Software Engineering Project

Password Management System

Felipe Azzolino Varella

Prof. Dr. Andreas Schaad

Introduction

This project has as main objective the development of a system following the main processes of a software development, with emphasis in Requirements, Design, Implementation and Testing. In this case, the software is a Password Management System (also called PMS) that is, in essence, a system used to store and manage passwords safety, where each user has a master password that allows them to add and check if other services linked to PMS had their passwords leaked.

Hence, in a very simplified way, the system developed during this project can add users to our system, allowing it to edit its password, add and edit applications (that will also have passwords), check its data, and delete itself from the system. By an external call, we can verify if the system or application password was leaked by using the API from the website *HaveIBeenPwned?.*

Furthermore, we have another entity that is an administrator which can activate, deactivate, delete, or reset the user's password and, so, it is used for technical questions regarding the PMS users.

Everything was developed using Python 3.8.3 with the micro framework Flask in the IDE Pycharm 2020.1.1.

Requirements

In this part of this report, we will discuss about Requirements Engineering, i.e., what our Password Management System should do – including all the restrictions that it must be developed and work – according with what is required by the client.

User requirements and system requirements:

User requirements:

- Our software must be developed in Python.
- The software must be a PMS that, basically, must be able to add users, edit user's password, add user's services, edit user's services, delete user's services, check user's data, delete users from the system and have an entity that simulates an administrator, which can activate/deactivate users, edit the user's password or even delete users from the system.
- The user email must be unique and should have a valid format.
- Every user may have as data a master password (related to the system), if this password has already been leaked, the creation date of this user, the date of the last password change, a list of services (which may also have a password, creation date, the last time edited and if this password has already been leaked) and if the user is active or not. All this data should be saved in disk.
- The user's system password must be strong and safely stored in disk.
- The user cannot have more than one service with the same name stored in our system.
- In respect to the administrator, it must have a strong authentication factor and be able to manipulate the user's data.

System requirements:

- Every request (e.g. add user, edit user's password etc.) must be made via JSON/REST calls with the specific method.
- Every response should return a JSON containing if the process was successful and a message explaining.
- All user's data should be stored in disk in JSON format.

- The system's password should follow some rules that must be saved in disk in JSON format.
- The administrator's authentication data should be saved in disk in JSON format.
- The stored data should be dynamic (i.e., should be updated immediately).

Functional and Non-functional requirements:

Furthermore, we can write all these requirements in functional (FR) or non-functional (NFR) requirements. Some non-functional requirements (e.g. "the user's system password must be strong and safely stored in disk" – written in the user's requirements) can be expanded into functional requirements that can be easily converted into programming language latter.

Id	Classification	Description
1	Non-Functional	The programming language must be Python.
2	Non-Functional	The user email should be unique and have a valid format.
2.1	Functional	The user must have and "@" and a "." after that.
3	Non-Functional	The password must be strong.
3.1	Functional	The password length should be between 8 and 64 characters.
3.2	Functional	The password must use only English characters (i.e., á, ö, ß are not allowed).
3.3	Functional	The password must have (by default) the following requirements: a lower case letter, a upper case letter, a number and a special character (i.e., !@#$%^&*()?/.<>,{}[]:;|\`'_-+=).
4	Functional	All password rules should be stored in disk in JSON format.
5	Non-Functional	The passwords should be safely stored in disk.
5.1	Functional	The passwords must be stored in disk using bcrypt.
6	Non-Functional	The administrator, must have a strong authentication factor.
6.1	Functional	The administrator should have one login and two passwords for authentication (by default we have admin_login: "admin@admin", admin_pw1: "ABC1234", admin_pw2: "DEF5678")
7	Non-Functional	The system must be able to manipulate all user data.
7.1	Functional	Function "add_user()" must be provided.
7.2	Functional	Function "edit_password()" must be provided.
7.3	Functional	Function "add_service()" must be provided.
7.4	Functional	Function "edit_service()" must be provided.
7.5	Functional	Function "del_service()" must be provided.

7.6	Functional	Function “check_user()” must be provided.
7.7	Functional	Function “del_user()” must be provided.
8	Non-Functional	The administrator must be able to edit some user data.
8.1	Functional	Function “admin_option()” must be provided.
9	Non-Functional	The administrator must be able to edit its data whenever necessary.
9.1	Functional	Function “admin_edit()” must be provided.
9.2	Functional	The administrator’s authenticators should be stored in disk in JSON format.
10	Functional	Every request must be made via JSON/REST calls.
11	Functional	All user’s data should be stored in disk in JSON format.
12	Non-Functional	The stored data should be dynamic.
12.1	Functional	We should overwrite the file every time we edit any user’s data.
13	Functional	Every response should return a JSON containing if the process was successful and a message explaining

Table 1: *All functional and non-functional requirements from our Password Management System*

User Stories:

Lastly, we can write it down two simple user stories related to the user and the administrator of the Password Management System to facilitate the understanding of each entity.

User Story:	Functionality: Manipulate all user data
As an...	System user
I want...	to add services, edit, check and delete my data
so that...	I can manipulate my data and verify if my service's password was leaked.

Table 2: *User Story of the system user.*

User Story:	Functionalities: Admin manipulation of user data and verificators
As an...	System administrator
I want...	to edit the user's data and modify my verificators
so that...	I can administrate the system whenever desired.

Table 3: *User Story of the system administrator.*

Design

In this part of the report, we will discuss about the Software Design, i.e., all the pre-development planning that is used to model the system, representing different views of it. Based in the Unified Modeling Language (UML), we will use diagrams notation to help in understanding how our Password Management System works.

In this work we will use 4 diagrams: Case diagram, Sequence diagram, State machine diagram and Class diagrams, where for each, an example using our Password Management System, its functionalities or its classes will be given.

<u>**Case diagrams**</u>:

Case diagrams, at its simplest, is a behavior diagram used to represent the interaction between users and system. Therefore, this represents how the user – that can be also a service – and the administrator interacts with our Password Management System (PMS). Below is shown the case diagram for our Software:

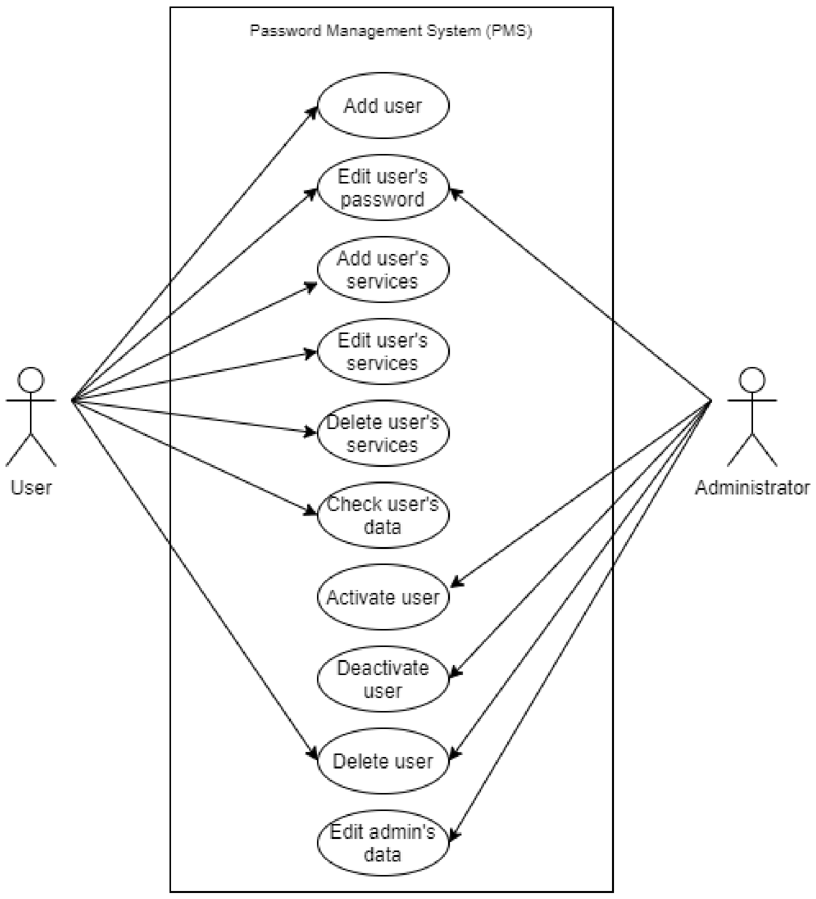


Diagram 1: *Case diagram for the PMS.*

<u>Class Diagram</u>:

This is a structure diagrams used to describe the structure of software by showing the system's classes with their attributes, methods, and, if possible, the relationships among its instances.

