in the first letter of each word separated by spaces.

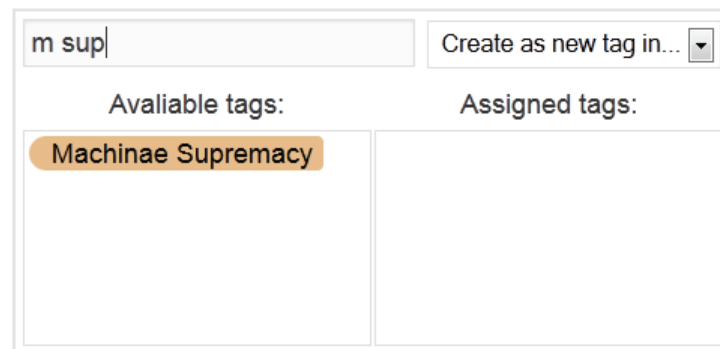


Fig. 24. Filtering the list of available tags

The JavaScript code that does the filtering and animates the addition and removal of the tags in the list is shown below. This `filter()` function is bound to the `keyup` event of the search field when the tagging interface initializes.

```
this.last_filter = '';
this.filter = function() {
    filter = search_field.val();
    if (last_filter != filter) {
        last_filter = filter;
        filter = filter.replace(/([\\\\/\|\.\*\+\?\(\)\[\]\{\}\])/g,
            '\\$1');
        words = filter.split(' ');
        r = [];
        $.each(words, function(key, val) {
            r.push(new RegExp(val, 'i'));
        });

        search_list.children().each(function(i, li){
            text = li.firstChild.innerHTML;
```

```

    ok = true;
    $.each(r, function(rk,rv){
        if (!rv.test(text)) {
            ok = false;
            return false;
        }
    })
    li = $(li);
    vis = li.is(':visible');
    if (ok && !vis) {
        li.slideDown(100);
    } else if (!ok && vis) {
        li.slideUp(100);
    }
})
}
}

```

New tags can be created by typing the desired name into the search field then selecting the category of the new tag from the list on the right side of the search field. The created tag automatically gets added to the Assigned tags list.

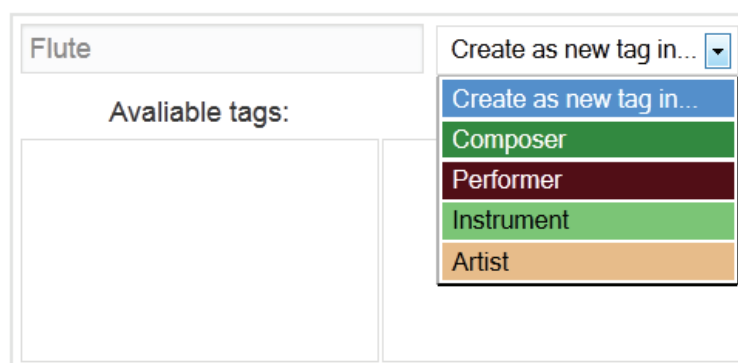


Fig. 25. Creating a new tag

When there are categories marked as mandatory, the tagging interface shows the list of these at the bottom. More specifically, the list of mandatory categories is shown from which the user hasn't assigned at least one tag to the current document or composition. When the user adds or removes a tag this list

gets updated and may get completely hidden if every mandatory requirement is fulfilled.

Fig. 26. The tagging interface listing mandatory categories

In the background JavaScript code modifies the content of a hidden form field, and the content of this field gets processed by the `save_tags_for_comp()` or `save_tags_for_doc()` method of the `kkm_tags` function library depending on the content type. The method for saving tags for a document is shown below.

```
public static function save_tags_for_doc($doc_id, $taglist) {
    global $wpdb;

    //Remove every tag except the format
    self::remove_tags_from_doc($doc_id, array(self::TYPE_FORMAT));

    /*
     * Get tags from the taglist
     */
    $tags = json_decode($taglist);

    //Filter tags into existing and new
    $new_tags = array();
    $tag_ids = array();
    foreach ($tags as $tag) {
```

```
if (is_numeric($tag)) {
    $tag_ids[] = (int)$tag;
} else {
    $tag = @json_decode($tag);
    if (is_array($tag) && is_numeric($tag[0]) &&
is_string($tag[1])) { //Check variable types
        $tag_name = trim($tag[1]); //Trim whitespace
        if ($tag[0] > 0 && strlen($tag_name) > 0) { //Do
not allow invalid values
            $new_tags[] = array($tag[0], $tag_name);
        }
    }
}

//Create new tags
if (!empty($new_tags)) {
    $new_ids = self::create_tags($new_tags);
    $tag_ids = array_merge($tag_ids, $new_ids);
}

if (!empty($tag_ids)) {
    //Build SQL query
    $sql_pairs = array(); //(docid, tid) pairs
    foreach ($tag_ids as $tid) {
        $sql_pairs[] = $wpdb->prepare(
            '(%d,%d)',
            $doc_id,
            $tid
        );
    }

    $wpdb->query(
        'insert into `kkm_doc_tags` (`docid`, `tid`) values
        '.implode(', ', $sql_pairs).' on duplicate key update
        `dtagid`=`dtagid`; '
    );
}
}
```


As the tagging interface itself is defined by a not-so-trivial and not-so-short HTML fragment, I have written a function to make it easier to include it on a page:

```
public static function render_tagging_box($field_name, $type = '',
    $assigned_tags = null, $suggested_tags = null,
    $excluded_categories = null) {
    $ret = '<table class="kkm_tagging"
    id="kkm_tagging_table_'. $field_name. '">';

    //Filter row
    $ret .= '<tr><td colspan="2">';
    $ret .= '<input type="text" name="kkm_tagging_search"
    id="kkm_tagging_search_'. $field_name. '" />';
    $ret .= '<select name="kkm_tagging_newcat"
    id="kkm_tagging_newcat_'. $field_name. '"><option
    value="0">'. __('Create as new tag
    in...', 'kkm'). '</option></select>';
    $ret .= '</td></tr>';

    //Taglist caption row
    $ret .= '<tr><td>';
    $ret .= __('Avaliable tags:', 'kkm');
    $ret .= '</td><td>';
    $ret .= __('Assigned tags:', 'kkm');
    $ret .= '</td></tr>';

    //Taglist row
    $ret .= '<tr><td>';
    $ret .= '<ul class="kkm_taglist"
    id="kkm_tagging_pool_'. $field_name. '"></ul>';
    $ret .= '</td><td>';
    $ret .= '<ul class="kkm_taglist"
    id="kkm_tagging_assigned_'. $field_name. '"></ul>';
    $ret .= '</td></tr>';

    //Mandatory row
```

```
$ret .= '<tr><td colspan="2">'.__('Categories you need to
choose from:', 'kkm'). '<div
id="kkm_tagging_needed_'.$field_name.'"></div></td></tr>';
$ret .= '</table>';
$ret .= '<input type="hidden" name="'.$field_name.'"
id="'.$field_name.'" />';
$params = array(
    'name'          => $field_name,
    'type'          => $type,
    'assigned'      => is_array($assigned_tags) ?
$assigned_tags : array(),
    'suggested'     => is_array($suggested_tags) ?
$suggested_tags : array(),
    'excluded'      => is_array($excluded_categories) ?
$excluded_categories : array()
);
$ret .= '<script
type="text/javascript">kkm_tagging_box('.json_encode($params).'
);</script>';
return $ret;
}
```

This function builds the HTML table, and passes the parameters to the JavaScript function that initializes the interface. These parameters include the list of tags currently assigned to the content, tag categories that won't be displayed on the interface and a list of suggested tags. I will explain where these suggested tags come from in section 4.5.6.

It is also worth noting that the tagging interface loads the list of tags in JSON format on the client side from the `/kkm/taglist` URL. The generation of this tag list is done by a special handler located in the `handlers/ajax-taglist.php` file.

4.5.4. Composition details page

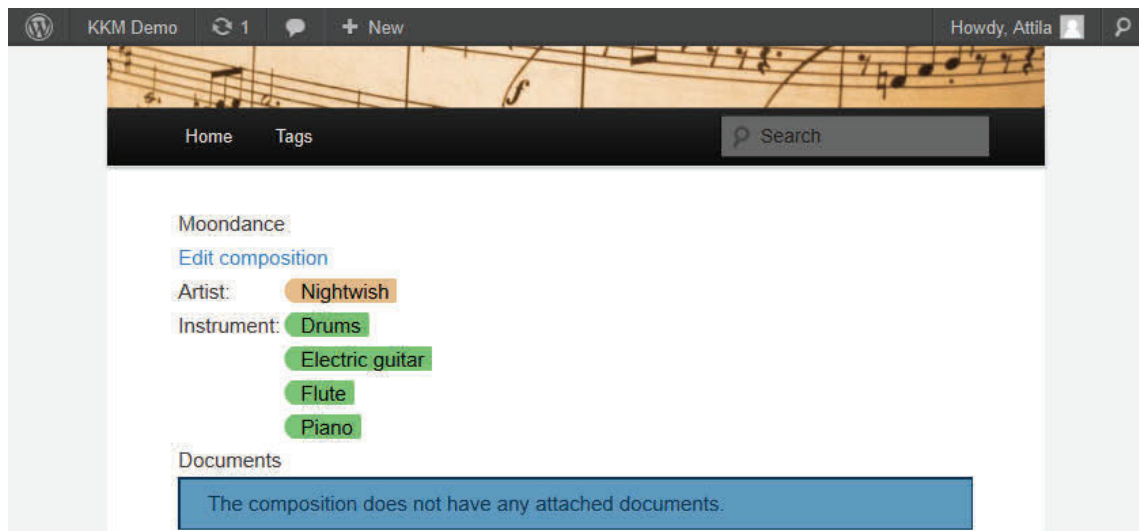


Fig. 27. Composition without documents

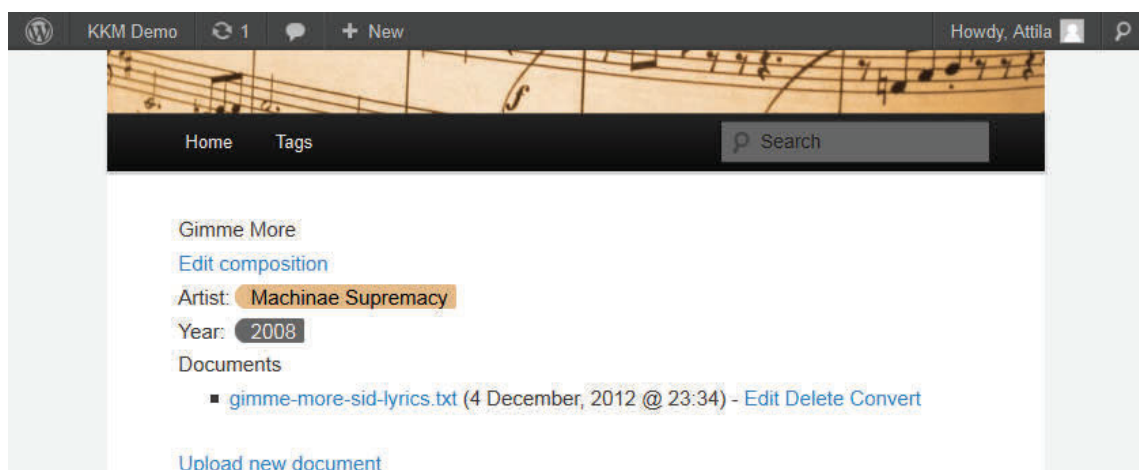


Fig. 28. Composition with a single document

The composition page lists every tag associated with the composition in a table, so the categories of the tags can be comprehended without knowing the colors associated with each category.

