SERVER PROJECT MAIL

NIVESSE Charles – LEDOUX Alexis – MARCHAL Baptiste – MEROLLA Mathis

Table des matières

[Introduction 1](#_Toc59203102)

[SQLite DATABASE 1](#_Toc59203103)

[Creation 1](#_Toc59203104)

[Connection and methods 2](#_Toc59203105)

[Mail Server 4](#_Toc59203106)

[Connection 4](#_Toc59203107)

[Recuperation 5](#_Toc59203108)

[Sending 5](#_Toc59203109)

[Interface 6](#_Toc59203110)

# Introduction

When reading the subject for the first time, we chose to interpret it as a “Mail” type app found on Windows 10.

This app allows you to regroup any mail address you want into one mailbox and uses a local database to stock the received mails locally.

This report will present the prototype we built, starting by the presentation of the SQLite local database, followed by the network relations, then will end on the presentation of the Interface, coded in a console.

# SQLite DATABASE

## Creation

First, we create the SQLite database. Here, we need 3 tables :

* email: a table with all the information about each account.

Columns:

* + Address: the email address of the account.
  + Password: the password of the account.
  + Service: the name of the mail service (gmail, outlook..)
* mail: a table with all the mails from each account.

Columns:

* + Sender: the email address of the sender.
  + Receiver: the email address of the receiver.
  + Body: the message.
  + Date: the date of the mail.
  + Subject: the subject of the mail.
* log: a table with all the log of connection to the mail server.

Columns:

* + Type: the type of connection (login, send, receive..)
  + Ip: the ip address of the machine.
  + Date: the date and time of the connection.
  + Service: the name of the mail service (gmail, outlook..)

## Connection and methods

Now that the database is created, we need a bunch of methods to manipulate it.

First, we need to establish the connection to the database. We create a “create\_connection” method which will connect python with the database file (here names “pythonsqlite.db”).

We use this call to sqlite3 to connect it :

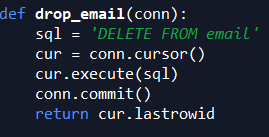


Now that we are connected to the database, let’s create each method.

1. DROP METHODS

We create drop methods to delete data stored in each table. Here we created 3 general drop methods (drop\_email, drop\_all\_mail and drop\_log) and a particular drop method which only drop the mails of a specified account (drop\_mail). This last method will be used when we recover each mails from a server (we’ll see later).

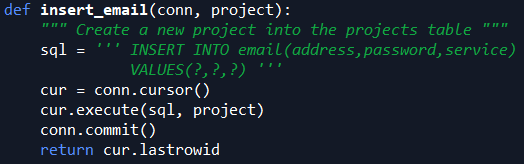
All 3 general methods look like this one :



1. INSERT INTO DB METHODS

We create insert methods to insert data in each table of the database. Here we create 3 general methods (insert\_email, insert\_mail and insert\_log).

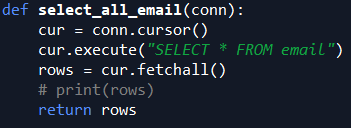
All 3 general methods look like :



1. SELECT FROM DB METHODS

We create select methods to select data when needed. Here we create 3 general methods (select\_email, select\_mail and select\_log).

All 3 general methods look like :

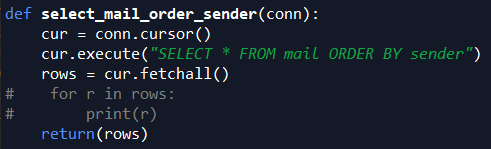


1. SORTING METHODS

We create sorting methods to sort the mails. Here we create 3 sorting methods:

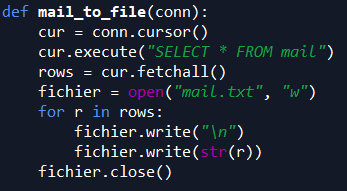
* + select\_mail\_order\_sender: order by sender.
  + select\_mail\_order\_receiver: order by receiver.
  + select\_mail\_order\_subject : order by subject.

All 3 methods look like :



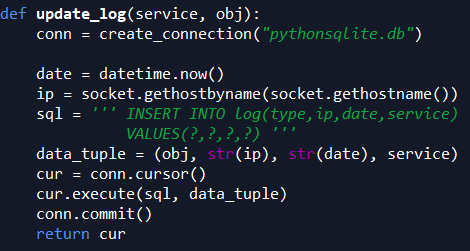
1. SAVING MAILS TO A FILE METHOD

We create a method to export every mails to an external file. We call the method mail\_to\_file. If the file does not exist yet, it’ll create it. Otherwise, it’ll overwrite the current one.



1. UPDATE LOG METHOD

Finally, we create a method to update the log table. We call it update\_log and we’ll use it everytime we have a connection to a mail server.