In our system, we will work with 3 classes:

- User: Used to create and manipulate all user data.
- Password: A static class (i.e., it only contains static methods and, therefore, do not contain attributes) that is used by User's class methods to validate its password.
- Admin: That is a static class which its methods are used to edit the User's instances data, validate if the administrator data is correct and edit the admin verifiers (i.e., login and passwords).

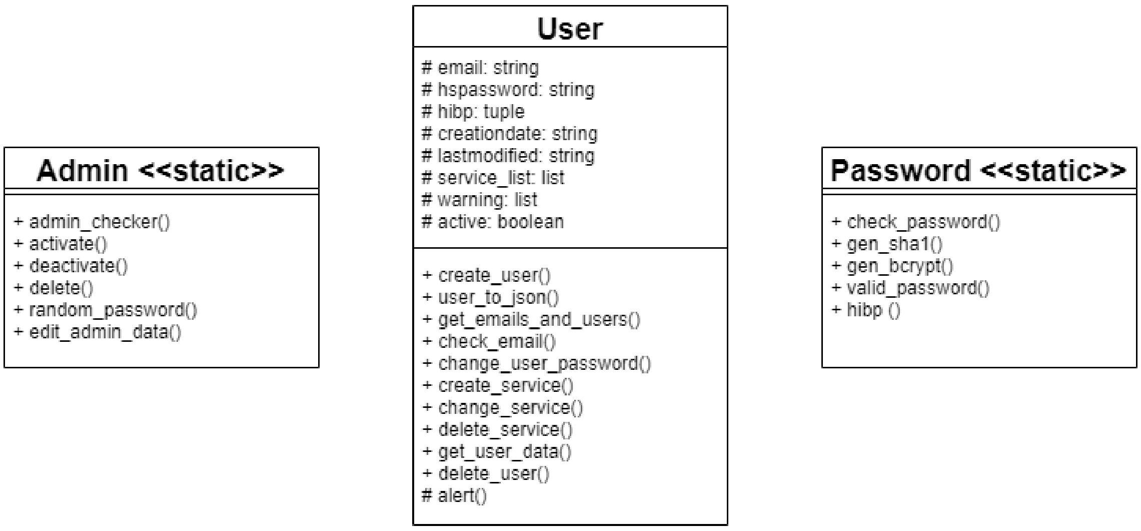


Diagram 2: *Class diagrams for the PMS.*

Sequence Diagram:

This is a behavior diagram which its use shows the interactions between objects in time, allowing to understand how a functionality from a system works chronologically. For our PMS, the functionality responsible for adding user in our system will be represented by this UML diagram as shown below:

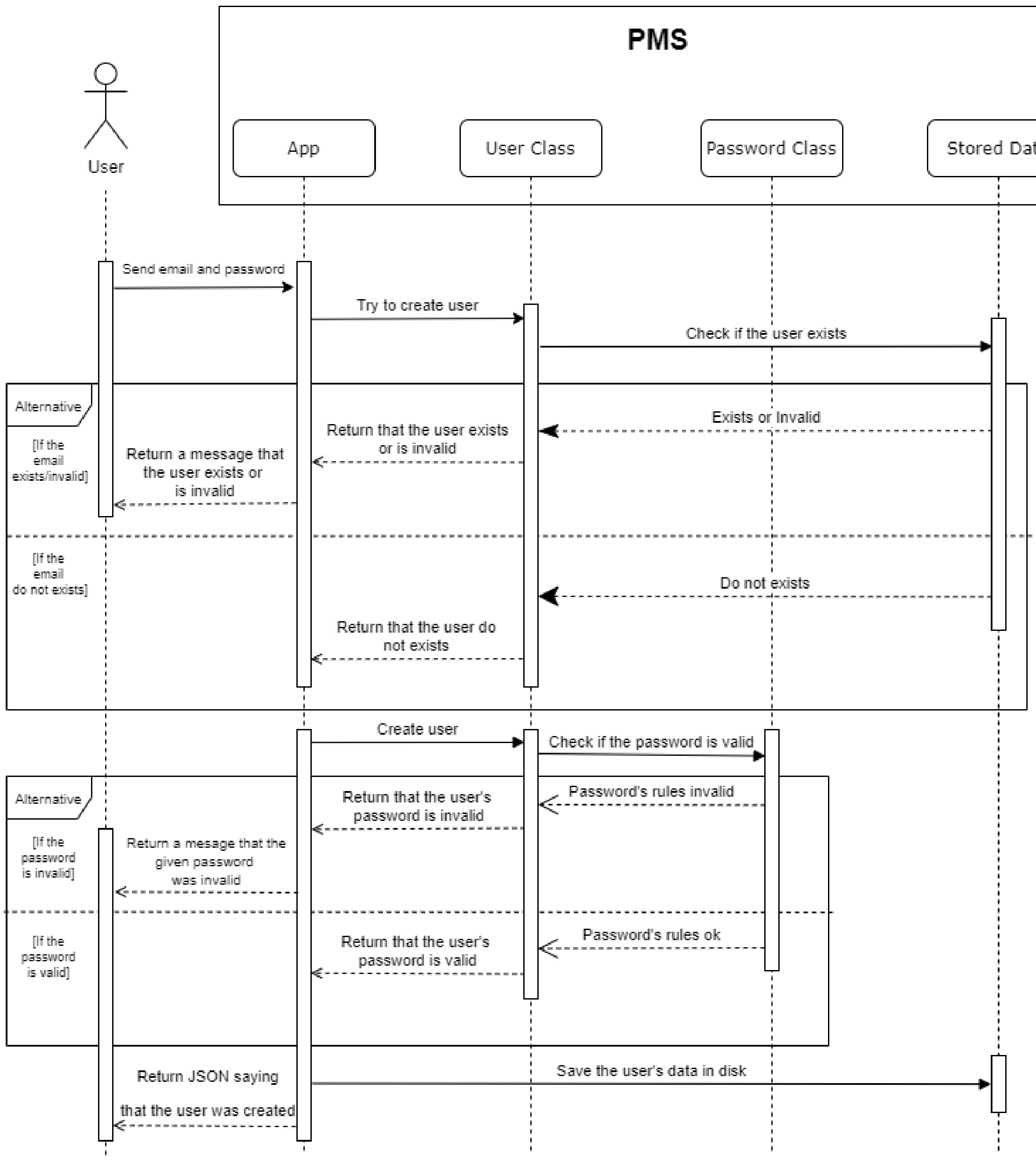

PMS
User
App
User Class
Password Class
Stored Data
Send email and password
Try to create user
Check if the user exists
Alternative
[If the email exists/invalid]
Exists or Invalid
Return that the user exists or is invalid
Return a message that the user exists or is invalid
[If the email do not exists]
Do not exists
Return that the user do not exists
Create user
Check if the password is valid
Alternative
[If the password is invalid]
Password's rules invalid
Return that the user's password is invalid
Return a mesage that the given password was invalid
[If the password is valid]
Password's rules ok
Return that the user's password is valid
Save the user's data in disk
Return JSON saying that the user was created

Diagram 3: *Sequence diagram for the PMS's functionality add user.*

<u>Activity diagram</u>:

This behavior diagram is used to represents the workflow (step by step) of actions and activities from a system. This diagram, basically, shows the activities involved in a process. In our PSM example shown below, we represent how the functionality to add user's applications (e.g., Facebook, Instagram etc.) can be expressed via this UML diagram

