

**Python Programming**

**USER MANAGER**

**Calatayud Giner, Jorge**

**Garcia Gascon, Luis**

**Manso Corde, Javier**

**Course 2022/2023**

**INDEX**

**0 ABSTRACT 3**

**1 INTRODUCTION 4**

**2 DEVELOPMENT 5**

**3 SUMMARY 10**

***4 BIBLIOGRAPHY 11***

# ABSTRACT

A simple and effective method of managing user data is what the User Manager project seeks to offer. Python3 was employed to create the project. This report outlines the issue we were trying to resolve, our approach, and the project's outcomes.

# INTRODUCTION

In numerous applications, managing user data is a frequent task. Nevertheless, it can be difficult to remember all the information, including usernames, passwords, and private information. By developing a tool that can effectively store and manage user data, the User Manager project hopes to offer a solution to this issue.

Python3, a well-liked programming language renowned for its clarity and usability, was used to create the program. To design a user-friendly interface and securely store user data, the program makes use of a number of libraries, including Tkinter, SQLite3, and hashlib.

# DEVELOPMENT

A modular approach was used in the development of the User Manager project, with each module in charge of a particular duty. There are several modules in the program, including:.

• Main module: This module controls the user interface of the program and creates the main window.

• The SQLite3 database used by the program to store user data must be created and maintained by this module.

• User data, such as usernames, passwords, and personal information, are created, edited, and deleted by the user module.

How each class Works:

**Gdu.pyw:**

All that this class does is run the program. Additionally, make sure you're running Windows—Linux is incompatible.

**Functions.py:**

The following functions are defined in this class and are used in the mainclass:.

**compare\_dates(y\_o, m\_o, d\_o)**: With the help of the y\_o, m\_o, and d\_o arguments, this function compares the current date to a specified date. It creates a date\_input object by using the datetime module to obtain the current date. Following that, it compares date\_input and date\_output (which were created using the year, month, and day provided) and returns True if date\_input is greater than or equal to date\_output and False if it is not.

**get\_date():** With the help of this function, you can get the time and date right now. The time is formatted as "HH:MM" using the datetime module to obtain the current date and time, and a list containing the formatted date and time is returned.

**payment() -> dict:** Payment and expiration dates are generated by this function. With the keys "payment" and "expiration," it creates a dictionary called date\_payment. It constructs strings for the payment and expiration dates and retrieves the current date using the date module. The month and, if necessary, the year are increased to determine the expiration date. The date\_payment dictionary is then returned after the function has assigned the payment and expiration dates to the appropriate keys.

**autoincrement\_id() -> int:** In order to create a new ID, this function takes the maximum ID from the "users" database table and increases it by one. The maximum ID is retrieved using database queries, and in the event that none are located, the new ID is set to 1. Returned is the incremented ID.

**check\_ci(ci) -> bool:** This function determines the validity of a supplied identification number (ci). It first turns ci into a string and determines its length. An additional confirmation that ci only contains digits is provided if the length is 8. The function returns True if both requirements are satisfied; otherwise, it returns False.

**get\_display\_size():** This procedure retrieves the system's display's screen resolution. A temporary Tkinter window is created, maximized, the screen height and width are obtained, and the window is then destroyed. It then returns the display's height and width.

**check\_ci\_exists():** From a database table called "users," this function retrieves a list of active identification numbers ("ci"). Then, it creates a list of all "ci" values by running a query to retrieve them all. It returns the list of "ci" values.

**debugging\_msg(msg):** A debugging message (msg) is printed to the console by this function. The message is simply printed, and True is returned.

**system\_log(msg):** In a text file at "log/system\_log," this function logs a system message (msg) along with a timestamp. txt". It obtains the current date and time using the get\_date() function, creates a message structure using the timestamp, and appends the result to the log file. It makes a new log file if the existing one doesn't exist. No return value is mentioned explicitly.

**check\_string(ci):** Using the supplied string (ci), this function determines whether any alphabetic characters are present. The isalpha() method is used to iteratively go through every character in the string and determine whether or not it is an alphabetic character. The function returns True if at least one character is alphabetic; otherwise, it returns False.

**Encontrar\_librerias.py:** It's simple to install them if we need them because this class only defines one method, which is used to write the module that we need in a text file.

**Config.py:** A config file is essentially what this class is. It has some global parameters and all the imports we require, making it simple to maintain consistency in our code. The method we defined in Encontrar\_librerias . py is also used in this method to verify that all the imports are correct.

