**Харківський національний економічний університет**

**імені Семена Кузнеця**

**ЗВІТ**

**З ВИКОНАННЯ Самостійної роботи**

**за дисципліною: *“Інформаційні системи та інтернет технології”***

**Варіант № 9**

**Виконав: студент факультету Інформаційних технологій**

**3 курсу, спец. Кібербезпека,**

**групи 6.04.125.010.21.2**

**Довбня Максим Віталійович**

**Перевірила:**

**Солодовник Ганна Валеріївна**

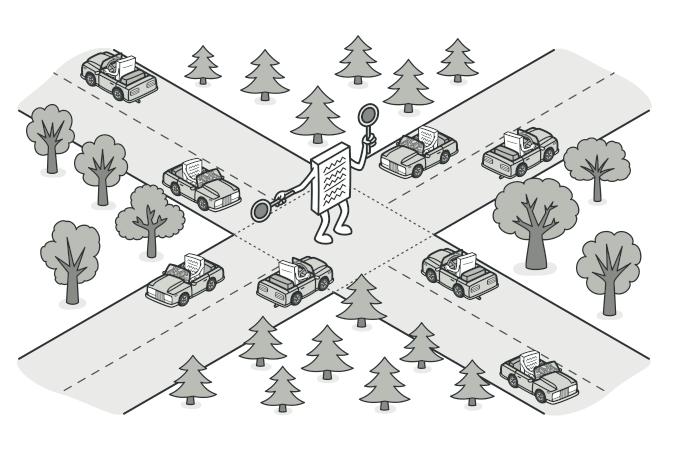
**ХНЕУ ім. С. Кузнеця**

**2024**

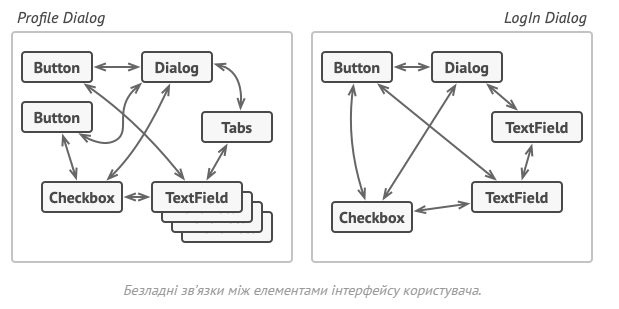
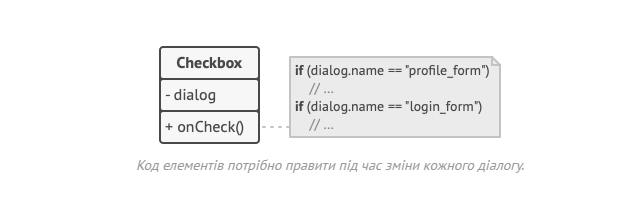
**Завдання:**

1. Надати опис патерну відповідно до номеру за журналом: Посередник (Mediator)
2. Знайти на теренах Інтернет приклад коду аутентифікації користувачів та надати пояснення до цього коду

**Опис патерну «Посередник»:**

**Посередник** — це поведінковий патерн проектування, що дає змогу зменшити зв’язаність великої кількості класів між собою, завдяки переміщенню цих зв’язків до одного класу-посередника.  


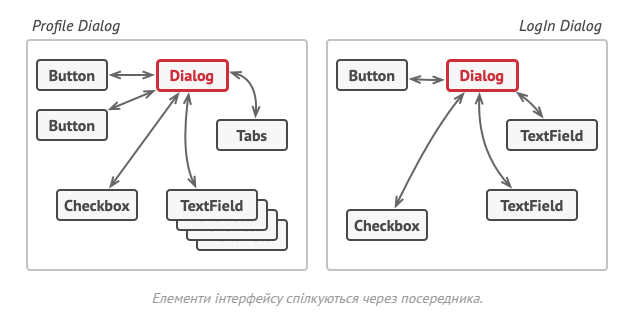
**Проблема**

Припустімо, що у вас є діалог створення профілю користувача. Він складається з різноманітних елементів керування: текстових полів, чекбоксів, кнопок.  
  
