Національний технічний університет України
„Київський політехнічний інститут імені Ігоря Сікорського”
Кафедра автоматизації проектування енергетичних процесів і систем

КУРСОВА РОБОТА

з дисципліни: «Бази даних»

тема: «Student Bot Assistant»

Керівник **ст. викладач Колумбет В.П**	**Виконала Круть К.О.**
Допущено до захисту	**Студентка 2 курсу**
«28» грудня 2021р.	**Групи ТІ-01**
Захищено з оцінкою	**залікова книжка**
_____95_____	**№ ТІ-0114**

Київ – 2021

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Кафедра автоматизації проектування енергетичних процесів і систем

КУРСОВА РОБОТА

з дисципліни «Бази Даних»
(назва дисципліни)

на тему: «Student Bot Assistant»

Студента групи _____ТІ-01_____
напряму підготовки 6.050103 Програмна інженерія
спеціальності 121"Інженерія програмного забезпечення"
_____Круть К.О._____
(прізвище та ініціали)

Керівник _____ст.вик. Дацюк О. А._____
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна оцінка _____

Кількість балів: _____ Оцінка ECTS _____

Члени комісії

_____	_____ст. вик. Дацюк О. А._____
(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали)
_____	_____ст. вик. Колумбет В. П._____
(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали)
_____	_____
(підпис)	(вчене звання, науковий ступінь, прізвище та ініціали

Київ – 2021

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет _____ТЕПЛОЕНЕРГЕТИЧНИЙ_____

Кафедра _____Автоматизації проектування енергетичних процесів і систем_____

Напрям підготовки _____6.050103 Програмна інженерія_____
(шифр і назва)

З А В Д А Н Н Я
НА КУРСОВУ РОБОТУ СТУДЕНТУ

_____Круть Катерині Олександрівні_____
(прізвище, ім’я, по батькові)

1. Тема роботи _____«Student Bot Assistant»_____

керівник курсової роботи – _____ст. вик. Дацюк Оксана Антонівна_____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи _____28.12.2021_____

3.Вихідні дані до проекту(роботи): мова програмування Python, середовище розробки PyCharm, система керування базами даних – MySQL.

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): розробити інформаційну систему «Student Bot Assistant», спроектувати базу даних.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання курсової роботи	Строк виконання етапів роботи	Примітка
1.	Затвердження теми роботи	22.10.2021	
2.	Вивчення та аналіз здачі	22.10.2021	
3.	Проектування структури бази даних	23.10.2021	
4.	Розробка алгоритму реалізації здачі	03.11.2021	
5.	Програмна реалізація здачі	14.11.2021	
6.	Тестування програми	29.11.2021	
7.	Оформлення пояснювальної записки	19.12.2021	

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник курсової роботи _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сьогодні, в час Цифрової революції особливий інтерес викликають програми, які були б здатні допомагати в повсякденному житті, роботи життя легшим та зручнішим. Хвиля популярності ботів все більше і більше захоплює світ. Програма «Student Bot Assistant» призначена саме щоденної допомоги. Вона дозволяє налаштувати свій розклад, використовувати нотатки та організувати власні фінанси. Програма орієнтована на студентів, проте підходить всім користувачам, адже є простою у використані та гнучкою у налаштуваннях, має багатий функціонал

Код написаний мовою програмування Python та реалізований у середовищах розробки PyCharm за допомогою сервера баз даних MySQL.

Обсяг пояснювальної записки 75 аркушів, 1 додаток.

ANNOTATION

Nowadays, in the time of Digital Revolution, programs that would be able to help us in everyday life and make it easier and more comfortable continue to arouse particular interest. The wave of popularity of bots is capturing the world more and more. The program «Student Bot Assistant» is intended for daily assistance. It allow to set up your schedule, use notes and organize your own finances. The program is aimed at students, but is suitable for all users, because it is easy to use and flexible in settings, also because it has rich functionality.

The code is written in the Python programming language and implemented in PyCharm integrated development environment, using the MySQL database server.

The volume of the explanatory note is 75 sheets, 1 appendix.

ЗМІСТ

ВСТУП

В XXI столітті, відомому також як цифрове, світ – невпинно швидкозмінний, і ми живемо в його бурхливому темпі з щільним графіком, отримуючи щодня купу інформації та завдань. Життя студента особливо насичене, тому важко все пам’ятати, організувати та за всім слідкувати. Як і тисячі років тому, гроші відіграють надзвичайно важливу роль у житті кожного з нас. Щодня ми взаємодіємо з фінансами, витрачаємо, отримуємо кошти, і саме тому надзвичайно важливою для сучасної людини є фінансова грамотність, а саме уміння контролювати свої кошти, спланувати майбутнє, для досягнення фінансового добробуту. Важливим для кожного студента/школяра, або ж просто зайнятої людини є ведення розкладу на щодень, на тиждень чи навіть рік, ведення нотаток – та пошук необхідної інформації. Тягати купу записників, щоденників та переглядати нескінченну кількість книг на бібліотечних полицях немає сенсу, адже сьогодні, в час цифрової революції, нам в цьому можуть допомогти програми-помічники, особливо зручними є ті, що є гнучкими, доступними та не потребують додаткових завантажень на пристрої.