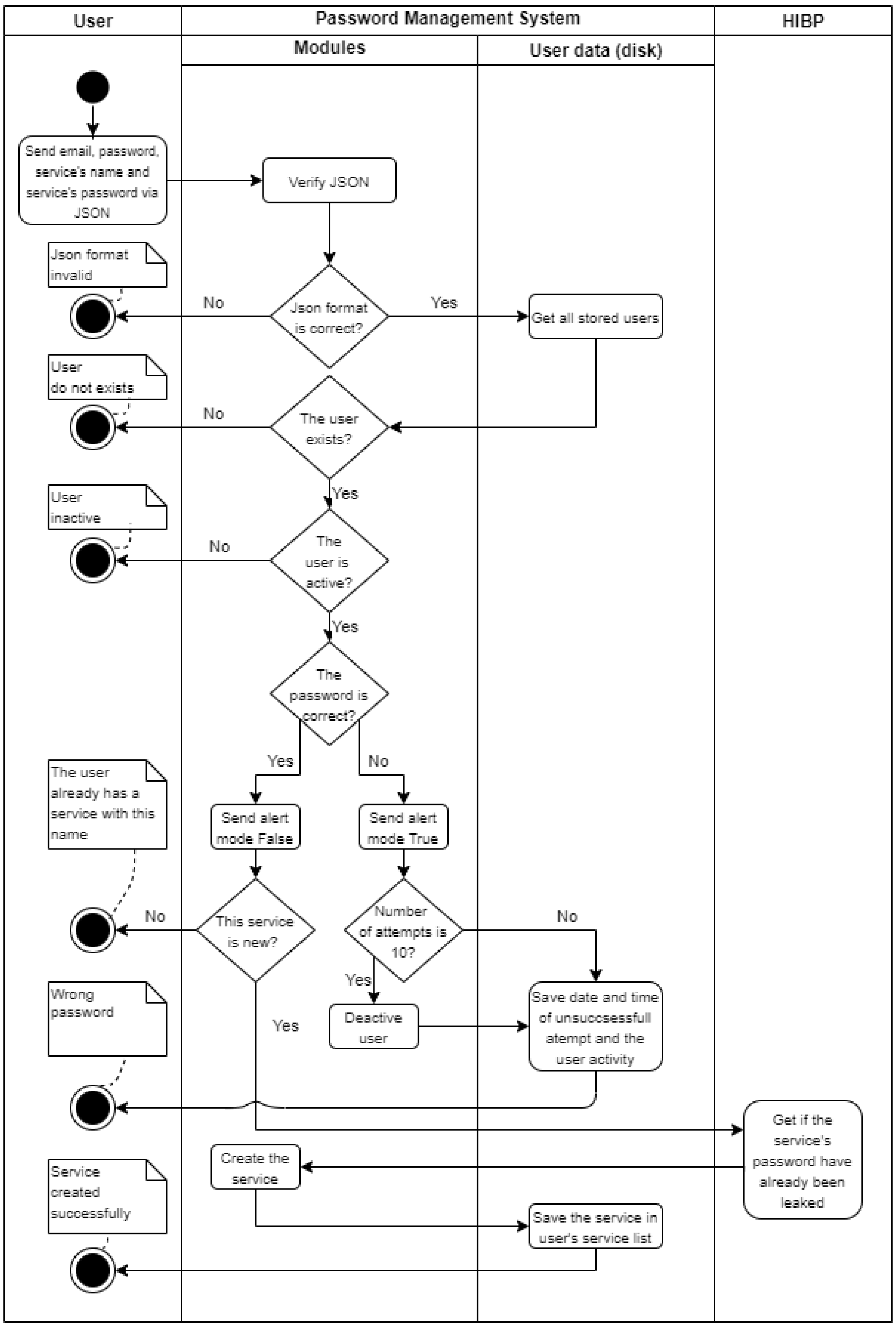

User
Password Management System
HIBP
Modules
User data (disk)
Send email, password, service's name and service's password via JSON
Verify JSON
Json format invalid
No
Json format is correct?
Yes
Get all stored users
User do not exists
No
The user exists?
Yes
User inactive
No
The user is active?
Yes
The password is correct?
Yes
No
The user already has a service with this name
Send alert mode False
Send alert mode True
No
This service is new?
Number of attempts is 10?
No
Yes
Wrong password
Yes
Deactive user
Save date and time of unsuccessfull atempt and the user activity
Get if the service's password have already been leaked
Create the service
Service created successfully
Save the service in user's service list

Diagram 4: *Sequence diagram for the PMS's functionality add service.*

State Machine Diagram:

This is the last behavior diagram that will be explored in this report. This type of diagram is used to model the behavior of an object in response to external events. In our PMS case, we can use this type of diagram with our object "user". Our object has only two possible states: active that is when the user can access/modify its data (it is set active by default when created) and inactive that is when the user cannot access/modify its data (this mode is set when the user enter the incorrect password 10 times in a row). Thus, this can be represented in this UML diagram as:

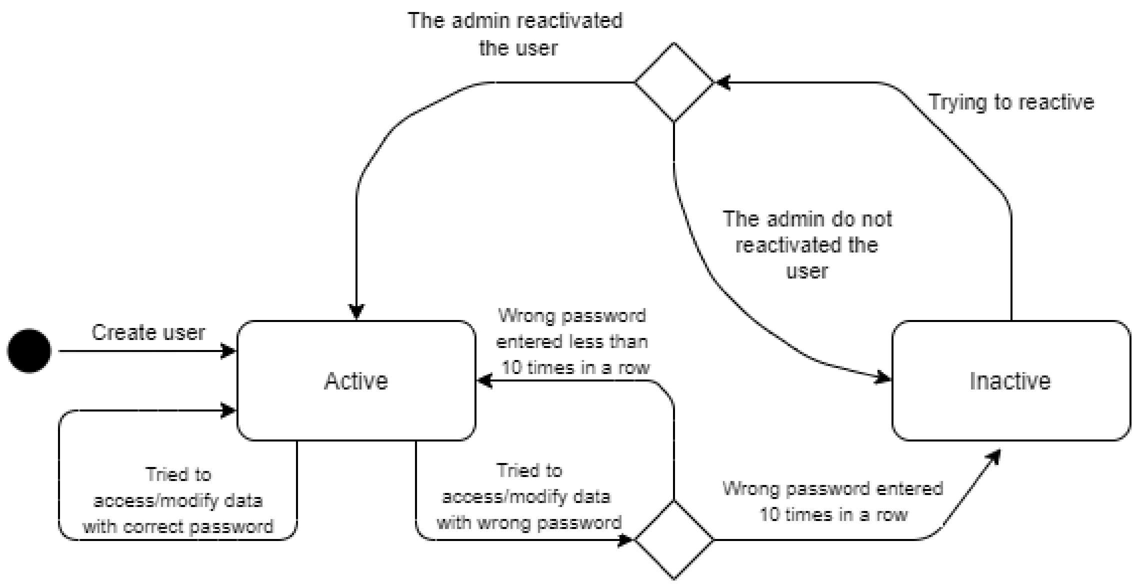
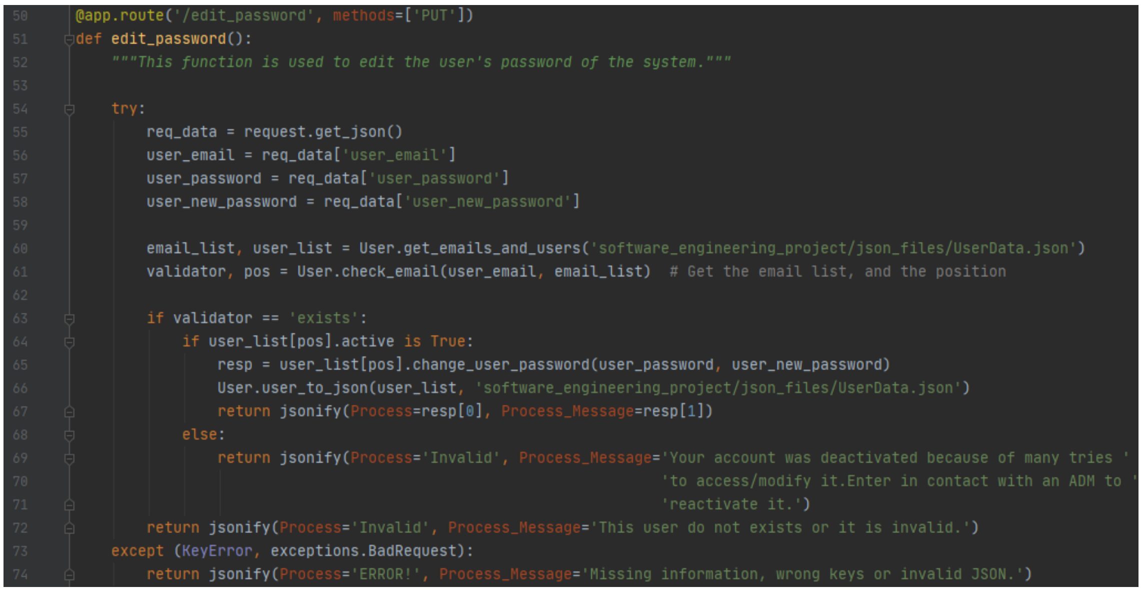