**Gdu.py:**  This class implements the GUI for our application. We are using the Tkinter library to create this GUI because it is simple to use and we felt it was the best option for our app.

Now we are going to explain each function:

*init*: This is the initialization method of the class. It sets up some properties of the main window, creates and places the interface components, and starts a thread to execute a background function.

*\_\_root\_config*: Configures the appearance and behavior of the main window. It sets the background color, size, resize capability, title, and icon of the window. It also sets up a function to handle the window's closing event.

*\_\_stop\_thread*: Stops the thread that was started in the "init" method and destroys the main window.

*\_\_create\_treeview*: Creates and configures two tree views that display information about users and users with expired memberships. It defines the columns of the tree views, inserts user data into the tree views, and adds scrollbars if there are many users.

*\_\_get\_selected\_user*: Retrieves the ID of the selected user from the user tree view.

*\_\_get\_selected\_expired\_user*: Retrieves the ID of the selected user from the expired memberships tree view.

*\_\_delete\_user*: Deletes the selected user from the database after confirming the deletion.

*\_\_show\_action*: Displays the corresponding action in the "actions" label based on the button or area that was clicked.

*\_\_deselect*: Deselects items in the user tree view when clicking on another area of the main window.

*\_\_create\_widgets:* Creates and places buttons and other interface components, including the images used in the buttons.

*\_\_time*: Displays the current time in the main window and updates it every second.

*\_\_confirm\_message*: Displays a confirmation dialog before performing an action, such as adding or modifying a user.

*\_\_on\_quit*: Handles the closing event of the child windows created to add, modify, or view details of a user.

*\_\_add\_user:* Opens a child window to add a new user.

*\_\_insert\_query:* Performs the insertion of the new user into the database.

*\_\_msg\_accept(self, message, operation):* A method that displays a message and performs a specific operation based on the user's choice.

*\_\_add\_partners(self):* A method for adding new partners (users) by creating a new window and collecting their information.

*\_\_modify\_partners(self):* A method for modifying existing partner information by creating a new window and pre-filling it with the partner's data.

*\_\_get\_partner(self):* A method for retrieving a partner's information from the database.

*\_\_view\_partner(self):* A method for viewing partner information by creating a new window and displaying the partner's data.

*\_\_add\_query(self):* A method for executing an SQL query to add a new user to the database.

*\_\_modify\_query(self):* A method for executing an SQL query to modify an existing user's information in the database.

*\_\_get\_view\_user(self):* A method for retrieving a user's information from the database for viewing.

*\_\_load\_expired\_partners(self):* A method for loading expired partners from the database and updating their state.

*\_\_get\_all\_users(self):* A method for retrieving all users' information from the database.

*\_\_refresh\_users(self):* A method for refreshing the display of user information in a tree view widget.

*\_\_refresh\_users\_expired(self):* A method for refreshing the display of expired users in a separate tree view widget.

*\_\_check\_state(self, p\_op):* A method for checking the state of a user's membership.

*\_\_pay\_membership(self, p):* A method for handling the payment process for a user's membership.

Tkinter, a Python library for developing graphical user interfaces, was used to develop the program's user interface. In addition to a login form for accessing the program, the interface has several forms for adding and editing user data.

Data for the program is kept in an SQLite3 database, which makes data storage and retrieval quick and easy. In order to securely store passwords and protect user data, the program uses hashing.

# SUMMARY

An easy and effective method of managing user data was offered by the User Manager project. The program's user interface is clear and simple to use, and its data storage system is effective and safe.

The program's modular design makes it simple to add new features or change existing ones, making it a flexible option for managing user data. In the event of a security breach, user data is safeguarded thanks to the program's use of hashing to store passwords securely.

The project might benefit from the addition of new features, like the capacity to import user data from a different source or export user data to a CSV file. Though most use cases can be satisfied by the project's current functionality.

# BIBLIOGRAPHY

Python3 documentation: <https://docs.python.org/3/>

Tkinter documentation: <https://docs.python.org/3/library/tkinter.html>

SQLite3 documentation: <https://docs.python.org/3/library/sqlite3.html>

hashlib documentation: <https://docs.python.org/3/library/hashlib.html>

In conclusion, the User Manager project provides a simple and efficient way to manage user data. The program's modular approach, user-friendly interface, and secure data storage system make it a flexible and reliable solution for managing user data.