Саме тому реалізація цієї програми виконана у формі бота-помічника в одному із найбільш популярних у світі мессенджерів – Telegram (станом на 2021 рік Telegram використовує більше ніж 500 млн людей у всьому світі). Оскільки найзручніше писати подібні програми підлаштовані під платформу Telegram найзручніше мовою Python, тому для реалізації цієї програми обрано саме її та фреймворк Aiogram, що дозволяє забезпечити багатофункціональність бота, модуль pymysql забезпечує під’єднання програми до бази даних, що не обхідна для функціонування бота-помічника.

Для реалізації бази даних була обрана система керування реляційними базами даних MySQL корпорації Oracle. Система MySQL дає можливість легко створювати бази даних та працювати з ними.

Мета цієї курсової роботи полягає в отримані нових та збагаченні вже отриманих знань про бази даних, удосконаленні навичок роботи з СУБД. Також застосування цих знань та умінь для розробки концептуальної моделі та структури необхідної для

програми бази даних, проектування інформаційної системи, звичайно, також програмна реалізація «Student Bot Assistant», в функціонал якого входить організація та контроль власних фінансів.

Завдання – написати програму, що дозволяла б керувати власним розкладом, створювати нотатки, шукати необхідні книги та записувати їх, управляти фінансами та організовувати їх, а саме додавати витрати/прибуток, переглядати їх, керувати бюджетом та переглядати статистику.

РОЗДІЛ 1. ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Предметна область роботи

Для цієї курсової роботи предметною областю є бот-помічник, а фрагментами цієї ПО є «Нотатки», «Розклад» та «Фінанси». Кожен фрагмент цієї системи має свої об’єкти, функціональну область та процеси, їх перечислено нижче у таблиці 1.1

Таблиця 1.1. Опис функціональної області та її процесів

Функціональна область	Процеси
Прибуток	Додавання прибутоку
	Видалити прибутоку
Витрати	Дадавання витрат
	Видалення витрат
Бюджет	Перегляд встановленого ліміту
	Редагування встановленого ліміту
Категорія	Додавання категорії
	Перегляд всіх категорій
Статистика	Перегляд статистику за тиждень
	Перегляд статистику за місяць
Інше	Перегляд витрати за сьогодні
	Перегляд витрати за тиждень
	Перегляд витрати за місяць
	Перегляд прибуток за сьогодні
	Перегляд прибуток за тиждень
	Перегляд прибуток за місяць
Групи	Додавання групи
	Парсинг груп з сайту розклад КПІ

	Редагування даних групи
	Видалення групи
Викладачі	Додавання нового викладача
	Парсинг викладачів з сайту розклад КПІ
	Редагування даних викладача
	Видалення викладача
Дисципліни	Додавання дисципліни
	Редагування дисципліни
	Видалення дисципліни
Аудиторії	Додавання аудиторії
	Редагування даних аудиторії
	Видалення аудиторії
Розклад	Відображення розкладу на сьогодні
	Відображення розкладу на наступний навчальний день
	Відображення наступної пари
	Відображення розкладу на обидва тижні
Пари	Додавання пари
	Редагування даних пари
	Видалення пари
Тема	Додавання нової теми
	Редагування теми
	Видалення теми
Назва	Додавання нової назви
	Редагування назви за темою
	Видалення назви за темою
Текст нотаток	Додавання нової нотатки
	Редагування редагування існуючої нотатки, за назвою та темою

	Видалення доданої нотатки, за назвою і темою
	Пошук доданої нотатки, для конкретного користувача, за ім’ям
	Пошук доданої нотатки, для конкретного користувача, за темою
	Вивід всіх, занесених конкретним користувачем, нотаток

1.2. Основний функціонал

Аналізуючи дані попередньої таблиці визначили основний функціонал створеної програми, доступний користувачеві:

- додавання розкладу
- редагування розкладу
- відображення розкладу на певний період часу
- створення нотаток
- пошук нотатки за певний критерієм
- редагування нотатки
- видалення нотатки
- додавання прибутку/витрат
- видалення прибутку/витрат
- додавання категорій
- перегляд всіх категорій
- перегляд статистики за певний період часу
- перегляд прибутку/витрат за певний період часу

РОЗДІЛ 2. ОПИС БАЗИ ДАНИХ

2.1. Концептуальна модель бази даних

Концептуальна модель — модель предметної області, що складається з переліку взаємопов'язаних понять, що використовуються для опису цієї області, разом з властивостями й характеристиками, класифікацією цих понять, за типами, ситуацій, ознаками в даній області і законів протікання процесів в ній. Концептуальна модель відноситься дод моделей, які формуються після процесу концептуалізації або узагальнення. Концептуальні моделі часто є абстракціями речей в реальному світі [1].

Концептуальна модель бази даних – інформаційна модель найвищого рівня абстракції для ІС «Student Bot Assistant». Всі дані ІС вміщено в шістнадцяти таблицях: «Користувач», «Група», «Теми», «Нотатки», «Пара», «День тижня», «Тиждень», «Час пари», «Дисципліна», «Аудиторія», «Тип пари», «Викладач», «Бюджет», «Категорія», «Витрати», «Прибутки».