When the users click on the Edit composition page, they will be directed to the same page I have described in section 4.5.2. The link to delete the composition will be shown on that page.

Documents associated with the composition (if there is any) will be listed at the bottom of the page. Clicking on the filename will bring up the document details page, and next to the file date there are action shortcuts similar to those displayed on the composition listing page. I will describe these actions in their own section.

4.5.5. Deleting a composition

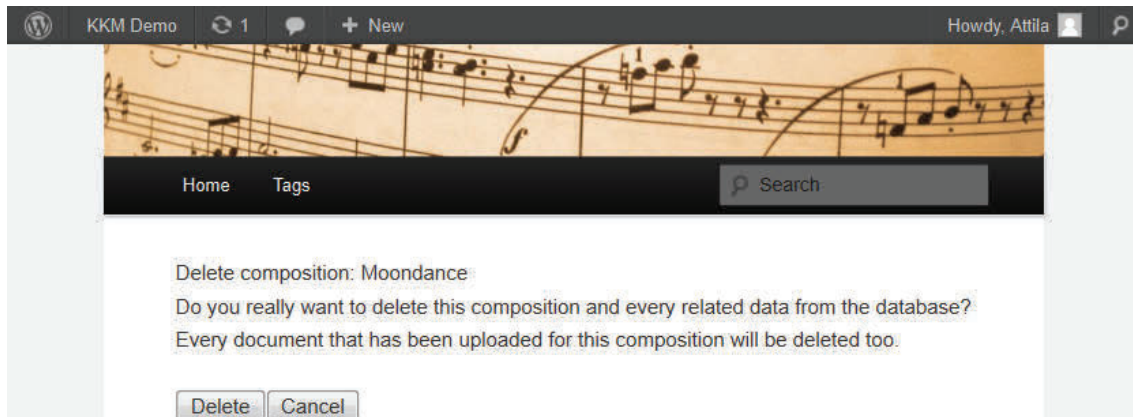


Fig. 29. Page asking the user to confirm the deletion

When the users click on the Delete link of a document or composition, they are asked to confirm this action on a separate page. This page explains the results of their action, notably that when a composition gets deleted, every related document will get deleted too.

4.5.6. Uploading a document

The main purpose of this system is to store documents, make them available, and assist the work of its users with format conversions.

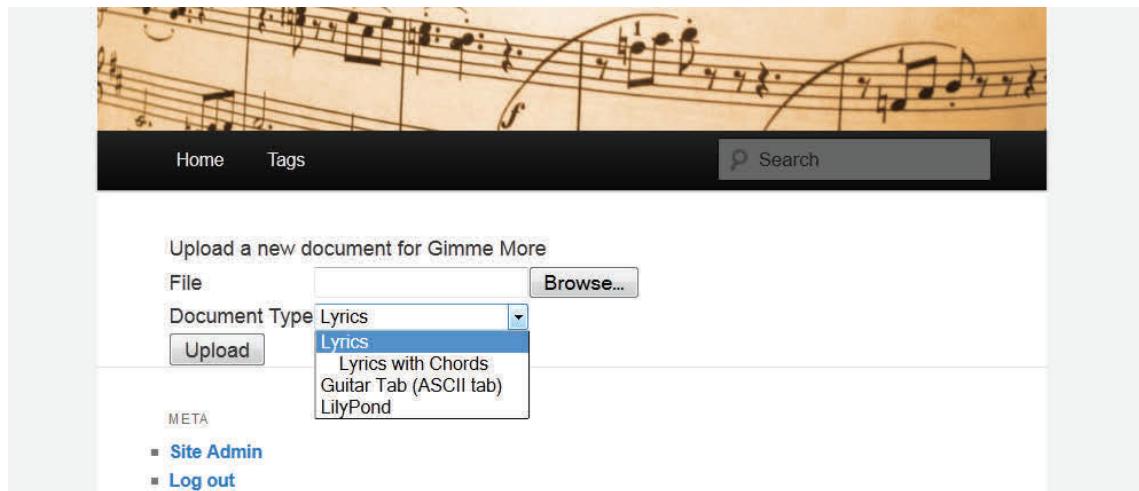


Fig. 30. Uploading a document – stage 1

Documents can be uploaded by clicking the Upload new document link on the details page of the composition (shown on Fig. 28). The uploading is a two-step process from the users' point of view. On the first page, they must select a file and choose its format from the Document Type dropdown menu.

On the second page, users can add additional information about the file to the database in the form of tags. There is also an Origins field at the bottom of the page, where any text can be inserted. Originally I created this field with the intention to allow users to save the URLs of the web page from which the document was saved, as for other sources (such as books, collections) tags may be more appropriate.

Validation results:

These lyrics do not contain any 'foo'. Should be revised.

Preview:

Lorem ipsum

Tags:

Available tags:

- Drums
- Electric guitar
- Flute
- High quality
- Low quality
- Machinae Supremacy
- Nightwish
- Papa Roach
- Piano

Assigned tags:

Origins :

Save

Fig. 31. Uploading a document – stage 2

On the screen shown in the above figure, the results of the *Dummy lyrics parser* module are visible at the top of the page. When users upload a document, the system queries the available parser modules, and tries to validate the file content with each that supports the type of the current document. The messages shown to the users are the aggregated results of any parser that has been run on the file and are colored based the type of the message: notices have a blue background, warnings are colored orange, and errors are rendered in red. If any parser reports a validation error, the upload process gets terminated, and the system deletes the uploaded file.

```

/**
 * Calls the validators and prints the result
 * @return boolean True if there was no error.
 */
function kkm_ul2_validate($parsers, $doc_path) {
    $results = kkm_modules::validate_document($doc_path, $parsers);
    if ($results == null) {
        return true; //Do not terminate upload process because of
        missing validators.
    } else {
        $has_error = false;
        //Print the results
        echo('<h3>'.__( 'Validation results:', 'kkm').'</h3>');
        echo('<div class="kkm_validation_area">');
        foreach ($results as $result) {
            echo('<div
            class="kkm_'. $result[0]. '_message">'. $result[1]. '</div>');
            if ($result[0] == 'error') {
                $has_error = true;
            }
        }
        echo('</div>');

        return !$has_error; //If there was no error, only then we want
        to continue.
    }
}

```

The above code is responsible for rendering the messages returned by the validators using the proper format for each message type. As I haven't showed yet how the functions of the modules get called by the system, I include the code from the `kkm_modules` function library that calls the parser modules to do the validation below.

Developing Music Score Management System for Communities

```
public static function validate_document($path, $parsers) {
    $results = array(); //Aggregated results of the validators.
    $validated = false; //Was there at least one parser with an
    impleteneted validate() function?

    //Call the validator method of each parser module
    foreach ($parsers as $parser) {
        $res = call_user_func(array($parser, 'validate'), $path);

        //For unimplemented validators $res will be null
        if (is_array($res)) {
            $validated = true;

            //Loop through messages
            $has_error = false;
            foreach ($res as $msg) {
                $results[] = $msg;
                if ($msg[0] == 'error') {
                    $has_error = true;
                }
            }

            //If a validator reports an error, stop calling the
            others.
            if ($has_error) {
                break;
            }
        }
    }

    if ($validated) {
        return $results;
    } else {
        return null;
    }
}
```


To create a preview, the system iterates over the parser modules until it can find one that supports preview generation for the file type. Modules must return HTML code fragment as preview, so they are free to choose any method to display the contents of the file: plain text for lyrics, syntax highlighted code for LilyPond documents, or even a rendered image can be shown to the users.

An other important feature of the parser modules is to extract metadata from the documents and write back the optionally modified metadata. When a user presses the Save button at the bottom of the document upload page, tags are saved into the database, and the same values get written back to the file using the parser modules. The overview of this process is shown on the figure below, while the functions that work in the background are documented in the Appendix (sections 8.2 and 9.3).