The admin reactivated the user
Trying to reactive
The admin do not reactivated the user
Create user
Active
Wrong password entered less than 10 times in a row
Inactive
Tried to access/modify data with correct password
Tried to access/modify data with wrong password
Wrong password entered 10 times in a row

Diagram 5: *State machine diagram for the PMS's user.*

Implementation

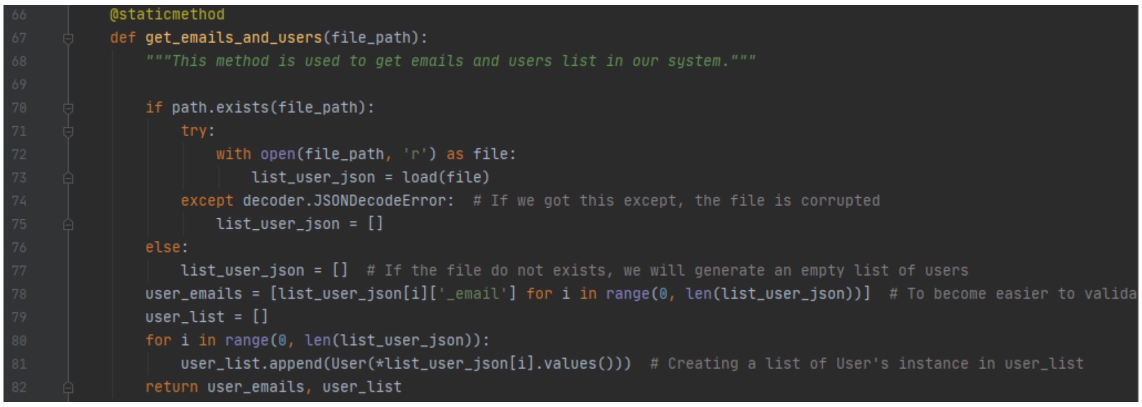
In this part of the report, we will discuss about the functionalities developed in Python for the PMS project. Due to the limited number of pages, we will choose one functionality and explore all methods called by this and then, in the next topic (i.e. "Testing"), we will demonstrate the unit test for the most important methods used. Therefore, we will explore the functionality responsible for <u>editing the user's password</u> – called *change_password()* – that can be found in the module "*app.py*" in the directory "*pms_softwareengineering*" and also in the image below:



```python
50  @app.route('/edit_password', methods=['PUT'])
51  def edit_password():
52      """This function is used to edit the user's password of the system."""
53  
54      try:
55          req_data = request.get_json()
56          user_email = req_data['user_email']
57          user_password = req_data['user_password']
58          user_new_password = req_data['user_new_password']
59  
60          email_list, user_list = User.get_emails_and_users('software_engineering_project/json_files/UserData.json')
61          validator, pos = User.check_email(user_email, email_list)  # Get the email list, and the position
62  
63          if validator == 'exists':
64              if user_list[pos].active is True:
65                  resp = user_list[pos].change_user_password(user_password, user_new_password)
66                  User.user_to_json(user_list, 'software_engineering_project/json_files/UserData.json')
67                  return jsonify(Process=resp[0], Process_Message=resp[1])
68              else:
69                  return jsonify(Process='Invalid', Process_Message='Your account was deactivated because of many tries '
70                                                                    'to access/modify it.Enter in contact with an ADM to '
71                                                                    'reactivate it.')
72          return jsonify(Process='Invalid', Process_Message='This user do not exists or it is invalid.')
73      except (KeyError, exceptions.BadRequest):
74          return jsonify(Process='ERROR!', Process_Message='Missing information, wrong keys or invalid JSON.')
```

First of all, inside the block "try", we request a JSON from the user containing its email, password and the new password with the keys "user_email", "user_password" and "user_new_password" (lines 55-58 – *app.py*), respectively. Due to this try-exception block, if the user do not insert a JSON containing the keys with the respective values or if this structure has an invalid format (e.g., missing a comma) an error in JSON format will be returned to the user explaining that.

If the JSON format was correct, we will retrieve from the disk (as seen in line 60 – *app.py*) a list containing all users and another one containing only its emails (this one to make easier the search for users) registered in our system. This method can be explicitly seen below:

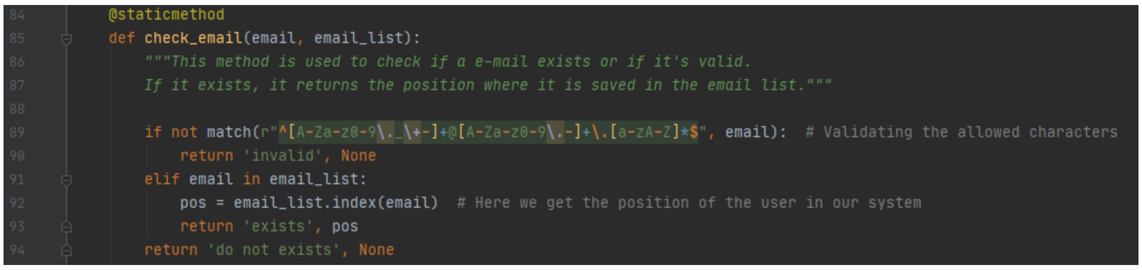


```python
    @staticmethod
    def get_emails_and_users(file_path):
        """This method is used to get emails and users list in our system."""

        if path.exists(file_path):
            try:
                with open(file_path, 'r') as file:
                    list_user_json = load(file)
            except decoder.JSONDecodeError:  # If we got this except, the file is corrupted
                list_user_json = []
        else:
            list_user_json = []  # If the file do not exists, we will generate an empty list of users
        user_emails = [list_user_json[i]['_email'] for i in range(0, len(list_user_json))]  # To become easier to valida
        user_list = []
        for i in range(0, len(list_user_json)):
            user_list.append(User(*list_user_json[i].values()))  # Creating a list of User's instance in user_list
        return user_emails, user_list
```

This way, the first thing that this method do is verify if exists a file with the given name for, then, try to read this. If the file exists, we need to make another verification but, this time, trying to verify if the file is in JSON format and if this is not corrupted (i.e., if the JSON objects saved was saved in the correct form). After that, we generate a list of users in JSON format (lines 73 or 75 – *user.py*), i.e., a list that each element is a dictionary containing the user's attributes.

Then, we generate a list of emails with the list of users in JSON by using a list generator (line 78 – *user.py*) with every value of '_email' iterating over every dictionary in the list and, also, a list which every element is an instance of the class User, i.e., we generate a list of User's objects (line 81 – *user.py*). Finally, these lists (email list – named "user_emails" – and list of User's objects – named "user_list" – are returned to the module *app.py*).

After this, another method (*check_emai()l*) is called (line 61 – *app.py*).