1. COMPLEMENTARY METHODS

We then have some leftover methods that we use in some scenarios:

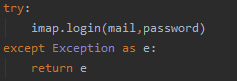
* + Select\_id\_email: return the id of an account.
  + Email\_exist: used to check if the email is already in the database.

# Mail Server

## Connection

Each time you want to connect to a mail Server, you have to call a login method (imaplib.IMAP4\_SSL.login or smtplib.SMTP.login). Each one verifies that your credentials are good and return errors if they aren’t.

To connect to a mail Server, we are using end-to-end encryption (SSL/tls) to make the transmission of data a bit more secure. We implemented two types of connection (two mail servers): ”gmail” and “outlook”.



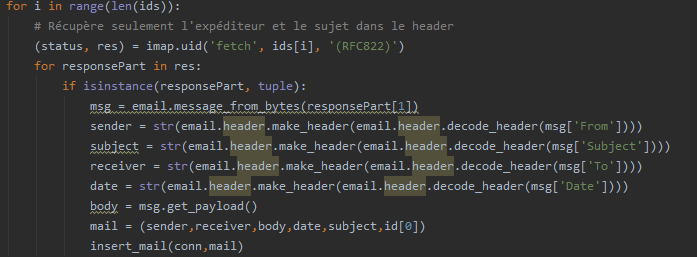
## Recuperation

To synchronise our local database with a mail server, we have two methods named “recup\_datagmail(mail,password)” and “recup\_dataoutlook(mail,password)”. They are basically the same, with just some values that changes, like the server you are speaking with.

After login in, this function calls drop\_mail(connection,mail\_adresse). The goal is to delete every mail in the local database for the account we are about to synchronise, in order not to have duplicates when we add them after.



Then, we simply recuperate all the data we need by the IMAP protocol, and for each mail we receive, we look for specific informations (sender, receiver, date, body,…) and we add it to the database with insert\_mail(conn,mail). After we add every mail we received, we simply close the connection.

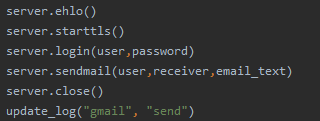


The IMAP protocol is used here and to make it easy to program, we are using imaplib, which is an integrated package of python since python 3.5.

## Sending

To send a new mail, we also have two methods (send\_outlook() and send\_gmail()), for the same reason as the recuperation of mails, which is because some values differs from one webmail server to another.

After login in using a secure connection, we simply use the smtplib.SMTP.sendmail() function to send a new mail. The library used is also an integrated one of python 3.5+. We just made sure that the text we send is well written. After sending, we simply can close the connection.



# Interface

For the interface we have the interface.py which contain the main menu of the app.

It works like this, when you launch the app you begin with a series of choice (in the console) for testing our different functions. It’s built like this:

* 1. SEND MAIL
     1. Send by outlook
     2. Send by gmail
     3. Return
  2. SYNCHRO MAIL
     1. Synchro outlook
     2. Synchro gmail
     3. Return
  3. LOCAL :
     1. Check your mails
     2. Check logs
     3. Save mails on a file
     4. Return
  4. QUIT

choice ‘a’, ‘b’, ‘c’ and ‘d’ are made with the function choice()

‘a’ use choice\_send\_mail() to give the choice for ‘I’, ‘ii’, ‘iii’ (in the code it’s ‘a’, ‘b’ and ‘c’)

‘b’ use choice\_synchro\_mail() to give the choice for ‘I’, ‘ii’, ‘iii’ (in the code it’s ‘a’, ‘b’ and ‘c’)

‘c’ use choice\_local() to give the choice for ‘I’, ‘ii’, ‘iii’, ‘iv’ (in the code it’s ‘a’, ‘b’, ‘c’ and ‘d’)

By the way every one of these ‘choice’ functions have a validation system so that there won’t be any error coming from the fact that someone put a wrong string.

For choice\_send\_mail():

* If ‘a’ is chosen, we call the function send\_outlook() from ConnectionMailServer.py to send an outlook’s mail.
* If ‘b’ is chosen, we call the function send\_gmail() from ConnectionMailServer.py to send a gmail’s mail.
* If ‘c’ is chosen, we return to choice()

For choice\_synchro\_mail():

* If ‘a’ is chosen, we call the function recup\_dataoutlook() from ConnectionMailServer.py to register the information of a outlook address to the database.
* If ‘b’ is chosen, we call the function recup\_datagmail() from ConnectionMailServer.py to register the information of a gmail address to the database.
* If ‘c’ is chosen, we return to choice()

For choice\_local():

* If ‘a’ is chosen, we call the function read\_mails() from Read\_mails.py to read all the mails from the email address
* If ‘b’ is chosen, we print the result of the function select\_log() from SQL\_retriever.py to check all the logs
* If ‘c’ is chosen, we call the result of the function mail\_to\_file() from SQL\_retriever.py to write all the data from the mails of the address mail to a file name mail.txt
* If ‘d’ is chosen, we return to choice()