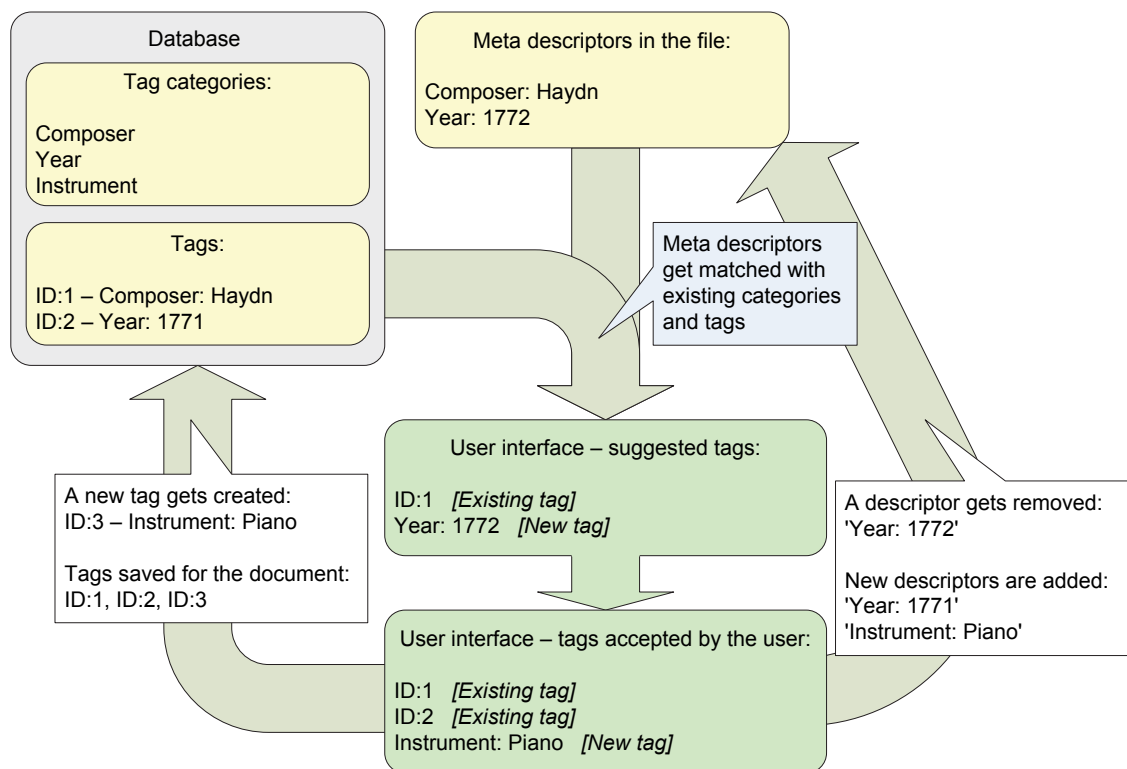


Fig. 32. Overview of the meta data processing upon document upload

4.5.7. Document details page

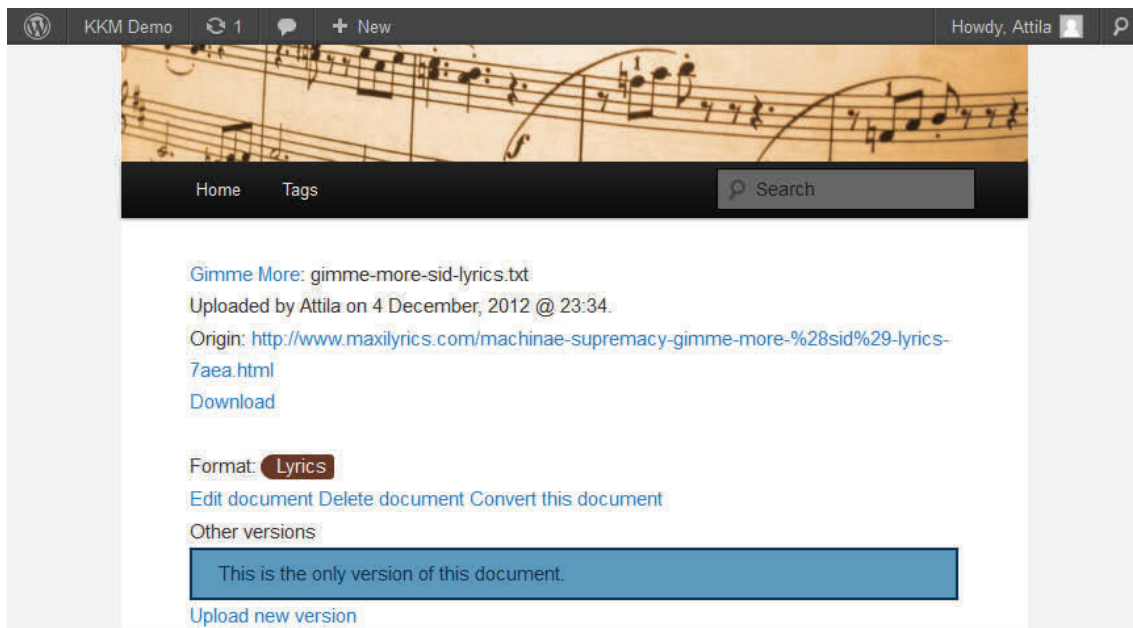


Fig. 33. Document details page

The document details page has a similar structure as the composition details page. For the sake of quick navigation, there is a link to the composition page on the left side of the filename. Just below this link are the date of the upload and the display name of the user who added the document.

It is worth noting that URLs in the Origin property gets rendered as clickable links. This feature is achieved by a single line of PHP code:

```
$origin = preg_replace('/\b(?:http|https|ftp|mailto):[^\s]+/', '<a href="$0">$0</a>', $origin);
```

4.5.8. Version control in work

Uploading a newer version of a document is mostly the same process as uploading an entirely new document. There are only two differences between them: the upload process can be started by clicking on the *Upload new version* link on the document page, and the new version must be of the same type as the original document. For this reason, instead of the dropdown menu for

selecting one, the format tag of the original document is rendered on the upload page as shown on the image below.

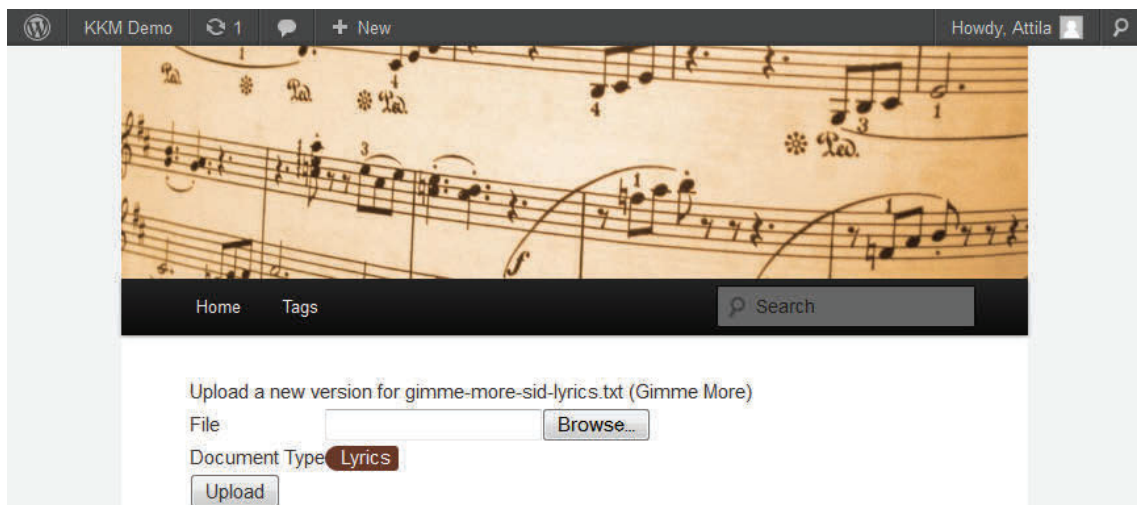


Fig. 34. Versions of the same documents must be of the same type

When the document has multiple versions, others than the currently viewed version get listed at the bottom of the page. For every item in this list, the upload date is shown next to the filename. It is worth noting that every version of a document can have a different filename, but of course, using the same name for every version is also possible. Files are stored on the server in a way that allows multiple files to have the same name without any conflict.

There is a notice visible above the list of file versions that tells whether this document is the current version, or an older – or more appropriately an inactive – one. Inactive documents can be marked as the current version by clicking on the *Make this the active version* link just below this notice.

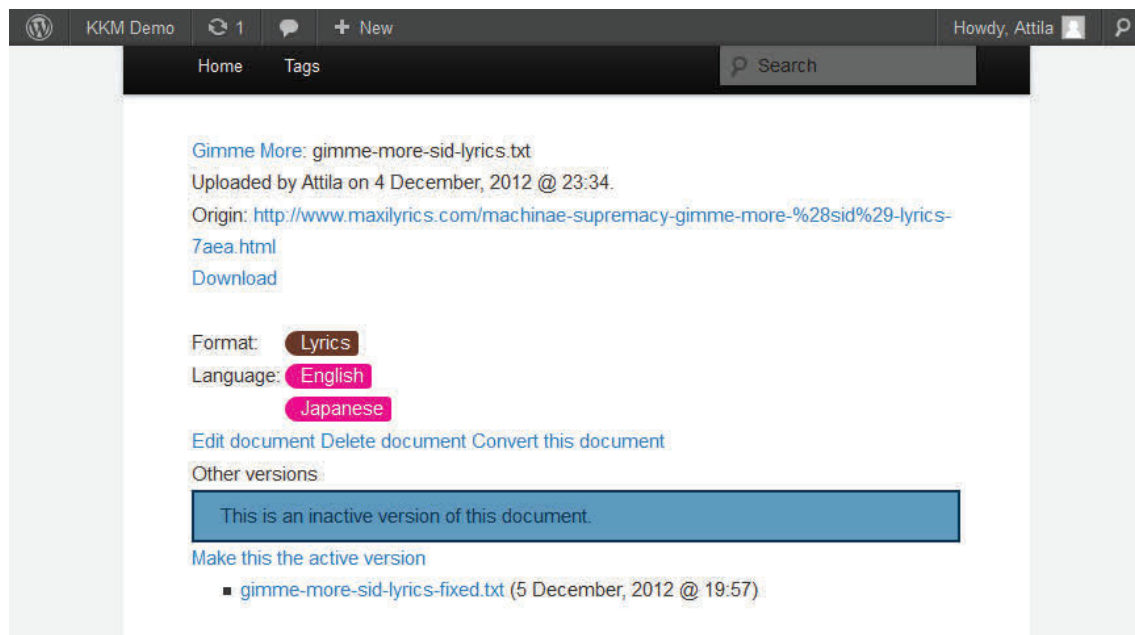


Fig. 35. The document page showing an inactive document version

4.5.9. Converting a document

By clicking either on the *Convert* link next to the filename of the document on the composition page, or on the *Convert this document* link on the document page begins a conversion process.

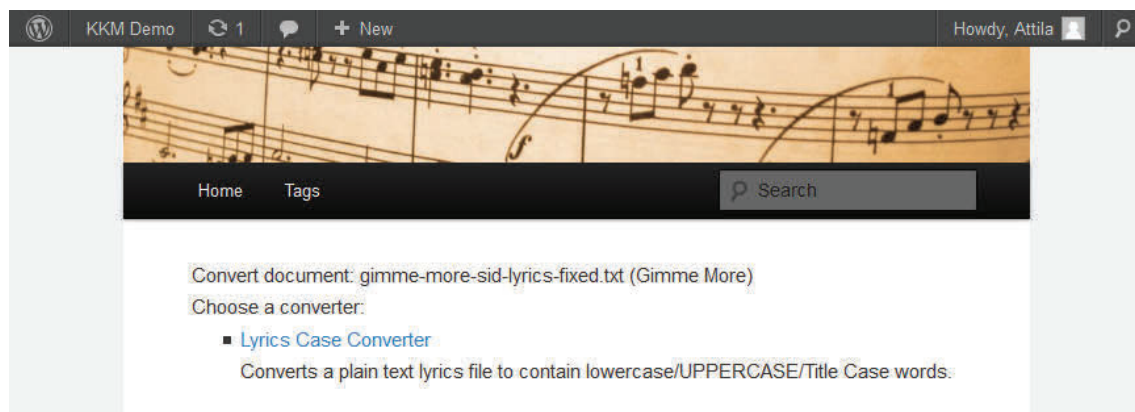


Fig. 36. Start page of a conversion

On the start page of the conversion process, converters usable with the source document format are listed to the user. The list consists of the names and short descriptions of the converters, which are loaded from the modules themselves using the converter module interface documented in the appendix (section 9.1).

Clicking on the name of a converter takes the users to the next page where they must specify a name for the converted document, and optionally, set the parameters for the conversion if they are required by the chosen converter. It is worth noting that the links belonging to the converters contain the class names of the converter modules. Using these instead of a generated index (e.g. using its position in the alphabetical list of modules) for identification makes it possible to save these links and go back to the page at a later time even if modules get added and removed after the users save the URL (of course, deleting the converter module the link belongs to breaks even this setup).

The screenshot shows a web interface for a document converter. At the top, there's a header with 'KKM Demo', a refresh icon, a comment icon, a '+ New' button, and a user profile 'Howdy, Attila'. Below the header is a navigation bar with 'Home' and 'Tags' links, and a search bar. The main content area displays the conversion options for a document named 'gimme-more-sid-lyrics-fixed.txt (Gimme More)'. The form contains four rows of input fields: 'Save file as:' with the value 'test file', 'Custom title to be inserted into the document:' with the value 'test title', 'Select the position of the title:' with a dropdown menu set to 'Below content', and 'Select target case:' with a dropdown menu set to 'Title Case'. A 'Start conversion' button is located at the bottom of the form.

