



25 August 2023

adriana.baldacchino.20@um.edu.mt

Integrating Overleaf in CERNBox

Adriana Baldacchino

IT-SD-GSS Summer Student 2023

Supervised by: Javier Ferrer, Giuseppe Lo Presti

Summary

This document illustrates the process of integrating support for Overleaf in CERNBox as a project for CERN's 2023 Summer Student Programme. The final implementation considers two actions: export to Overleaf, which pushes CERNBox data to Overleaf; and import from Overleaf, which retrieves and assimilates data received from Overleaf. The export action is implemented using a public API provided by Overleaf, while the import action proves more difficult, as it requires user authentication. Therefore, we implement a proof of concept which can be realised once a secure way of obtaining user credentials is available. This project also includes the user interface, and hence front end functionality is also implemented and explained in detail in this document.

Throughout this report, we define the problem proposed, any pitfalls encountered, solutions considered and the final implementation details. This will serve to both document thoroughly the Overleaf service, and as a window into my experience and work at CERN.

Contents

1	Introduction	3
2	Similar Work	3
3	Problem Specification	3
4	Design	4
4.1	Export functionality	4
4.2	Import functionality – A proof of concept	5
4.3	Structure of our implementation	7

4.3.1	Initial approach	7
4.3.2	User interface	8
4.3.3	Overleaf as a service	9
5	Problems encountered	9
5.1	Solved issues	9
5.1.1	Archiver modifications	9
5.1.2	Exposing token in project name	10
5.1.3	Encoding name as an extended attribute	10
5.1.4	Temporary rebase for CERNBox	10
5.2	Unsolved issues	11
5.2.1	Filtering out LaTeX projects	11
6	Technical details	12
6.1	Overleaf APIs	12
6.1.1	Export to Overleaf	12
6.1.2	Import to Overleaf	12
6.2	Front end configuration	12
6.3	Overleaf Service Configuration	13
6.3.1	Adding Overleaf service to Gateway	13
6.3.2	Adding Overleaf service to nginx	13
6.4	Git structure	13
7	Future work	14
7.1	Migration to a CERNBox Extension	14
7.2	Implementation of organizational sign on for Overleaf	14
7.2.1	Overwriting exported projects	15
7.2.2	Continuous integration	15

1 Introduction

CERN's collaboration hub, CERNBox, is utilised by more than 37k¹ users to store data and collaborate. Users benefit from many features, such as data synchronisation, backup, sharing and integration with various apps such as Office, Markdown and Draw.io. The app integration feature allows users to edit documents more easily and comfortably, which incentivizes users to use CERNBox as their primary storage application.

Currently, no such implementation exists for any editors for the LaTeX language, which is widely used in the scientific community to write technical documents and articles due to its facilitation of writing complex mathematical equations and precise formatting. This would be a useful feature for the CERN community, given most academic papers and reports are written in LaTeX.

There are many editors available for writing in LaTeX, however one of the most widely used is Overleaf, due to its friendly user interface and collaboration features. Moreover, CERN provides premium Overleaf features to all its staff, making it an ideal candidate to integrate with CERNBox. Hence, this summer student project is dedicated to providing functionality to sync CERNBox contents with Overleaf.

2 Similar Work

Similar work was done by Daniel Müller on integrating Overleaf with ownCloud². This integration follows the typical app provider pattern: it opens a local version of Overleaf inside ownCloud, which allows users to edit Overleaf files without leaving the site. This integration works by creating an Overleaf account for each ownCloud user and hence escapes any authentication issues. However, this would not be the best approach for us as this would force us to use the free version of Overleaf. Since CERNBox users are also CERN users, they have access to the premium version of Overleaf, they would likely not wish to downgrade. However this work is still useful to us, as in this process Müller documented internal APIs used to interact with Overleaf. One of these is used in the Overleaf import feature, to retrieve the project data.

