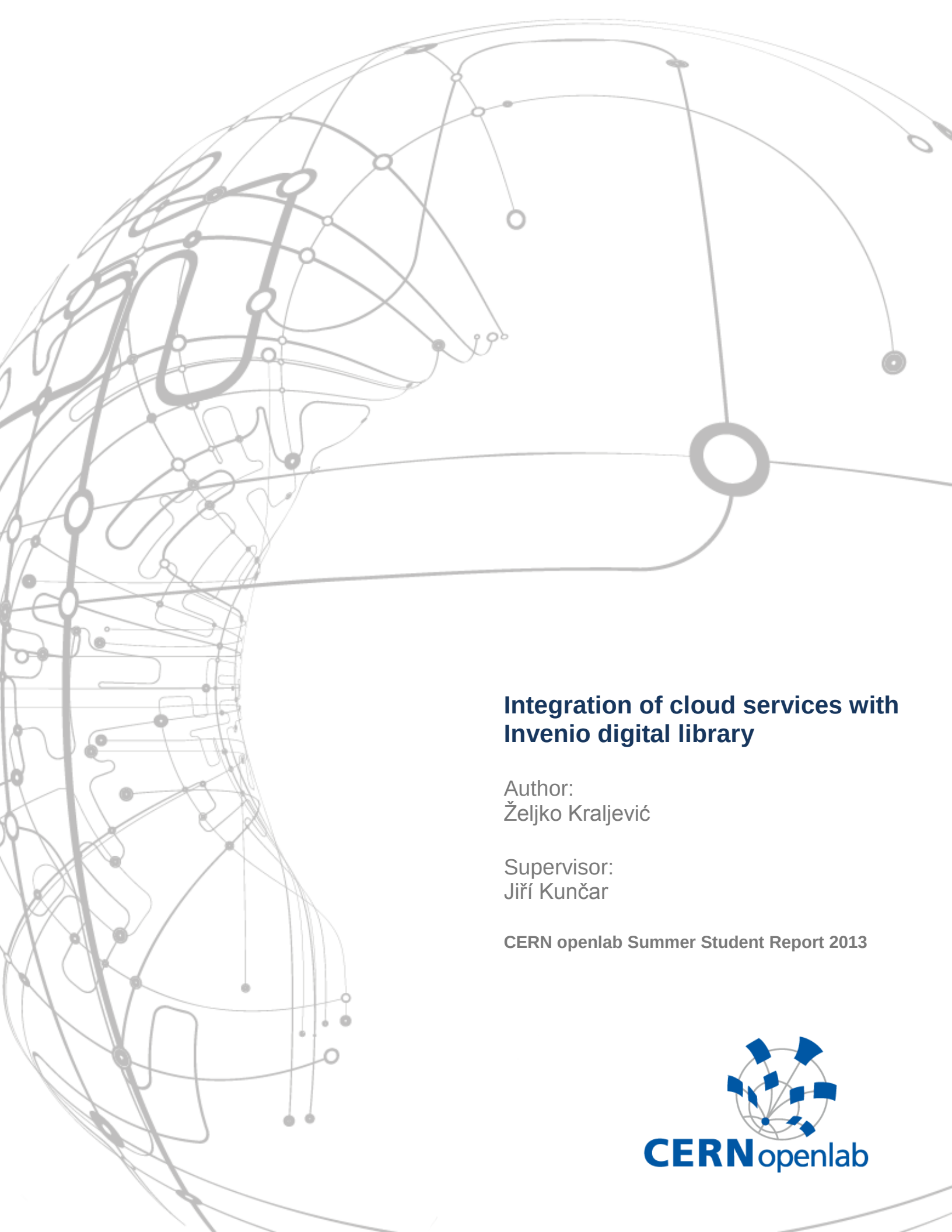

Integration of cloud services with Invenio digital library

Author:
Željko Kraljević

Supervisor:
Jiří Kunčar

CERN openlab Summer Student Report 2013

CERN openlab

Project Specification

Connect Invenio digital library with popular cloud storage services such as Google Drive, Dropbox and Sky Drive.

