Authorization Tool for Users and Media Archives in WordPress
A Role Based Authorization Plugin

Anton Björkenvall
Abstract

Authorization Tool for Users and Media Archives in WordPress

Anton Björkenvall

Access control is the central element of computer security and its principal objectives are to implement secure authentication and authorization in a system.

Authentication is the verification that the credentials of a user are valid and authorization is the granting of permissions to a user to access system resources.

This thesis describes the design and implementation of an authorization plugin, called File Authorization Manager, for WordPress built upon an already existing authentication plugin, called privateContent. Due to various factors mentioned in the thesis the implementation of said authorization plugin is, in its current state, not feature complete and therefore not working as intended.

Due to these factors this thesis not only describes the design and implementation of an authorization plugin, but also the intricacies with developing plugins that alter the core functions of WordPress.
## Contents

1 **Introduction** ........................................ 1
    1.1 Background ........................................ 1
    1.2 Access Control ...................................... 1
    1.3 Access Control Policies ......................... 2
        1.3.1 Discretionary Access Control (DAC) ............ 3
        1.3.2 Role Based Access Control (RBAC) .............. 3
        1.3.3 Mandatory Access Control (MAC) ................ 3
    1.4 WordPress .......................................... 3
    1.5 Enhanced Media Library PRO ........................ 6
    1.6 privateContent .................................... 7
    1.7 WampServer ........................................ 8
    1.8 The .htaccess file .................................. 8

2 **Requirements** ...................................... 10
    2.1 External Requirements .............................. 10
    2.2 Internal Requirements .............................. 11

3 **Design** .............................................. 12
    3.1 Design Overview .................................... 12
    3.2 Database Layout .................................... 13
    3.3 User Experience .................................... 15
        3.3.1 Graphical User Interface Solution ................ 15
        3.3.2 Invisible Server Side Solution .................. 17
        3.3.3 The Settings Page ............................... 18
    3.4 Replace a File ...................................... 20

4 **Implementation** ..................................... 22
    4.1 Framework ......................................... 22
        4.1.1 FAM and WordPress ............................... 22
        4.1.2 Activation and Deactivation ..................... 24
        4.1.3 The Database .................................... 25
        4.1.4 Frontend to Backend ............................. 25
        4.1.5 Upgradeability .................................. 26
    4.2 Interconnectivity .................................. 26
        4.2.1 privateContent .................................. 26
        4.2.2 Enhanced Media Library PRO ...................... 28
        4.2.3 .htaccess ....................................... 28
            4.2.3.1 The changes made to the .htaccess-file .......... 28
            4.2.3.2 The interception of a file request and the autho-
                   rization involved ................................ 30
1 Introduction

When a user requests a file from a website many moving parts are involved. In order to make sure that the user requesting a file is truly who they say they are, an authentication tool needs to be in place. In addition to this authentication tool the system needs to be able to determine what files should be made available to this user. The latter step is called authorization and this is based on the information gathered from the authentication step as well as other mechanisms decided by a system administrator.

This thesis will examine how such an authorization plugin can be designed and implemented upon an already existing authentication plugin.

1.1 Background

Svensk Kärnbränslehantering AB, abbreviated SKB, is a Swedish nuclear fuel and waste management company which uses an intranet based on WordPress. This intranet is used by their employees and customers in Sweden as well as abroad. In the current WordPress-setup SKB uses privateContent, an authentication plugin that provides users with a login/logout, password handling and assignment to groups, and Enhanced Media Library PRO, a plugin that enables the users to create media categories and assign these categories to media uploads. This setup enables SKB to enable users to log in and out of the intranet as well as categorizing media when uploading it. The system in its current implementation lacks an authorization tool which can restrict the access between users and files and setup an interface in which the system administrator can manage the restrictions and enable access for user groups based on filetype, the date when the file was uploaded and the Enhanced Media Library PRO-category of the file. The plugin will use the privateContent user-data and its user-categories to authorize access to files. This thesis will focus on the design and implementation of said plugin as well as discuss the intricacies of dealing with server based access rules. The plugin is named File Authorization Manager and is sometimes in this thesis, when talking about certain files and other associations, abbreviated ab-fam or FAM. The ab, in ab-fam, is the author’s initials and are used since WordPress can use any object, or variable, globally and therefore a more unique naming scheme is preferred.

1.2 Access Control

Access control is the central element of computer security and its principal objectives are to prevent unauthorized users from gaining access to resources, to prevent legitimate users from accessing resources in an unauthorized manner and to enable legitimate users to access resources in an authorized manner. When looking at access control two major components stand out: authorization and authentication.

Authentication is a verification that the credentials of a user are valid. This
function therefore makes sure Person X is indeed Person X and not an illegitimate user.

Authorization is the granting of permissions to a user to access system resources. This function therefore determines who is trusted for a given purpose. In this step it is known that Person X is truly Person X and the function determines which, if any, resources Person X should have access to.

[Figure 1] explains the relationship between authorization and authentication and also introduces another entity, the security administrator. It is the security administrator’s job to decide what type of access to which system resources is allowed for a certain user. [1]

![Figure 1: The relationship between authorization and authentication.](image)

1.3 Access Control Policies

When designing an authorization database the authorization itself needs to be based on policies. These policies in turn dictates what types of access are permitted, under what circumstances, and by whom. Policies like these are often grouped into categories that serves as frameworks for implementing access control. Even though these policies can seem to be mutually exclusive many of today’s system’s access control policies are combinations and alterations of
the policies mentioned below. Deciding which policy to use is crucial to the system since changing policy in an already implemented system can be a complex undertaking. In the designing of File Authorization Manager it was decided that a role based access control was favored since SKB already had access control policies based on user roles implemented in their company structure. However the access control in File Authorization Manager can also be considered as mandatory since users are not allowed to change the classification and access of files, whether they are their own or public. [1]

1.3.1 Discretionary Access Control (DAC)
Discretionary Access Control controls access based on the identity of the requestor and rules set by the administrator that dictates what this requestor is allowed to do. This policy therefore makes it possible for a user to dictate its own rules, if allowed by the administrator, and allow other users to access its files. An analogy of this is Facebook where a user can control access among its friends to view pictures, status updates etc. [1]

1.3.2 Role Based Access Control (RBAC)
Role Based Access Control base the access control on user groups, or roles, instead of the individual. This policy is therefore typical in corporations where, for example, the marketing division should be able to access files the finance division should not. [1]

