Final Project:

Password Manager Program

EE 202 – C2 - Team 1
Dr. Saud Wasly
February 19, 2023

Mahdi Bathallath
2236809

Taha Barzanji
2135092

Mohammed AlShehri
2042124

Contents

Self-assessment Rubric

		Mark Out of 10
REPORT	Well Formatted according to PTW checklist	✓
	Spelling and grammar checked	✓
	Clear Writing Style	✓
	Includes detailed problem statement, project objectives, and impact	✓
	Effective use of illustrative figures and tables	✓
	The report covers all aspects of the project/assignment, including pictures and diagrams of the implementation	✓
	Includes detailed justifications for all design choices	✓
	Reflects on learned lessons and failed experiments	✓
	Use of References	✓
Implementation	Code is Working as Expected	✓
	Meets all the requirements	✓
	Extra Features	✓
	The code is readable and meet good coding Standards	✓

Table of Contributions

TASKS	**Mahdi Bathallath**	**Taha Barzanji**	**Mohammed AlShehri**
UI Sketch {5%}	✓	✓	x
UI Creation {5%}	✓	✓	x
Data Structure {20%}	✓	✓	x
UI with triggers {10%}	✓	✓	x
Open/Save files {10%}	✓	✓	x
Open URL {5%}	✓	✓	x
Encryption {5%}	✓	✓	x
Packaging {20%}	✓	✓	x
Extra Features {20%}	✓	✓	x
TOTAL Contribution:	**51%**	**49%**	**0%**

Context

What is Password Manager?

Password Managers are programs that enable users to save passwords digitally and in a safe way. These tend to be stored in encrypted files which can be decrypted using a master password or any other authentication method. Since these files are encrypted, they can also be saved on cloud storage services, such as Google Drive and Dropbox, or locally on a device. These aspects form the specifications of Team one's Password Manager. After fathoming these specifications it was time to design the UI (user-interface) using QtDesigner. Then it was possible to implement the coding necessary to bring the program to life. Finally, testing was necessary to eradicate any bugs in the program.

Project Objectives, and Impact

The aim of this project is to make a safe environment for the user to store passwords. The user has to have a range of options when it comes to saving, modifying, and retrieving said passwords. This project will give students an understanding of how to make a program from scratch, starting with sketching/designing the user interface to packaging and making distributable installers. One of the fundamental concepts that are covered is encryption and its implementation. Due to the sensitive nature of stored data, encryption is needed to keep personal passwords from falling into the wrong hands. These might include hackers, a nosy family member or anyone who has access to said files.

Design choices and steps

Sketching the user interface

Before implementation a concept layout needed to be established. This is to help augment the initial design in a series of improved designs. However, for this project an initial sketch design was used to ensure that the UI was user friendly and intuitive. The layout was improved over the course of the project in QtDesigner. This happened after consulting the supervisor on the design. As seen in Figure 1, the initial design is functional but not very intuitive.