Abstract

Cloudutils is a standalone python module which provides an interface to popular cloud storage services (e.g. Google Drive, Skydrive…). It is built on top of the PyFilesystem module. In this project Cloudutils module was developed and used to connect Invenio digital library and popular cloud storage services.

Table of Contents

1 Introduction

My project is divided into two parts. The first part was to develop a standalone Python module that provides a common and easy to use interface to most popular cloud storage services (e.g. Google Drive, SkyDrive). Today a lot of applications need a connection with cloud storage services. It is mostly because users would like to have access to their documents from anywhere but also a cloud storage service is a reliable backup option. Connecting your software with existing services is easier than building such service from scratch and building something that already exists would break a dozen of programming principles.

As far as we know there is currently no off the shelf open source product that provides a common interface to popular cloud storage services, hence we developed module `Cloudutils` that in its simplest form can be used for downloading and uploading files from popular cloud storage services.

The second part of my project was to connect Invenio digital library to popular cloud storage services using the module developed in the first part of my project. One can ask following common questions:

- What would an Invenio user gain from this?
  - Every user would be able to easily upload the file he finds on Invenio to cloud storage.
- Why would Invenio users need this?
  - It's easy to share files when they are on cloud storage.
  - Invenio user doesn't want to download a file to his computer but would still like to easily access the file latter. A few reasons for not wanting to download the file could be:
    - He/She is not using his/her computer.
    - He/She is using his/her mobile phone.
    - He/She has a very slow internet connection.
  - Invenio user would like to access the file from all his devices that have internet connection without downloading the file to every device and without searching every time for the file on Invenio.
- How will this be good for Invenio administrators?
  - All cloud storage services are a backup option.
  - Files in the Invenio digital library don't need to be in one place, they can be distributed to many cloud storage services and still easily be provided to the end user using the `Cloudutils` module.

2 Cloudutils

`Cloudutils` is a standalone python module which provides a common interface to major cloud storage services. Its connection with cloud storage services and user applications (e.g. Invenio) is show in Figure 1.