1.3.3 Mandatory Access Control (MAC)
Mandatory Access Control is the strictest of all policies and bases the control on comparing security labels with security clearances. This policy is typically used in critical systems and governments where each file has a classification and where users should not be able to control access by their own volition. [1]

1.4 WordPress
WordPress is a full content management system built on PHP and MySQL. WordPress is installed on a network server and serves as a framework upon which other templates and plugins can be built. Plugins are PHP scripts which uses WordPress’ hooks and filters, instructions given by WordPress at certain events, and upon calling these hooks or filters can alter the way a web page is working both internally and visually. Themes are like plugins but only alter the user interaction interface of a web page. When installing WordPress it sets up a MySQL-database which is used for storing internal data as well as plugin and theme data. WordPress uses globally accessible objects to access and change the database as well as building new tables. To access settings of WordPress as well as manage installed plugins and themes WordPress uses an interface called Dashboard [2]:

[1]: https://example.com
[2]: https://example.com
Figure 2: The WordPress Dashboard where all settings can be accessed as well as installation and alteration of themes and plugins.
Figure 3: The Plugins-menu where the admin access the management of plugins. When a plugin-folder is stored in the right sub-directory, /wp-content/plugins/, as well as the naming of the PHP-files and applying certain headers in the main PHP-file, WordPress will treat it as a plugin and will show it on this page. The admin can then install, de-install, activate and deactivate the plugin as long as this functionality is implemented in the plugin.

Since WordPress relies upon third-party plugins in order to be as customizable and upgradeable as possible a system with hooks, called actions and filters, is implemented throughout WordPress. These actions and filters provide a third-party plugin developer with a way to insert code to be run when certain events are triggered. The following list of actions is an example of how a hook can be used in a certain event:

1. A user logs in. This triggers an action event which a plugin developer can use in order to insert code into WordPress.

2. In order to insert code after a user logs in, but before any other code is run by WordPress, the plugin developer inserts the hook `wp_login` in the plugin code along with the additional PHP instructions to be run.

3. WordPress inserts this code right after the user login happens and continues to run whatever instructions it is going to run next.
<?php

function your_function() {
    // your code
}

add_action('wp_login', 'your_function');
?>

Figure 4: In order to insert code into WordPress when a user logs in the action hook `wp_login` is used. The function `add_action` is used to call the action hook and `your_function` is the function that contains the PHP code to be inserted into WordPress when the action hook is triggered. [3]

1.5 Enhanced Media Library PRO

Enhanced Media Library PRO is a categorizing plugin which will present the user with the choice of applying categories to a file upon file upload. The plugin also rewrites the file upload sub-menu and creates a file-browser where the user can filter the files based on Enhanced Media Library PRO’s categories, dates and file types. [4] In this thesis Enhanced Media Library PRO will sometimes be abbreviated EMLP.
Figure 5: This figure shows the sub-menu Media Library when Enhanced Media Library Pro is activated. The highlighted part in the upper field shows the filtering options the admin is presented with. The highlighted part in the middle field shows each file’s media category. In this example each media file is only assigned to one media category but any file can have any number of media categories assigned to it.

1.6 privateContent

`privateContent` is a plugin that provides WordPress with a multi level membership platform. This authentication plugin enables the admin to create and organize user categories and further assign users to these categories. The plugin also provides `WordPress` with a registration form in which the users can register and a login function. The data used by `privateContent` is stored in the `WordPress` database tables, created by `privateContent`, named `wp_pc_user_meta` and `wp_pc_users`. 
1.7 WampServer

WampServer is a Windows web development environment which allows users to create web applications with Apache, PHP and MySQL databases. Alongside these tools, WampServer includes PhpMyAdmin to manage the databases. [8] Apache is a commercial-grade and freely-available source code implementation of an HTTP web server. [9]

1.8 The .htaccess file

The .htaccess file is a per-directory configuration file which means that each directory in the Apache installation can have its own .htaccess file. These files provide the server with compartmentalized configuration and can be used to delegate administration tasks. In the case Apache receives a request for a file, /local/foo/bar.html, it will look for the per-directory configuration file in /, /lo-
cal, /local/foo directories, in that order. The use of .htaccess has a performance penalty, however, since Apache needs to look for the per-directory files each time a request is received. This penalty is not a problem when it comes to developing plugins for WordPress since WordPress already makes use of .htaccess files in the server and therefore the rules needed for File Authorization Manager can be added to already existent .htaccess files with already existent penalties. [11]
2 Requirements

“A complete understanding of software requirements is essential to the success of a software development effort. No matter how well designed or well coded, a poorly analyzed and specified program will disappoint the user and bring grief to the developer.”[11] Well written requirements made prior to the design phase is critical not only in order to meet the demands of the customer of the software but also to ease the designing of the software. This section contains the requirements made by the author and SKB and are divided into two sections, external requirements and internal requirements. The external requirements focus on what SKB require of File Authorization Manager and the internal requirements focus on interconnectivity requirements between the plugin and the framework.

2.1 External Requirements

Certain external requirements were set upon the plugin by SKB during the planning of the project in regards to safety, support, future implementations and functionality. The following requirements were set by SKB:

1. The plugin will manage authorization for files uploaded to the WordPress media archive.

2. In order to authorize a user’s access the plugin must be able to work with a separate authentication plugin, in this case privateContent, and its user database.

3. SKB uses Enhanced Media Library PRO in order to categorize the media files and the plugin must therefore be able to work in conjunction with this plugin.

4. Inline linking, sometimes referred to as hotlinking, should be prevented by the plugin.

5. If an old media file is replaced by a new media file, the plugin will make sure that the new media file keeps the old file’s dependencies and name in order to make sure that the web page is unaltered by the change. If a file is not of the same file type as the one it’s replacing the user should be warned and the replacement should be aborted.

6. The plugin should handle user data in a secure manner.

7. The plugin must make sure it can be continuously supported and upgradeable. In addition it must be able to include functionality to be activated, deactivated, installed and deinstalled.

8. The plugin should have a graphical user interface that makes it easy for the administrator to set up and maintain access rights.
In privateContent each user can be a part of any number of user categories and it is these user categories that serve as the foundation of the authorization part of the plugin. In addition Enhanced Media Library PRO injects its own categories into the media archive and these categories will as well be a part of the authorization process of the plugin. Since the plugin is to be used in conjunction with both privateContent and Enhanced Media Library PRO it will base the user-to-file authorization on:

1. The groups, or user categories, the user belongs to.
2. The date the file was added to the media archive.
3. The Enhanced Media Library Pro-categories the file belongs to.
4. The file type.