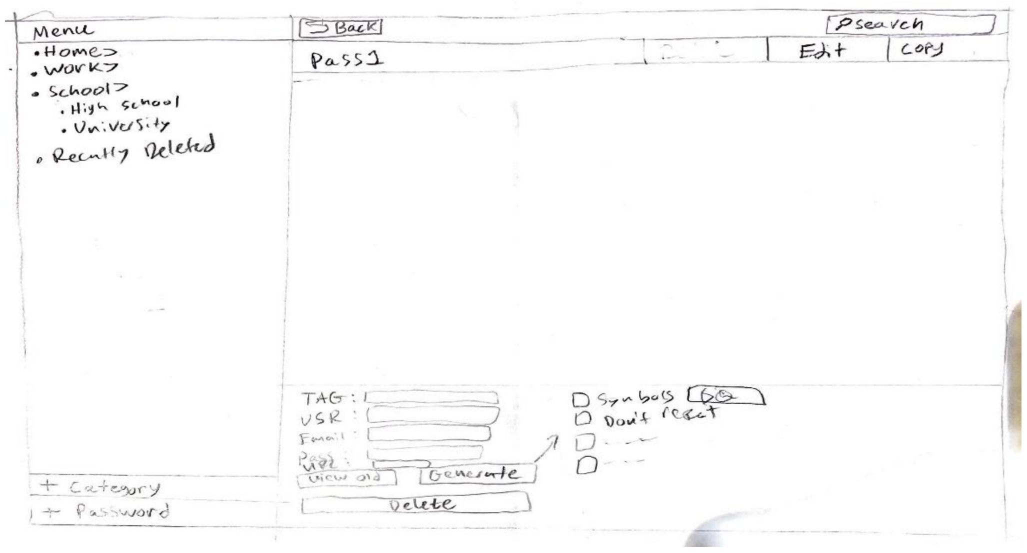


Figure 1: Sketch of Initial Design

Designing the user interface

QtDesigner was used to turn the initial rough sketch into a laid-out user interface. As seen in Figures 2.1 through 2.6 there are noticeable differences between the initial rough sketch and the ones in the UI. Firstly, the edit and copy button are no longer integrated into the QTreewidget item (pass 1). Also, the “Add Password” button is no longer under the categories widget, instead, it is below the accounts widget. These changes came after a meeting conducted by the team members to simplify the user interface. We also decided to add more windows to further simplify the main window design. Figure 2.2 shows the window created for adding and editing a password. When comparing this to that of the original sketch it seems to be more intuitive to the user and easier to implement. The generate button was also changed from the original design. Originally the choices for the password were meant to be enabled and disabled when not required. Creating a window for it, as seen in Figure 2.3, makes the main page look simpler. The windows for creating a master password and entering one was not part of the original design because the team had an idea of how it would look. These can be seen in figures 2.4 to 2.5. This is also true for viewing old passwords which can be seen in Figure 2.6. Lastly, we decided to add a window to allow the user to restore an account from the “Recently Deleted” category later in the implementation stage. This was done because the previous category might have been deleted so this gives the user an option to restore an account and relocate it wherever they like.

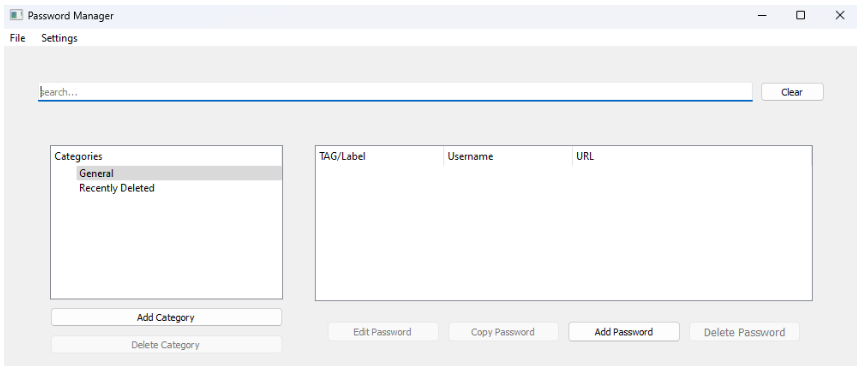
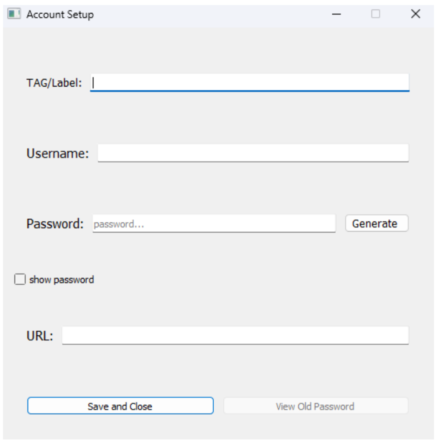


Figure 2.1: Final UI Main page Design



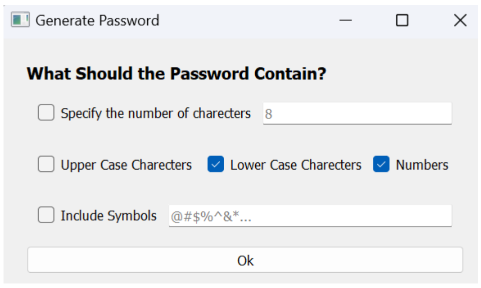
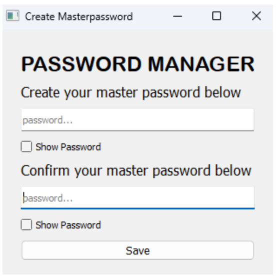