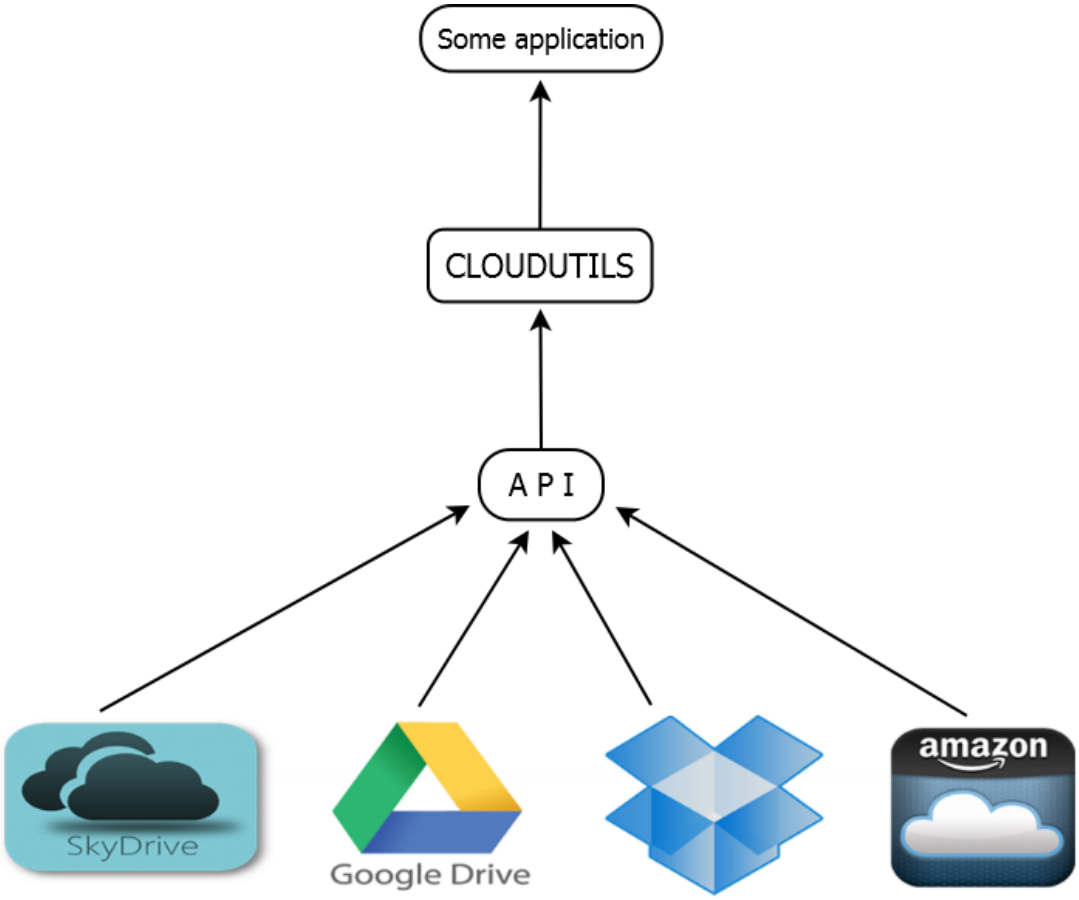

Figure 1. Cloudutils as a middleware between cloud storage services and some python application

The role of `Cloudutils` is to simplify the use of cloud storage services and to make every one of them look the same. If we take a look a bit deeper into the structure of `Cloudutils` module, but still maintain a high level of abstraction we get Figure 2.

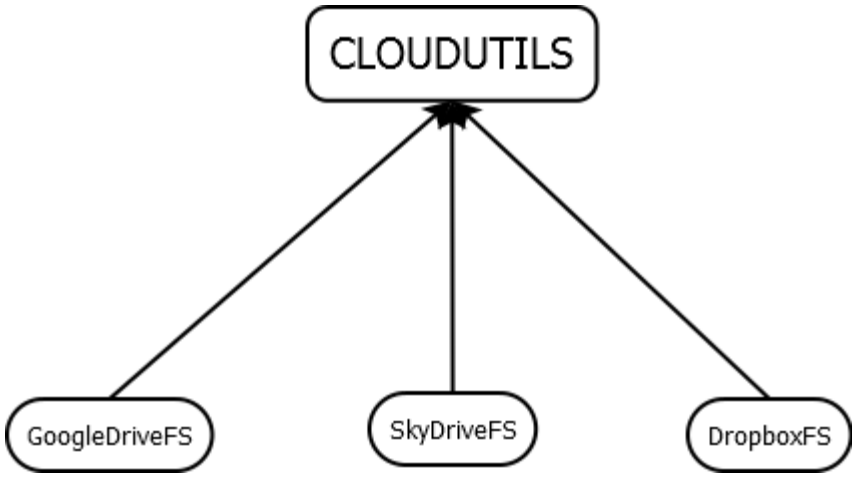

Figure 2. Cloudutils module at a very high abstraction level.

On the previous figure we can see that the `Cloudutils` module consists of three parts which are in fact cloud storage service filesystem abstractions.

2.1 Cloud storage filesystem

Every cloud storage filesystem abstraction consists of four classes:

- Cloud storage client
- Cloud storage cache item
- Cloud storage cache
- CloudStorageFS

2.1.1 Cloud storage client

This class is a wrapper around a cloud storage API. It provides unified methods and converts errors and exceptions that a cloud storage service returns to PyFilesystem exceptions.

2.1.2 Cloud storage cache

Downloading some data form a cloud storage service can be very time consuming. For example on my computer it takes around 5 seconds to list a directory. In this example the amount of data transferred is very low and doesn't consume much time. The problem is a cloud storage service has to authorize the client (e.g. some Invenio user) and also the client needs to authorize the cloud storage service. Because of this it was needed to implement a cache. Cache is a component that transparently stores data so that future requests for that data can be served faster.

Every implemented cloud storage filesystem abstraction caches the following data:

- Metadata about a file/folder (e.g. name, owner, size etc.).
- All children from a folder.
- All parents from a file/folder.

