Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eiffel ROC CMS: File Uploader

Software Engineering Laboratory: Open-Source EiffelStudio

Fabian Murer

supervised by
Jocelyn Fiat

February 5, 2016

Contents

1 Summary

This report is about a students project which was done in a Software Engineering Laboratory Course at ETH Zurich. The goal of this project was to extend the *Eiffel ROC CMS*[1] with a module to easily upload files.
The students project has been very well assisted by Jocelyn Fiat from Eiffel Software[2].

2 Introduction

The *Eiffel ROC CMS* is currently a library, based on the *Eiffel Web Framework*[3], with which one can built its own content management system (CMS). It is still under development and there are constantly new modules or tools added.

Prior to this project, there have already been some possibilities to upload files. It could have been done by simply create a HTML POST request. Doing this wherever the user needs to upload files can be very troublesome. The goal of this project was to provide a simple solution to that issue by creating a module. The module can easily be included by two lines of code. It provides several simple functionalities, such as choosing one or more files and upload them. It also lists all of the already uploaded files with some additional information such as the time, when it has been uploaded or the name of the user who has uploaded it.

3 Project Task

The Project Task was to build a simple solution to upload and store files on the CMS. It had to support at least the most required functionalities for an uploading tool, namely

1. Choose and upload files from the current filesystem,
2. Store the uploaded files on the CMS environment,
3. List the files that have been uploaded,
4. Display some additional information such as uploading time, size of the file or the user who has uploaded the file,
5. Remove a file.

[1] https://github.com/EiffelWebFramework/ROC
[2] https://www.eiffel.com
[3] https://github.com/EiffelWebFramework/EWF

4 Implementation

The module has mapped some commands to certain routes of the webpage. This means, when the user is on a specific page, the corresponding command is executed. The *File Uploader Module* has three of such mappings, shown in Figure 1.

```
setup_router (a_router: WSF_ROUTER; a_api: CMS_API)
        -- <Precursor>
    do
        map_uri_template_agent (a_router, "/upload/", agent execute_upload (?, ?, a_api), Void) -- Accepts any method GET, HEAD, POST, PUT, DELETE, ...
        map_uri_template_agent (a_router, "/upload/{filename}", agent display_uploaded_file_info (?, ?, a_api), a_router.methods_get)
        map_uri_template_agent (a_router, "/upload/remove/{filename}", agent remove (?, ?, a_api), a_router.methods_get)
    end
```

Figure 1: Map commands to certain routes of the webpage.

For example, always when the user comes on the page *"/upload/"*, the command `execute_upload` is executed. This is the main feature, which then itself calls some other features.

4.1 Upload Files

Uploading is done by an Open-Source library called DropzoneJS[4], which provides a *drag-and-drop tool* to easily choose files. The tool is based on a single javascript-file, which actually does a simple HTML POST request.

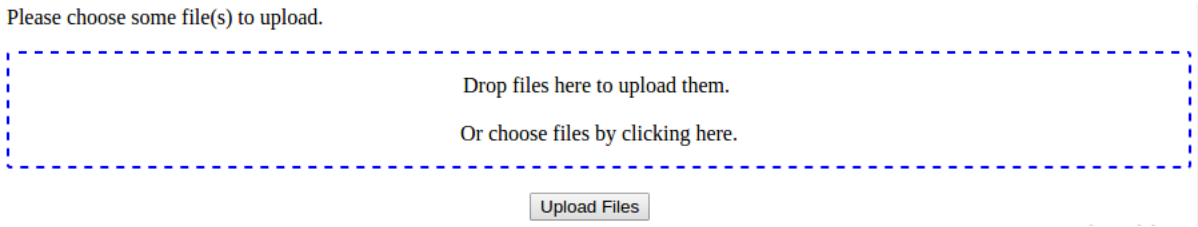


Figure 2: Simple drag-and-drop tool by DropzoneJS

The tool basically allows the user to either choose the files per drag and drop or as usually by browsing through the file system of the users computer.