Figure 2.2: Add or Edit Accounts Window

Figure 2.3: Generate Password Window



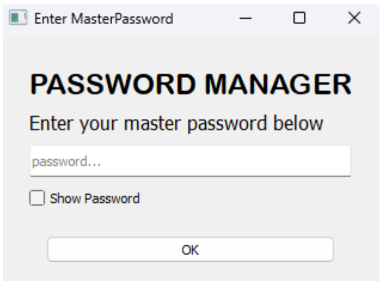
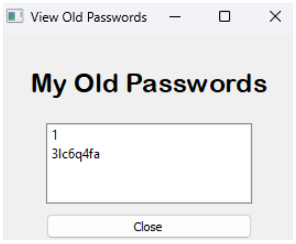


Figure 2.4: Master password creation Window

Figure 2.5: Master password entry Window



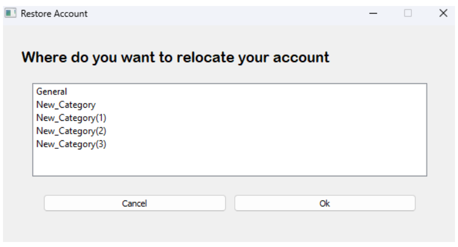


Figure 2.6: View Old Passwords Window

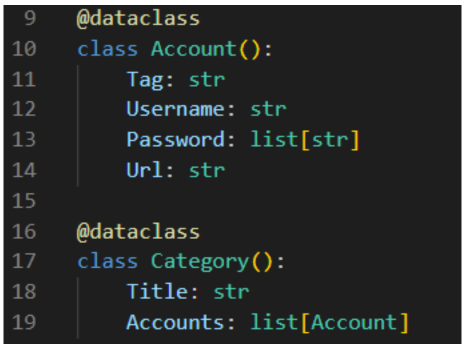
Figure 2.7: *Restore Accounts Window*

Implementing user interface functionalities

Implementing functionalities took place over the course of the project. We first started by compiling the UI files and connecting the buttons with their respective windows and functionalities. Each team member was tasked with doing the methods separately and then a meeting was conducted to choose the best approach. By the end of the meeting the team had methods for adding a category, adding an account, deleting a category, deleting an account, editing an account and a password generator. Then we worked on our initial data structure which was a dictionary of nested dictionaries. This will be discussed later. After that it was time to implement the opening and saving features for files. The files opened could be remote or local in nature; however, saving files was only done locally. In the interface we needed to get the URL in case of remote files or the path of local files. This was achieved through the getOpenFileName and getSaveFileName methods. Afterwards it had to be linked to a method in the data structure class.

Creating the Data Structure

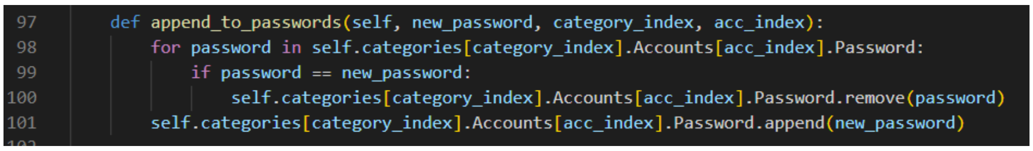
The data structure was implemented in a separate python file to help modularize and organize the program. This file contained a class that controls everything relating to the data. Initially the team decided that it would be best to create the data structure using a dictionary of nested dictionaries. However, after trying to implement some of the functionalities, such as deleting accounts and editing them, we decided to use the "@dataclass" decorator. In fact we created two different classes shown in Figure 3.1. One was used to organize the accounts and the other to associate these accounts with their categories. Instead of using key and value pairs everything was accessible through indexing. This made it easier to change the tag names instead of deleting the whole dictionary and adding a new. Additionally, the password attribute in the Account class was of type list. This is because it had to store all the previous passwords for the user to be able to access all their old passwords in the UI. Essentially, the newest password was appended to the end of the list and accessed by the index -1. If the password was repeated, the method in figure 3.2 deletes it and appends it to the end of the list.



```python
@dataclass
class Account():
    Tag: str
    Username: str
    Password: list[str]
    Url: str

@dataclass
class Category():
    Title: str
    Accounts: list[Account]
```

Figure 3.1: Data class



```python
    def append_to_passwords(self, new_password, category_index, acc_index):
        for password in self.categories[category_index].Accounts[acc_index].Password:
            if password == new_password:
                self.categories[category_index].Accounts[acc_index].Password.remove(password)
        self.categories[category_index].Accounts[acc_index].Password.append(new_password)
```