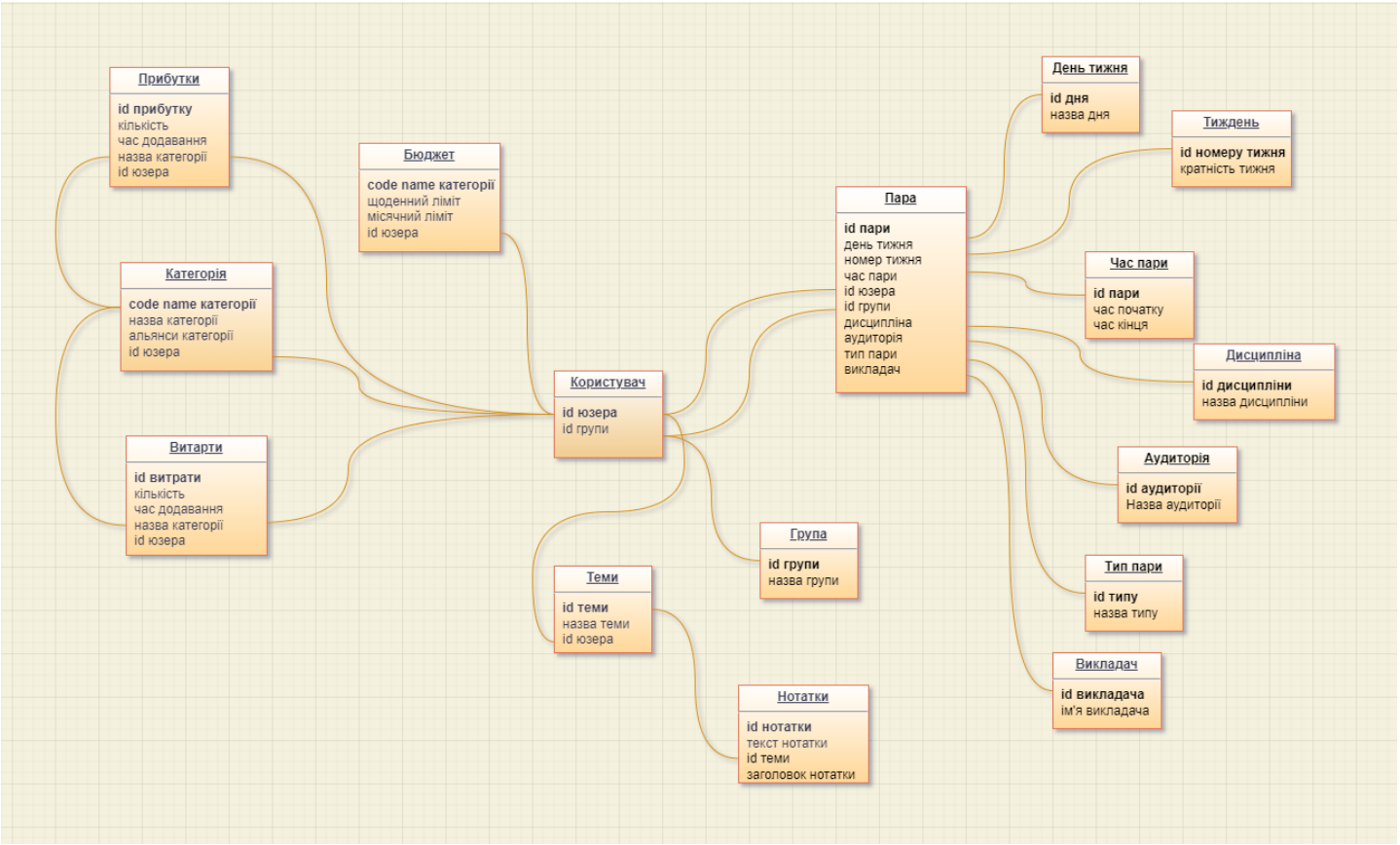

Рис. 2.1.1 – Концептуальна модель бази даних

Як вище було сказано, метою цієї роботи є створення функціональної частини бота-помічника, що відповідає за організацію і контроль фінансів, яка не може функціонувати без бази даних, що зберігала б всі необхідні для користувача та системи дані. Їх вміщено в п’яти таблицях, а саме: «Користувач», «Бюджет», «Категорія», «Витрати» та «Прибутки».