2.2 Internal Requirements

In order for the plugin to properly work with WordPress it must follow the WordPress coding standard for plugins. This includes placing the files in the right subdirectory in order for WordPress to find the plugin. In addition the header of the main file must follow a coding and naming convention:

```php
/*
 * Plugin name: File Authorization Manager
 * Plugin URI: http://example.com/wordpress-plugins/file-authorization-manager
 * Description: An authorization plugin.
 * Version: 1.0
 * Author: Anton Björkman
 * License: GPLv2
 */
```

Figure 7: This figure shows the header requirements that WordPress needs in order to process the file as a plugin.

When a file meets these standards it is parsed by WordPress as a plugin and can use the PHP-framework upon which WordPress is built. This enables the plugin to use WordPress' hooks and filters which are triggers that the plugin can use to run code in certain events. When it comes to the server settings it is important to make sure that WordPress is allowed to alter and use files in the server's, in this case Apache's, root since WordPress itself makes and alters rewrite rules.
3 Design

The development process focuses on how a program should be implemented. The first step of deciding this how is made in the design step. In order to design a plugin like FAM the requirements of the previous section needs to be first translated and then addressed.

3.1 Design Overview

When planning the overall design of the plugin, FAM, the interconnectivity between FAM, privateContent, Enhanced Media Library PRO and WordPress itself is of vast importance. In order to make these inter-connections work, two weeks week of the project’s total time frame was allocated towards this - a great part considering the scope of the project. Since FAM, the authorization plugin, has to work in conjunction with privateContent, the authentication plugin, the design of FAM has to take privateContent’s API into consideration and also make sure that changes in some of privateContent’s user data, such as deletion and addition of users, are reflected in FAM’s data. Enhanced Media Library PRO is of equal importance regarding the design of the plugin but since EMLP only affects the relationships inside the FAM-database its implementation is straightforward and therefore a lot of the design work that is a part of the connectivity between FAM and EMLP is omitted in the design overview. The following figure shows the overall design of the plugin:

Figure 8: The design of the interaction between WordPress, privateContent and File Authorization Manager.

The figure shows what happens when a user wants to access both a singular file and multiple files. The following steps describes how the plugin handles this user request:
1. Get the user credentials from privateContent’s database. These credentials are used to later access the groups that the user belongs to. This is because FAM doesn’t base the authorization on the user itself, but the groups the user is assigned to.

2. When the user credentials are fetched the plugin uses these in order to access the groups that the user is assigned to by privateContent.

3. The FAM-database contains the relationships between user groups and the file types, dates and EMLP-categories the user group is authorized to access. These relationships are fetched with the user groups obtained in the previous step.

4. Since the user, in this case, wants to access one or many files, these files are known by WordPress. The meta-information is stored in, and obtained from, each file and checked against the relationships fetched in the previous step.

5. All files that pass the previous step’s authorization test are returned to WordPress. Since all the files that are returned to WordPress have been tested against the rules obtained from the FAM-database and the user itself the plugin will know that the user is authorized to view each of the files.

3.2 Database Layout

When designing the database many factors have to be considered in order for the plugin’s tables to be both simple and sturdy. Since a lot of the operations of the plugin involve privateContent and Enhanced Media Library PRO the database has to reflect both the internal workings of the plugin as well as the external connection to the other plugins. Since this is imperative for the plugin to work the design of the database was the first design part of the project and the foundation upon which the plugin was built around. Since WordPress uses MySQL and has both functions and objects involving operations with MySQL there is no need for the plugin to be able to handle any other database management systems. In order to design the database the understanding of privateContent and EMLP is crucial. Since the plugin needs to save all relationships between users as well as internal data the key for a good database design is simplicity and constantly avoiding to save data that is already stored elsewhere, data duplication. In order to know what data needs to be stored all steps of the user interaction with WordPress connected to the plugin will have to be addressed:

1. privateContent-data that the plugin will use:
   
   (a) The privateContent-group ID:s which the user is a part of.
   
   (b) The name privateContent has given the group. (This is only saved in order to make the database easier to understand and debug during implementation. In the final release of the plugin this can be omitted.)
2. Enhanced Media Library PRO-data the plugin will use:
   (a) The EMLP-categories the group is allowed to access.

3. The plugin’s internal data.
   (a) The group-to-file relationship.
   (b) The group-to-EMLP category relationship.
   (c) The file types of the files uploaded to WordPress’ media library.
   (d) The date and time span the user group can access a file.
   (e) If a user group can access all dates, i.e. no time restrictions, this needs to be stored.

Figure 9: An entity relationship diagram showing the File Authorization Manager’s database tables.

In Figure 9 the final design of the database is shown. In this diagram the following tables and relationships were added:

- *ab_fam_group*. This table is where the data connected to a group’s name, ID and its time-restrictions are stored.

- *ab_fam_file_type*. All file types that are uploaded to WordPress’ media library are stored here. This data can initially be accessed in existing tables already created by WordPress but here lies a problem. If a user
deletes all the files, the file type definitions associated with the files will also be deleted. This becomes a problem for File Authorization Manager due to the need to display for the admin which file types the users should be able to have access to. Due to these circumstances it was important to keep a separate table for file types for File Authorization Manager. This table will never contain more entries than there are different file types in the media library.

- *ab_fam_group_to_category*. This table is needed in order for the plugin to store the relationship between a user group and its authorized EMLP-categories. A primary key-pair is composited by the group ID and the EMLP-category ID. The group ID is a foreign key from the ab_fam_group table and the category ID is from an existing WordPress table called *term_taxonomy*.

- *ab_fam_group_to_file*. This table is like the previous table used to store a relationship but in this case between the user group ID and the file type ID’s it is authorized to access. The primary key is a composite of the group ID, which is from the ab_fam_group table, and the file ID, which is from the ab_fam_filetype table.

### 3.3 User Experience

#### 3.3.1 Graphical User Interface Solution

The first design that was taken into consideration was based on WordPress’ widget-functionality. From a user-perspective a widget is a small icon with drag-and-drop functionality that can be accessed from the Dashboard. If a user drags this icon into the web page it behaves as a standalone program within the web page. WordPress widgets offer customization and options such as forms to fill out, includes or excludes of data and information, optional images, and other customization features.[12]

In the first design draft it was thought that the use of such a widget would be a good implementation in regards to the authorization plugin. The design idea was to let the administrator insert a widget in the web page, and upon activation of said widget it would present the users with a browser-interface showing only the files the user was authorized to access.
As Figure 10 shows, the widget is placed inside a web page through WordPress’ Dashboard. This widget, when viewed from the web page, runs the PHP code which presents the user with a browser window in which the user can view and sort the files. The user can not view or access files that the user is not authorized by File Authorization Manager to access.