It is also possible to set a cache expiration time; this can be useful in the following scenario:

- We cached the content from a remote folder but in the meantime from our mobile phone we uploaded a new file in that remote folder. If we request again the same folder from our application, we will not see the newly added file because we will get the cached copy from the requested folder.

To set when will some data be considered expired we use the CACHE_TTL constant which by default is set to 5min. If an item is expired and a user asks for that item it will be fetched again from the cloud storage service, then cached and returned to the requester.

The `Cloudutils` module caches data when a user is:

- Listing a folder
- Opening a file/folder
- Creating a new file/folder
- Modifying content from a file
- Modifying file/folder metadata
- Copying a file

Caching is not done if the requested data is already present in cash and has not expired.

When caching a file/folder all the data important for that file/folder is put inside the `CacheItem` object and saved to memory.

2.1.3 CacheItem

The `CacheItem` is the class that `CloudStorageCache` uses as a data container that holds all the necessary information about one file/folder. The `CashItem` contains:

- File/Folder metadata.
- Children from a folder (if they exist).
- Parents from a file/folder (if they exists).
- Timestamp – the time when the `CachItem` was created/refreshed. This is used to check if an item inside cache has expired.

2.1.4 CloudStorageFS

This is the main class of every cloud storage filesystem abstraction. It is built on top of `fs.base.FS` class from `PyFilesystem.`

2.1.4.1 PyFilesystem

`PyFilesystem` is a Python module that provides a common interface to any filesystem. The easiest way is to think of `PyFilesystem FS` objects as the next logical step to Python's *file* class. Just as file-like objects abstract a single *file*, `FS` objects abstract the whole filesystem by providing a common interface to operations such as reading directories, getting file information, opening/copying/deleting files etc. `PyFilesystem` has many features and many classes. From that I have used only a small subset which provides all the functionalities I need.

I have used:

- `fs.base.FS` class, as the base class to all filesystem interfaces that I have developed. Most methods from `fs.base.FS` where overwritten so that they can work efficiently with cloud storage services.
- `fs.filelike.SpooledTemporaryFile` class, as a file wrapper for managing large remote files.
- `fs.remote.RemoteFileBuffer` class, as a `SpooledTemporaryFile` wrapper that controlees writing and reading from a remote file.
- `fs.errors` as the only errors that a filesystem interface implementation class can raise.
- `fs.path` for converting the user given path to a path that a cloud storage filesystem class can understand. In some interfaces methods from this class were overwritten, because some cloud storage services don't use paths.

PyFilesystem is developed really nicely and with lot of additional features, but it' still has some bugs when someone digs a bit deeper. For example:

- `fs.remote.RemoteFileBuffer` – This class is used when working with a remote file. It should write to a remote file only when `flush` or `close` is called. But for some reason when someone calls `close` on a `RemoteFileBuffer` it writes the same content two times to the remote file.

With a few workarounds I have fixed this and some other bugs and developed a common interface for three cloud storage filesystems.

2.1.4.2 Implementation of CloudStorageFS

This is the main part of every cloud storage filesystem abstraction. It is built on top of `fs.base.FS` class from `PyFilesystem`. Some basic methods in `CloudStorageFS` are:

- `Createfile` - creates a new file with empty content.
- `Open` - gets the content from a remote file and wraps it in a `SpooledTemporaryFile`. Because of this if a file is larger than 5MB (this is a constant and can be changed) it gets written to hard disk and if it is smaller it is kept inside RAM. The `SpooledTemporaryFile` is then wrapped in a `RemoteFileBuffer` and returned to the requester.
- `Copy` – copies a file from one location inside the `CloudStorageFS` to another. The copy method currently doesn't support copying between different filesystems.
- `Remove` – removes a file.
- `Getinfo` – returns information about a file (e.g. size, name, owner etc.)

This are just some basic methods, the list of all methods can be found on http://pythonhosted.org/fs/. Most of the methods from `fs.base.FS` are overwritten and optimized for cloud storage services.

In this project I have developed a filesytem abstraction for Google Drive, Dropbox and SkyDrive. Even though these are all cloud storage services they differ a lot in the API they provide. In the next few chapters I will describe the most important differences between this cloud storage services.