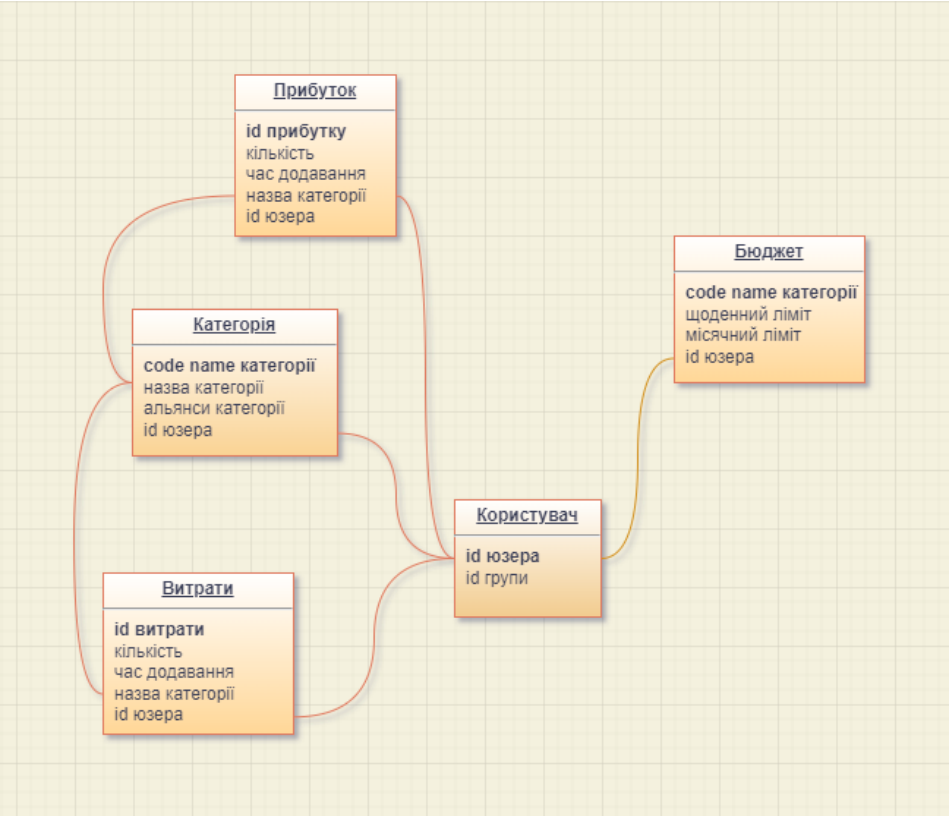

Рис. 2.2.1 Концептуальна модель БД складової «Фінанси» ІС «Student Bot Assistant»

Таблиця «Користувач» є головною таблицею взагалі у всій інформаційній системі «Student Bot Assistant», з нею пов’язані всі інші таблиці. Вона зберігає найважливіші дані – telegram id користувача та id групи.

«Бюджет» пов’язаний лише з головною таблицею, вона вміщує в собі інформацію про щоденний та місячний бюджет користувача, тобто певний ліміт, який він собі встановлює.

Таблиця «Категорія» також дуже важлива, адже без неї неможливе функціонування таблиць «Витрати» та «Прибутки». Вона зберігає літеральний код категорії, її ім’я та

також альянси, тобто допоміжні ключові слова, які забезпечують додавання витрат/прибутку до бази даних так само, як і назва категорії.

Структура таблиць «Витрати» і «Прибутки» майже ідентична: вона відрізняється лише id, проте призначення їх різне. За назвою зрозуміло, що «Витрати» містить інформацію про витрати записані користувачем: кількість грошей в його валюті, точний час додавання запису та код назви категорії. Такі ж дані зберігає і таблиця «Прибутки», проте для прибутків користувача.

2.2. Фізична модель бази даних

Фізична модель бази даних – це модель даних, яка визначає яким чином представляються дані і все зберігає всі деталі, необхідні СУБД для створення бази даних. Вона включає всі таблиці, колонки, зв’язки та властивості для фізичної реалізації бази даних [2].

Нижче, на рисунку 2.2.1, представлена фізична модель всієї бази даних «Student Bot Assistant». Вона складається з шістнадцяти таблиць.