This idea proved, however, to be sub-optimal for SKB since it relied on a widget to show the authorized files. SKB instead needed to be able to prevent access to files across all of their intranet, not just only wherever a widget was present. Another problem with this idea was that it didn’t provide a solution for the hotlinking-problem SKB were facing. Since the widget didn’t alter the user-to-file-permissions in any way but only provided a visualization tool for administrators to present to the users it didn’t solve the hotlinking issues; any file could still be accessed by any user, logged in or not, by typing in the address for said file and therefore accessing it.

Due to these factors another solution had to be designed with these aspects in mind.
3.3.2 Invisible Server Side Solution

The next design iteration needed to take all aspects of user-to-file authorization, hot linking and being universally applicable across the whole system, not just a local solution as the previous design draft. Since all file requests go through the server the plugin needed to somehow access the server itself, fetch the request, manipulate the response and then let the server take care of the rest.

Figure 11: This is an illustration of the user-server dialogue. A user clicks on a link and therefore sends a request to the server. The server uses PHP and CSS to assemble the data into a web page and then sends this data, the response, back to the user.

In order to inject code into the server right in between the request is received and the response is sent back to the user the plugin needed to dynamically run code inside the server yet also take security and upgrade-ability into consideration. Since SKB’s external web is constantly being upgraded and updated it was of utmost importance that the alterations by the plugin was as minimal as possible. This is where the .htaccess file comes into action; the .htaccess file is a configuration file for use on web servers running the Apache Web Server software. When a .htaccess file is placed in a directory, which is in turn loaded via the Apache Web Server, then the .htaccess file is detected and executed by the Apache Web Server software. These .htaccess files can be used to alter the configuration of the Apache Web Server software to enable or disable functionality and features.

Since WordPress stores all files that have been uploaded to the web page in a certain folder and its subfolders, creating and manipulating a .htaccess file inside these folders can strictly prevent a user from accessing files in these folders which is exactly what the intent of the plugin is. Since we know all the user credentials through privateContent’s API we can, in real-time, fetch this information, check the user group’s authorization and decide whether we want to send the file to that user or not. In order to make a file accessible we make no changes to the .htaccess file and let the user access the file as they normally would. If a user shouldn’t have access, due to group settings not being in conjunction with the file to be accessed, we make changes to the .htaccess file which in turn locks the file for downloads. Since no user that doesn’t have the right group permissions will be able to access a file this design therefore solves both the user-to-file authorization as well as the hot-linking problems SKB are facing.
3.3.3 The Settings Page

Since all the plugin’s settings in regards to individual groups needed to be stored and accessed by the administrator, one week was allocated in order to make this interface as simple and effective as possible. The initial plan was to make each user have its settings stored individually but after meetings with SKB we decided to make a solution where all settings are based on groups instead of individual users. This direction made it easier to visualize a settings page and design it in a way that enabled the administrator to easily access and alter each group’s authorization settings. In order to be a fully functioning settings page the interface needed to include three different settings for each group:

1. The Enhanced Media Library PRO-categories the group is authorized to access.
2. The time span for a individual file which the group is authorized to access. Since the group might be authorized to access all dates this had to be accounted for as well.
3. The file types the group is authorized to access. This could be a jpg, pdf etc.

Since SKB might have any number of groups associated with the plugin each group were to be accessed through a drop down list in order to make the page feel less cluttered and therefore more effective. A downside to this approach is that the administrator can only see the settings for the group that is currently chosen. This might be a problem since one group’s authorization settings can be loosely based on another group’s authorization settings and therefore make it harder for the administrator to alter a group’s settings. Since both solutions had their upsides and downsides it was decided that the latter solution were to be used in the implementation stage, for its superior overview, and the former solution were to be used in the final implementation of the plugin. Since settings are only saved when the “save changes”-button is clicked there is nothing that prevents the administrator to browse between groups in order to get a overview and at the same time only see a single group’s settings and therefore keep the page uncluttered which is why the latter solution were chosen to be the final design.
Figure 12: The settings page where all the interaction between the plugin and the administrator takes place. Each “authorization-part” is separated in order to elucidate the work-flow of the administrator. Since the settings are only saved when the “Save settings”-button is clicked the administrator can flip between user groups in order to get an overview of the other groups’ authorization settings. All inputs are checkboxes besides the time span’s inputs which utilizes both checkboxes and drop-down calendars which are activated by clicking on the date. When a date is chosen the drop-down calendar disappears and instead a date-formatted text takes its place.
3.4 Replace a File

When the thesis-specification was outlined it was decided that the plugin should include functionality to replace a file. During talks with SKB this was later changed to make an additional plugin with replace file-functionality in order to keep each plugin work specifically with their intended functionality.

As can be seen in Figure 13 the user selects the file to be replaced in the first box, and the file to be replaced with in the second box. All metadata can be fetched from the WordPress database table `wp_posts` and the new file should inherit all the metadata of the old file. The easiest and most effective way to approach this is to simply replace the address of the old file, what WordPress calls the GUID, with the address of the new file. These operations are described in the following steps:

1. Check whether the new file and the old file are of the same filetype. If they are continue to step 2. If not let the user know this by showing an error-message.
2. Save new file in the same folder as the old file.
3. Fetch the post entry of the old file. This is found in the table `wp_posts`.
4. If this post entry is an attachment, i.e. this is an uploaded file, change its GUID to the new file type’s address. If the old file is not an attachment let the user know that the file is not an uploaded file and can therefore not be found.

Figure 13: A design overview of the interface with the replace file-functionality.
5. Let the user know that the files were replaced.

Since the replace file-plugin is a standalone plugin it also incorporates all internal requirements specified in section 2.2.
4 Implementation

The implementation of a software is turning the system specification into an executable system.\[15\] The implementation stage is therefore the continuation of the design stage and where the blueprint turns into executable code.

4.1 Framework

When discussing the framework in this thesis it encompasses the entire system provided by WordPress. This in turn includes WordPress itself and its database, Enhanced Media Library PRO, privateContent, Apache and all other sub-systems provided by the mentioned parts.