3 Problem Specification

The aim of this project is to provide integration of Overleaf with CERNBox. Due to the limited time of 8 weeks, this is narrowed down to implementing two features:

1. **Export to Overleaf;** Pushing data from CERNBox to Overleaf
2. **Import from Overleaf;** Receiving data in CERNBox from Overleaf

In this implementation, all syncing of data is managed by the user — the user decides when to export their project to Overleaf and import back the data. Since no syncing is

¹<http://cds.cern.ch/record/2855323>

²<https://github.com/Daniel-WWU-IT/overleaf-owncloud>

implemented, data will most likely be overwritten during import or export. This should be made very clear to the user in the front end, with a dialogue box issuing a warning.

The option to export or import the resource should be available on right clicking the resource, and should only be available for resources for which exporting and importing from Overleaf makes sense. This means we want to implement a file action, which appears in the list of actions on a right click. Restricting this to only suitable resources is a design question, and is discussed further in section 4 and 5.2.1.

Another important part of this challenge was the fact that we do not have access to the user's Overleaf credentials, and it would not be good practice to ask the user for their password to a different site. This makes the process of importing from Overleaf, and overriding existing Overleaf projects impossible, since this can only be done with proper authentication. However, the export feature still works fine with the current infrastructure, which hints at decoupling the two processes.

4 Design

4.1 Export functionality

Export to Overleaf is implemented without any use of user credentials, in order for it to be usable in the current set up of Overleaf. This comes with the price that every time a user exports a project that is already exported, it creates a new Overleaf project rather than overwriting the previously created one. To remain transparent to the user, this is communicated whenever they export a project that they have already exported, as shown in figure 1. If in future we can access user credentials, be it by single sign on or other means, this should be changed to overwrite the previously exported project. This is discussed further in detail in section 7.2.1.

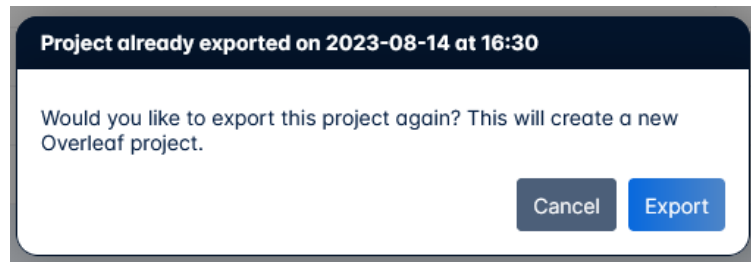


Figure 1: Modal window displayed to inform user that export will create new project.

Each time the project is exported, we store the project name as sent to Overleaf and the time. This allows us to show more user friendly error messages containing when the project was exported, as well as allowing us to match the project during import. This way if export is available before import is, we can still re-import projects that were exported before the import feature is added. However the project name sent to Overleaf might differ to the actual project name, as another project might already exist with the same name, or Overleaf might deduce the project name from the file contents. In the latter case, the user would have to export the project again to establish an active link to Overleaf.

To send the folder to Overleaf, we make use of a public documented API³ which when given a link to a zip file, unzips the file and creates a new project using the contents. This endpoint is documented further in section 6.1.1. The link we serve Overleaf is a link to the archiver, which is a service used in CERNBox to zip and download folders.

Technically, we can export any file or folder to Overleaf as our process always creates a zip file containing the file selected or the files found in the folder. However it would be less than ideal to show the prompt for every file and folder, as most have no relation to LaTeX. The initial plan was to limit our scope to either tex files, or folders containing tex files, however this deemed to be very difficult to implement as it would require turning a synchronous action to an asynchronous one, as discussed further in section 5.2.1. Currently, we show this prompt to both .tex files and folders, as well as download files.

At a high level, the process occurs as following:

1. User clicks ‘Export to Overleaf’ button on a tex file or a folder.
2. If project has already been exported, ask for user confirmation that they wish to create a *new* project.
3. Store the current time and the name of the project.
4. Redirect user to a new tab with a link to call the Overleaf API, with our archiver url as a parameter.
5. This creates a new project containing the exported folder/file.