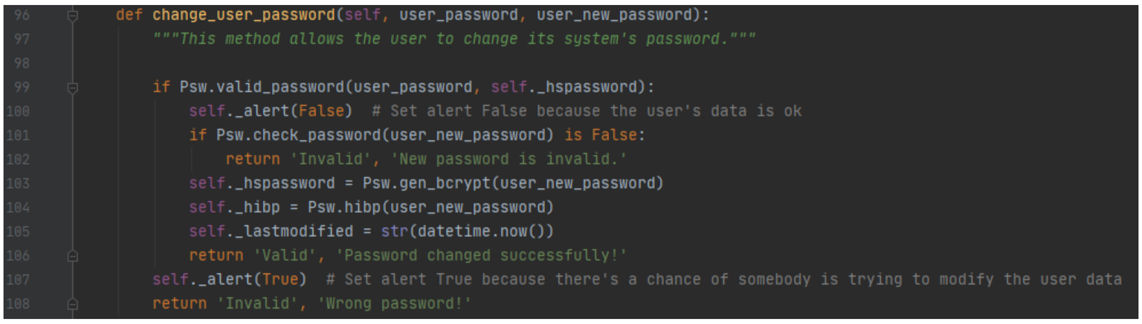


```
84      @staticmethod
85      def check_email(email, email_list):
86          """This method is used to check if a e-mail exists or if it's valid.
87          If it exists, it returns the position where it is saved in the email list."""
88
89          if not match(r"^[A-Za-z0-9\._\+-]+@[A-Za-z0-9\.-]+\.[a-zA-Z]*$", email):  # Validating the allowed characters
90              return 'invalid', None
91          elif email in email_list:
92              pos = email_list.index(email)  # Here we get the position of the user in our system
93              return 'exists', pos
94          return 'do not exists', None
```

This method receives two parameters: "email" and "email_list". The first one is the email that is given by the user (i.e., the email which its password needs to be changed) and the second one is the list of emails registered in our system (generated in line 60 – *app.py*). This method basically verifies if the given email is valid, i.e., if it contains characters from A-Z, a-z, numbers, special characters, a "@" and a "." after that (line 89 – *user.py*) for, then, verify if the emails exists (returning the position of the user in the list) or if it do not exist, returning this to the application.

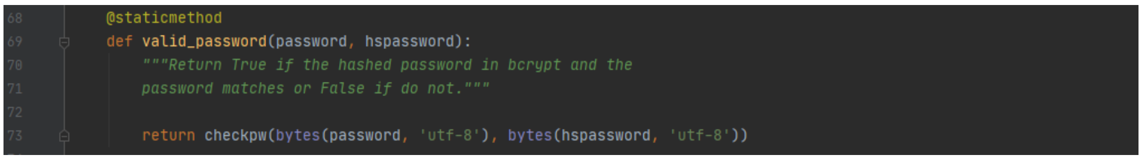
Now, we can finally procced to the attempt of changing the user's password (line 63 – *app.py*). If the given email does not exists or if its format is invalid, a message in JSON format will be returned to the user explaining that. Else, we will procced and try to verify if the user is active or not by analyzing the attribute "active"; if this attribute is *True*, that means that the user is active and, therefore, it is able to procced. Else, if this attribute is *False*, that means that the user was deactivated because its account was in danger and, so, a message in JSON format will be return informing that.

Assuming that the user is active, we can procced to the process of password change. For that, the method "*change_user_password()*" (line 65 – *app.py*) is called.



```python
96      def change_user_password(self, user_password, user_new_password):
97          """This method allows the user to change its system's password."""
98
99          if Psw.valid_password(user_password, self._hspassword):
100             self._alert(False)  # Set alert False because the user's data is ok
101             if Psw.check_password(user_new_password) is False:
102                 return 'Invalid', 'New password is invalid.'
103             self._hspassword = Psw.gen_bcrypt(user_new_password)
104             self._hibp = Psw.hibp(user_new_password)
105             self._lastmodified = str(datetime.now())
106             return 'Valid', 'Password changed successfully!'
107         self._alert(True)  # Set alert True because there's a chance of somebody is trying to modify the user data
108         return 'Invalid', 'Wrong password!'
```

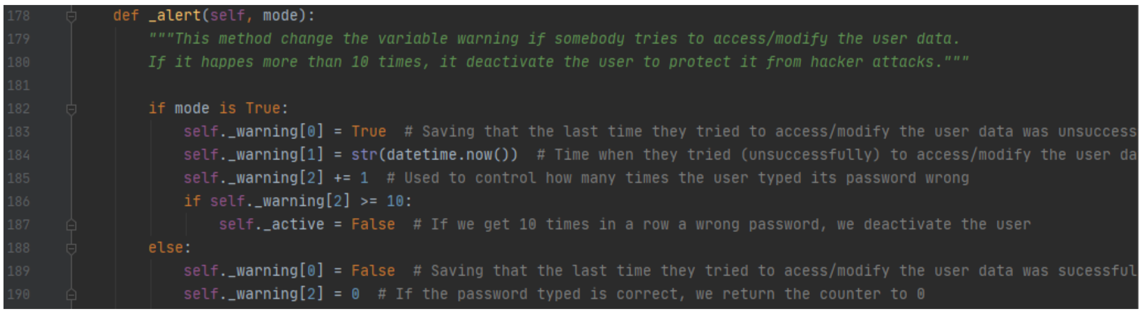
This method receives two parameters: “user_password”, that is the actual user password; and “user_new_password”, that is the desired new password. Moreover, this method calls another one (“*valid_password()”* in the class Password in the file *password.py*).



```python
    @staticmethod
    def valid_password(password, hspassword):
        """Return True if the hashed password in bcrypt and the
        password matches or False if do not."""

        return checkpw(bytes(password, 'utf-8'), bytes(hspassword, 'utf-8'))
```

This method is quite simple and was created to simplify the notation. This is used to verify if the given password can be validated with the hashed password stored in the user data. For that, we use the external method “*checkpw()*” from the library “*bcrypt*” that returns *True* if the password is correct or *False* if not.

After using this method, we can conclude if the given password is correct or not. In both cases (line 100 for success and line 107 for unsuccess – *user.py*), we send an alert to the system via the method “*_alert()*” (also saved in the User class) that can be seen below:



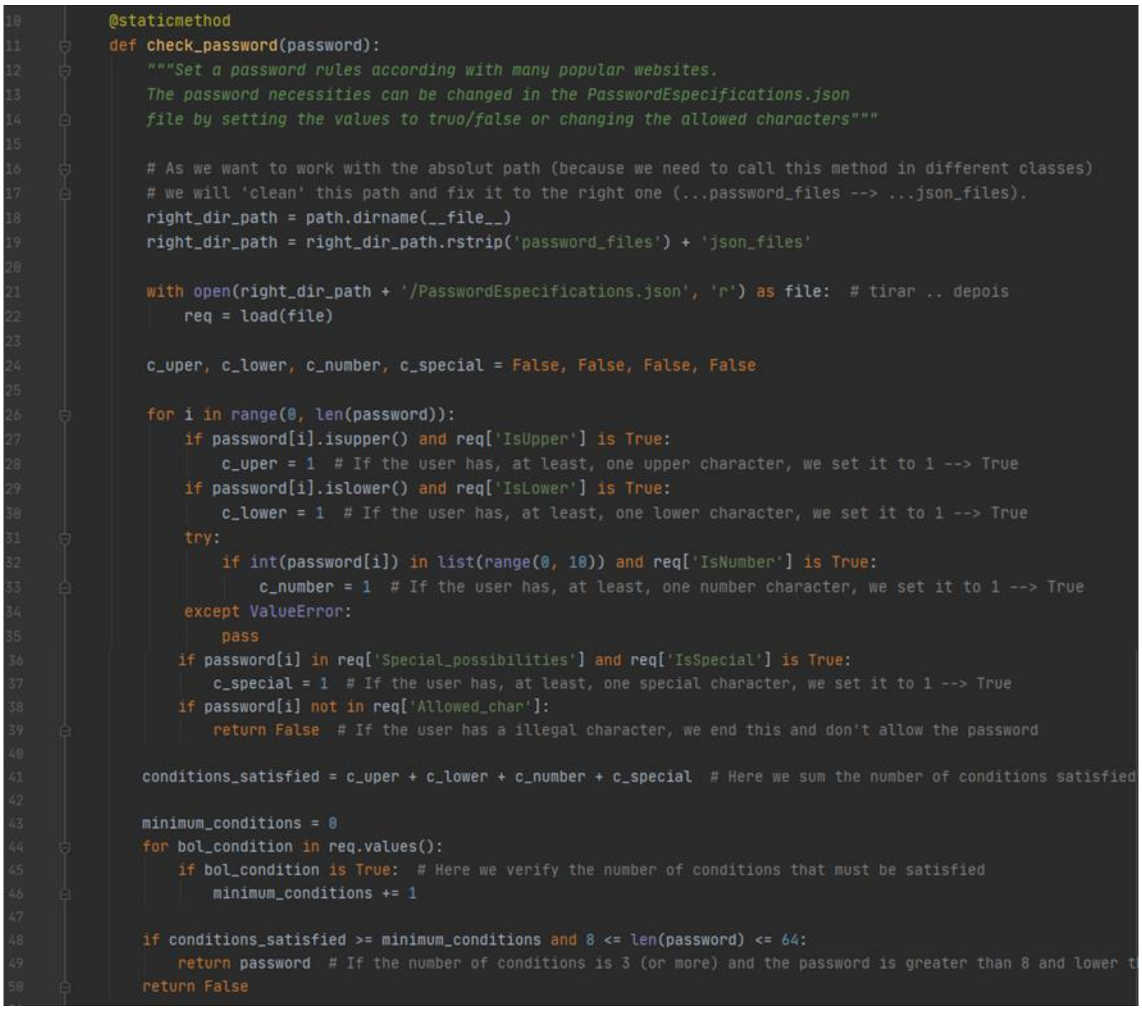
```python
    def _alert(self, mode):
        """This method change the variable warning if somebody tries to access/modify the user data.
        If it happes more than 10 times, it deactivate the user to protect it from hacker attacks."""

        if mode is True:
            self._warning[0] = True  # Saving that the last time they tried to access/modify the user data was unsuccess
            self._warning[1] = str(datetime.now())  # Time when they tried (unsuccessfully) to access/modify the user da
            self._warning[2] += 1  # Used to control how many times the user typed its password wrong
            if self._warning[2] >= 10:
                self._active = False  # If we get 10 times in a row a wrong password, we deactivate the user
        else:
            self._warning[0] = False  # Saving that the last time they tried to acess/modify the user data was sucessful
            self._warning[2] = 0  # If the password typed is correct, we return the counter to 0
```