4.2 Store Files

The *Eiffel Web Framework Library* already has some classes and features to store files. `WSF_REQUEST.uploaded_files`[5] contains the recently uploaded files and with the `WSF_UPLOADED_FILE.move_to` command, one can store a file in a predefined folder on the CMS environment.
For each uploaded file, another file with a special extension (filename.cms-metadata) is

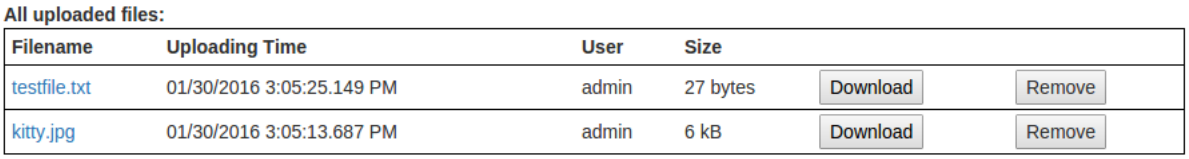
[4] see http://www.dropzonejs.com

[5] to see more about how requests are handled in the Eiffel Web Framework, check https://github.com/EiffelWebFramework/ewf_examples/blob/master/workbook/workbook.md

created and stored in a hidden folder. These files contain metadata of the uploaded files, such as the date and time the file has been uploaded, its size and the name of the user who has uploaded it.

4.3 List the Files

The uploaded files are then listed in a table with its metadata. This is done by an iteration through the directory where the files are stored. The corresponding metadata is read out of the hidden files and also shown in the table.
To read data out of a file the class `RAW_FILE` (inheriting from class `FILE`) from the *ELKS*-library is used. This class provides some useful commands and queries such as `read_line` or `last_string`.


All uploaded files:

Filename	Uploading Time	User	Size		
testfile.txt	01/30/2016 3:05:25.149 PM	admin	27 bytes	Download	Remove
kitty.jpg	01/30/2016 3:05:13.687 PM	admin	6 kB	Download	Remove

Figure 3: Example of how the uploaded files are listed.

4.4 Details of a file

Each file also has an own page, on which the user can check the details again. Additionally to the metadata listed before, the user can see the type of the file as well as a little preview of the file. This preview shows either a small image if the file itself is a image, or a little default thumbnail.

4.5 Remove a File

Removing a file is done by a simple click on a button. By clicking it, the user gets redirected onto a new page *"/upload/remove/{filename}"*. As you may have seen at the beginning of Section 4, a command `remove` is mapped to this route.
With the query `WSF_REQUEST.path_parameter` it is possible to read out the filename which is attached to the route. Similarly to create a file (as it was done when creating the metafiles), the class `RAW_FILE` with its command `RAW_FILE.delete` is used to remove the file with the filename from the path-parameter. Additionally to the file, the corresponding metadata-file is also deleted.

5 Using the Module

To use this module, the user needs to do a few simple steps:

1. Include the module in the execution class of the CMS (see Figure 4).
2. Create a new folder in the CMS directory. This is the folder where the files will be stored. Within this folder, the user also needs to create a hidden folder for the meta-data and another one for the preview images. In the class `CMS_FILE_UPLOADER_API`, the fields for the folder-paths need to be set accordingly.

```
    -- uploader
create {CMS_FILE_UPLOADER_MODULE} m.make
a_setup.register_module (m)
```

Figure 4: Lines to include the module.

6 Functionalities for the future

Unfortunately, there are still some useful functions missing.
At the moment, the module can not manage directories. This means, all the uploaded files have to be in one folder and can not be ordered by putting them into separate folders. The details page of a file is still a bit empty. It might be more useful, when users could comment on some files, rate them or even do some editing. For a better overview, one could add some more specific thumbnails for different types of files.

7 Conclusion

To conclude this report, it was first hard to get into the existing CMS system and get familiar with all the classes and features (also from the *Eiffel Web Framework*). So it took me a long time to discover the right classes needed to create the module and its features. The documentation[6] of the CMS helped me a lot there.
Furthermore, a lot of time has been spent on managing to run the CMS on a virtual machine, which was sometimes a bit frustrating. However, this probably leads back to a mistake on my side, since I once had an older version of EiffelStudio installed, which did not support some features from the newer EWF library or vice versa. I had also some troubles to set up the environment variables correctly. The Eiffel installation description[7] helped there.

[6] see https://github.com/EiffelWebFramework/ROC/tree/master/doc
[7] see https://docs.eiffel.com/book/eiffelstudio/eiffelstudio-linux#Setting_up_EiffelStudio

In the end, it has been a very good project! I have learned a lot, especially how it is to work on a bigger project and how a content management system works in the very backend. Last but not least, I have learned how to write a technical report like this!
Finally, I would like to thank Jocelyn Fiat, who has helped me a lot during this project.