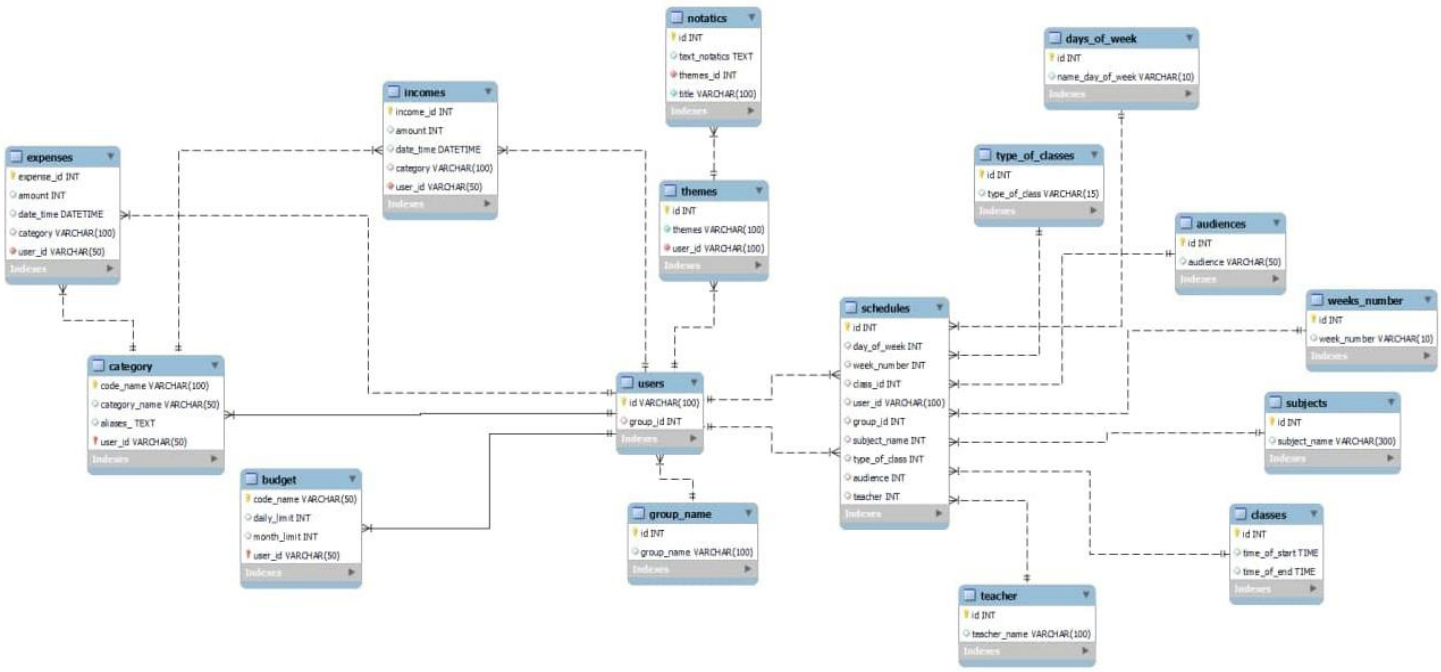

Рис. 2.2.1 – Фізична модель бази даних «Student Bot Assistant»

Частина «Фінанси» складається з чотирьох таблиць, її фізична модель представлена на рисунку 2.2.3.

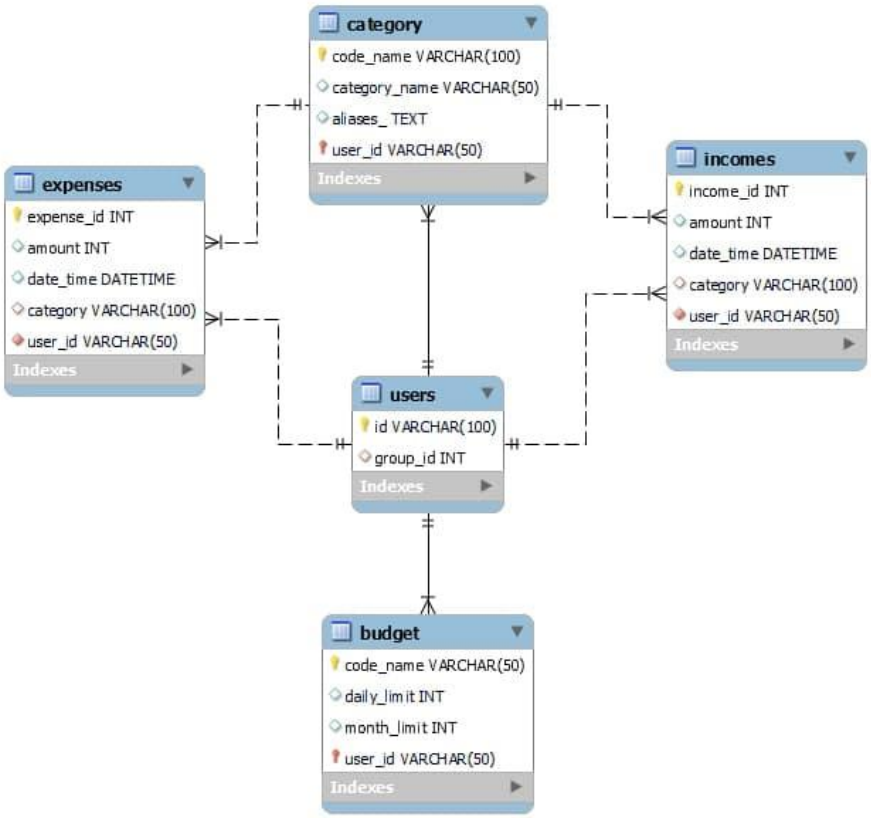

Рис. 2.2.3. Фізична модель фінансової частини БД «Student Bot Assistant»

2.3. Структура таблиць бази даних

Головною таблицею є – «users», що зберігає дані про юзера. Вона складається з трьох полів: *id* типу VARCHAR (обмеження до 100 символів), *group_id* типу INT та *username* VARCHAR (обмеження – 50 символів). Поле *id* – PRIMARY KEY, *username* має властивість – UNIQUE KEY, також *group_id* виступає як FOREIGN KEY та посилається на поле *id* в таблиці *group_name*.

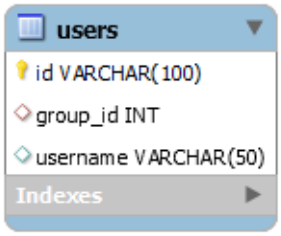

Рис. 2.3.1 Таблиця «users»

Таблиця «budget» містить дані про встановлений користувачем ліміт на щодень та місяць. Складається вона з полів: *code_name* VARCHAR (обмеження – 50 символів та значенням за замовчуванням – 'general'), *daily_limit* типу INT (DEFAULT 0), *month_limit* типу INT (DEFAULT 0), *user_id* VARCAHR (обмеження – 50 символів), що не може набувати значення NULL та має властивість FOREIGN KEY, тобто посилається на поле *id* таблиці *users*. PRIMARY KEY виступає комбінація полів *code_name* та *user_id*.