4.1.1 FAM and WordPress

In order to inject code into WordPress, File Authorization Manager needed to take advantage of numerous action hooks, both WordPress' own internal hooks as well as privateContent and Enhanced Media Library PRO's. The implementation of the latter action hooks and in what way FAM interconnects with privateContent and Enhanced Media Library PRO is discussed further in Section 4.2. When it comes to FAM's interaction with WordPress a multitude of action hooks are used and the following list depicts the most important action hooks and how FAM takes advantage of them:

1. \texttt{register\_activation\_hook}. This hook is run when the user activates the plugin in the WordPress Dashboard. Since this is the first time the plugin is run by WordPress this is where the database and its dependencies are created. A check is made to make sure that this is the first time the plugin has been activated. If it is not the first activation of the plugin the database is not created or altered. A check is also made in order to make sure that privateContent is installed and activated. Since FAM relies so heavily on privateContent and its functionality FAM is only activated if privateContent is activated. If privateContent is not activated an error message is sent to the the user telling it to first install privateContent and then try to activate FAM. Upon activation a line is also added to the .htaccess file and this is further covered in Section 4.2.3.

2. \texttt{register\_deactivation\_hook}. This is, as the name implies, a hook that is run after a user deactivates the plugin. Since the database should be intact if the user decides to activate the plugin again, it is left unaltered. The line added to .htaccess is now removed from the file and this is further explained in Section 4.2.3.

3. \texttt{add\_action('admin\_menu', 'ab\_fam\_create\_menu')}. Since the user needs to be able to change settings for the plugin a menu button as well as a settings page needs to be inserted in the Dashboard. WordPress provides a hook for this which is triggered when the Dashboard is loaded and it is
this hook that the plugin uses in order to provide the user with the afore-
mentioned interface. When the action hook 'admin_menu' is run, the
FAM-function 'ab_fam_create_menu' is run which in turn calls numer-
ous other functions. These other functions creates the html, css, javascript
and php code needed to create a settings page and a button for the Word-
Press Dashboard which is used by the user in order to access this settings
page.

4. \texttt{add\_filter('wp\_handle\_upload', 'ab\_fam\_update\_filetypes\_on\_upload')}.
This action filter is triggered when a file is uploaded to the uploads folder.
Since the plugin needs to have all the file extensions saved in its database,
each time a user uploads a file this file extension is checked by the func-
tion 'ab_fam_update_filetypes_on_upload' and if it doesn’t exist in
the database it is added. If it already exists nothing gets added to the
database.
Figure 14: The figure shows the settings page and the menu buttons added to the Dashboard. This version of the settings page only shows the implementation stage design, i.e. each user category is shown and accessed by scrolling down the page. In the release version a user would choose a user category in a drop down list and access each user category individually.

4.1.2 Activation and Deactivation

The activation and deactivation of the plugin are managed by the hooks mentioned in the previous section. Since the only part of the plugin that remains active is the database all other functionality the plugin provides is deactivated.
This includes the settings page, the dashboard menu items and the lines added to the .htaccess file. The database is left unaltered since all data must be saved for later use. When it comes to the uninstall functionality the plugin deletes all its data including the database. This part is managed by the uninstall.php file located in the same folder as the other plugin files. When the uninstall button in the Dashboard is clicked by the user the uninstall.php file is automatically executed by WordPress.

4.1.3 The Database

When it comes to creating databases for WordPress the system provides a global object variable, called wpdb, which the programmer can use to access, manipulate and save databases. This object is used throughout the entirety of File Authorization Manager’s implementation, from fetching table prefixes to inserting data, since it provides the plugin with a safe and easy way to access the database by, for example, automatically escaping strings in queries and providing advanced queries in easy-to-use methods.

4.1.4 Frontend to Backend

Most of the work that is done by File Authorization Manager is invisible to the user but there is one exception, namely the settings page. When implementing the settings page the connection between the front end interface and the backend database is crucial to the functionality of the plugin. The following steps are executed in the file ab-fam-settings-form.php when a user saves their settings:

1. Each group’s settings, which can be seen in Figure 14, are comprised of a html form which upon clicking the button “Send” is submitted.

2. In order to make sure that each form is handled upon submittal, remember that there can be an unlimited amount of groups, there is a jQuery-function that executes when any form is being submitted. This function checks which of the forms that have been submitted by fetching the form name, which is the same as the group ID, and then pushing the data saved in the form to arrays representing the categories, time span and file types.

3. These arrays are pushed to a single array in order to be sent through Ajax.

4. An Ajax-call is created that contains:

   (a) A URL to the back-end code where the data will be handled, in this case the file ab-fam-settings-form-submit.php.

   (b) The type of the Ajax-request, in this case “POST”.

   (c) The data to be sent. In this case the array created in step 3.

   (d) The data type of the data to be sent. In this case “text”.
5. The file ab-fam-settings-form-submit.php receives the Ajax-call and processes the data. The database is updated with the new data that has been sent by the settings page.

6. If the data has been received and sent to the database successfully a success-message is sent back to the file ab-fam-settings-form.php. If some connection fails the message will instead be an error message.

7. Ab-fam-settings-form.php receives the message and outputs it to the user by including the message in a result div that each form, and by that each group, has.

4.1.5 Upgradeability
In order for File Authorization Manager to be upgradeable the implementation of the plugin had to address this factor. Since the plugin relies on data provided by privateContent and Enhanced Media Library PRO the functions that import this data are, throughout the plugin, as unilateral as possible. Since these functions are unilateral any changes in the data provided by privateContent or Enhanced Media Library PRO, due to upgrades, can easily be altered in accordance to the changes. Since File Authorization Manager also relies on its own data, for instance in databases and global variables, these parts are also divided into small and easily maintainable blocks of code. In the first implementation of the database all file types were stored as, for example, “.jpg” or “.png” but since WordPress uses its own syntax for file types, the plugin instead was altered to reflect WordPress’ own file structures, in this example “image/jpg” or “image/png”. This change was made in order to only rely on WordPress’ already stored file types allowing the plugin to dynamically adapt to upgrades for WordPress itself. The file type table can seem to be redundant but since WordPress allows for user to delete posts these posts will therefore be deleted in the WordPress database and, as a result, the file types of the deleted post as well. This is not optimal for File Authorization Manager since the plugin needs remember which file types a certain user group is allowed to access and therefore File Authorization Manager needed its own table for file types.