Fig. 37. Conversion options page rendered for the demo converter

The form on the conversion options page is rendered dynamically for the converter module the users currently use. As I have documented in the appendix, converter modules have a function that the system can use to ask it about a list of options they need. Text fields and dropdown menus are then generated based on this list using functions in the `kkm_options` function library (its documentation is available in the appendix, section 8.3).

When the user clicks on the *Start conversion* button, the specified options get validated before submitting the form. This is done by an AJAX query that submits the options to the options validator handler (located in the `/handlers/options-validate.php` file), which in turn returns the messages

Developing Music Score Management System for Communities

generated by the module. If there were any errors, they are displayed under the form.

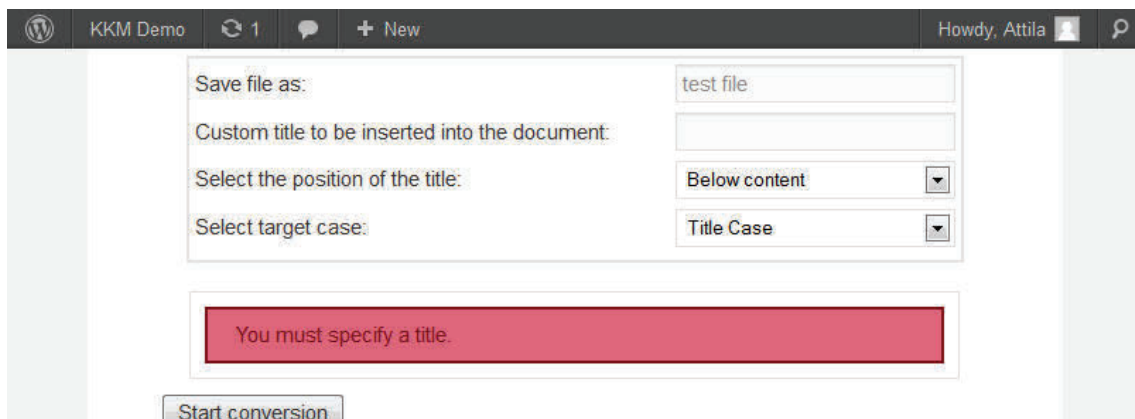
The screenshot shows a web interface for a music score management system. At the top, there's a header with a WordPress logo, 'KKM Demo', a refresh button, a comment icon, a '+ New' button, and a user profile 'Howdy, Attila'. The main content area contains a form with four fields: 'Save file as:' with a text input containing 'test file', 'Custom title to be inserted into the document:' with an empty text input, 'Select the position of the title:' with a dropdown menu set to 'Below content', and 'Select target case:' with a dropdown menu set to 'Title Case'. Below these fields is a red error message box that says 'You must specify a title.' At the bottom of the form is a 'Start conversion' button.

Fig. 38. Conversion options page showing an error

Only if the AJAX call returns without any error gets the form submitted. Of course, this counts only as a client side validation, which doesn't even run if the user disables JavaScript, so the validation is run again immediately before starting the actual conversion on the server side.

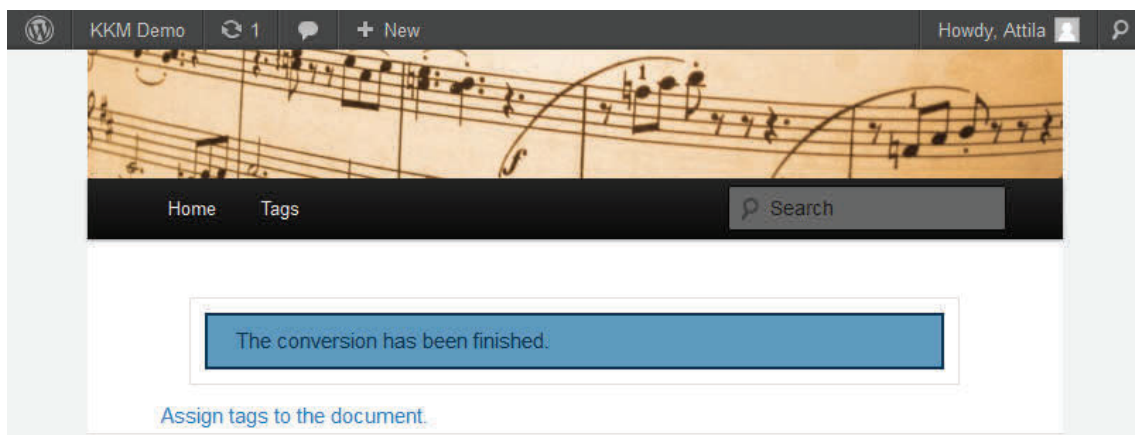
The screenshot shows the result page of a conversion. The header is the same as in Fig. 38. Below the header is a banner image of a musical score. Under the banner is a navigation bar with 'Home' and 'Tags' links, and a search bar. The main content area features a blue success message box that says 'The conversion has been finished.' Below this message is a link that says 'Assign tags to the document.'

Fig. 39. Result page of a conversion

The converters can log messages during the conversion process, which will be shown to the users at the end. These messages also get saved to the database, and using the URL of the result page they can be viewed anytime.

After the conversion, tags can be added to the document in the same way as to uploaded documents.

4.5.10. Searching the database

I have implemented the search function in a way that any search field native to WordPress can be used to start a search query. This includes the search field sidebar widget and custom search fields defined by the active theme.

The search query string entered by the users gets segmented into individual search terms. From these terms, common words such as 'a', 'the' or 'of' gets removed, and the remaining terms will be used to filter the content. Only items that match every term will be listed on the result page.

A composition is considered as a valid result if the words in its name and the names of its associated tags together contains the search terms – every word can be matched partially, so there is no need to specify whole words in the query.

Documents are returned only if their composition does not fulfill every criterion, but if with their filenames and associated tags added to the list of words used to match the compositions they do.

Type	Composition name	Associated tags	Document filename	Associated tags	Shown in result?
Composition	Alpha	Beta			No
Document			Gamma	Delta	Yes
Document			Epsilon		No
Composition	Alpha Beta	Gamma			Yes
Document			Delta		No

Fig. 40. Example data to illustrate the search function

Let us say that for example the users search for the string 'An Alpha of Beta, Gamma'. In the first step, the string gets segmented into single words: 'An', 'Alpha', 'of', 'Beta', 'Gamma'. Because 'an' and 'of' are common words, they get removed from this list, leaving only 'Alpha', 'Beta' and 'Gamma'.

The content stored in the system for this example is shown on Fig. 40. The first composition only has two of the required search terms, so it won't be

Developing Music Score Management System for Communities

included in the results. However, it has a document with a filename that matches the last term, so the document will be shown on the result page. The other document of this composition only has an irrelevant word in its name, so it is ignored. The second composition has two of the terms in its name, and the third as an associated tag, so it is considered a match. Its document, while technically has every required term associated with it, won't be listed because the composition is already included in the results.

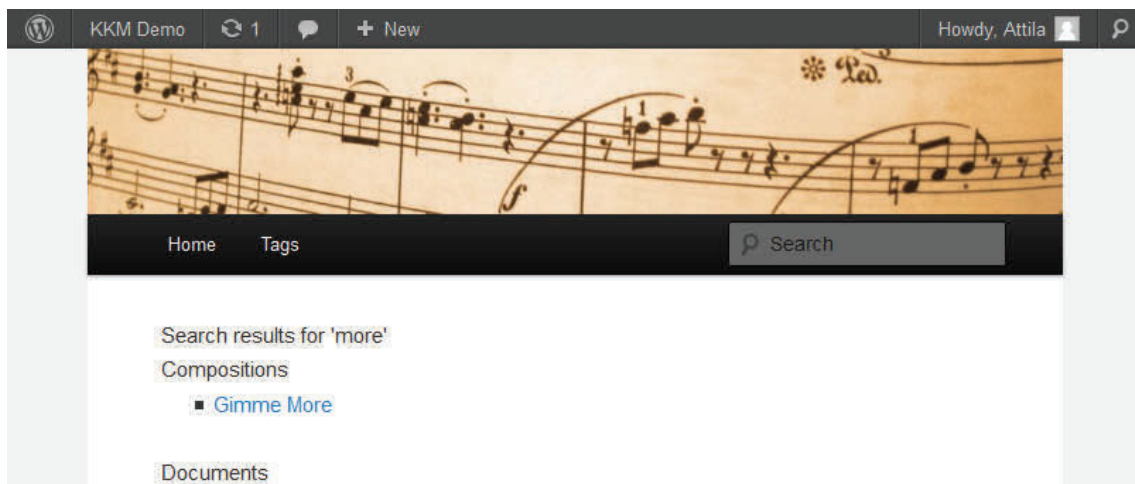


Fig. 41. The search result page showing a matched composition

4.5.11. Tag list

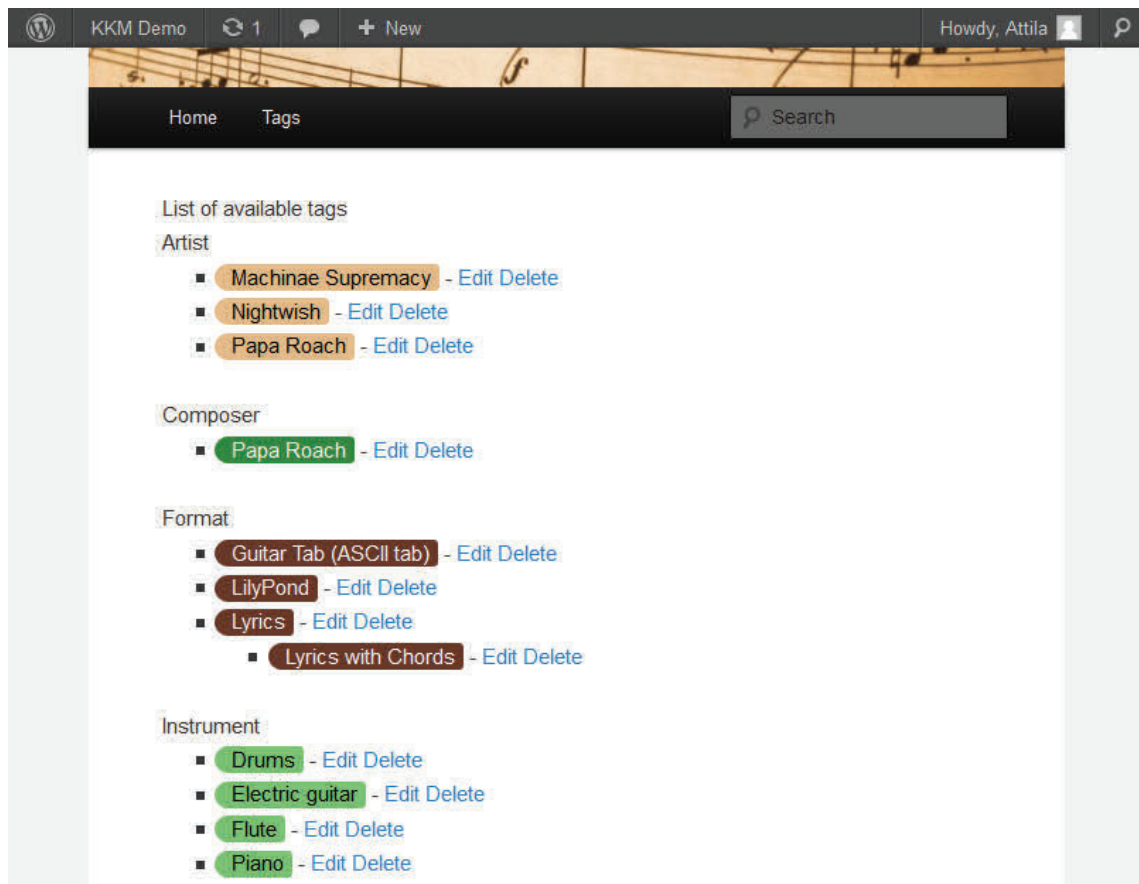


Fig. 42. Page showing the available tags

There is a page mapped to the `/kkm/tags` URL that lists every tag from the database. Tags are grouped by their category and rendered in alphabetical order. Tags that have a parent are rendered below their parent in a sub-list.