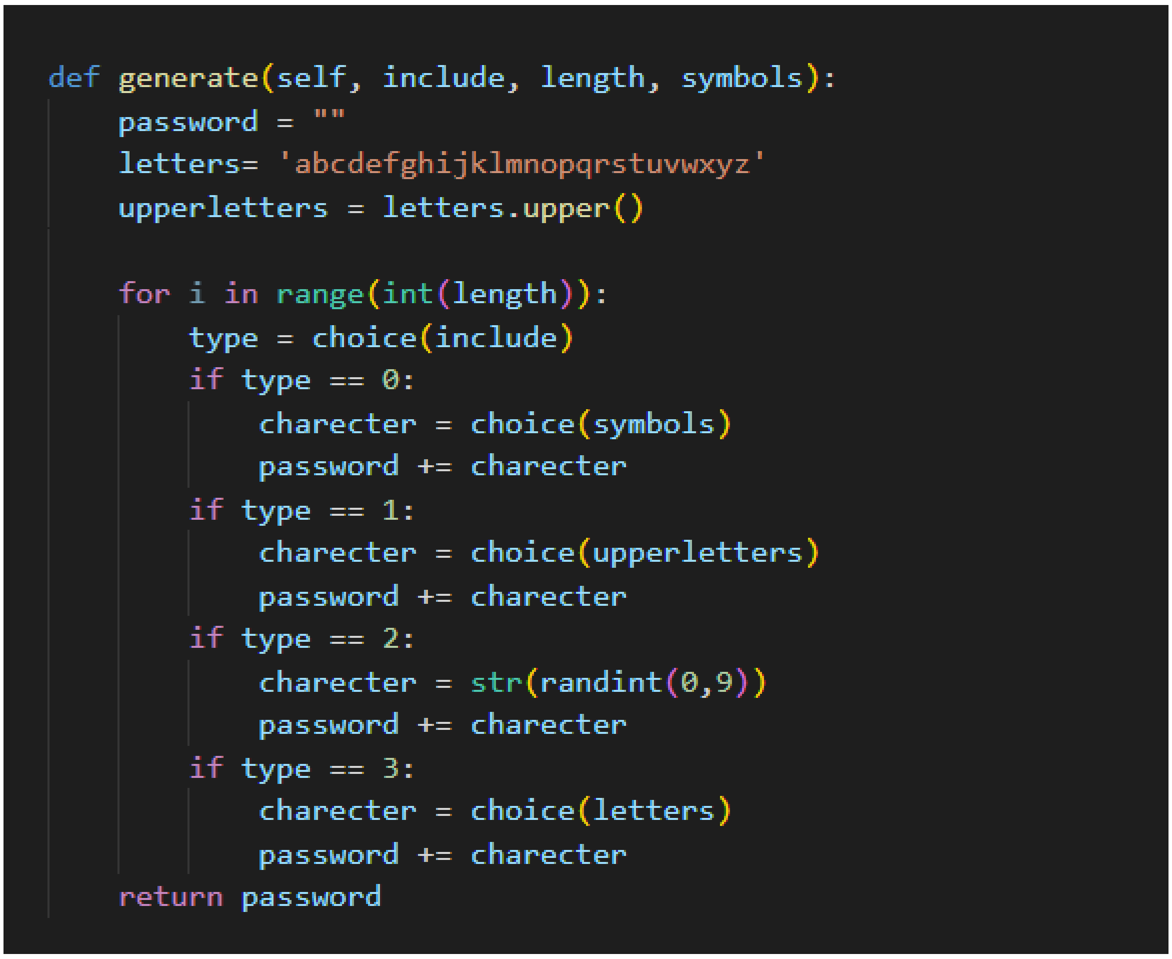
Figure 3.2: Appending passwords

The importance of the data structure lies in the need to save the file and resume it later. Thus, the saving and opening features had to be done in the data structure file. The Pickle and Requests modules where used in this case. From Pickle we used dumps to serialize the data structure and loads to desterilize it when opening a local file. In the case of remote files, the get method was used to get the contents of the file then deserialize it using loads.