2.2 GoogleDriveFS

Every cloud storage filesystem abstraction depends on the API that the cloud storage service provides. Because of this the difficulty of implementing a cloud storage filesystem abstraction greatly depends on the API and the documentation a cloud storage service provides.

A quote from Google Drive page '*The Drive SDK gives you a group of APIs along with client libraries, language-specific examples, and documentation to help you develop apps that integrate with Drive'*. And here when they say help it really means help, because this is the best documented and overall the easiest API to work with. I had some problems with implementation, but they were connected with the way a PyFilesystem should work.

2.2.1 Implementation details

The major problem with implementing GoogleDriveFS was that Google Drive doesn't work with paths. To implement a filesystem with PyFilesystem it is necessary to have paths, but here it wasn't possible because of:

- Every file/folder inside of Google Drive is represented with an ID and it is not possible to get a path.
- Inside of a folder in Google Drive there can exist multiple files/folders with the same name, because the name is not important only the ID. This was one of the main reasons why it was not possible to convert an ID to a path.
- A file/folder can have multiple parents.

Because of this a workaround was needed. GoogleDriveFS was still made with paths, but some rules need to be followed when using paths. For example when creating a new folder path has to be in one of the following forms:

- parent_id/new_folder_name (when recursive is False)
- parent_id/new_folder1/new_folder2... (when recursive is True)
- /new_folder_name to create a new folder in root directory
- /new_folder1/new_folder2... to recursively create a new folder in root directory

All other methods in GoogleDriveFS where implemented according to the rule ID should be used as a one level deep path with a few modifications when creating a new folder/file.

2.3 SkyDriveFS

For me it was really hard to implement SkyDriveFS. The documentation is not really good and to make it worse they don't provide an official Sky Drive python API.

2.3.1 Implementation details

Sky Drive like Google Drive uses an ID to identify a file, but this problem was already solved when I implemented GoogleDriveFS. Here a new problem arrived, Sky Drive allows a user to use 'well known' folder names instead of an ID (e.g. me/pictures, me/documents etc.). A slight modification was needed, but in the end not a big problem.

Another problem was that Sky Drive doesn't provide a python API, but luckily I found a Python command-line interface for Microsoft LiveConnect SkyDrive REST API v5.0 which was used as a replacement for the official API.

The list of the problems is very big so to sum it up I will repeat myself and say that it was really hard to implement SkyDriveFS, not because of the big problems (mainly there were none), but because of at least a hundred of small problems (caused by a bug or insufficient documentation or something working differently than described etc.) that where not so easy to fix.

2.4 DropboxFS

It was very easy to implement DropboxFS because it uses paths for all actions like downloading, uploading, creating a new file etc. Dropbox also provides a python API even though it is not very documented. The Dropbox API is very simple compared with Google Drives, but it had all the necessary functions.

2.5 Organization of the Cloudutils module

In this chapter I will explain the basic structure of the cloudutils module and also provide a simple usage example. The module is explained from top to bottom.

The cloudutils module has a class `cloudutils_factory` and only this class should be used by someone who is using this module.

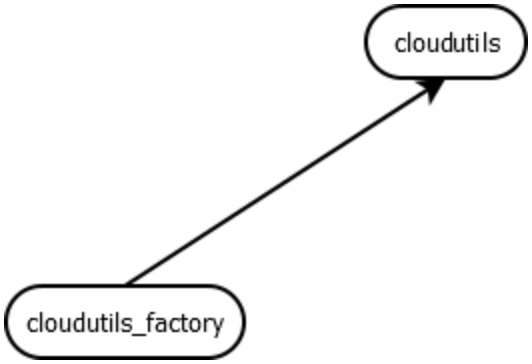

Figure 3. Second level of the cloudutils module.

The cloudutils_factory is used to create every CloudStorageFS that I have implemented. Basically one needs to provide credentials for a specific Cloud Storage service and the factory will do all the rest for you.

The cloudutils_factory uses three factories to build each CloudStorageFS.