Clicking on a tag brings up the search result page that lists both compositions and documents having the selected tag.

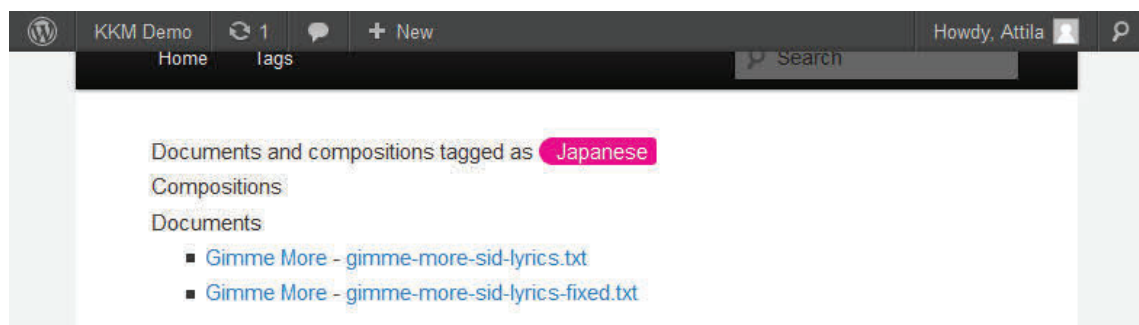


Fig. 43. Listing content that have a specific tag associated with them

By clicking the *Edit* link, tags can be renamed and their parent tag can be specified. For parent tag, any tag from the same category can be selected as long as the current tag is not an ancestor of it.

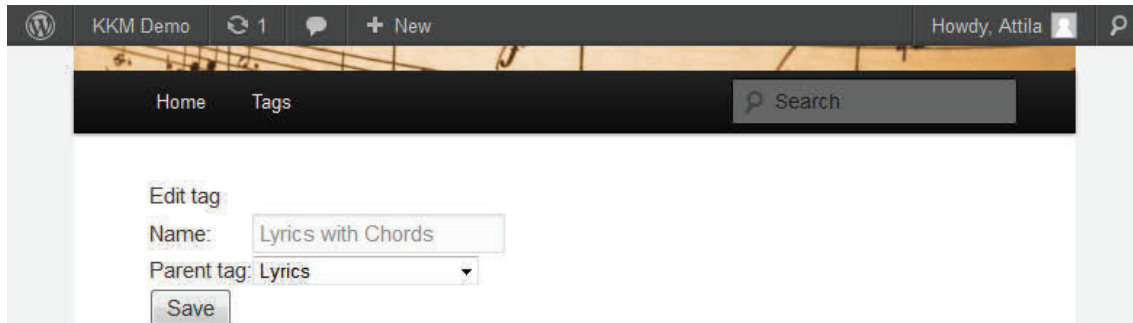


Fig. 44. Editing a tag

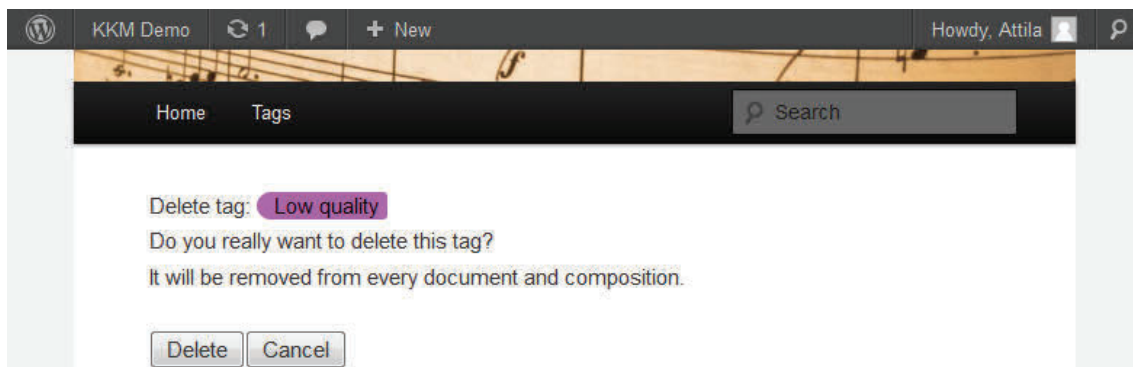


Fig. 45. Deleting a tag

To delete a tag the users must confirm its removal in the same way as in the case of documents and compositions.

4.6. Localization

WordPress supports internationalization and localization using GNU Gettext [23]. Its framework includes functions to load localized string packages and to manage the rendering of texts that may or may not have their localized versions stored in these packages.

Plugins can take advantage of this support, as they can use the same functions the WordPress core uses. In my plugin I have written every string that gets displayed to the users in a way that they can be translated using

existing and supported tools. To achieve this, I have wrapped every string in the `__()` or `_e()` functions as seen in the code below.

```
$date = mysql2date('j F, Y @ G:i', $document['date']);  
/* translators: First string is the display name of the uploader,  
the second is the upload date. */  
$subtitle = sprintf(__('Uploaded by %1$s on %2$s.', 'kkm'),  
    $document['username'], $date);  
echo('<div class="kkm_document_subtitle">'.$subtitle.'</div>');
```

The first line of this code converts the date stored in the database to a localized format – the `mysql2date()` function tries to substitute the given format string with a localized one defined in the translation package of the WordPress core, and uses this value along with the display name of the user in the subtitle on the document page. I have also made sure to leave comments explaining the content of the used variables for the translators in a strict format parsable by the tools used to translate this plugin, as they do not necessarily read the code while they translate the messages.

As the process of the translation is independent from my plugin, and as guides on the usage of the required tools are already available (and quite lengthy), I omit this part from my thesis. For the sake of completeness, however, I would like to point anyone interested in this topic to the tutorials available in [24] and [25].

5. Evaluation and future plans

While some of the designed functionality did not get implemented in the end, I think I have managed to create a solid base for any further development. I have aimed to write high quality code – by strictly following coding conventions – that can be understood, maintained and extended with ease.

I also believe that by the means of the dynamically managed modules a portion of required functionality can be easily added, and by combining the two halves of the project (the core I wrote with the modules written in parallel by Erzsébet Frigó) we get a product that will be useful for the communities we designed it for.

As for the future of the project, a usability test should be conducted within real choirs or bands, and the software should be modified to be in accordance of their requirements. This could possibly require the software to be translated into the native language of the test groups.

We have also collected some wild ideas along the past year – these include an online notation editor for example where users can collaborate in the editing process, a topic that could well worth a thesis in itself – that could be elaborated to add more useful functionality to the project.

6. List of figures

Fig. 1.	IMSLP/Petrucchi Music Library – a MediaWiki based collection [1]	11
Fig. 2.	8notes.com homepage – well structured and easy to navigate [2]	12
Fig. 3.	The project page on Assembla [4].....	16
Fig. 4.	Flowchart (overview)	22
Fig. 5.	Flowchart (prepare new document).....	23
Fig. 6.	Illustration to the hierarchical tags example.....	24
Fig. 7.	Cloud computing layers [7].....	25
Fig. 8.	Update page in the admin section of a WordPress installation	28
Fig. 9.	Serving downloads from file system (green) and from database (orange) using the same host for every service application	30
Fig. 10.	Pricing of GoDaddy, one of the most popular host providers [19] ..	31
Fig. 11.	Overview of the modularized software architecture.....	31
Fig. 12.	Directory structure of a WordPress installation.....	35
Fig. 13.	The activated plugin on the admin interface showing its description 36	
Fig. 14.	Directory structure of the plugin.....	37
Fig. 15.	Overview of the database structure	40
Fig. 16.	Content of the handlers directory.....	42
Fig. 17.	Types of requests and their handing.....	43
Fig. 18.	The modules directory with two modules.....	45
Fig. 19.	List of declared classes at different stages of the module loading ...	47
Fig. 20.	The admin menu section that belongs to the plugin	49
Fig. 21.	Tag categories admin page	49
Fig. 22.	Home page listing the compositions.....	51
Fig. 23.	Page for adding or editing compositions	52
Fig. 24.	Filtering the list of available tags.....	53
Fig. 25.	Creating a new tag	54
Fig. 26.	The tagging interface listing mandatory categories.....	55
Fig. 27.	Composition without documents.....	59
Fig. 28.	Composition with a single document.....	59
Fig. 29.	Page asking the user to confirm the deletion.....	60
Fig. 30.	Uploading a document – stage 1	61
Fig. 31.	Uploading a document – stage 2	62
Fig. 32.	Overview of the meta data processing upon document upload	65
Fig. 33.	Document details page.....	66
Fig. 34.	Versions of the same documents must be of the same type	67
Fig. 35.	The document page showing an inactive document version.....	68
Fig. 36.	Start page of a conversion.....	68
Fig. 37.	Conversion options page rendered for the demo converter	69
Fig. 38.	Conversion options page showing an error.....	70

Developing Music Score Management System for Communities

Fig. 39.	Result page of a conversion	70
Fig. 40.	Example data to illustrate the search function.....	71
Fig. 41.	The search result page showing a matched composition.....	72
Fig. 42.	Page showing the available tags	73
Fig. 43.	Listing content that have a specific tag associated with them.....	73
Fig. 44.	Editing a tag	74
Fig. 45.	Deleting a tag	74