This method receives only one parameter: “mode”. This parameter can assume only Boolean values which:

- *True* represents that the user is in danger and, so, we will set the warning mode to *True* (signalizing that the last attempt to access the user data was not with the correct password), saving in String format the date when they tried to modify the user data and adding 1 to the counter (used to represent the number in a row they tried to access the user data for, then, deactivate the user if this number reaches 10);

- *False* represents that the user's attempt to access its data was successfully and, so, the last person that manipulated the user data was, probably, itself. This way, we will set the warning mode to *False* and reset the counter (i.e., set it to 0).

Now, returning to talk about the "*change_password()*'' method, if the written password was incorrect (i.e., do not matches with the stored password), we will set the alert mode to *True* and will return a message for the user informing that the given password is wrong. Else, we will set the alert mode to *False* and will procced to the Password's method "*check_password()*".



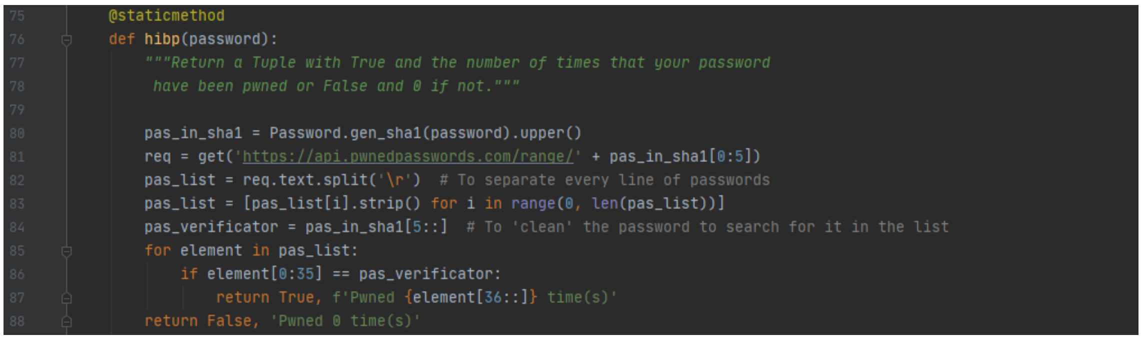
```python
    @staticmethod
    def check_password(password):
        """Set a password rules according with many popular websites.
        The password necessities can be changed in the PasswordEspecifications.json
        file by setting the values to truo/false or changing the allowed characters"""

        # As we want to work with the absolut path (because we need to call this method in different classes)
        # we will 'clean' this path and fix it to the right one (...password_files --> ...json_files).
        right_dir_path = path.dirname(__file__)
        right_dir_path = right_dir_path.rstrip('password_files') + 'json_files'

        with open(right_dir_path + '/PasswordEspecifications.json', 'r') as file:  # tirar .. depois
            req = load(file)

        c_uper, c_lower, c_number, c_special = False, False, False, False

        for i in range(0, len(password)):
            if password[i].isupper() and req['IsUpper'] is True:
                c_uper = 1  # If the user has, at least, one upper character, we set it to 1 --> True
            if password[i].islower() and req['IsLower'] is True:
                c_lower = 1  # If the user has, at least, one lower character, we set it to 1 --> True
            try:
                if int(password[i]) in list(range(0, 10)) and req['IsNumber'] is True:
                    c_number = 1  # If the user has, at least, one number character, we set it to 1 --> True
            except ValueError:
                pass
            if password[i] in req['Special_possibilities'] and req['IsSpecial'] is True:
                c_special = 1  # If the user has, at least, one special character, we set it to 1 --> True
            if password[i] not in req['Allowed_char']:
                return False  # If the user has a illegal character, we end this and don't allow the password

        conditions_satisfied = c_uper + c_lower + c_number + c_special  # Here we sum the number of conditions satisfied

        minimum_conditions = 0
        for bol_condition in req.values():
            if bol_condition is True:  # Here we verify the number of conditions that must be satisfied
                minimum_conditions += 1

        if conditions_satisfied >= minimum_conditions and 8 <= len(password) <= 64:
            return password  # If the number of conditions is 3 (or more) and the password is greater than 8 and lower t
        return False
```

Although this is a quite big method, the way it works is very simple: first of all we receive a password as the only parameter and verify it according with what is saved in our file "*PasswordEspecifications.json*", that contains as keys "IsUpper", "IsLower", "IsNumber", "IsSpecial", "Special_possibilities" and "Allowed_char". The first 4 first keys, contains as values a boolean that shows if its necessary to have this kind of value (e.g., if the "IsLower" is *False*, that means that is not necessary to have a lower case character in our password, if the "IsSpecial" is *True*, that means that we need to have a special character and so on), the

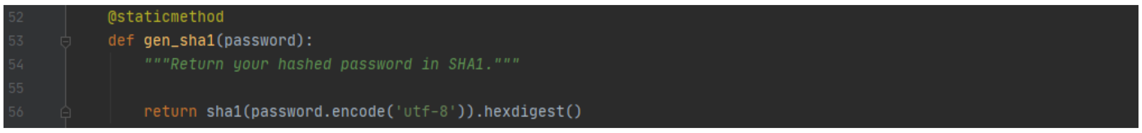
“Special_possibilities” key contains what is considered a special characters and “Allowed_char” key contains all the allowed characters.

Furthermore, we just check if the number of conditions was satisfied (line 44-46 – *password.py*), i.e., if the password has everything that was required, for, finally, verify if the password is between 8 and 64 to allow this password. If these conditions were satisfied, we will return the password; else, we will return False (signalizing that this process failed).

At last, if this process was successful, we finally will be able to edit the user’s password, save when this change happened and update the HIBP for the new password with the method *“hibp()”* (from the class Password in *password.py*).