4.2 Import functionality – A proof of concept

As a workaround to the fact that we don’t have user credentials, we make use of a pre-defined cookie obtained by logging into Overleaf. To be able to make this work on CERNBox, we would need to be able to have access to the user’s data, which would most likely necessitate implementation of organisational sign on to Overleaf using CERN credentials. While this feature is available from Overleaf’s side, it is not yet implemented for CERN, and there are no plans to implement this in the foreseeable future. However, we hope to show that this would be a useful feature as it would allow this synchronisation. More details on how this feature could be implemented in the future can be seen in section 7.2.

As stated before, importing a folder will overwrite the contents of the folder, and this is made clear by a modal window which requires the user to confirm they want to overwrite their current folder. There is no worry about data being lost here as both CERNBox and Overleaf keep backups of content, however it is always good to keep the user informed and avoid them the hassle of loading the backup. This is done as seen in figure 2.

The import is done by calling a protected endpoint in Overleaf, using the project identifier to determine which project we are downloading. This is documented thoroughly in section 6.1.2. Once the import functionality is feasible, we would be able to change our export functionality such that we know the project identifier of the project we are downloading. However in this current scenario, we might not know the project identifier immediately, so

³<https://www.overleaf.com/devs>

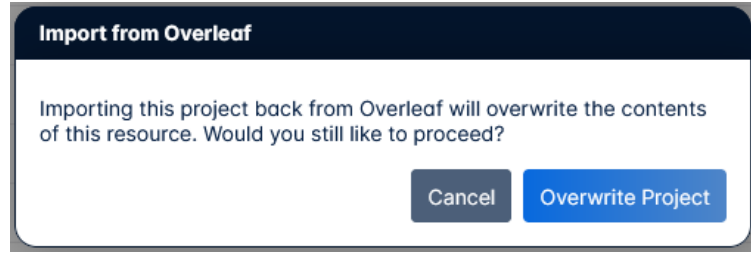


Figure 2: Modal window displayed to confirm overwriting of project.

there are extra steps taken to determine a matching project from the project name and project export time.

Adding this increases backwards compatibility, for if the project export feature is integrated before import, we would still be able to find the matching Overleaf project of a resource exported before import is implemented. However this solution will not work when Overleaf deduces the project name from the project title rather than using the `snip_name` parameter, in which case the user would have to export the file again to create a link with Overleaf.

Matching the project using name and time is done by fetching all information on projects whose name matches `<project_name>` (`<number>`), and finding the one with the most feasible time. This is determined using the following rules:

1. The project must have been created *after* being exported in CERNBox.
2. The project must have been created at most 30 seconds after being exported.
3. If multiple projects match this description, the one created closest to the export time is taken.

Note here that 30 seconds is arbitrary and can easily be changed in the case that export will take much longer or much shorter. It is quite a long time, however it is justifiable if a project is significantly large. We have this conflict with multiple projects as we are not aware if the project name in Overleaf is the one we specified during export. Specifically, if another project already existed with that name, it would be followed by a number in brackets.

Similar to export, we would like to filter out the projects which would make sense to re-import from Overleaf. Ideally this would include only projects exported previously, but unfortunately we run into the same problems as before, as fetching the extended attribute of a file would require an asynchronous action, as discussed in section 5.2.1.

Here we also run into a different problem in re-importing projects created from tex files. Since exporting a file to Overleaf creates a project, the user may add other files to the project. The behaviour when re-importing such a project is not obvious to the user. However the case where the project contains only one file is straightforward – simply replace the current file with the one in the project. This problem is resolved by prompting the user whenever the project they are importing contains more files, and asking them if they would like to upload all the files in the current directory, as seen in figure 3.