Окремі елементи діалогу повинні взаємодіяти одне з одним. Так, наприклад, чекбокс «у мене є собака» відкриває приховане поле для введення імені домашнього улюбленця, а клік по кнопці збереження запускає перевірку значень усіх полів форми.  


Прописавши цю логіку безпосередньо в коді елементів керування, ви поставите хрест на їхньому повторному використанні в інших місцях програми. Вони стануть занадто тісно пов’язаними з елементами діалогу редагування профілю, які не потрібні в інших контекстах. Отже ви зможете або використовувати всі елементи відразу, або не використовувати жоден.

**Рішення**

Патерн Посередник змушує об’єкти спілкуватися через окремий об’єкт-посередник, який знає, кому потрібно перенаправити той або інший запит. Завдяки цьому компоненти системи залежатимуть тільки від посередника, а не від десятків інших компонентів.

У нашому прикладі посередником міг би стати діалог. Імовірно, клас діалогу вже знає, з яких елементів він складається. Тому жодних нових зв’язків додавати до нього не доведеться.

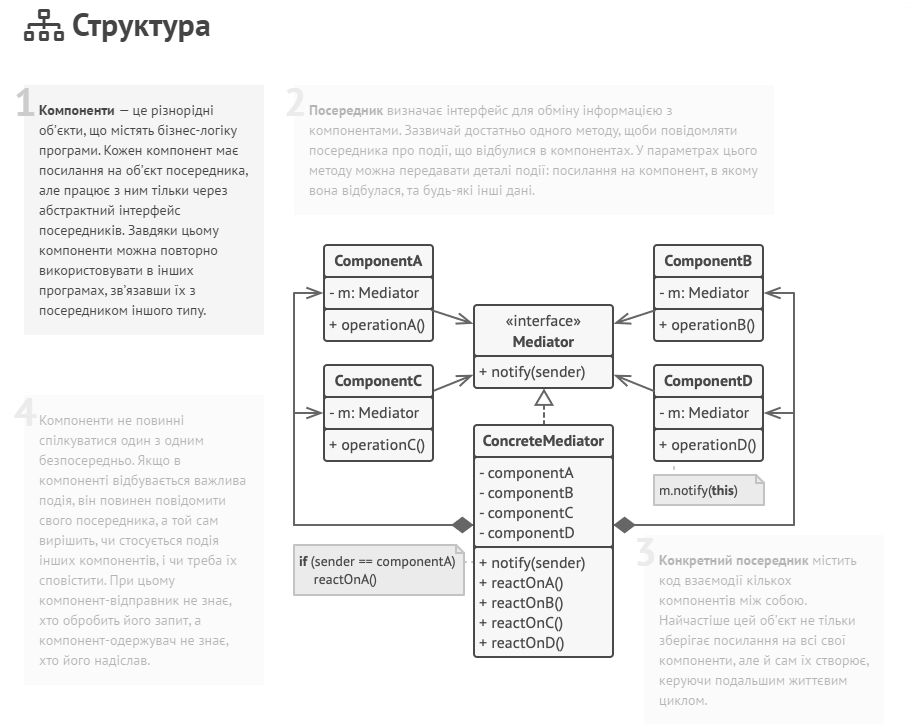
Основні зміни відбудуться всередині окремих елементів діалогу. Якщо раніше при отриманні кліка від користувача об’єкт кнопки самостійно перевіряв значення полів діалогу, то тепер його єдиний обов’язок — повідомити діалогу про те, що відбувся клік. Отримавши повідомлення, діалог виконає всі необхідні перевірки полів. Таким чином, замість кількох залежностей від інших елементів кнопка отримає лише одну — від самого діалогу.

Щоб зробити код ще гнучкішим, можна виділити єдиний інтерфейс для всіх посередників, тобто діалогів програми. Наша кнопка стане залежною не від конкретного діалогу створення користувача, а від абстрактного, що дозволить використовувати її і в інших діалогах.