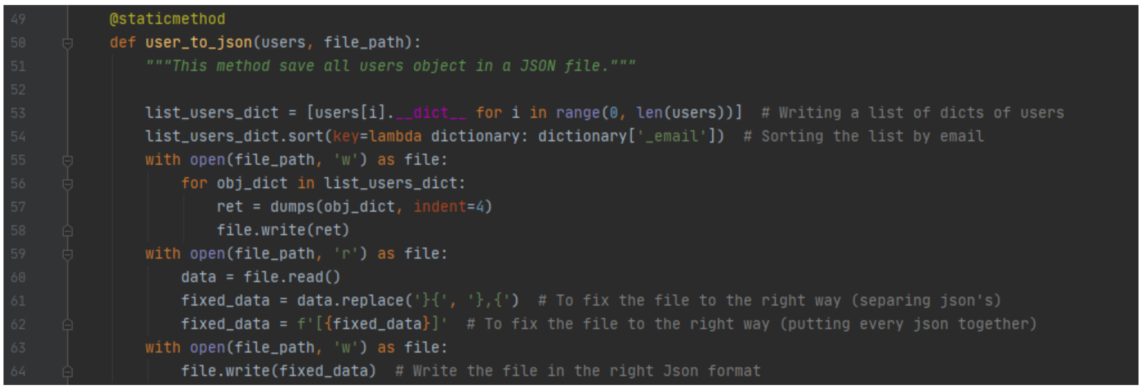
```python
    @staticmethod
    def hibp(password):
        """Return a Tuple with True and the number of times that your password
         have been pwned or False and 0 if not."""

        pas_in_sha1 = Password.gen_sha1(password).upper()
        req = get('https://api.pwnedpasswords.com/range/' + pas_in_sha1[0:5])
        pas_list = req.text.split('\r')  # To separate every line of passwords
        pas_list = [pas_list[i].strip() for i in range(0, len(pas_list))]
        pas_verificator = pas_in_sha1[5::]  # To 'clean' the password to search for it in the list
        for element in pas_list:
            if element[0:35] == pas_verificator:
                return True, f'Pwned {element[36::]} time(s)'
        return False, 'Pwned 0 time(s)'
```



```python
    @staticmethod
    def gen_sha1(password):
        """Return your hashed password in SHA1."""

        return sha1(password.encode('utf-8')).hexdigest()
```

This method calls the method *gen_sha1* (also shown above) that generates the given password in SHA1 (via the method “*sha1()”* from the library “*hashlib*”) for, then, send a get request in an external API that provides us the number of times that a password has already been leaked. With that, we can update the HIBP data of the new password with *True* (if this password has already been leaked) and the number of times or *False* and 0.

Now, the user calls the last method “*user_to_json()*” (line 65 – *app.py*) also saved in the *user.py* file.



```python
    @staticmethod
    def user_to_json(users, file_path):
        """This method save all users object in a JSON file."""

        list_users_dict = [users[i].__dict__ for i in range(0, len(users))]  # Writing a list of dicts of users
        list_users_dict.sort(key=lambda dictionary: dictionary['_email'])  # Sorting the list by email
        with open(file_path, 'w') as file:
            for obj_dict in list_users_dict:
                ret = dumps(obj_dict, indent=4)
                file.write(ret)
        with open(file_path, 'r') as file:
            data = file.read()
            fixed_data = data.replace('}{', '},{')  # To fix the file to the right way (separing json's)
            fixed_data = f'[{fixed_data}]'  # To fix the file to the right way (putting every json together)
        with open(file_path, 'w') as file:
            file.write(fixed_data)  # Write the file in the right Json format
```

This method receive 2 parameters: “users”, that is a list containing all the users and “file_path”, that the path where is desired to save this users. This method basically generates a list of dictionaries with every user and save this in the correct JSON format (i.e., separating every user in JSON with comma and delimitating all them with brackets).

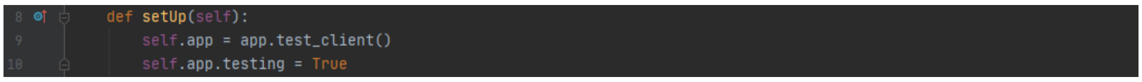
After all that, the user should receive a message in JSON format saying that the process was successfully and, so, the password was updated.

Testing

In this part of the report, we will discuss about the unit tests created for the PMS project. As explained in the last topic, here we will explain the unit test created for the most important methods used in the last explored functionality, i.e., "*edit_password()*".

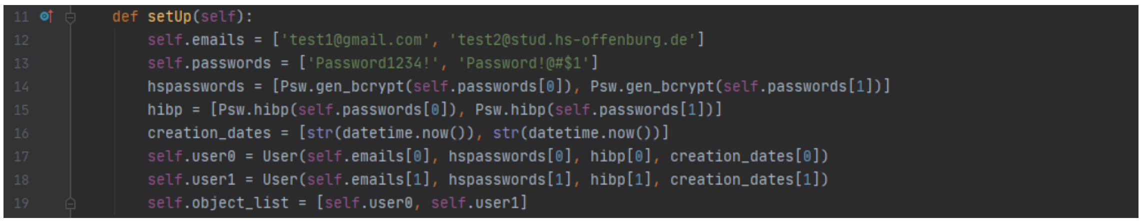
First of all, as we will use this for almost all the tests, we will show the *setUp()* method used for the User, Password and app tests. Before continuing:

For the app we have:



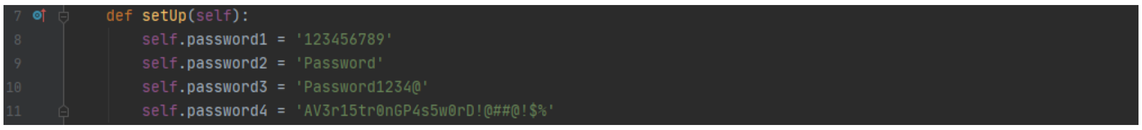
```python
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True
```

For the User class we have:



```python
    def setUp(self):
        self.emails = ['test1@gmail.com', 'test2@stud.hs-offenburg.de']
        self.passwords = ['Password1234!', 'Password!@#$1']
        hspasswords = [Psw.gen_bcrypt(self.passwords[0]), Psw.gen_bcrypt(self.passwords[1])]
        hibp = [Psw.hibp(self.passwords[0]), Psw.hibp(self.passwords[1])]
        creation_dates = [str(datetime.now()), str(datetime.now())]
        self.user0 = User(self.emails[0], hspasswords[0], hibp[0], creation_dates[0])
        self.user1 = User(self.emails[1], hspasswords[1], hibp[1], creation_dates[1])
        self.object_list = [self.user0, self.user1]
```

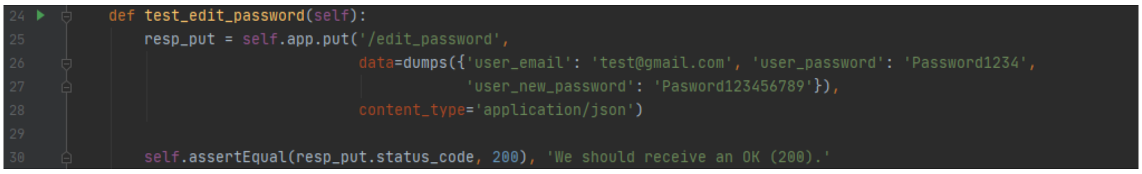
For the Password class we have:



```python
    def setUp(self):
        self.password1 = '123456789'
        self.password2 = 'Password'
        self.password3 = 'Password1234@'
        self.password4 = 'AV3r15tr0nGP4s5w0rD!@##@!$%'
```

Now we are able to test the chosen methods.

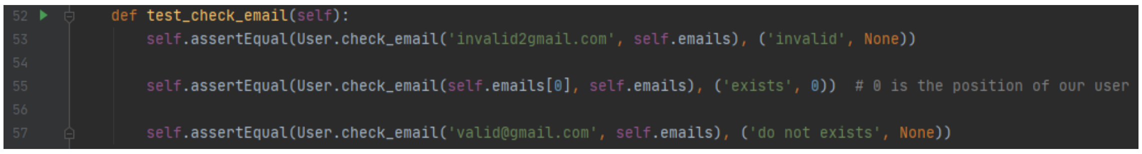
1) edit_password() (created in the module *app.py*).



```python
    def test_edit_password(self):
        resp_put = self.app.put('/edit_password',
                                data=dumps({'user_email': 'test@gmail.com', 'user_password': 'Password1234',
                                            'user_new_password': 'Pasword123456789'}),
                                content_type='application/json')

        self.assertEqual(resp_put.status_code, 200), 'We should receive an OK (200).'
```

In this test, our only objective is verifying if we are receiving the correct status code for our verb (PUT). In this case, we are testing if we receive the expected status code, i.e., 200 (OK).