After methods of the data structure where created it was important to link them with methods in the main UI application. The main sequence of events for the program is as follows:

1- User interacts with the front-end of the program
2- Interaction is caught and an appropriate method in the data structure class is invoked
3- The method produces data and is sent back the user interface section of the code
4- The data is then translated into a specific front-end reaction
5- The user is shown the reaction and based upon it decides to return to step 1 or not

The previously mentioned steps apply to almost all methods that are in the program, with some exceptions, such as methods whose whole purpose is just to show a certain window. In reality, anything that dealt with the data being stored in the data structure was in the data structure class. There was a discrepancy when it came to placing the generate password method in the data structure class. Nevertheless, the team decided to keep it there since it is still considered as data, much like editing a file. The method mentioned can be seen in figure 3.3.



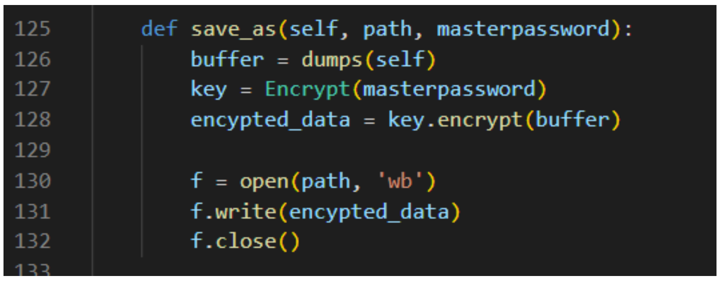
```python
def generate(self, include, length, symbols):
    password = ""
    letters= 'abcdefghijklmnopqrstuvwxyz'
    upperletters = letters.upper()

    for i in range(int(length)):
        type = choice(include)
        if type == 0:
            charecter = choice(symbols)
            password += charecter
        if type == 1:
            charecter = choice(upperletters)
            password += charecter
        if type == 2:
            charecter = str(randint(0,9))
            password += charecter
        if type == 3:
            charecter = choice(letters)
            password += charecter
    return password
```

Figure 3.3: generate password method

Encrypting data

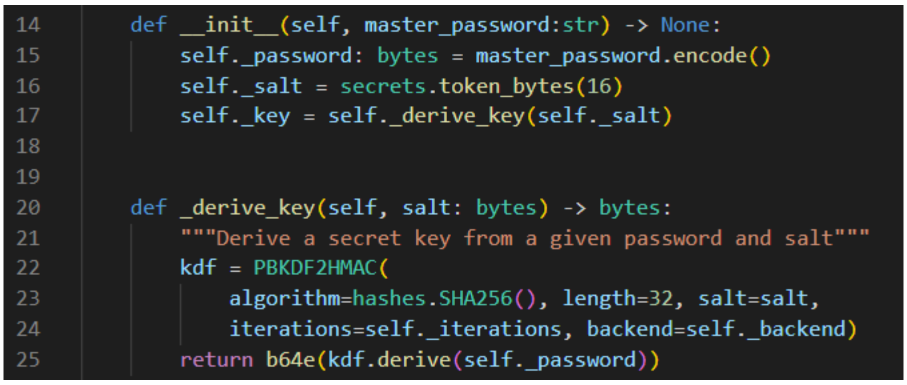
Encrypting data is a vital step in building most programs that deal with an end user, let alone a password manager. For this project, the choice between using an Encryption module and creating one from scratch was available. After attempting to create one from scratch the team decided it would be best to use the module since the one that was created had many bugs. As seen in figure 4.1, the file is encrypted after it is serialized. In the user's frontend, this method is invoked after the user selects the desired file path and master password. In line 127 of figure 4.1, an object called key is created. In its instantiation, shown in figure 4.2, a random salt is generated, and a key is created by combining the salt and master password then hashing. The salt is added to make it nearly impossible to derive the original master password. The salt is also stored with the file to ensure that when the file is opened, the same key is generated. When the file is later opened the UI will ask for a corresponding master password. If it is not corresponding, it will raise an exception, as seen in figure 4.3, and give a Qmessagebox warning at the user's front end. If it is correct the file will be decrypted, then desterilized.



```
    def save_as(self, path, masterpassword):
        buffer = dumps(self)
        key = Encrypt(masterpassword)
        encyped_data = key.encrypt(buffer)

        f = open(path, 'wb')
        f.write(encypted_data)
        f.close()
```