Таким чином, посередник приховує у собі всі складні зв’язки й залежності між класами окремих компонентів програми. А чим менше зв’язків мають класи, тим простіше їх змінювати, розширювати й повторно використовувати.

## Аналогія з життя

Пілоти літаків, що сідають або злітають, не спілкуються з іншими пілотами безпосередньо. Замість цього вони зв’язуються з диспетчером, який координує політ кількох літаків одночасно. Без диспетчера пілотам доводилося б увесь час бути напоготові і стежити самостійно за всіма літаками навколо. Це часто призводило б до катастроф у небі.

Важливо розуміти, що диспетчер не потрібен під час всього польоту. Він задіяний тільки в зоні аеропорту, коли потрібно координувати взаємодію багатьох літаків.  


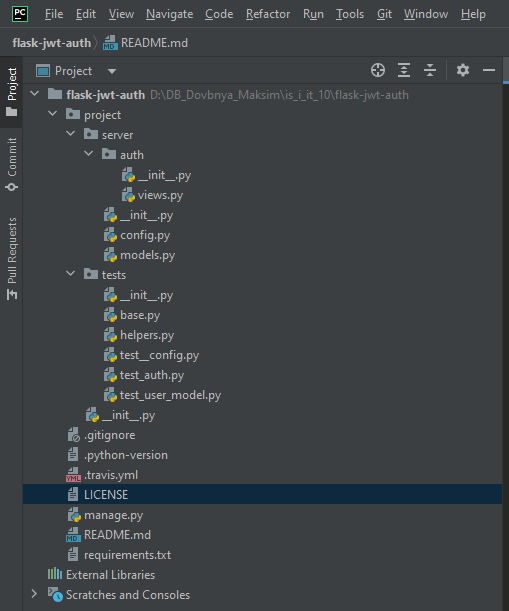
## Застосування

* Коли вам складно змінювати деякі класи через те, що вони мають величезну кількість хаотичних зв’язків з іншими класами.
* Посередник дозволяє розмістити усі ці зв’язки в одному класі. Після цього вам буде легше їх відрефакторити, зробити більш зрозумілими й гнучкими.
* Коли ви не можете повторно використовувати клас, оскільки він залежить від безлічі інших класів.
* Після застосування патерна компоненти втрачають колишні зв’язки з іншими компонентами, а все їхнє спілкування відбувається опосередковано, через об’єкт посередника.
* Коли вам доводиться створювати багато підкласів компонентів, щоб використовувати одні й ті самі компоненти в різних контекстах.
* Якщо раніше зміна відносин в одному компоненті могла призвести до лавини змін в усіх інших компонентах, то тепер вам достатньо створити підклас посередника та змінити в ньому зв’язки між компонентами..

**Приклад коду аутентифікації користувачів та надати пояснення до цього коду**

Для аналізу коду я обрав наступний код:   
<https://github.com/realpython/flask-jwt-auth>

Для більш зручного аналізу зроблю клон проекту

Спочатку пропоную подивитись на структуру проекту   


Можна побачити, що головний файл - /manage.py та що від нього починається запуск проекту

Далі можна побачити папку /project, вона містить всю іншу частину проекту

В свою чергу папка містить ще дві /project/server – з кодом для автентифікації та /project/tests – з тестами для тестування того, як працює код

В свою чергу папка /project/server має всередині папку /project/server/auth з в’ю моделлю для автентифікації

Якщо дивитись більш детально по файлом тоді маємо наступне:

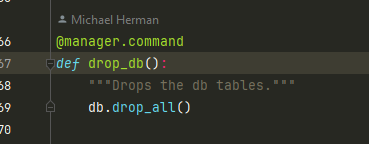
Для налаштування у файлі README.MD є інструкція