4.2 Interconnectivity

4.2.1 privateContent
Since privateContent, the authentication tool, is so crucial to the workings of File Authorization Manager, the authorization tool, the first time privateContent is called is in the activation of File Authorization Manager. The hook register_activation_hook is called, in order to activate the plugin, with the function ab_fam_on_activate which executes the code needed for activation, like installing databases and fetching variables to check whether the plugin has been installed before. Before the database and all other functionality of the plugin are executed, the plugin checks whether privateContent is activated which
If privateContent is activated File Authorization Manager continues to install all its activation code but if privateContent is not activated the plugin instead aborts the activation and sends a notice to the admin, with the hook admin_notices, that File Authorization Manager couldn’t be activated due to privateContent not being active.

```php
function ab_fam_on_activate()
{
    if (!is_plugin_active('private-content/private_content.php')) {
        // not active - shut down plugin activation and show an admin notice
        function no_plugin_warning()
        {
            echo '<div class="error">Please activate PrivateContent plugin to use this plugin</div>';
        }
        add_action('admin_notices', 'no_plugin_warning');
    } else {
        //active - run all activation code
    }
}
```

Figure 15: The activation of File Authorization Manager

Most of the other functionality connected to privateContent can be found in the code connected to the database. Since privateContent doesn’t provide any hook for when a user group is added, updated or removed this check is done by File Authorization Manager against the WordPress database tables `wp_terms` and `wp_term_taxonomy` each time a user accesses the WordPress Dashboard. This is a suboptimal solution but until privateContent can provide developers with a full API covering these issues there is no way of dynamically updating the database with privateContent’s user groups. The privateContent API is, however, used whenever File Authorization Manager checks which user is logged in and what groups that user is connected to. The check to see which user is logged in is done through the function `pc_user_logged`. This function returns an ID, it can return any meta information but the only one used by the plugin is the ID, which in turn is used as input for the global object pc_user’s method `get_user_fields`. File Authorization Manager uses this method to get the user groups connected to the currently logged in user and when this data has been returned the “connection” with privateContent is shut-down, further privateContent data is no longer needed until the next time a user wants to access a file, and all later authorization-based calculations are done internally in File Authorization Manager.
4.2.2 Enhanced Media Library Pro

Since Enhanced Media Library Pro doesn’t provide any hooks or other API functions all connections to EMLP and File Authorization Manager is made through the WordPress database. Like privateContent, EMLP uses the tables `wp_terms` and `wp_term_taxonomy` to store the file upload categories, which File Authorization Manager needs in order to connect each user group to the file category, or categories, it is associated with. Just like in the case with privateContent, a fully covered API would make File Authorization Manager’s code both simpler and easier to maintain but since this API is not implemented the only way to access Enhanced Media Library PRO’s categories is through the database.

4.2.3 .htaccess

In order for File Authorization Manager to grant or deny access to a file for a user a connection to the very core of the server was needed and it is here the very foundation of the implementation of the plugin enters the stage; the .htaccess-file. When the decision of using .htaccess was made two major questions arose: Where should the changes to the .htaccess be located? When should these changes be activated or deactivated? Since the denying of a file should be immediate from when a user activates the plugin, and the granting immediately after the deactivation of the plugin, it became obvious that the changes made to the .htaccess-file needed to be executed in the, respectively, `ab_fam_on_activate` and `ab_fam_on_deactivate` functions. In order for the implementation to be lucid and easy to grasp the coding was focused on two parts.

4.2.3.1 The changes made to the .htaccess-file

In order for the code responsible for activating and deactivating the interceptive behaviour of the plugin to write text to the .htaccess file numerous functions were needed. The plugin needed to write a line to .htaccess upon activation, remove that line and replace it with a placeholder upon deactivation and remove all lines written by the plugin in case the user removed the plugin altogether from the WordPress installation.

The first function that involves .htaccess is `ab_fam_initiate_line_in_htaccess` which is pictured in Figure 4.3. In order for the plugin to initially write a line to .htaccess this function is needed to traverse the text in .htaccess, find the right line and insert a text, or rather a rewrite rule in text format, that can easily be interchanged when activating and deactivating the plugin. The following steps describes the workings of the function:

1. The variable `newMessage` is set to a string. The rewrite rule is further explained in the below section.

2. The variable `htaccesspath` is set to WordPress’ home path appended with the name of the file.
3. The algorithm iterates over each of the lines in .htaccess and inserts newMessage before the last line, in this case marked by the string “If-Module”.

4. In the last step it overwrites .htaccess with the new text structure and saves the changes.

The .htaccess file is now updated with a rewrite rule that tells the server that whenever a file located somewhere in “wp-content/uploads/” is requested the server shall redirect that request to “/index.php?get_file=$1”. The functionality of .htaccess is now complete and will work for anyone trying to access a file, whether this user is logged in, logged out, authorized to view the file or not. The mechanics to allow or disallow the user to view the file is further explained in Section 4.2.3.2.

If the administrator of the website decides to deactivate File Authorization Manager some functionality for this scenario must be implemented. We have gone through the changes made in the other parts of File Authorization Manager upon deactivation but the rewrite rule added by the plugin to .htaccess must also be able to be deactivated. This is made by removing the rewrite rule and instead adding a placeholder that serves as a marker for future activation.
and is made by the function ab_fam_remove_line_from_htaccess pictured in Figure 17. The following steps describe how this operation is made:

1. The variable \texttt{oldMessage} is set to the string containing the rewrite rule that currently exists in .htaccess.
2. \texttt{newMessage} is set to a placeholder line starting with a “#” indicating that it is to be considered by the server as a comment. This string only serves as a placeholder and doesn’t add any functionality to .htaccess.
3. The function \texttt{str_replace} is used in order to delete all occurrences of the rewrite rule and, in that place, add the placeholder message.

Additionally there is a function that reverses the operations made in \texttt{ab_fam_remove_line_from_htaccess}, described in Figure 17, and is executed when File Authorization Manager is activated.

\subsection{The interception of a file request and the authorization involved}

When the server has received the file request it re-routes the request to the address “/index.php?get_file=$1” which tells WordPress to fetch the metadata and contents of the file requested. Since WordPress now has all the information of the file, derived from the database table \texttt{wp_posts}, this information is now available to File Authorization Manager. The interception of the file request is executed in the functions \texttt{ab_fam_add_get_file_query_var} and \texttt{ab_fam_intercept_file_request}. The former function latches onto the filter “query_vars”, which injects the query, in this case “get_file=$1”, and processes this request. The latter function is the major workhorse and is responsible for fetching the metadata associated with the file, assessing the user’s privileges, checking whether the user is allowed to access the file and finally allowing or disallowing the user to access the file. The execution of the function \texttt{ab_fam_intercept_file_request} is explained in the following steps:

1. Firstly the function checks whether the query variable is of the form “get_file”. If it is the execution of the function continues and if it is not the function is terminated.
2. The metadata of the file is fetched from the wp_posts database table.
3. The function determines, through sub-functions, whether the user is permitted to access the file or not. If the user is not allowed to access the file the function exits with error code “404”. If the user is allowed to access the file the function appends the metadata into the header and echos back the file path to WordPress, which is how WordPress shows a file to a user.