When downloading a project, first we upload each file contained in the zip file returned by Overleaf. This is quite easy using `zip.NewReader`. The unzipping of the file is done in

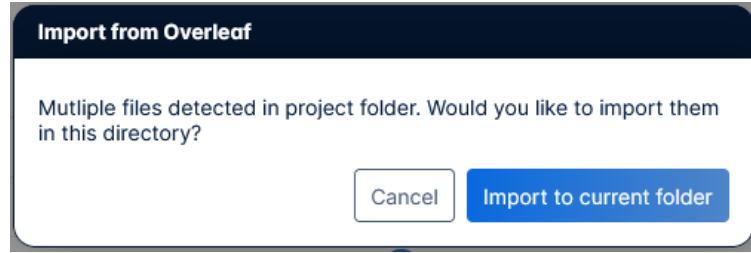


Figure 3: Modal window displayed when project contains multiple files.

memory, which is not ideal when it comes to large projects. Due to this, we set a limit to the size of the zip file. Note also that due to the way the zip reader is implemented, this uploads only files, not folders, hence if there is an empty folder in Overleaf, it is unfortunately not carried over to CERNBox.

After uploading the files, we then walk through the directory, deleting any files or folders which are not found in the Overleaf version, by checking if we can open that file or folder using the zip reader. Uploading each file means if a file with the same name is uploaded, it is considered as a newer version of the old file. Another alternative to this would be deleting the entire folder and then uploading Overleaf contents, but this was not chosen as it leads to different versions of the project ending up in the recycle bin instead of using versioning, which is the cleaner option.

At a high level, the process occurs as following:

1. User clicks ‘Re-import from Overleaf’ button on a tex file or a folder.
2. If project has not already been exported, display an error message and stop.
3. If the project identifier is not saved, deduce the project using the name and export time. If no matching project is found, display an error and stop.
4. Download project from Overleaf.
5. Unzip project and overwrite the resource’s contents with project contents.

4.3 Structure of our implementation

4.3.1 Initial approach

Many instances already exist of apps which serve to expose some app interface to the user, like CodiMD and Collabora, so the initial idea is to make use of the same interface used to launch these app providers. These apps all make use of the *appprovider* service. This service is very convenient as it handles permissions and checking that the file/folder is suitable. Moreover, the infrastructure already exists, even up to detecting the matching files in the front end automatically once an app provider is registered.

However, it is quite restricted for the scope of our problem. Communication to an app is done in one way — by sending an Open in App request, which then returns back an Open in app response. This response consists typically of a status and a URL, which the front

end will open in an iframe, which is a nested browsing context inside the CERNBox website. Already this process is not ideal, as import will not require us to return any URL. Moreover, Overleaf does not open in an iframe due to protections in the browser, which rules out the export possibility.

In the beginning, the idea was to modify the front end interface to allow for these changes, by adding a **Target** field which determines whether we open an app in a new tab or an iframe, and an **Action** field which determines the prompt of the app (as by default, every app appeared as “Open in App”). For CERNBox to display warnings when exporting a previously exported file, the UI needed to be modified even further to contain the logic that catches the conflict error and prompts the user for feedback.

These changes accumulate, and make our implementation of the open in app functionality diverge significantly from the ownCloud branch, of which CERNBox web is a fork of. This would be an unacceptable change, since the current aim is to rely on ownCloud as much as possible for front end, which would entail us moving closer to their implementation rather than further away. This, along with the fact that the app provider structure is not exactly ideal for our use case, was the final nail in the coffin for the idea to use the app provider.

4.3.2 User interface

The alternative to using the app provider infrastructure is to create an extension for ownCloud which contains our necessary user interface. Fortunately, ownCloud has recently launched a way to add different kinds of app interactions with the use of an extension, along with an example of how this works: a file importer⁴. Unfortunately, this feature does not yet allow us to mount the export/import to Overleaf as a file action — which means we can’t make it so the user right clicks on a file and gets the option to export or import the file.