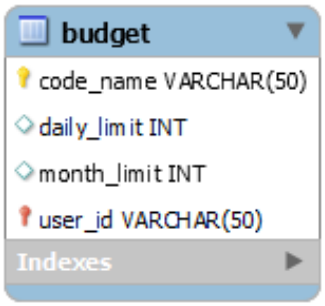

Рис. 2.3.2 Таблиця «budget»

Таблиця «category» містить дані про категорію для витрат або прибутків. Складається з *code_name* типу VARCHAR (обмеження – 100 символів), *category_name* – VARCHAR (обмеження – 50 символів), *aliases_* типу TEXT та поля *user_id* VARCHAR (обмеження – 50 символів, не може бути NULL). PRIMARY KEY цієї таблиці – це комбінація полів *code_name* та *user_id*. А FOREIGN KEY є *user_id*.

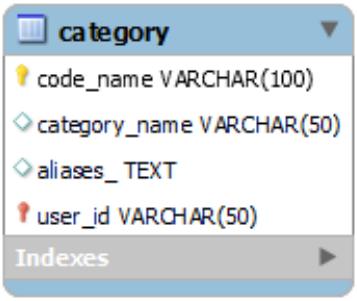

Рис. 2.3.3 Таблиця «category»

Четвертою таблицею є «expenses», що містить п’ять полів. Головне поле *expense_id* типу INT (має властивості PRIMARY KEY, NOT NULL та AUTO_INCREMENT); *amount*, що є INT; *date_time* типу DATETIME, *category* –

VARCHAR (з обмеженням в 100 символів) та є FOREIGN KEY й посилається на поле в таблиці «users». Властивіть FOREIGN KEY має також поле *category* й посилається на *code_name* з таблиці «category».

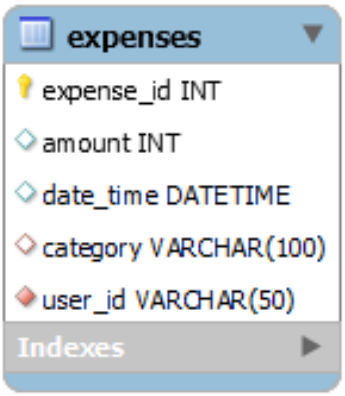

Рис. 2.3.4 Таблиця «expenses»

П'ята таблиця – «incomes», що є дуже подібною до попередньої і також містить п’ять полів. Головне поле *income_id* типу INT (RIMARY KEY, NOT NULL та AUTO_INCREMENT); *amount*, що є INT; *date_time* типу DATETIME, *category* – VARCHAR (з обмеженням в 100 символів) та є FOREIGN KEY й посилається на поле в таблиці «users». Властивіть FOREIGN KEY має також поле *category* й посилається на *code_name* з таблиці «category».

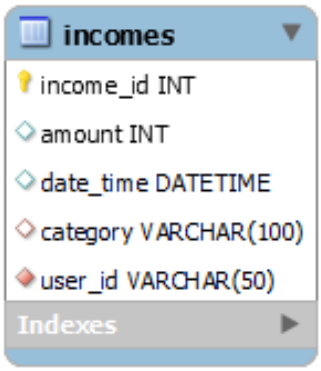

Рис. 2.3.5 Таблиця «expenses»

РОЗДІЛ 3. ОПИС ПРОЕКТУ ПРОГРАМНОГО ПРОДУКТУ

3.1. Використане програмне забезпечення

Для розробки бази даних був використаний інструмент MySQL Workbench, призначений для візуального проектування баз даних та інтегроване середовище для розробки програмного забезпечення мовою програмування Python – PyCharm компанії JetBrains.

MySQL Workbench – це комплексний візуальний інструмент, що використовується архітекторами, розробниками та адміністраторами баз даних. MySQL Workbench забезпечує всебічне моделювання даних, засоби розробки та методи управління SQL для конфігурації сервера, крім того засоби управління користувачами, резервного копіювання та ще багато іншого [3].

Інструмент MySQL Workbench є популярним, адже надає візуальні інструменти для створення, виконання та оптимізації запитів SQL, що роблять програму простою та легкозрозумілою у використанні. Редактор SQL забезпечує виділення кольорового синтаксису, автоматичне заповнення, повторне використання фрагментів SQL та історію виконання SQL. Панель Database Connections дозволяє розробникам легко керувати стандартними підключеннями до бази даних, включаючи MySQL Fabric. Браузер об’єктів забезпечує миттєвий доступ до схеми та об’єктів бази даних.

MySQL Workbench надає можливість вибору де працювати – безпосередньо в програмі чи в консолі, що призначена для управління середовищами MySQL є простою у використані і водночас містить все для повноцінної роботи. Крім того, надається можливість використання візуальних засобів для налаштування серверів, управління користувачами, резервного копіювання та відновлення, перевірки даних аудиту та перегляду стану бази даних, що є надзвичайно корисним для розробників та адміністраторів БД [4].