The function itself is latched onto the action “wp” which is executed when the object \texttt{wp}, which in turn is used for high-level filtering and validation \cite{17}, is
set up. The sub-functions called by *ab_fam_intercept_file* are all used to check the file’s data against the user’s permissions. Each type of the authorization; the file’s date and time, the Enhanced Media Library PRO categories the file belongs to and the file type of the file are represented by the sub-functions called when intercepting the file and they all check their respective part of the authorization.
5 Evaluation

When evaluating a project like the one covered in this thesis many aspects must be covered: Does the plugin work as intended? Are all parts of the specification met? Is the plugin secure? Did the allocated time for each part of the project conform with the time it actually took?

The short answer to the first question is no, the plugin doesn’t work as intended. The following section covers why the plugin isn’t feature complete and working as intended and also evaluates each individual part of the project.

5.1 The Rewrite Rules Conundrum

The sole reason why the plugin doesn’t work as intended lies in the way WordPress handles its rewrite rules. When outlying the time budget and the design document for the project it was based on the principle that WordPress stores its rewrite rules in the .htaccess file and by altering this .htaccess file the plugin could re-direct users through PHP code created by the plugin and by that authorizing users for access to files. This, however, proved to be a much too simplistic view of how WordPress handles its .htaccess rules. Instead of only relying on saving rules in the .htaccess file, WordPress uses an intricate way of storing and accessing its rules by additionally having an object representing the rewrite rules. Furthermore some of the rewrite rules are stored in the database table wp_options under the option name rewrite_rules. One could think that it should be clear to someone implementing a plugin that this is the way WordPress handles its rewrite rules but a combination of issues makes this somewhat difficult for a plugin developer; the database is very big and quite difficult to understand due to minimal documentation in the WordPress manual regarding the database implementation, the object used for storing rewrite rules is covered very briefly in the manual and most importantly, it is never explained when and how these three ways of storing and accessing rewrite rules are used. Due to these issues the plugin cannot redirect a file request to the PHP code responsible for the authorization. Since the revelation of WordPress’ intricate rewrite rule handling came late in the implementation the time frame didn’t allow for a thorough walkthrough that could have brought a solution to light. Instead the remainder of the time left of the thesis was focused on tying loose ends together and making sure the parts implemented were of high code quality and that each compartment of the code was up to standard. By making sure that each part of the code worked, a future update to WordPress enabling easier access to rewrite rules would, by a minimal implementation, make the plugin work as intended. Another approach of solving this issue would be to go through each of WordPress’ rewrite rules and mapping out when and why they are stored where they are stored and by this understanding be able to find a pattern which to use for the plugin. These possible future implementations are covered in Section 6.1.
5.2 The Specified Parts

Following below is each parameter, defined in the specification of the thesis, and with them their evaluation:

1. The plugin should base its authorization on a file’s Enhanced Media Library PRO categories, file type and the date which the file was uploaded. The plugin should also work with existing privateContent groups.

   The plugin uses privateContent and Enhanced Media library PRO extensively throughout the implementation and this behaviour is further explained in section 4.2. The implementation of said parts initially proved to be quite difficult to implement due to the lack of some API-functions, in regard to privateContent, and lack of an API altogether, in regard to Enhanced Media Library PRO. Even though this made the implementation more complex it didn’t affect the project’s time table since the interconnectivity-part of the project was allocated sufficient time.

2. The plugin should prevent hot-linking.

   Since the thesis focused on implementing authorization in the .htaccess-file this also solved the hotlinking issue. This is further discussed in section 4.2.3. Due to the nature of the .htaccess-file and how it prevents the user from downloading a file via its rewrite rules, the hotlink prevention is not made explicitly and therefore will not be evaluated in this thesis.

3. The plugin should add functionality to replace a file with unchanged metadata.

   In talks with SKB it was decided that the replace file-part of the plugin should be a separate plugin due to it being a different functionality altogether from the original plugin and therefore it didn’t make sense to have a replace-file functionality in an authorization plugin. Sadly this plugin was never implemented due to the original plugin, upon which the focus laid on, never being finished. This is further discussed in section 3.4.

4. The plugin should be platform independent.

   In the thesis specification SKB clearly stated that the plugin should be able to work cross-platform, both in terms of operating systems and servers. As of writing this thesis the plugin works with any operating system that supports a web browser that in turn supports WordPress. When it comes to server platform independence however the plugin is only implemented to work with Apache due to possible differences in rewrite rules and regular expressions. The decision to work with Apache was made together with SKB since Apache is the only platform they currently use. In a future implementation this could be handled by checking which platform is in use and then execute code specifically for that platform. This is further discussed in section 6.2.
5.3 Is the Plugin Secure?

When evaluating the security of this plugin one must determine where a security breach can occur. Since this plugin doesn’t manage users or user’s passwords directly but instead uses privateContent’s API to check whether a user is logged in and upon this check determines whether downloads should be possible a lot of the possible security issues have been addressed. However since the plugin also uses database interactions this also needed to be addressed. In order for the plugin to use safe database interactions, for instance escaped special characters, it throughout the code uses the WordPress function `prepare`, sometimes explicitly and other times implicitly by using other WordPress functions that internally use `prepare`. By using `prepare` we can be sure that the plugin escapes special characters but the other interactions by the user are equally important to address. This is taken care of by only allowing the user to select values with pre-defined boundaries such as check-boxes and drop-down calendars which only accepts values in certain domains which in turn enables the plugin to only execute database code that is “reasonable”. All data sent by the user is validated server-side to make sure that it only affects the File Authorization Manager’s data and also that the data itself corresponds to the expectations. Due to the plugin being an authorization plugin only it does not control the authentication token nor does it validate it. This is a potential threat to SKB and the company must ensure that the framework or plugin used for user groups and authentication ensures safe handling of the token.