While ownCloud plans on adding this functionality eventually, it would not be ready by the time allocated to this project, and having our export and import as a file action is integral to the project. Hence, we use a workaround and implement Overleaf as a context action. This is exactly the same as what was done for the [open remote file](#) action. Eventually, both of these actions would need to be moved to extensions.

Since all we need to do for the export and import is send http requests, we were considering the idea of implementing the entire functionality in the front end. This was quite late into the project, so existing code would have had to be migrated from the back end to the front end, but the logic was already there, so it would not have been too difficult. However, we opt to implement the core logic into the back end for two main reasons; firstly, complicating things in the front end might make things harder in converting our implementation into an extension, and secondly it is good practice to put logic in the back end, which is typically less prone to breaking changes, which means the code would need less maintenance.

Hence, our front end contains of two typescript files for the context actions import and export respectively. These handle interfacing with the user such as displaying error messages or modals, enabling or disabling themselves given the type of resource, and handling calling the Overleaf service described below.

⁴<https://github.com/ownCloud/web/tree/master/packages/web-app-importer>

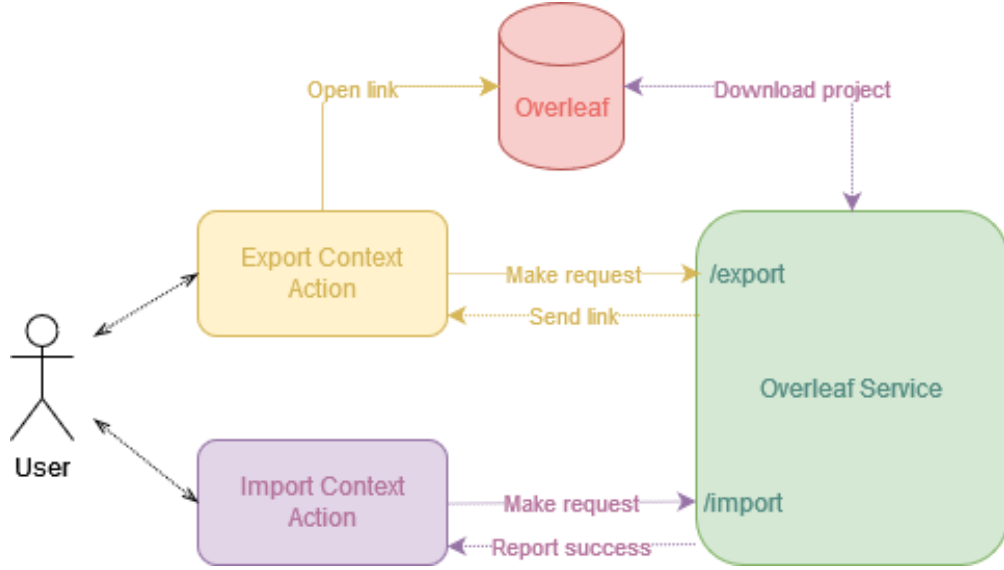


Figure 4: High level overview of interaction between components

4.3.3 Overleaf as a service

Since we cannot use the *appprovider* service any longer, we create a new Overleaf service, using the prefix `/overleaf`. For our purposes, it is enough to implement an http service, since we mainly send requests and set attributes. Our service has two endpoints, `overleaf/export` and `overleaf/import` which allow users to export or import files as expected.

Under the hood, the *appprovider* service validates parameters, extracts the resource information by finding the resource reference using the path and ensures the resource can be accessible by checking whether it exists and whether the user has the correct permissions. While the majority of code is copied from the *appprovider* gateway and http services, it differs in some ways when it comes to view mode. Since we know the permissions needed to do import and export, we check this directly instead, ensuring that the user has read permission for the export, and read and write permissions for the import.