7. Bibliography and references

- [1] IMSLP/Petrucchi Music Library: Free Public Domain Sheet Music, <http://imslp.org/wiki/> (2012-04-22)
- [2] Free sheet music on 8notes.com, <http://www.8notes.com/> (2012-04-22)
- [3] Assembla, <https://www.assembla.com/home> (2012-12-01)
- [4] Project homepage on Assembla(bme kotta önlab), <https://trac.assembla.com/bme-kotta-onlab> (2012-12-01)
- [5] Original flowchart by Zoltán Horváth, <https://trac.assembla.com/bme-kotta-onlab/attachment/wiki/WikiStart/Szerkeszt%C3%A9si%20folyamat.vsd>, (2012-12-01)
- [6] Judith Hurwitz, Robin Bloor, Marcia Kaufman, and Dr. Fern Halper: Cloud Computing For Dummies (ISBN: 978-0-470-48470-8), p.18
- [7] Wikipedia, http://en.wikipedia.org/wiki/File:Cloud_computing_layers.png (2012-12-01)
- [8] Apache HTTP Server Project, <http://httpd.apache.org/> (2012-12-02)
- [9] PHP: Hypertext Preprocessor, <http://php.net/> (2012-12-02)
- [10] MySQL :: The world's most popular open source database, <http://www.mysql.com/> (2012-12-02)
- [11] MediaWiki, <http://www.mediawiki.org> (2012-12-02)
- [12] MediaWiki API documentation, http://www.mediawiki.org/wiki/API:Main_page (2012-04-28)
- [13] WordPress homepage, <http://wordpress.org/> (2012-09-30)
- [14] WordPress Plugin API documentation, http://codex.wordpress.org/Plugin_API (2012-09-30)
- [15] Updating WordPress, http://codex.wordpress.org/Updating_WordPress#Automatic_Update (2012-09-30)
- [16] PHP: http_send_file – Manual, <http://php.net/manual/en/function.http-send-file.php> (2012-09-30)
- [17] XSendfile, <http://wiki.nginx.org/XSendfile> (2012-09-30)
- [18] MySQL :: MySQL 5.0 Reference Manual :: 11.1.6.3 The BLOB and TEXT Types, <http://dev.mysql.com/doc/refman/5.0/en/blob.html> (2012-09-30)
- [19] Web Hosting | Secure Hosting Plans with Unlimited Bandwidth - Go Daddy, <http://www.godaddy.com/compare/hosting.aspx?isc=IAP199H2> (2012-09-30)

- [20] Brad Williams, Ozh Richard, Justin Tadlock: Professional WordPress Plugin Development (ISBN: 978-0-470-91622-3), p.14
- [21] Yannick Lefebvre: WordPress Plugin Development Cookbook (ISBN: 978-1-84951-768-3), p.46
- [22] MySQL Workbench homepage,
<http://www.mysql.com/products/workbench/> (2012-12-03)
- [23] GNU gettext project, <http://www.gnu.org/software/gettext/> (2012-12-06)
- [24] I18n for WordPress Developers,
http://codex.wordpress.org/I18n_for_WordPress_Developers (2012-12-06)
- [25] Yannick Lefebvre: WordPress Plugin Development Cookbook (ISBN: 978-1-84951-768-3), p.275

8. Appendix: Function library documentation (API)

In my thesis project I wanted to implement the software in an extensible way. The open source world is all about forking projects and implementing new functions when someone needs them, so new developers may join a project after it was abandoned by the original author. In these projects, it is important that the code itself is understandable on its own. Clean and well documented code with transparent logic is the base for high quality and maintainable software development.

As in the previous chapters I only highlighted code snippets that were important for understanding the methods by which the main functions work, most of the library functions were omitted. However, a large part of my work was writing these functions, structured and grouped based on their functionality, to provide the smaller bricks from which the main functions could be built. In this appendix, I provide the documentation for these libraries for two reasons: the description of smaller internal functions that didn't get any focus may help the understanding of the whole project, and to provide a reference for possible further development.

8.1. File and upload related functions (*kkm_files*)

`create_dir_struct($path)` (*private static function*)

Creates the directory given in `$path`, and creates every parent if they do not already exist.

Parameters:

- string `$path` Path of the directory to create, relative to `WP_CONTENT_DIR`, without slash at the beginning.

Returns:

Developing Music Score Management System for Communities

- **boolean** True, if the target directory exists at the end of the function, false otherwise.

create_document_path(\$doc_id, \$safe_name) (*public static function*)

Returns the path to the document, and creates the containing directory hierarchy if needed.

Parameters:

- integer **\$doc_id** Document ID from the DB.
- string **\$safe_name** The sanitized filename (as stored in the filesystem).

Returns:

- **string|boolean** Path relative to WP_CONTENT_DIR or WP_CONTENT_URL pointing to the document or false if an error occurred.

delete_document(\$doc_id, \$purge = false) (*public static function*)

Deletes a document from the filesystem, and marks it as deleted in the database (when \$purge is false).

When \$purge is set to true, the complete DB record gets deleted too. This method should be used only when the entire document tree gets deleted (other versions, converted documents, etc).

Parameters:

- integer **\$doc_id** Document ID.
- boolean **\$purge** If false, the DB record is only marked as deleted but doesn't get removed.

Returns:

- **boolean** True if the document was deleted successfully, false if an error occurred.

get_dir_for_id(\$doc_id) (*private static function*)

Returns the path to the directory holding the given document.

Parameters:

- **integer \$doc_id** Document ID from the DB.

Returns:

- **string** The path relative to WP_CONTENT_DIR.

get_document_path(\$doc_id, \$safe_name) (*public static function*)

Returns the path to the document.

Documents are clustered into directories of 100, which dirs are further clustered.

Example paths for ID=1 and ID=15984:

kkm/docs/00/00/000001-safe-name.ext

kkm/docs/01/59/015984-safe-name.ext

Parameters:

- **integer \$doc_id** Document ID from the DB.
- **string \$safe_name** The sanitized filename (as stored in the filesystem).

Returns:

- **string** Path relative to WP_CONTENT_DIR or WP_CONTENT_URL.

get_safe_name(\$name) (*public static function*)

Developing Music Score Management System for Communities

Checks and escapes the filename to be safe to store in the filesystem. Longer filenames also get truncated.

Parameters:

- **string \$name** Original file name.

Returns:

- **string** Safe file name used to store the attachment.

get_temp_dir()

Returns the absolute path to the temporary directory.

Converters may use this directory to store intermediate files, and the framework may store output generated by a handler here.

Returns:

- **string** Absolute path to the temporary directory.

**save_document(\$file_arr, \$comp_id, \$user_id = null,
\$previous_version = null)** (*public static function*)

Saves an uploaded file as a document for a given composition.

Parameters:

- **array \$file_arr** A single item in `$_FILES`, containing the data for the uploaded document.
- **integer \$comp_id** Composition ID in the database.
- **integer \$user_id** Optional. If not provided, the current user's id is used.
- **integer \$previous_version** Optional. If this upload is a newer version of an already stored document, its docid should be provided.

Returns:

- **boolean|integer** The path (relative to the avatars dir) of the processed file on success, or false if there was an error.

stream_file(\$path, \$name) (*public static function*)

Streams the given file to the user.

If `http_send_file()` is available (pre PHP 5.3 with PECL extension, or PHP 5.3+), it will be used, or a PHP 5.0 compatible script will handle it.

Parameters:

- string **\$path** File path, should be absolute.
- string **\$name** Original file name. The browser will suggest this name in the Save As dialog.

stream_file_segment(\$file, \$start, \$end, \$throttle_sleep, \$throttle_buffer) (*private static function*)

Streams a part of the given file (script method).

Parameters:

- resource **\$file** File handler.
- integer **\$start** First byte to send (start of interval, inclusive).
- integer **\$end** Last byte to send (end of interval, inclusive).
- integer **\$throttle_sleep** Sleep time in seconds, as used in `http_throttle()`.
- integer **\$throttle_buffer** Buffer size in bytes, as used in `http_throttle()`.

stream_file_with_http_send_file(\$path, \$name) (*private static function*)

Streams the file with the PECL extension. (Built in function for PHP 5.3+)

Parameters:

- string **\$path** File path, should be absolute.
- string **\$name** Original file name. The browser will suggest this name in the Save As dialog.

stream_file_with_script(\$path, \$name) (*private static function*)

Streams the file with a custom handler script that provides the same functionality as the http_send_file() function.

Parameters:

- string **\$path** File path, should be absolute.
- string **\$name** Original file name. The browser will suggest this name in the Save As dialog.

8.2. Module related functions (*kkm_modules*)

convert_document(\$module, \$source_doc_id, \$source_path, \$dest_doc_id, \$target_path, \$options, \$user) (*public static function*)

Calls the conversion function of the given converter module. Messages generated by the converter will be saved into the database.

The log id will be returned. If an error occurred, the destination file gets deleted.

Parameters:

8. Appendix: Function library documentation (API)

- string **\$module** Converter module class name.
- integer **\$source_doc_id** Document ID of the source.
- string **\$source_path** Absolute path of the source file.
- integer **\$dest_doc_id** Document ID of the source.
- string **\$target_path** Absolute path of the destination file.
- array **\$options** Converter options.
- integer **\$user** User ID, optional. If provided, the conversion will be logged as user action.

Returns:

- integer ID of the log in the database.

get_converters(\$format_from, \$format_to) (*public static function*)

Returns converter classes with that support the given format conversion.

Both parameters are optional. If one is null, no filtering will be done on that end of the conversion. If both parameters are null, every converter will be returned.

Parameters:

- string **\$format_from** Document type to convert from. Optional.
- string **\$format_to** Document type to convert into. Optional.

get_differs(\$format) (*public static function*)

Returns diff classes for the given type.

Developing Music Score Management System for Communities

Parameters:

- string **\$format** Document type, optional. If not set, diff modules will be returned without filtering.

Returns:

- **array** Array of class names for the diff modules.

get_metadata(\$path, \$parsers) (*public static function*)

Calls the parser modules to load metadata from the document. The aggregated results of the parsers will be returned.

In the returned associative array, every value will be an array as opposed to the return value of the module where single values for a given key may be returned as-is.

An example of a valid return value:

```
array(  
    'composer' => array('Jonathan Davis', 'Richard  
Gibbs'),  
    'title' => array('Not meant for me')  
)
```

Parameters:

- string **\$path** Absolute path of the document.
- array **\$parsers** Array of parser classes to use. Should be the result of `get_parsers()`.

Returns:

- **array** Metadata loaded from the file.

get_module_name(\$module) (*public static function*)

Returns the name and description of a module.

Parameters:

- string **\$module** Module class name.

get_parsers(\$format) (*public static function*)

Returns parser classes for the given type.

Parameters:

- string **\$format** Document type, optional. If not set, parsers will be returned without filtering.

Returns:

- array Array of class names for the parser modules.

get_preview(\$path, \$parsers) (*public static function*)

Calls the parser modules to generate a preview. All modules will be called until one returns a non-empty string.

Parameters:

- string **\$path** Absolute path of the document.
- array **\$parsers** Array of parser classes to use. Should be the result of `get_parsers()`.

Returns:

- string HTML fragment for preview. If a preview cannot be generated, an empty string will be returned.

get_target_extension(\$module) (*public static function*)

Returns the file extension that belongs to the output file generated by the specified converter module.

Parameters:

- string **\$module** Converter module class name.

load_interfaces() (*private static function*)

Developing Music Score Management System for Communities

Includes the interface definitions.

load_modules() (*private static function*)

Loads modules from the kkm/modules directory and stores information about them into the static `$_modules` class variables.