Для початку потрібно налаштувати змінні оточення   


Також потрібно створити бд у psql

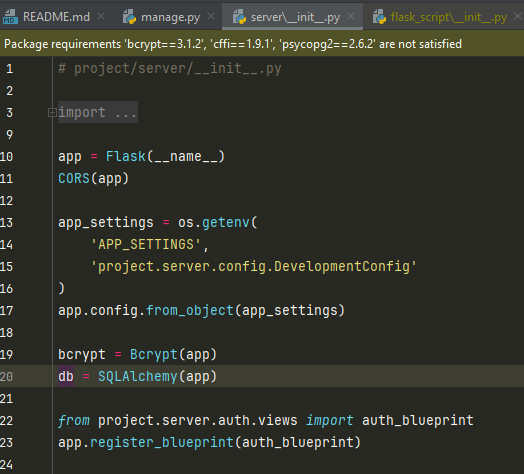
Та запустити відповідний код для налаштування   


Якщо подивитись на даний код зсередини, тоді можна побачити наступний код



Декоратор «manager.command» - надає інформацію, що дану команду можна запустити з терміналу   
  
в свою чергу db.drop\_all() це використання SQLAlchemy як систему з роботою з БД

Код з підключення до БД наступний:



Заглибнувшись більше у код можна зрозуміти, що файл /manage.py лише запускає проект, проте налаштування застосунку знаходиться у файлі /project/server/\_\_init\_\_.py

Також можна побачити, що налаштування беруться з змінних оточення

Також у кінці файлу можна побачити, що під’єднуються файли з автентифікації в’ю

В’ю знаходиться за наступним шляхом /project/server/auth/views.py

Якщо просто відкрити файл, тоді можна побачити, що файл всередині має 4 класи як окремі в’ю моделі, що вони всі зареєстровані як окремі моделі та що всі вони мають різні правила



Так для реєстрації є лише метод «POST» за шляхом /auth/register

Для входу в систему метод «POST» та шлях - /auth/login

Для отримання статусу – «GET» та шлях – /auth/status

Та для виходу «POST» та шлях - /auth/logout

Почну розглядати код з класу «RegisterAPI»

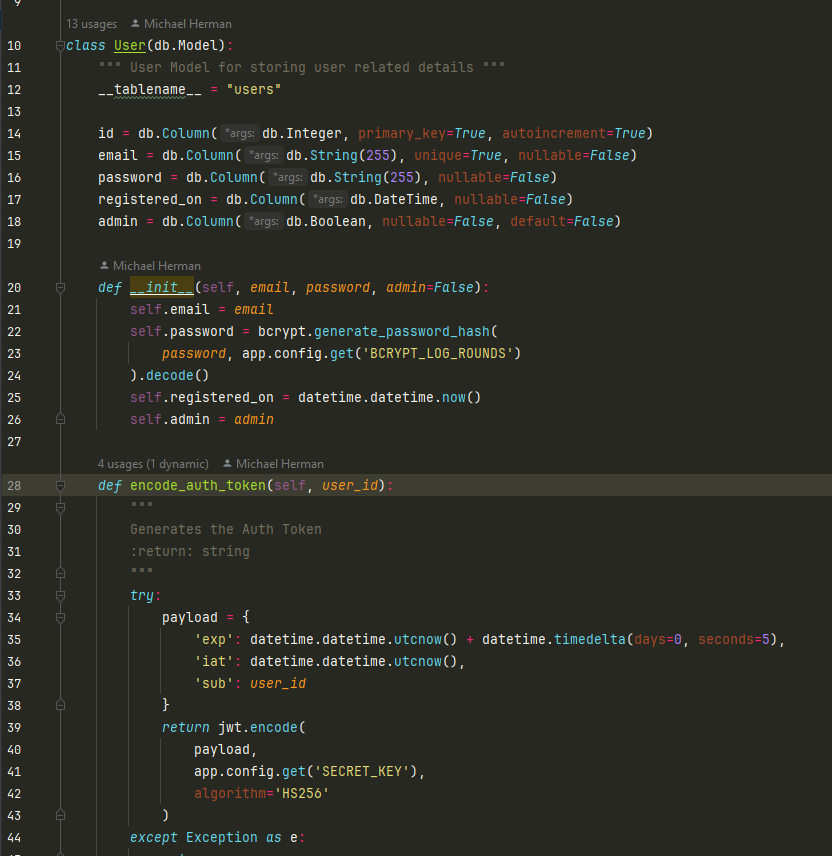


З опису одразу видно, що це просто метод, який всередині має лише код для обробки запиту типу «POST»