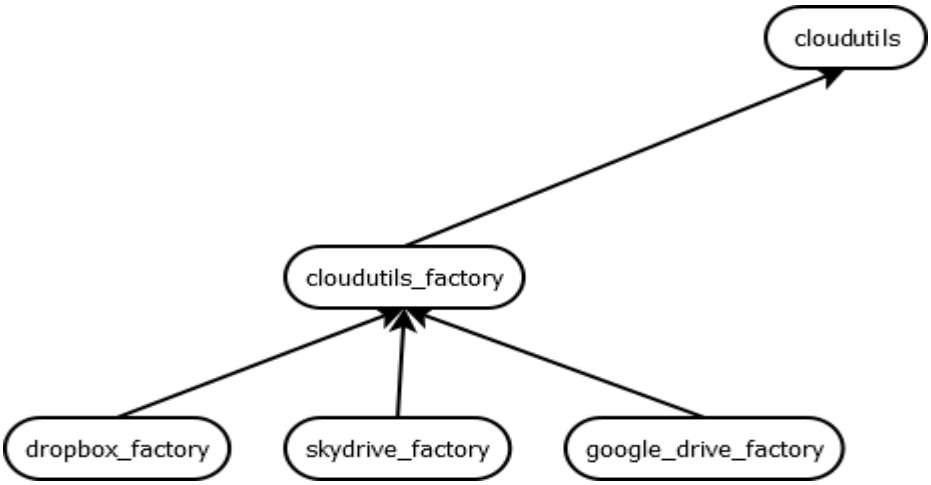

Figure 4. Second level of the cloudutils module.

Every of these factories is connected with a CloudStorageFS class which is connected with the cloudutils_config class, that provides some basic configuration for every CloudStorageFS.

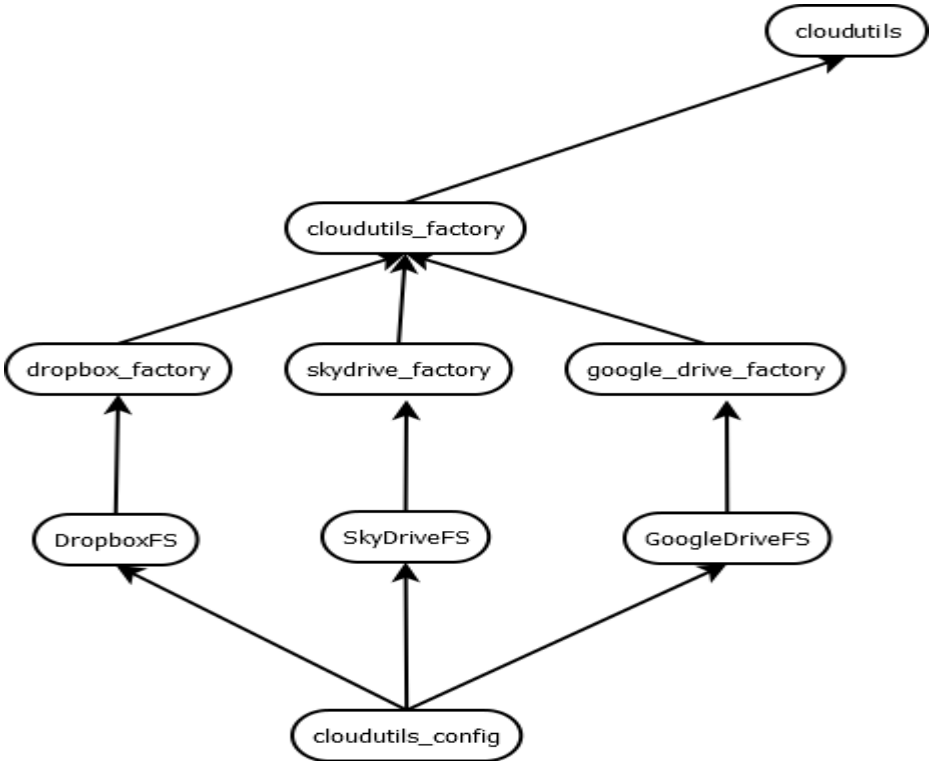

Figure 5. Third and fourth level of the cloudutils module.

The factories are in their current state configured to work with Invenio, but they can be easily rewritten to work with almost any python application.