Modules are static classes, and we load them by comparing the declared classes list before and after the inclusion of the files.

load_modules_from_dir(\$path) (*private static function*)

Loads modules from a directory. This is called for each module type separately.

A module must have a `module.php` file in its directory, as that will be included. In those files every module defines a static class implementing the proper interface.

Parameters:

- string **\$path** Absolute path of the directory to load. Trailing / needed.

save_metadata(\$path, \$parsers, \$meta) (*public static function*)

Calls the `save_metadata()` function on each given parser to write the given meta into the file.

Parameters:

string **\$path** Absolute path of the document.

array **\$parsers** Array of parser classes to use. Should be the result of `get_parsers()`.

array **\$meta** Associative array containing the metadata.

validate_document(\$path, \$parsers) (*public static function*)

Validates the file with the given parser modules. All modules will be called until one returns an error, or there's no more module.

Parameters:

- string **\$path** Absolute path of the document.
- array **\$parsers** Array of parser classes to use. Should be the result of `get_parsers()`.

Returns:

- **array** Array of aggregated messages from all validators. If the file couldn't be validated, null will be returned. For clean and valid files an empty array is the result.

validate_options(\$module, \$path, \$options) (*public static function*)

Returns the result of the option validation of the given converter module.

Parameters:

- string **\$module** Converter module class name.
- string **\$path** Absolute path of the document.
- array **\$options** Options to validate (coming from the user).

Returns:

- **array** Array of messages returned by the validator. If no errors occurred, an empty array is returned.

8.3. Module options processing related functions (*kkm_options*)

`render_form_field($option_name, $type, $params)` (*private static function*)

Renders a form element for the given option variable.

Parameters:

- string **\$option_name** Name of the field.
- string **\$type** Type of the field. Accepted values are 'text' and 'list'.
- array **\$params** Parameters for the field.

Returns:

- **string** The generated HTML fragment.

`render_list_field($option_name, $params)` (*private static function*)

Renders a list (<select>) field to be displayed on the form.

Parameters:

- string **\$option_name** Name of the field.
- array **\$params** List items. If non-numerical keys are defined, they will be used as the values for the list elements.

Returns:

- **string** The generated HTML fragment.

`render_options_form($doc_id, $module, $source_file, $form_action)` (*public static function*)

Renders a form for choosing the options for a converter.

Parameters:

- integer **\$doc_id** Document ID in the database.
- string **\$module** Converter module class name.
- string **\$source_file** Absolute URL of the file to convert. Forwarded to the module.
- string **\$form_action** Handler name.

render_text_field(\$option_name) (*private static function*)

Renders a text field to be displayed on the form.

Parameters:

- string **\$option_name** Name of the field.

Returns:

- **string** The generated HTML fragment.

8.4. Search related functions (*kkm_search*)

get_matching_compositions(\$terms, \$tags) (*private static function*)

Loads compositions from the database that have a title matching the search terms, or has one of the matching tags.

Parameters:

- array **\$terms** Array of strings. These terms are included into the SQL query without any check, so escape terms properly before passing it to this function. If null is passed, the title of the composition will not be matched.
- array **\$tags** Array of integers, containing tag IDs.

Returns:

Developing Music Score Management System for Communities

- **array** Composition rows from the database.

get_matching_documents(\$terms, \$tags, \$compositions) (*private static function*)

Loads documents from the database that have a filename matching the search terms, belongs to a matched composition, or has one of the matching tags.

Parameters:

- **array \$terms** Array of strings. These terms are included into the SQL query without any check, so escape terms properly before passing it to this function. If null is passed, filename won't be checked.
- **array \$tags** Array of integers, containing tag IDs.
- **array \$compositions** Array of integers, containing composition IDs.

Returns:

- **array** Document rows from the database.

get_matching_tags(\$terms) (*private static function*)

Returns the tags from the database that match at least one of the search terms.

Parameters:

- **array \$terms** Array of strings. These terms are included into the SQL query without any check, so escape terms properly before passing it to this function.

Returns:

- **array** Rows from the database.

get_search_results(\$search_string) *(public static function)*

Runs a search in the database and returns matching documents and compositions.

In the returned array there're two elements with string keys: an array of document rows and an array of compositions rows from the DB (each is an array in itself returned by `$wpdb->get_result()`).

Parameters:

- string **\$search_string** Search query string entered by the user.

Returns:

- **array** Array of arrays. If no results found, both sub-arrays will be empty. If there's no valid search term present, no results will be provided.

get_search_results_by_tag(\$tag_id) *(public static function)*

Returns the documents and compositions that have the given tag.

In the returned array there're two elements with string keys: an array of document rows and an array of compositions rows from the DB (each is an array in itself returned by `$wpdb->get_result()`).

Parameters:

- integer **\$tag_id** Tag ID.

Returns:

- **array** Array of arrays. If no results found, both sub-arrays will be empty. If there's no valid search term present, no results will be provided.

`get_search_terms($search_string)` (*private static function*)

Returns the normalized search terms from the user entered search query string.

During normalization every word gets converted into lowercase, and common words get excluded. The resulting array will contain every word only once (repetitions are excluded), and will be in alphabetic order.

Parameters:

- string **\$search_string** User entered string.

Returns:

- array Array of strings.

`is_every_term_fulfilled($terms, $words, $strong_check)` (*private static function*)

Checks whether the given list of words fulfill every search term.

If strong check is enabled, every search term must be present in the list of provided words as-is. If it's disabled, a regular expression will be used that may match a term as a part of a word.

Parameters:

- array **\$terms** Array of strings. Normalized search terms.
- array **\$words** Array of strings. Normalized index of the document/composition.
- boolean **\$strong_check** Determines the comparison method. Defaults to false (soft checking with regular expression).

Returns:

- **boolean** True if every term has its match in \$words.

8.5. Tag management related functions (*kkm_tags*)

create_tags(\$tags) (*public static function*)

Saves new tags into the database.

Parameters:

- **array \$tags** Array of (tag category ID - tag name) pairs.

Returns:

- **array** Tag IDs of newly created tags.

delete_tag(\$tag_id) (*public static function*)

Deletes a tag from the database.

The tag will be removed from every document and composition, and children tags will be updated so as their parent will be set to the parent of this tag. Tag hierarchy is maintained.

Parameters:

- **integer \$tag_id** Tag ID.

Returns:

- **boolean** True on success, false on failure.

get_format_tag_for_document(\$doc_id) (*public static function*)

Loads the format tag of a document.

Parameters:

- **integer \$doc_id** Document ID in the database.

Developing Music Score Management System for Communities

Returns:

- **array** Tag from the database. Category name and color will be appended to the row. If the tag could not be found, null will be returned.

get_mandatory_name(\$flag) (*public static function*)

Returns the localized name of the mandatory flag.

Parameters:

- integer **\$flag** The mandatory flag for the tag.

Returns:

- **string** Localized name, e.g. 'Mandatory', 'Optional'

get_tags_for_composition(\$comp_id, \$mandatory_filter) (*public static function*)

Returns tags for the given composition.

The **\$mandatory_filter** parameter restricts the tag categories from which the tags will be returned:

- 3 - Only mandatory tags
- 2 - Mandatory and visible optional tags
- 1 - Every tag assigned to the composition

Parameters:

- integer **\$comp_id** Composition ID in the database.
- integer **\$mandatory_filter** One of the **FILTER_*** constants.

Returns:

- **array** Tags from the database. Category names and colors will be appended to the rows. If no tags could be found, an empty array will be returned.

get_tags_for_document(\$doc_id, \$mandatory_filter) (*public static function*)

Returns tags for the given document.

The `$mandatory_filter` parameter restricts the tag categories from which the tags will be returned:

- 3 - Only mandatory tags
- 2 - Mandatory and visible optional tags
- 1 - Every tag assigned to the composition

Parameters:

- integer **\$doc_id** Document ID in the database.
- integer **\$mandatory_filter** One of the `FILTER_*` constants.

Returns:

- **array** Tags from the database. Category names and colors will be appended to the rows. If no tags could be found, an empty array will be returned.

is_valid_format_tag(\$tag_id) (*public static function*)

Checks whether the given tid belongs to an existing tag in the Format category. Used to validate value sent by the user.

Parameters:

- integer **\$tag_id** The tid of a tag.

Returns:

- **boolean** True if it's a valid format tag, false otherwise.

remove_tags_from_comp(\$comp_id, \$excluded_categories) (*private static function*)

Developing Music Score Management System for Communities

Removes every tag from the composition that does not belong to the list of excluded categories.

Parameters:

- integer **\$comp_id** Composition ID in the database.
- array **\$excluded_categories** Optional. Array of tag category IDs.

remove_tags_from_doc(\$doc_id, \$excluded_categories) (*private static function*)

Removes every tag from the document that does not belong to the list of excluded categories.

Parameters:

- integer **\$doc_id** Document ID in the database.
- array **\$excluded_categories** Optional. Array of tag category IDs.

render_tag(\$name, \$bg_color, \$title, \$return) (*public static function*)

Renders a tag.

Parameters:

- string **\$name** Label of the tag.
- string **\$bg_color** Background color of the tag.
- string **\$title** Title of the tag.
- boolean **\$return** If set to true, the HTML fragment of the tag will be returned, otherwise it will be echoed. Defaults to false.

`render_tag_list($category, $multiple, $name, $attributes,
$selected, $show_empty, $return)` (*public static
function*)

Renders a list of tags from the given category as a HTML
<select>.

Parameters:

- integer|string **\$category** Category name or ID.
- boolean **\$multiple** If true is given, a multiple choice list will be rendered.
- string **\$name** Name and ID of the <select> tag.
- string **\$attributes** Optional. This string will be inserted into the opening tag. May contain CSS class and JavaScript event definitions.
- array|integer **\$selected** Optional. If an array of integers is given, the tags with these IDs will be selected by default. For dropdown lists, a single integer may be used.
- boolean **\$show_empty** Optional. If set to true, an empty row will be rendered at the top with it's value set to 0.
- boolean **\$return** Optional. If it's set to true, the HTML fragment will be returned instead of echoed.

`render_tag_list_items($tags, $parent, $selected, $level)`
(*private static function*)

Renders <option> elements for tags that belong to the given parent tag. If a tag has children, this function will be called again.

Parameters:

- array **\$tags** Tags from the DB.

Developing Music Score Management System for Communities

- integer **\$parent** Parent tag ID.
- array **\$selected** If an array of integers are given, the tags with these IDs will be selected by default.
- integer **\$level** Indentation level. Defaults to 0.

Returns:

- **string** HTML fragment.

render_tag_table(\$tags) (*private static function*)

Renders the tags in a HTML <table> element.

Parameters:

- array **\$tags** The result of `get_tags_for_composition()` or `get_tags_for_document()`.

render_tag_table_for_composition(\$comp_id) (*public static function*)

Renders the tags for the given composition as a HTML <table>.

Parameters:

- integer **\$comp_id** Composition ID.

render_tag_table_for_document(\$doc_id) (*public static function*)

Renders the tags for the given composition as a HTML <table>.