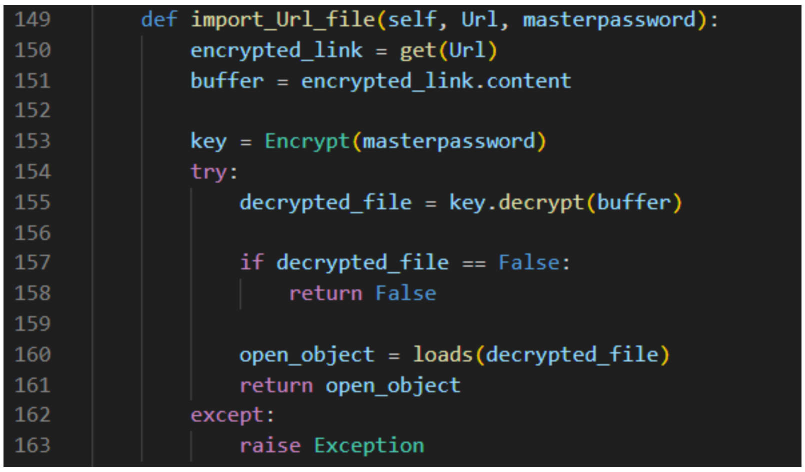
Figure 4.1: Saving a local files



```
    def __init__(self, master_password:str) -> None:
        self._password: bytes = master_password.encode()
        self._salt = secrets.token_bytes(16)
        self._key = self._derive_key(self._salt)


    def _derive_key(self, salt: bytes) -> bytes:
        """Derive a secret key from a given password and salt"""
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(), length=32, salt=salt,
            iterations=self._iterations, backend=self._backend)
        return b64e(kdf.derive(self._password))
```

Figure 4.2: Backend Encryption



```
    def import_Url_file(self, Url, masterpassword):
        encrypted_link = get(Url)
        buffer = encrypted_link.content

        key = Encrypt(masterpassword)
        try:
            decrypted_file = key.decrypt(buffer)

            if decrypted_file == False:
                return False

            open_object = loads(decrypted_file)
            return open_object
        except:
            raise Exception
```

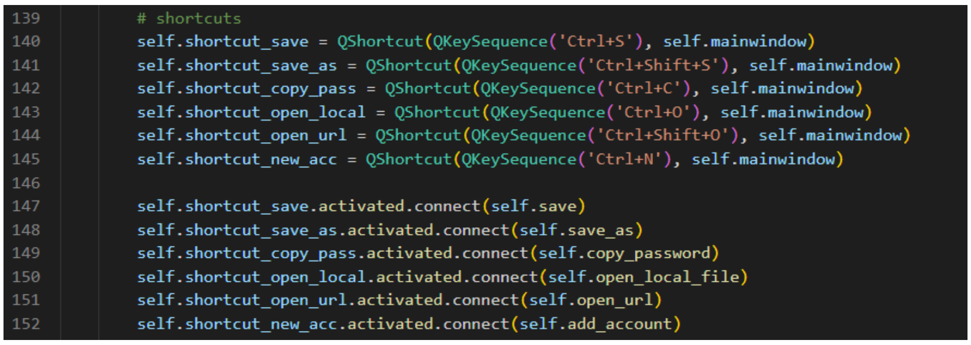
Figure 4.3: Opening remote files via URL

Now that the program is completed, according to the initial specifications, it was time to add our own features.

Added features

When the core functionalities of the program were completed, the team decided to implement some extra features to improve the user experience. One of the first features that were added are the shortcuts. The shortcuts were one of the easiest ideas to implement due to the pyqt5 library having made it very easy and convenient to add shortcuts to programs as seen in Figure 5.1.

