MashUp:

Merging Cloud Storages

# Motivation

Today's free cloud storage options tend to have extremely limited space (for example Dropbox only offers 2 GB). This makes it impossible to store larger files. Also, if a user wants to store more data for free, they are bound to create multiple cloud accounts in order to meet all of their needs. This makes it difficult to manage the user's files, as they are stored in different locations around the Internet. Remembering credentials for all those services might be an additional problem. MashUp is a service aiming to solve that problem. It lets the user register their cloud storage accounts with a web-service, which becomes a single, unified cloud storage account. When users upload files to MashUp, MashUp disassembles the files into parts and stores them in different cloud accounts. Then, when the file is to be downloaded, MashUp will assemble it back and present it to the user.

MashUp is a web service written in Python running on Gunicorn HTTPS server.

# Components of MashUp v0.1

## Authentication system:

Users can create accounts and log in to their accounts. When registering, user's password is hashed using bcrypt, and random salt is generated. That hash will be used for later authentication. Also, a hash for a PBKDF2-password-derived key is created. The username, bcrypt hash of the password and PBKDF2 hash of the password are added to the SQLite database.

When a user logs in, their password is checked against the hash created during the registration. If the hashes agree, a new session is created, and PBKDF2 key is derived from the password. This way, at no point are the password or PBKDF2 key stored outside of memory.

***Challenge:*** *Design of the security system was quite difficult, since I did not want to store user’s passwords or token in my database. In the end I decided to use two different types of cryptographic hashes with different salts. Apart from that, I had to create a wrapper for Python’s AES encryption to account for the initialization vector and padding. This experience has definitely given me some insight to how sensitive information should be stored, and how to properly implement it in my application.*

Unit tests for Authentication system are included in the service’s code.

***Challenge:*** *Writing unit tests for MashUp required spawning separate processes in the setUp method. Since I have not done such an operation in Python previously, it gave me a lot of new knowledge regarding Python processes libraries.*

## Cloud Storage Management

In order for the service not to be implemented separately for each cloud storage provider, the service includes a layer of abstraction for the clouds. This way, the service can use the same code for any of the clouds regardless of the cloud provider.

MashUp v0.1 implements Dropbox using the Dropbox Python SDK, and OneDrive using HTTP requests (OneDrive Python SDK lacks features that are crucial for MashUp, and is very poorly documented). For OneDrive, automatic key refreshing is implemented (OneDrive access tokens expires after an hour).

***Challenge:*** *Even though I did not end up using OneDrive Python SDK, I spent several hours browsing the source code of the SDK. In order to make some basic functionalities work, I replaced OneDrive SDK methods with my own. This taught me how a real life REST SDK is written, and also taught me not to use undocumented APIs, since they can be a waste of time.*

In order for MashUp to get access tokens to the clouds, OAuth support was implemented. The request for an authorization link binds the link to the session ID, so when the access token comes back to the application, the service knows whose cloud it refers to. Then, the access token is added to the database, encrypted with the password derived key described in the authentication section. This way, the access tokens are not readable unless the owner is logged in.

***Challenge:*** *Designing the cloud implementations so that OAuth support can be encapsulated was quite difficult – especially, since in the future, not all implemented clouds need to be authorized through OAuth. One of the problems was designing a proper URL for the authorization request, another one was properly adding the newly created access token to the right account - in the end, I used OAuth’s state parameter. As a result, I have gained a lot of insight in designing URLs for RESTful services, as well as using OAuth. I have also learnt new techniques for Object-Oriented-Programming in Python such as virtual class methods.*

## File System

File system is responsible for deciding about the disassembly and assembly of files, storing files in appropriate clouds, as well as accepting and sending files to and from clients.

The module is using two SQL tables. The items database stores folders and files’ metadata – unique ID, parent ID, name and the type: folder or file. The disassembly database stores information about the parts the files were split into – which file the part belongs to, and which bytes of the file are stored in that part.

The module also supports basic cataloging – one can create folders and put files or other folders into them. Listing contents and creation of catalogs have been implemented, as well as removal and moving of both files and folders.

***Challenge:*** *Designing the API call for moving items was quite difficult, since moving is not really a CRUD operation. In order to use the most sensible possible call for that purpose, I used PUT, and the destination path as the URL. The call behaves as if something was “put” in the destination path; a query parameter specifies, that the item put in that location is an item moved from another location in the service. The previous path to the item is sent in the body. I believe this problem showed me how to create a sensible restful design for operations that do not easily fall into the CRUD paradigm.*

The service supports partial file downloads – that is, the user does not have to request the entire file at once, and can specify the range of bytes they want to download. Using that data, the service performs a query on the disassembly database to find the relevant chunks, retrieves them from the clouds, and puts them together into the desired response.

For file uploading, upload sessions were implemented. Clients can upload the files in multiple requests – they just have to mark the last request with an “end” query parameter, so that the service knows the upload is finished. On the service side, the session will accept new data, and once it accumulates enough to form a chunk, it uploads the chunk to one of the user clouds. The cloud with the lowest used\_space/quota ratio is selected.

***Challenge:*** *For a long time, I was thinking about implementing a separate protocol on top of HTTP to send and retrieve files in multiple requests. However, I analyzed different cloud storage APIs and decided to mimic their approach to uploading large files, and simply use upload sessions. This has given me insight into how applications communicate with clouds.*

# Client

The client is used to demonstrate MashUp v0.1 as well as serves as an example client for MashUp. It provides a familiar interface for bash users, by exposing cd, ls, rm, mkdir, and mv on both local and MashUp file systems. It also provides fetch and store commands for transferring files between the user’s machine and MashUp.

***Challenge:*** *Creating the file system client was quite interesting – I had to find a way to properly manage both the remote and local file system. That’s why I thought of having two modes in the application – local and remote, and use fetch and store as a way to communicate between those two systems. Trying to emulate the behavior of the Linux bash using my own service, was challenging, but also showed me how I can present my service to the end user with a familiar interface.*

Apart from the file system, MashUp also enables users to add new cloud accounts to their MashUp accounts, as well as register and log in to the service.

Instead of using hardcoded command line program flow, MashUp’s client manages to encapsulate various parts of the flow using separate Menus and MenuEntries. Every menu is a collection of MenuEntries, and since different menu entries might need to do different tasks – the MenuEntry class is declared virtual. Also, the whole program uses a shared memory dictionary which stores the connection to the MashUp server and the Session ID. This way, one can create modular menus in an objective manner.