2) *check_email()* (created in the class User in the module *user.py*):


```python
def test_check_email(self):
    self.assertEqual(User.check_email('invalid2gmail.com', self.emails), ('invalid', None))

    self.assertEqual(User.check_email(self.emails[0], self.emails), ('exists', 0))  # 0 is the position of our user

    self.assertEqual(User.check_email('valid@gmail.com', self.emails), ('do not exists', None))
```

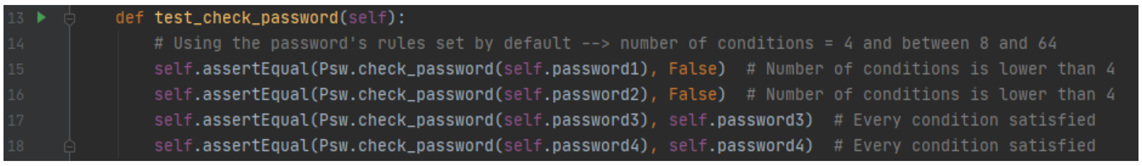
In our example, we are testing the 3 possible returns:

- In the first case, we test an invalid email (because the missing "@") expecting to receive a tuple containing that this is invalid and *None* (because the email is invalid and, therefore, it is not contains in our list).

- In the second case, we test a valid email that is also in our list, hoping to receive a tuple containing that it is exist (so it is valid) and 0 (its position in the list).

- In the third case, we test a valid email that is not in the list, so we expect that our tuple contains that this do not exists (but is valid) and *None* (because this email is not in our list).

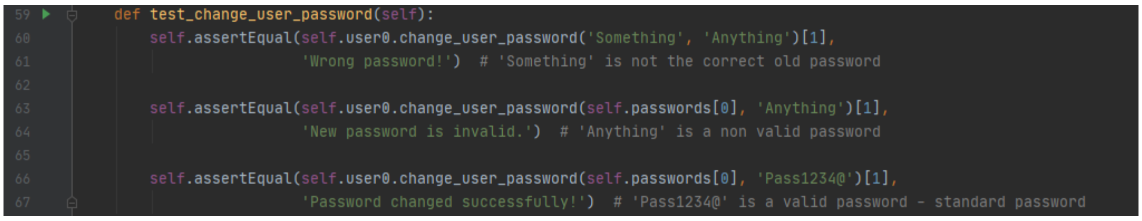
3) *check_password()* (created in the class Password in the module *password.py*):


```python
def test_check_password(self):
    # Using the password's rules set by default --> number of conditions = 4 and between 8 and 64
    self.assertEqual(Psw.check_password(self.password1), False)  # Number of conditions is lower than 4
    self.assertEqual(Psw.check_password(self.password2), False)  # Number of conditions is lower than 4
    self.assertEqual(Psw.check_password(self.password3), self.password3)  # Every condition satisfied
    self.assertEqual(Psw.check_password(self.password4), self.password4)  # Every condition satisfied
```
Although we have only two possible returns, we made 4 tests to verify the method for every password in the *setUp()* method. So, in our example:

- In the first and second case, we test two weak passwords, i.e., two passwords that do not follow the minimum requirements (have a lower case, upper case, special and numerical character), hoping to get *False* (signalizing that the password is not allowed).

- In the third and fourth case, we test two strong passwords with all the 4 requirements, expecting to get the password itself as response (signalizing that the password is allowed).

4) *change_user_password()* (created in the class User in the module *user.py*):


```python
def test_change_user_password(self):
    self.assertEqual(self.user0.change_user_password('Something', 'Anything')[1],
                     'Wrong password!')  # 'Something' is not the correct old password

    self.assertEqual(self.user0.change_user_password(self.passwords[0], 'Anything')[1],
                     'New password is invalid.')  # 'Anything' is a non valid password

    self.assertEqual(self.user0.change_user_password(self.passwords[0], 'Pass1234@')[1],
                     'Password changed successfully!')  # 'Pass1234@' is a valid password - standard password
```

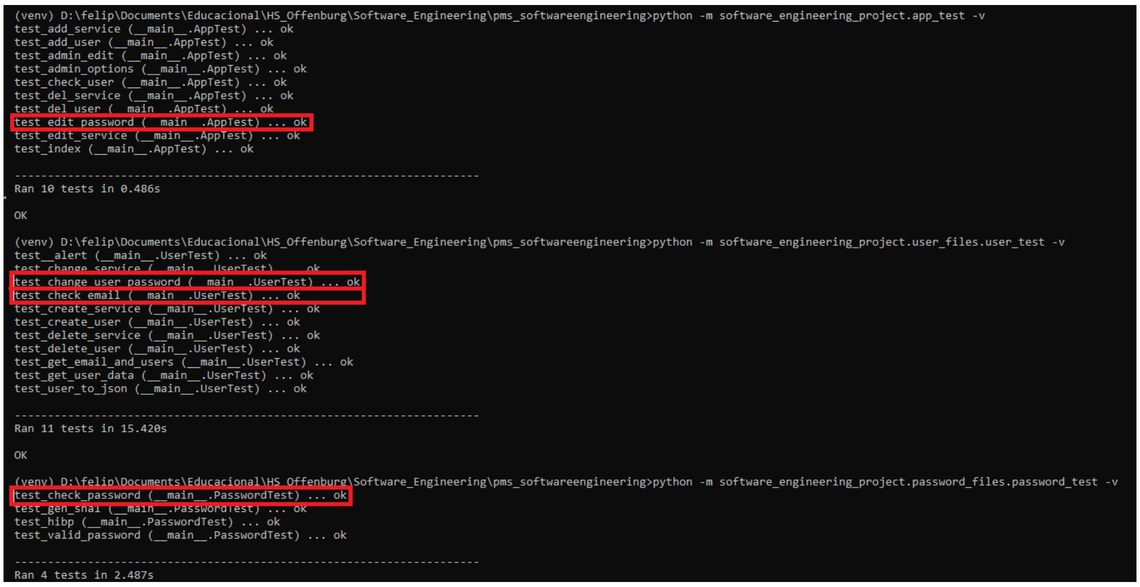
In this method, there are 3 possible returns and, therefore, we will test each one.

- In the first one, we are trying to change the password sending an incorrect password as the old one, expecting get a message saying that the wrong one was informed.

- In the second case, the old password is correct but the new one is invalid (i.e., it does not have the minimum requirements) and, so, we expect to receive a message saying that.

- In the third case, the old password is correct and the new one is valid. This way, we should receive a message saying that we are able to change the password.

Lastly, an image showing that all these tests was executed successfully is shown below:



```
(venv) D:\felip\Documents\Educacional\HS_Offenburg\Software_Engineering\pms_softwareengineering>python -m software_engineering_project.app_test -v
test_add_service (__main__.AppTest) ... ok
test_add_user (__main__.AppTest) ... ok
test_admin_edit (__main__.AppTest) ... ok
test_admin_options (__main__.AppTest) ... ok
test_check_user (__main__.AppTest) ... ok
test_del_service (__main__.AppTest) ... ok
test_del_user (__main__.AppTest) ... ok
test_edit_password (__main__.AppTest) ... ok
test_edit_service (__main__.AppTest) ... ok
test_index (__main__.AppTest) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.486s

OK

(venv) D:\felip\Documents\Educacional\HS_Offenburg\Software_Engineering\pms_softwareengineering>python -m software_engineering_project.user_files.user_test -v
test__alert (__main__.UserTest) ... ok
test_change_service (__main__.UserTest) ... ok
test_change_user_password (__main__.UserTest) ... ok
test_check_email (__main__.UserTest) ... ok
test_create_service (__main__.UserTest) ... ok
test_create_user (__main__.UserTest) ... ok
test_delete_service (__main__.UserTest) ... ok
test_delete_user (__main__.UserTest) ... ok
test_get_email_and_users (__main__.UserTest) ... ok
test_get_user_data (__main__.UserTest) ... ok
test_user_to_json (__main__.UserTest) ... ok

----------------------------------------------------------------------
Ran 11 tests in 15.420s

OK

(venv) D:\felip\Documents\Educacional\HS_Offenburg\Software_Engineering\pms_softwareengineering>python -m software_engineering_project.password_files.password_test -v
test_check_password (__main__.PasswordTest) ... ok
test_gen_sha1 (__main__.PasswordTest) ... ok
test_hibp (__main__.PasswordTest) ... ok
test_valid_password (__main__.PasswordTest) ... ok

----------------------------------------------------------------------
Ran 4 tests in 2.487s
```

Edugit link

The code that contains all modules and all tests for each method created can be found in the Hochschule Offenburg GitLab, by researching the "PMS_SoftwareEngineering" project (from fazzolin) or in the link below:

https://edugit.hs-offenburg.de/fazzolin/pms_softwareengineering