In addition to the mentioned security approaches the plugin only uses user data internally and never outputs them in any form or way. Since text injections to the .htaccess-file are also prone to security breaches the only text injections that are executed are predefined and hardcoded since this ensures the plugin to never dynamically write to the file which in turn would be hazardous. Another important question that arises is whether the administrator can be trusted? This security concern is important to address for SKB and make sure that the authentication framework used is taking this into consideration. File Authorization Manager assumes that the administrator is an authenticated administrator and also authorized according to SKB’s internal authorization guidelines.

Due to the time scope of this thesis a thorough threat model was not implemented. In addition to plugin covered by this thesis, SKB uses WordPress, the framework itself, and two additional plugins to handle file categorization and user group authentication. Due to this complexity a threat model would have taken a considerable amount of time to plan and execute. It is important however to mention threat modeling and how the plugin would benefit by having such a plan in place, even before the actual implementation of the plugin. The Open Web Application Security Project [18] lays out three steps that should be followed when implementing a threat model for an application:

1. De-construct the application into parts in order to gain an understanding of the application and how it interacts with external entities.

2. Categorize and rank the threats the application is vulnerable to. This
should be done both from the attacker’s perspective as well as the defender’s perspective.

3. Determine what countermeasures should be implemented.

As stated earlier, this threat model approach was not implemented in this thesis, but it is easy to see how an application like File Authorization Manager would benefit from taking threat modeling into account. In this thesis a threat model covering the plugin itself could have been implemented in the allocated time but it would have been difficult to find and address security concerns regarding the external entities due to lacking documentation regarding security provided by privateContent and WordPress. This is not an excuse however for not implementing a threat model for File Authorization Manager and as stated earlier: File Authorization Manager should have taken threat modeling into account in the design phase and before the actual implementation.

5.4 Time Allocation

When planning time allocation for software development it is always tricky to know how long time each segment needs. In the case of working with a platform and code language, WordPress and PHP, without previous experience it will always come down to certain elements of guesswork. In this thesis the major allocated time therefore went into reading up on the literature, manuals and forums to get a good grip on how to develop code for an authorization plugin. Due to the reading bit taking up such a large part of the planning the coding was allocated a much smaller part, also taking into consideration that report writing were to be done simultaneously as the coding.

In retrospect the planning proved to be good and not much would change despite that different paths were chosen during the implementation, for example going with the .htaccess idea instead of the original idea. The reading material initially chosen would however change some due to the changes made in the implementation. This would however have been impossible to know before the actual changes were made during the implementation.

Due to the plugin not being finished as of writing this report it could also be said that the allocated time for the project was not enough and that certain aspects of the implementation could have been left out. An example of this is the additional plugin that was planned to be made, the file replacement plugin, since it would have been difficult to find time for it, even if the .htaccess-part of the implementation could have been finished in the allocated time. In retrospect it seems that dividing the thesis into a two person project would have been more manageable since one person could have focused on solving the .htaccess-issues while the other was working on other parts of the implementation simultaneously.
6 Conclusions

From the outset it was very clear what File Authorization Manager was supposed to do; authorize users for file access. However, when discussing how this authorization were to be made it was not as clear how the plugin should be implemented. This required the allocated time of the project to lean very heavy towards reading and understanding the problem in order to find a solution. The solution turned out to be based on authorizing users using the .htaccess file as gate keeper. This solution, however, was outlined by reading literature and gathering other information that didn’t necessarily correspond with how WordPress accessed and manipulated the .htaccess file. Due to the somewhat lackluster information on WordPress’ part regarding the rewrite rules in addition to the literature and other gathered information not taking WordPress into account, this caused the solution to the problem to be difficult to grasp.

In hindsight the idea to first implement the framework, consisting of GUI, database interactions etc., was perhaps not the best approach. Instead the main focus should have been, from the start, to work out a solution for the authorization itself and later set up a framework and implement a fully functional plugin. On the other hand this approach would have made it harder to debug the code due to a lacking framework in which to test it and perhaps most importantly: How could one initially have known that WordPress’ management of rewrite rules would interfere with code written to .htaccess file to such an extent? It could be debated that all information needed to understand the rewrite rules can be accessed in the WordPress manual. It could even be debated that manipulating the .htaccess file is not best practice at all due to the security concerns aired in section 5.3.

To address the first concern one must realise that to understand the rewrite rules one must also understand WordPress as a whole. Understanding WordPress as a whole on the other hand takes time, perhaps more time than the thesis scope can provide, which in turn means that a developer has to move, quite rapidly, from a meta perspective to a micro perspective. In this case “zooming in” on the authorization problem meant that other parts of the WordPress platform were left out of the information gathering. This in turn made implementing the framework more time consuming and, as a result, less time was allocated towards the more intricate parts of the plugin implementation.

In address to the second concern the approach to write text directly into the .htaccess file was decided upon together with SKB. This approach was determined to be secure enough and since the text was hard coded it would be very difficult for a user to use this feature for malicious intent.

6.1 Further Work

As has been stated previously in this thesis the plugin is not feature complete. In order to make the plugin work as intended more focus needs to be laid on understanding how WordPress handles its rewrite rules which in turn would solve the problems that keep the plugin from being feature complete.
The GUI as it is implemented right now is very bare bones and it could need an overhaul to make it look more in line with WordPress’ settings page. This is not something that was focused on during the implementation but with a little work it could be made to look more professional and inviting to a user.

In order for the plugin to have full upgradeability and be future proof, to the extent that is possible, phasing out privateContent would be a time consuming task but not without its benefits. As of now the plugin relies heavily upon privateContent and in a scenario where privateContent changes dramatically, or perhaps is even dropped from the WordPress plugin catalogue, the plugin could really benefit from being both an authorization and an authentication tool. As stated this would require a complete rework, but if such an authentication plugin would be developed, either as a standalone plugin or as an addition to File Authorization Manager, the rewards would be substantial. SKB would have complete control over the entire authentication-authorization process and in addition, if a standalone approach to the authorization plugin were to be chosen, the impacts to File Authorization Manager would be minimal since the authorization plugin could use an API similar to privateContent’s. This approach could also simplify the database since several tables could be shared by the two plugins.

6.2 A Final Word

As a closing statement this thesis has provided SKB with information regarding the difficulties to work with WordPress and how difficult it is to bend the rules when it comes to manipulating core features of WordPress. This plugin is not impossible to implement, not at all, but the time needed for such a plugin that this thesis covers might be greater than the time that was provided. Allocating more time into understanding the core workings of WordPress and its rewrite rules would likely reap the reward of a functional authorization plugin.
References