5 Problems encountered

This section details the road bumps encountered on the way, and what steps (if possible) we took to circumvent them. Explaining these details is done to give insight into not just *how* certain actions were taken, but *why*.

5.1 Solved issues

5.1.1 Archiver modifications

While downloading a zip file was working well when we were testing it, serving the archiver link to Overleaf was returning a ‘Not found’ error. After investigating further, we concluded that this is because the archiver returns a zip or a tar file depending on what operating

system the user is on. More than likely, Overleaf servers were not using Windows, so the default download type would be tar, which Overleaf cannot parse.

This was solved by adding a new parameter to the archiver, called `archtype` which is either zip or tar. If this attribute is set, the archiver will download the folder as a zip or a tar file respectively, and if not set will default to checking the user operating system. Then in our archiver link, we simply set `archtype=zip`.

5.1.2 Exposing token in project name

When exporting a project to Overleaf, typically the name of the project is decided by the name of what is deemed to be the main tex file of the project, using the title defined inside the project. Now, if a project does not contain a tex file, or the file deemed the main tex file by Overleaf does not have a title, the name is taken from the link, as the string after the terminating slash. This is bad news for us, as this contains our query parameters, including the access token which can be used to access a user's files. This is not only unsightly, but a potential problem if a user shares this project.

Hence, we make use of the parameter `snip_name` provided by the Overleaf API used on export. As a value, we pass the name of the file/folder without any extension. This will always overwrite the project name in case of any projects with a missing name, and will also overwrite in cases where the name cannot immediately be determined by Overleaf.

5.1.3 Encoding name as an extended attribute

During export to Overleaf, we would like to store the name we are sending to Overleaf, so that we can better match the project later on during the import stage. This is done by storing the name in an extended attribute of the file/folder. When testing some files with this, specifically files with spaces or equal signs, we realised these would be stored properly but on decoding would appear truncated. This is due to how extended attributes are delimited in EOS.

To get around this, before storing the name of a folder or a file we encode it using base 64. This way there are no spaces or equals in the attribute, which means no unexpected behaviour occurs

5.1.4 Temporary rebase for CERNBox

CERNBox backend follows the development of Reva⁵, an Inter Operability Platform which effectively acts as a middle man between services and storage. Any changes made to the backend would have to be committed to Reva. However, at the time of this project there were some changes to the way configuration is parsed in Reva, which meant the CERNBox configuration was slightly outdated. This means that on top of the current version of Reva, CERNBox was not working properly.

This was handled by using a temporary rebase on the current Reva, removing any commits that do not match with the CERNBox configuration. Doing this allows us to test the changes we make. This led to development being done in two separate branches, the *local*

⁵<https://github.com/cs3org/reva/>

branch, based on the rebase and the up to date branch, which could not be tested during the majority of this project.

During the end of the project, CERNBox was up to date with current Reva master, so the Overleaf import and export features could be tested exactly as they are committed on the most recent version of CERNBox. These changes were tested on different branches, which had the same files. More information on the git setup of the project can be found in section 6.4.

5.2 Unsolved issues

5.2.1 Filtering out LaTeX projects

As mentioned previously, it would be nice that we do not expose Overleaf functionality for every single folder, as a user typically has many folders, the majority of which have nothing to do with LaTeX. We attempted to fix this by limiting projects such that ‘Export to Overleaf’ only appears on folders containing tex files and ‘Import to Overleaf’ only appears on previously exported folders. While this is easy to state, given the current structure of the code and the actions necessary for this, we were not able to implement this in the scope of this Summer project.

The main difficulty arises in fetching certain information about the resource such as its children or its extended attributes, since this requires a PROPFIND request. This is necessarily an asynchronous operation which returns a promise instead of an actual value. Once a user right clicks on a resource, the front end starts finding out which actions to show based on a method `isEnabled`. This is abstracted away by nested calls, all of which would have to handle the possibility of being returned either a promise or an actual response, and wait accordingly.