Parameters:

- integer **\$doc_id** Document ID.

render_tagging_box(\$field_name, \$type, \$assigned_tags, \$suggested_tags, \$excluded_categories) (*public static function*)

Renders a HTML fragment containing the interface for assigning tags to a document or composition. The generated code is returned.

Parameters:

- string **\$field_name** Name for the form element that stores the selected tags.
- string **\$type** Enforce mandatory tag categories for documents ('doc') or compositions ('comp'). When left empty, no enforcement will be done.
- array **\$assigned_tags** Array of tag IDs. Optional.
- array **\$suggested_tags** Array of integers and [category ID - name] pairs.
- array **\$excluded_categories** Array of integers (category IDs). Tags from these categories won't be displayed.

Returns:

- **string** HTML fragment.

render_tagging_box_for_comp(\$comp_id, \$field_name, \$suggested_tags) (*public static function*)

Renders an HTML interface for assigning tags to a composition. The generated code is returned.

Parameters:

- integer **\$comp_id** Composition ID in the database.
- string **\$field_name** Name for the form element that stores the selected tags.
- array **\$suggested_tags** Optional array of tags that are currently not assigned to the document. Elements of

Developing Music Score Management System for Communities

this array may be integers (tag IDs), or string arrays (tag category - name) pairs or arrays of (category ID - name) pairs.

Returns:

- **string** HTML fragment.

```
render_tagging_box_for_doc($doc_id, $field_name,  
    $suggested_tags) (public static function)
```

Renders an HTML interface for assigning tags to the document. The generated code is returned.

Parameters:

- integer **\$doc_id** Document ID in the database.
- string **\$field_name** Name for the form element that stores the selected tags.
- array **\$suggested_tags** Optional array of tags that are currently not assigned to the document. Elements of this array may be integers (tag IDs), or string arrays (tag category - name) pairs or arrays of (category ID - name) pairs.

Returns:

- **string** HTML fragment.

```
sanitize_suggested_tags($suggested_tags) (private static  
function)
```

Tries to load existing tag IDs for the given category-name pairs and removes tags that do not fit any existing category.

Parameters:

- array **\$suggested_tags** Array of tags that are currently not assigned to the document. Elements of this array may be integers (tag IDs), or string arrays (tag category - name) pairs or arrays of (category ID - name) pairs.

save_tags_for_comp(\$comp_id, \$taglist) (*public static function*)

Saves the tags assigned to the composition on the HTML interface by the user.

Parameters:

- integer **\$comp_id** Composition ID in the database.
- string **\$taglist** The value returned by the tagging interface.

save_tags_for_doc(\$doc_id, \$taglist) (*public static function*)

Saves the tags assigned to the composition on the HTML interface by the user.

Parameters:

- integer **\$ doc_id** Document ID in the database.
- string **\$taglist** The value returned by the tagging interface.

update_tag_parent(\$tag_id, \$parent) (*public static function*)

Updates the tag hierarchy so that the given tag will have \$parent as its new parent tag.

This function updates both the kkm_tags and kkm_tag_relations tables. The parent of a tag cannot be itself or one of its descendants. If its called to do such an update, the function will return with false without updating the tables.

Parameters:

- **integer** `$tag_id` Tag ID.
- **mixed** `$parent` ID of new parent tag or 'null'.

Returns:

- **boolean** Returns false on failure, true on success.

8.6. User related functions (kkm_user)

`get_rights()` (*public static function*)

Returns the highest access level the user possesses.

Returns:

- **integer** The permission level of the current user.
Comparable to the `RIGHT_*` constants of this class.

9. Appendix: Module interface documentation (API)

As I described in my thesis, the modularized architecture is one of the main features of this plugin. I have already explained the methods by which the plugin loads the converter, parser, and diff modules, and in this appendix I'd like to provide the full documentation of the interfaces to help the reader get a better view of the inner workings as well as to provide the reference necessary for developing modules that extend the functionality of our software.

9.1. Converter module interface

get_name() (*public static function*)

Returns the human-readable name of the module along with a short description. These will be displayed both on the admin page and on the frontend.

An example of a valid return value:

```
array(
    'name' => 'PDF Converter',
    'description' => 'Converty LilyPond documents into
PDF.'
```

Returns:

- **array** An array of strings.

is_automatic() (*public static function*)

Returns whether the converter can be called automatically upon a file upload. If the module may require information specific to a single input file, it should return false.

Returns:

Developing Music Score Management System for Communities

- **boolean** True if the converter can be run unattended, false otherwise.

get_conversion_formats() (*public static function*)

Returns an array of document format pairs that this class is capable to convert between.

An example of a valid return value:

```
array(  
    array('LilyPond', 'PDF Sheet')  
)
```

Returns:

- **array** An array of document format pairs.

get_target_extension() (*public static function*)

Returns a file extension to be used in the name of the destination file.

Returns:

- **string** Extension used in the filename.

get_conversion_options(\$path) (*public static function*)

Returns an array of options that are presented to the user.

An example of a valid return value:

```
array(  
    'title'=> array('text', 'Custom title for the  
document:'),  
    'size' => array('list', 'Select paper size:',  
array('A4','A5')),  
    'orientation' => array('list', 'Select paper  
orientation:', array('p' => 'Portrait', 'l' =>  
'Landscape'))  
)
```

9. Appendix: Module interface documentation (API)

Text options are displayed as a text type input field, lists are displayed as dropdown menus with the options listed in the third element of the array. If the keys of the array are string, it will be used in the value attribute of the <option> tag.

Parameters:

- string **\$path** Full (absolute) path to the input file.

Returns:

- array Array of option descriptions.

validate_conversion_options(\$path, \$options) (*public static function*)

Validates the conversion options entered by the user.

This function will get the options as an array:

```
array(  
    'title' => '',  
    'size' => 'A4',  
    'orientation' => 'p'  
)
```

Parameters:

- string **\$path** Full (absolute) path to the input file.
- array **\$options** Options as an array.

convert(\$source, \$destination, \$options) (*public static function*)

Converts the input file into the output file.

An example of a valid return value:

```
array(  
    array('warning', 'No instrument defined, using Grand
```

```
Piano by default.'),  
    array('note', 'Conversion done.')  
)
```

Parameters:

- string **\$source** Full (absolute) path to the input file.
- string **\$destination** Full (absolute) path to the output file.
- array **\$options** Options as an array. See `validate_conversion_options` for more information.

Returns:

- **array** Messages that will be presented to the user. A message is an array containing its type ('note', 'warning' or 'error') and the message string itself. If error is returned, the generated file (if it exists) will be deleted. Notes and warnings are presented to the user in different ways, but do not have any effect on the conversion pipeline.

9.2. *Diff module interface*

`get_name()` (*public static function*)

Returns the human-readable name of the module along with a short description. These will be displayed both on the admin page and on the frontend.

An example of a valid return value:

```
array(  
    'name' => 'Lyrics diff',  
    'description' => 'Displays the two versions of the  
lyrics highlighting the differences.'  
)
```

Returns:

- **array** An array of strings.

get_accepted_formats() (*public static function*)

Returns an array of the accepted document formats.

An example of a valid return value:

```
array(  
    'LilyPond'  
)
```

Returns:

- **array** An array of strings containing file format labels.

get_diff(\$path1, \$path2) (*public static function*)

Calculates the difference between two files of the same format.

The returned HTML code will be inserted into the diff page and presented to the user.

Parameters:

- **string \$path1** Full (absolute) path to the file.
- **string \$path2** Full (absolute) path to the file.

Returns:

- **string** HTML fragment.

9.3. Parser module interface

get_name() (*public static function*)

Returns the human-readable name of the module along with a short description. These will be displayed both on the admin page and on the frontend.

Developing Music Score Management System for Communities

An example of a valid return value:

```
array(  
    'name' => 'LilyPond parser',  
    'description' => 'Can load .ly files.'  
)
```

Returns:

- **array** An array of strings.

get_accepted_formats() (*public static function*)

Returns an array of the accepted document formats.

An example of a valid return value:

```
array(  
    'LilyPond'  
)
```

Returns:

- **array** An array of strings containing file format labels.

validate(\$path) (*public static function*)

Loads and validates the given file.

The validation will be only called with files that belongs to one of the formats defined by the `get_accepted_formats()` function.

An example of a valid return value:

```
array(  
    array('warning', 'No author specified.'),  
    array('error', 'Invalid file format: this is not a  
valid LilyPond file.')  
)
```

If an error is returned, the file upload will be terminated. If only warning messages are returned, they

will be shown to the users, and they might choose to continue the uploading/saving progress.

Parameters:

- string **\$path** Full (absolute) path to the file.

Returns:

- **array** An array of error and warning messages. A message is an array containing its type ('warning' or 'error') and the message string itself. For valid files an empty array should be returned. If the validation is not or cannot be implemented, null should be returned (e.g. it's only metadata loader module).

get_metadata(\$path) (*public static function*)

Loads metadata from the file.

An example of a valid return value:

```
array(  
    'composer' => array('Jonathan Davis', 'Richard  
Gibbs'),  
    'title' => 'Not meant for me'  
)
```

The key is treated as a Tag category name, and the system may select existing tags from this category or suggest creating new ones with the label specified in the value. If multiple values belong to a key, they should be grouped into an array.

Parameters:

- string **\$path** Full (absolute) path to the file.

Returns:

Developing Music Score Management System for Communities

- **array** Array containing metadata as key-value pairs. An empty array or null should be returned for files that do not contain metadata, or in the case the class cannot load it.

save_metadata(\$path, \$meta) (*public static function*)

Saves modified metadata into the file.

The system may call this function before saving the uploaded file into the storage, if any metadata has been modified or assigned to the document.

Parameters:

- **string \$path** Full (absolute) path to the file.
- **array \$meta** An array of metadata. The format is the same as for the return value of `get_metadata()` function.

get_preview(\$path) (*public static function*)

Loads the file and generates a preview for it. This preview will be presented to the user during the document uploading/saving process.

Parameters:

- **string \$path** Full (absolute) path to the file.

Returns:

- **string** HTML segment that will be included into the page and displayed to the user during the uploading/saving process.

10. Appendix: Content of the attached CD

I have attached a CD to my thesis with files that could provide useful to any reader or further developer. In this appendix I provide an overview on its content.

\Attila Wagner - Thesis.pdf

My thesis in Portable Document Format. It's compatible with Adobe Acrobat X and later versions.

\Software\kkm

This is the WordPress plugin itself. As with any other plugin, installation is done by uploading this directory to the `wp-content/plugins` directory then enabling it from the administration interface of WordPress.

This attached version of the plugin only includes code written entirely by myself. Those who are interested in the full functionality of the developed software, or those who plan on contributing to its development should download it from the SVN repository accessible from the homepage of the project [4].

\Software\wordpress-3.4.2.zip

This is the full installation package for WordPress version 3.4.2 available on the WordPress homepage [13]. I have included it so the plugin can be tested in the future even in the case a later version of WordPress becomes incompatible with the plugin.

\Source images

This directory contains the image files that have been included as figures in my thesis.