Середовище розробки PyCharm призначене для програмування мовою Python, та визнана найкращим серед сьогодні існуючих. PyCharm надає багато можливостей для зручної роботи: зміна користувацького інтерфейсу, різні теми та підсвітка синтаксису,

налаштування користувацьких комбінацій клавіш. PyCharm робить розробку максимально продуктивною завдяки функція автодоповнення, аналізу коду, підствіткою помилок та швидким виправленням. Доступна навігація, яка дозволяє швидко переходити до будь-якого класу, файлу, вікну чи навіть символу в один клік, для швидкого переміщення проектом. PyCharm містить також вбудовані інструменти для розробки: відладчик, інструмент запуску тестів, повноцінний термінал, підтримує наукові бібліотеки Pandas, Numpy, Matplotlib та метроди віддаленої розробки Git, Docker і Vagran та web розробку.

Крім того, включає інструменти для роботи з базами даних: має доступ до Oracle, SQL Server, PostgreSQL, MySQL й інших баз даних. PyCharm навіт допомагає редагувати SQL-код, виконувати запити, преглядати схеми та змінювати схеми [5].

РОЗДІЛ 4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1. Загальний опис програми

Програма являє собою інтерфейс для допомоги студенту в різних аспектах діяльності: налаштування розкладу, ведення нотаток та контроль фінансів. Функціонал фінансової частини програмного продукту розміщений в модулях *Bot_Assistant.py, Finances.py, Statistic.py, markup.py, exceptions.py та databases.py,* він дозволяє організовувати та контролювати власні кошти: встановлювати ліміт та редагувати його, додавати та переглядати категорії, додавати витрати, прибуток й видаляти ці записи з бази даних. Крім того доступна можливість перегляду статистики за певний період часу.

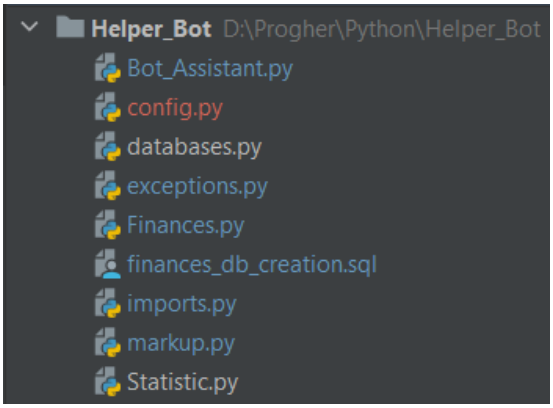

Рис. 4.1.1 – Модулі програмного продукту

Модуль *Bot_Assistant.py* – головний модуль програми, в якому розміщені всі хендлери (анг. *handlers*) та класи станів необхідні для обробки команд, встановлення станів – функціонування бота. Нижче перечислені класи та хендлери забезпечують роботу фінансової частини програми.

Класи:

- class HandlerIncomes;
- class HandlerExpenses;
- class HandlerStatistic;
- class HandlerCategory;
- class HandlerBudget;
- class HandlerOtherFinances;

Хендлери для роботи з фінансами:

- finance_menu;
- budget_menu;
- statistic_menu;
- other_finance_menu;
- add_expense_handler;
- adding_expense_message_handler;
- add_incomes_handler;
- adding_incomes_message_handler;
- edit_budget_handler;
- editing_budget_message_handler;
- add_category_handler;
- creating_finance_category;
- del_expense_handler;
- del_income_handler;
- categories_viewing_handler;
- today_expenses_handler;
- week_expenses_handler;
- month_expenses_handler;
- today_incomes_handler;
- week_incomes_handler;
- month_incomes_handler;
- this_week_statistic_handler;
- this_month_statistic_handler;
- back_to_finance_menu;
- back_to_other_finance_menu.

Модуль *markup.py* – модуль для створення кнопок бота.

- inline_button_finance;
- inline_button_add_finance_category;
- inline_button_add_expense;
- inline_button_add_incomes;
- inline_button_budget;
- inline_button_finance_statistic;
- inline_button_finance_other;
- inline_button_back_to_finance;
- inline_button_statistic_by_week;
- inline_button_statistic_by_month;
- inline_button_edit_budget;
- inline_button_see_today_expenses;
- inline_button_see_week_expenses;
- inline_button_see_month_expenses;
- inline_button_see_today_incomes;
- inline_button_see_week_incomes;
- inline_button_see_month_incomes;
- inline_button_back_to_other_finance;
- inline_button_see_categories;
- inline_keyboard_budget_menu;
- inline_keyboard_statistic_menu;
- inline_keyboard_finance_menu;
- inline_keyboard_other_menu.

Модуль *Finances.py* – модуль, в якому розміщені всі класи та методи, що забезпечують логіку роботи фінансової частини бота. Функції забезпечують парсинг повідомлень, додавання, видалення фінансових записів, перегляд даних користувача, тобто відбувається взаємодія з базою даних: вивід даних, додавання, видалення та редагування.

Класи:

- class Categories;
- class CreateCategory;
- class CategoryMessage;
- class Category;
- class Message;
- class BudgetMessage;
- class IncomeExpense;
- class UserData.

Методи:

- _parse_user_data;
- parse_category;

- _parse_message;
- _parse_budget_message;
- check_user_exists;
- add_user;
- add_expense;
- edit_budget;
- add_incomes;
- create_category_finance;
- see_categories;
- delete_income;
- _get_now_formatted;
- _get_now_datetime;
- set_default_budget;
- get_budget_month_limit;
- get_budget_daily_limit;
- today_expenses;
- this_week_expenses;
- this_month_expenses;
- today_incomes;
- this_week_incomes;
- this_month_incomes.