На 20-у рядку отримаємо дату з json пейлоаду

На 22-у намагаємось отримати користувача за емейлом

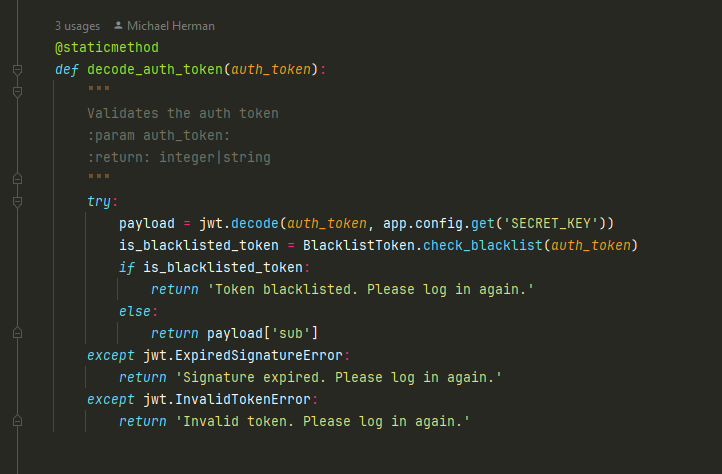
В свою чергу клас User – це модель у файлі /project/servr/models.py



Всередині цього класу можна побачити налаштування БД, ініціалізація нового користувача (рядок у БД)

Також є метод для шифрування ключа автентифікації відповідно до user\_id

Також у цьому класі є статичний метод для декодування токену



Всередині є перевірка токену на валідність та знаходження у чорному списку

В свою чергу чорний список це теж модель БД



Який всередині має лише налаштування для БД та первірку на наявність запису у таблиці

Тепер повернемось до нашої в’ю моделі

Якщо користувача буде знайдено, тоді в респонсі буде видано, що такий користувач вже існує, а якщо користувача не має, тоді буде спроба створити нового користувача, створивши користувача на 25-й строчці, відповідно отримавши дані з пейлоаду, до речі всередині класу User пароль хешується, а id надається автоматично та є ключовим полем у таблиці

Наступним кроком запис додається до таблиці на строчці 30

Та зберігається на строчці 31

Після чого буде згенеровано токен автентифікації та наданий у респонсі

Якщо в процесі виконання даного коду відбудеться помилка, тоді користувачу буде повернута помилка у відповіді від сервера

Тепер розглянемо наступний клас з логуванням користувачів



Знову це в’ю з методом, який має лише обробку запитів типу «POST»

Одразу отримаємо дані з json-у, який надішле користувач,

Наступним кроком на 63-й стрічці перевіримо, що користувач з таким емейлом існує, та якщо не існує – повертаємо помилку на 82-у рядку,  
а якщо існує, тоді перевіряємо, що хеш пароля у бд та хеш з пароля, який ми отримали у пейлоаді однакові, якщо так – генеруємо токен на 69-у рядку

Також, якщо в процесі виникла будь-яка помилка – користувачу буде повернуто помилку, та напис, що потрібно спробувати ще раз

Тепер перейдемо до класу UserAPI



Він так само має, лише обробку одного типу запиту, але це «GET» запит

Тут перевірка відбувається за допомогою хедеру, так на 98-й строчці ми намагаємось, отримати хедер, та отримати значення, якщо хедеру не має, або його тип буде не вірний – користувач отримає помилку,

Але, якщо токен існує, тоді він буде декодований та у разі його валідності – дані будуть повернуті користувачу

Та останній клас LogoutAPI



Код схожий на попередній клас, єдина різниця, що після декодування токену – створюється запис до БД, що даний токен знаходиться у чорному списку та у разі успішного запису – повертається код з успіхом, проте, якщо виникне помилка на будь якому з етапів – користувачу буде повернута помилка