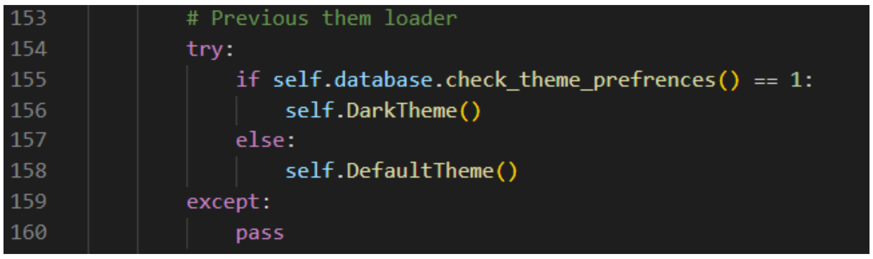
Another added feature was the Dark mode theme. For this method the team used the “QPalette” method to implement the theme but, after some further research, it was decided that the team will proceed with the “setStyleSheet. The fact that “QPalette” is not able to change the background while “setStyleSheet” method gives you the option to change almost everything about the user interface and its elements made the team choose the latter. The other reason is that the “QPalette” depends on the operating system which means that it is difficult to cater for other operating systems. The method created, to save the user’s theme preference, is in the data structure class. It essentially works by creating a text file that stores the name of the last activated theme. Now when the user opens the program, the program will read the text file upon instantiation. This can be seen in Figure 5.2 and the UI in dark mode can be seen in Figure 5.3.



```
        # shortcuts
        self.shortcut_save = QShortcut(QKeySequence('Ctrl+S'), self.mainwindow)
        self.shortcut_save_as = QShortcut(QKeySequence('Ctrl+Shift+S'), self.mainwindow)
        self.shortcut_copy_pass = QShortcut(QKeySequence('Ctrl+C'), self.mainwindow)
        self.shortcut_open_local = QShortcut(QKeySequence('Ctrl+O'), self.mainwindow)
        self.shortcut_open_url = QShortcut(QKeySequence('Ctrl+Shift+O'), self.mainwindow)
        self.shortcut_new_acc = QShortcut(QKeySequence('Ctrl+N'), self.mainwindow)

        self.shortcut_save.activated.connect(self.save)
        self.shortcut_save_as.activated.connect(self.save_as)
        self.shortcut_copy_pass.activated.connect(self.copy_password)
        self.shortcut_open_local.activated.connect(self.open_local_file)
        self.shortcut_open_url.activated.connect(self.open_url)
        self.shortcut_new_acc.activated.connect(self.add_account)
```

Figure 5.1: Shortcuts implementation



```
        # Previous them loader
        try:
            if self.database.check_theme_prefrences() == 1:
                self.DarkTheme()
            else:
                self.DefaultTheme()
        except:
            pass
```

Figure 5.2: Set the Theme according to preference

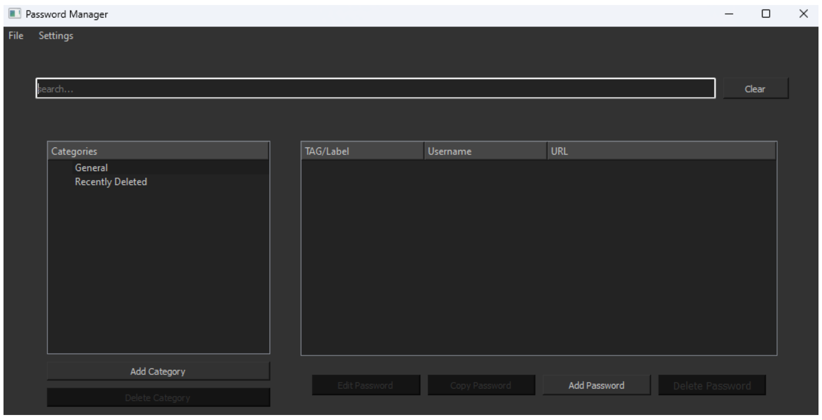


Figure 5.3: Dark Mode theme

Packaging

After the final touches were made to ensure that the program is working well, it was time to package the program for distribution. For this step, the "pyinstaller" library was used to make a distributable folder. Online converters have been utilized to change the format of the images used as the app icon. These formats are ".ico" and ".svg". These were added to the program in the spec files. Then, installForge was used to make an installer. Some settings had to be adjusted, such as adding an installer and making the user able to create a desktop shortcut icon. Lastly, the installer was built by pressing the build button and specifying the path where the installer is to be placed.

Discussion

What was learnt

While this report makes this project look easy enough to make, it was actually quite challenging. The main challenge was trying to learn a set of tasks in a short amount of time then implement it for the next showcase. Thus, time management was among the many skills gained. The process of building a program was also taught and using the design cycle was another vital skill. Teamwork and inter-team collaboration was also used during this project.

What to do next

In hindsight, more features could have been added if there was more time. In addition, we could have better controlled the expectation handling of the program by giving different error messages. This is because when the current program attempts to do a task and any exception is raised, only a given message is propagated to the front end user interface. Improving these features is what can be done to improve the program in version 2.0.