After trying to implement this, and discussing with the ownCloud team, this task was deemed unfeasible. The use case of having an asynchronous action was not one that they had considered, and would have to pertain a large restructuring of code. Hence, this was deemed out of scope for the project. While it is an improvement to the user experience, it would not break the Overleaf functionality to not have this feature.

However, if there is significant user feedback that this is a very unwanted change, we can try two approaches to fixing this:

- Making `isEnabled` and the rest of code calling it asynchronous.
- Appending the item for the export/import action to the menu *after* the menu is loaded, in an asynchronous way. This avoids changing a bunch of pre-existing code, but might be harder to do.

6 Technical details

6.1 Overleaf APIs

6.1.1 Export to Overleaf

Export to Overleaf works in a very simple way, by creating the request as a link and sending it to the front end, which opens the link in a new tab. Essentially this means the request is actually made by the user and Overleaf sends the response to them directly.

The endpoint is [/docs](#) with method **POST**. Parameters are found below in table 1.

Header name	Description	Value
snip_url	URL to the project to be downloaded	archiver URL
snip_name	Optional name of the project	Name of the file/folder

Table 1: Parameters for Overleaf Export.

We do not have access to the response since it is sent to the user, but on a successful attempt the request returns a 200 status. On errors, Overleaf displays a user friendly message.

6.1.2 Import to Overleaf

Downloading a project as a zip can be accessed with method **GET** through the endpoint [/project/<project_id>/download/zip](#)

For authentication we need to include the cookie of the user, and the host to match [www.overleaf.com](#), as otherwise sometimes the request gets rejected. These headers are detailed in table 2.

Parameter name	Description	Value
Cookie	Cookie of the current user's Overleaf	Pre-set cookie for one account
Host	Host sending request	www.overleaf.com

Table 2: Headers for Overleaf Export.

On success, we get a response with status 200, containing the zipped project as the body.

6.2 Front end configuration

The import and export functions are implemented as context actions in the front end. This is done by defining two methods, `isEnabled`, and the trigger method. We also have to add some methods to the `menuItemsActions` in `ContextActions.vue`, so that the front end will recognise that we added a new action.

Any communication with the user which requires user input is done through modal windows, otherwise it is done through messages in the right hand corner. All logic regarding

to exporting a project that's already been exported, or warning the user that an import will overwrite their current files is done in the front end.

6.3 Overleaf Service Configuration

To run the Overleaf service, we need to make some configuration changes. Currently, these configuration changes are manually typed in, but eventually we want to code these changes into puppet, hence we are documenting these changes thoroughly.

6.3.1 Adding Overleaf service to Gateway

Since the Overleaf service is lightweight, we can easily add it to the gateway configuration file. This can be done by adding the following lines:

```
[http.services.overleaf]
prefix = "overleaf"
app_name = "Overleaf"
archiver_url = "https://cbox-ocisdev-adriana-live.cern.ch/archiver"
app_url = "https://www.overleaf.com"
insecure = true
```

Note that for Overleaf import to work, you also have to add the user cookie as part of the configuration by adding `cookie = <user_cookie>`.

6.3.2 Adding Overleaf service to nginx

For the system to recognise the Overleaf service, I added the following code to the configuration files *etc/nginx/nginx.conf*.

```
location ^~ /overleaf/
{
    proxy_pass http://localhost:9143;
    proxy_set_header Host $host;
}
```

6.4 Git structure

As explained in section 5.1.4, development was done on two git branches: one for the local version, and one for the version up to date with the current Reva master. This is not the entire picture, since Overleaf export currently would work, while the import will need some more time due to authentication, I decided to also have different branches, one containing only the export functionality, and the other containing both export and import functionality, deemed the import branch.

After CERNBox became up to date, I also created two more testing branches *cernbox-test* and *cernbox-test-import*, to test the code with the current version of Reva. This also allowed testing of functions which should be included but were not yet implemented in the rebase,

for example the function used to parse the configuration of the service. Since there were no problems encountered in testing, the local branches based on the rebase can safely be ignored, and the up to date versions used.