3 Invenio and Cloudutils

The second part is connecting Invenio and cloud storage services using Cloudutils. A few things were needed here:

- Web development – HTML/CSS/JS.
- Customization of the Cloudutils factories.

3.1 Web development

It was needed to provide a user friendly web interface for:

- Connecting with cloud storage service
- Viewing files/folders
- Deleting files/folder
- Uploading files

3.1.1 Connecting with cloud storage service

An Invenio user is able to explicitly ask for a connection or can try to upload a file and is then automatically asked for credentials. When a user connects to a cloud storage he is never again asked for credentials, except when he explicitly disconnects from that service.

3.1.2 Viewing and deleting of files and folders

An Invenio user is able to view files and folders that were created using Invenio from every cloud storage service he is connected to. Also, every file from his cloud storage that he is able to see can be deleted.

3.1.3 Uploading files

An Invenio user can upload any file he can find on Invenio to his cloud storage service with just one click.

3.2 Customization of the Cloudutils factories

Cloudutils factories need a bit of customization to work properly with Invenio. It was needed to achieve:

- When creating a CloudStorageFS check in the database if the user is already connected to that service.
- Save all user configurations in the database.

All of this was achieved with slight changes in the main implementation of Cloudutils and the time need for this was under one day.

4 Conclusion

In this project I have developed a common easy to use interface to popular cloud storage services (e.g. Goolge Drive, SkyDrive and Dropbox) in the form of a standalone Python module named Cloudutils. It is open source and can be used by anyone and anywhere.

The whole module was built on top of the PyFilesystem module which is used to abstract any filesystem. Because of this a lot of additional features were added and an already tested approach was used.

The Cloudutils module was used to connect Invenio digital library with popular cloud storage services. This can serve as a proof for the ‘easy to use’ statement, because only a few lines of python code where needed to implement this.

5 References

[1] – PyFilesystem - http://pythonhosted.org/fs/

[2] – GoogleDrive API doc. - https://developers.google.com/drive

[3] – SkyDrive API doc. - http://msdn.microsoft.com/en-us/library/live/hh826521.aspx

[4] – Dropbox API doc. - https://www.dropbox.com/developers

6 How to register your app

To be able to use cloud storage services with Invenio or some other application you need to register your application.

Dropbox:

1. Go to https://www.dropbox.com/developers/apps
2. Click on Create app
3. Select Dropbox API app
4. Select Files and Datastores
5. Select Yes – my app only needs access to files it creates (this depends on what would you like for your application).
6. Write the name of your app (e.g. Invenio)
7. Write down the app key and app secret (you will need to fill this in the config section of cloudutils module)
8. Write the redirect URI to the ”OAuth redirect URIs” box. Please use **https** here, sometimes http doesn’t work. Write the full URI. (The best option would be not to use a port number, but if you have to try with and without it. Because in general there are a lot of people with the port number problem).

Google Drive:

1. https://cloud.google.com/console#/flows/enableapi?apiid=drive
2. Select create a new project
3. Click continue and wait a bit
4. Write the name and select Web Application
5. Click Register
6. Click on OAuth 2.0 Client ID
7. Fill the web origin and redirect URI
    a. Please use **https** for web origin and write a port number (People usually have problems with this, sometimes it will work with the port number and sometimes not. So I can’t say what the right way is).
    b. For redirect URI also try to use https and write the full redirect URI with port number and everything. (The same problem with the port number shows up even here, so you should try both ways).
    c. For all of this it would be the best to not use the port number, but if you have to, please follow the instructions above. For me, I didn’t use a port number and everything worked.
8. Please click Generate button now
9. Write down the Client ID and Client Secret.

10. On the left side menu click on APIs and check if Drive API is enabled, by default it should be.
11. This should be all.

SkyDrive

1. Go to https://account.live.com/developers/applications
2. Click on create application
3. Write Application name and click “I accept” button
4. Write the redirect domain
    a. Please note SkyDrive doesn’t allow the https://localhost:1212 form
    b. The URI has to be https://localhost.com .Basically a .com/.net or some else is needed.
5. Click on Save
6. Write down the Client Secret and Client ID