The structure of the project is as following:

- *overleaf* – Contains the Overleaf export feature, based on Reva master.
- *overleaf-import* – Contains the Overleaf export and import features, based on Reva master.
- *overleaf-local* – Overleaf export feature based on *temp-cernbox-rebase*: **Outdated**.
- *overleaf-import-local* – Overleaf export and export features based on *temp-cernbox-rebase*: **Outdated**.
- *cernbox-test* – Contains Overleaf export feature based on current version of CERNBox for testing purposes.
- *cernbox-test-import* – Contains Overleaf export and import features based on current version of CERNBox for testing purposes.

Both *overleaf* and *overleaf-import* branches can be found in my [fork of Reva](#), while front end code can be found in my [fork of CERNBox web](#). This contains both import and export features, with the export disabled by commenting out some lines in *ContextActions.vue* on branch *overleaf-web*.

7 Future work

7.1 Migration to a CERNBox Extension

While currently the code that would allow us to implement these actions as an extension does not exist, it was promised to be implemented in the near future. Depending on the way this is implemented, this would change the how we could convert the context action to an extension, so details about this are still murky. Once this is done, the import and export actions would be stored as CERNBox extensions for ownCloud⁶.

7.2 Implementation of organizational sign on for Overleaf

Future work to be done for this project to become feasible must include the implementation of organisational sign on for Overleaf. This is a feature that is accessible to CERN as a paying Overleaf customer, and has several benefits, including:

- CERN users who do not have an Overleaf account can create one easily through their organisation.

⁶<https://github.com/cernbox/web-extensions>

- Since users do not have to write a new password for their Overleaf account, this discourages them from reusing the same password, which is a security vulnerability for CERN.
- This would allow for interactions with Overleaf, such as the ones outlined in the document, to be implemented far more easily, which is beneficial for an organization that makes heavy use of LaTeX.

7.2.1 Overwriting exported projects

If organizational sign on were to be accessible, one would still need to implement a method to overwrite an Overleaf project instead of creating a new one each time. Although this feature is not currently implemented since it depends on this organizational sign on, methods to do this were explored and will be outlined here in case of future expansion.

To upload a project and know its contents, one can use the endpoint [/project/new/upload](#) with method **POST**. The request body is formatted as multipart/form-data with a specified boundary and includes the attributes *name*: name of the project, *type*: type of encoding and *qqfile*: contains the encoded zip file. The response contains whether this succeeded or not in variable **success**, and the variable **project_id** which contains the unique project identifier. Alternatively, we could create a new project using the [/project/new](#) endpoint to create a new project and add files using the endpoint described below.

The overwriting part is less simple. As briefly discussed in section 2, Müller had documented some useful API endpoints. This includes the endpoint [/project/<project_id>/upload?folder_id=<folder_id>](#) to upload a file, which overwrites and files already there. This could be solution by adding each changed file. The challenge includes finding which files have changed, and also figuring out how to handle files that were deleted in CERNBox but remain in Overleaf.

Since we have access to premium features, there is also the possibility of using the git integration to update the document. Given a project identifier, one can access the project using `git clone https://git.overleaf.com/<project_id>`. Then, once any changes occur to the document they could be found using the fetch command. The Overleaf project could also be changed by pushing the changes on CERNBox to Overleaf, force pushing if necessary if we need to overwrite the project in Overleaf.

7.2.2 Continuous integration

The above mentioned git feature could also be used to implement a real-time integration with Overleaf. Since we would be able to know whenever the user has made any changes, we check intermittently whether we are up to date with the project, and if not, update the project accordingly. Such an implementation could let us mirror the same behaviour as other app providers. It would allow the user to click on the project, be redirected to Overleaf, edit their LaTeX documents there and have all their changes synced to CERNBox.