Роботу в модулі з БД, представлено на прикладі функції *this_month_expenses*, що забезпечує «витягування» всіх витрат за поточний місяць.

```
def this_month_expenses(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.expense_id, a.amount, b.category_name '
            f'FROM expenses a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (MONTH(date_time) = MONTH(CURDATE()) AND '
            f'YEAR(date_time) = YEAR(CURDATE()) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]
```

Рис. 4.1.2 – Приклад SELECT в модулі Finances.py

```
def add_incomes(message, user_id):
    parsed_message = _parse_message(message)
    category = Categories().get_category(
        parsed_message.category_text)
    db.insert(
        "incomes",
        {
            "amount": parsed_message.amount,
            "date_time": _get_now_formatted(),
            "category": category.codename,
            "user_id": str(user_id)
        }
    )
    return IncomeExpense(id=None, amount=parsed_message.amount, category_name=category.name)
```

Рис. 4.1.3 – Приклад INSERT в модулі Finances.py

У модулі *Finances.py* insert, delete, fetchall здійснюється через перезавантаженні методи бібліотеки pymysql в модулі *databases.py*

Модуль *Statistic.py* забезпечує аналіз та виведення його результатів стосовно витрат та прибутків користувача за певний період часу. Результат подається у вигляді

графіку з кривими витрат та прибутку, зображення прикріплюються сумою *expenses & incomes* та чистий дохід (анг. *pure profit*).

Методи:

- _get_formatted;
- merging_list;
- delete_stats_image;
- get_week_expenses_for_stats;
- get_week_incomes_for_stats;
- get_month_expenses_for_stats;
- get_month_incomes_for_stats;
- week_data_for_stats;
- month_data_for_stats;
- stats_for_current_week;
- stats_for_current_month;
- create_diagram_for_stats;
- calculating_results;
- resulting_for_the_current_week;
- resulting_for_the_current_month.

Робота в цьому модулі з базою даних подібна до роботи в *Finances.py*, проте призначення функцій та значення, що вони повертають відрізняється.

```
def get_week_expenses_for_stats(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT SUM(amount), CAST(date_time AS DATE) AS Date_, date_time FROM expenses '
            f'WHERE yearweek(date_time, 1) = yearweek(CURDATE(), 1) AND user_id = {user_id} '
            f'GROUP BY CAST(date_time AS DATE) '
            f'ORDER BY date_time ASC'
        )
        rows = cursor.fetchall()
        return [i[0] for i in rows], [_get_formatted(j[1]) for j in rows]
```

Рис. 4.1.2 – Приклад SELECT в модулі Finances.py

В модулі *databeses.py* виконано перезавантаження стандартних функцій бібліотеки *pymysql*, призначених для роботи з базами даних, а саме з спроектованими в MySQL. Хоч функції стандартні, проте тут вони перезавантажені так, щоб з їх допомогою можна було просто виконувати подібні запити без повторів, тобто ці функції є універсальними і працюють для будь-яких таблиць з бази даних з різними комбінація. Крім того, вони надають можливість виконання команди не лише для одного рядка значень, а для будь якої кількості.

Методи:

- insert
- fetchall_
- fetchone_for_budget
- update_

4.2. Опис інтерфейсу програми

Після запуску головної команди бота «*/start*» користувачу стає доступним головне меню «помічника». Бот виводить повідомлення-привітання та кнопки *Library*, *Schedule*, *Notes* та *Finance*.

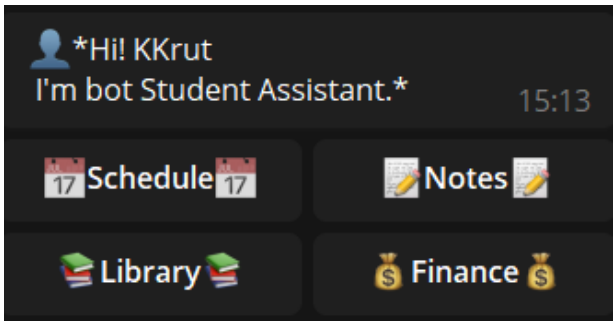

Рис. 4.2.1 – Головне меню програми «Student Bot Assistant»

Після натискання кнопки Finance стає доступним меню для менеджменту коштів. Надається можливість додати прибуток/витрати/категорію, перейти в меню статистики, бюджету та іншого

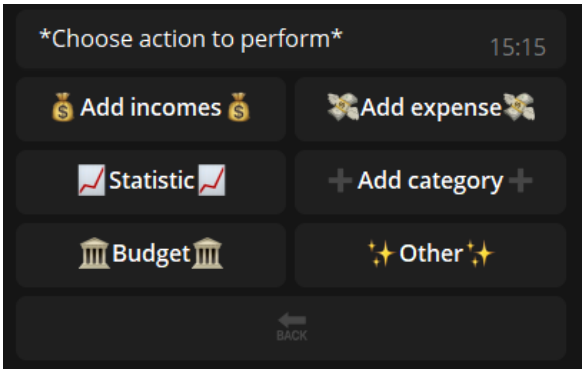

Рис. 4.2.2 – Вигляд меню «Finances»

Після натискання кнопки «Add category» користувач може додати категорію витрат/доходів.

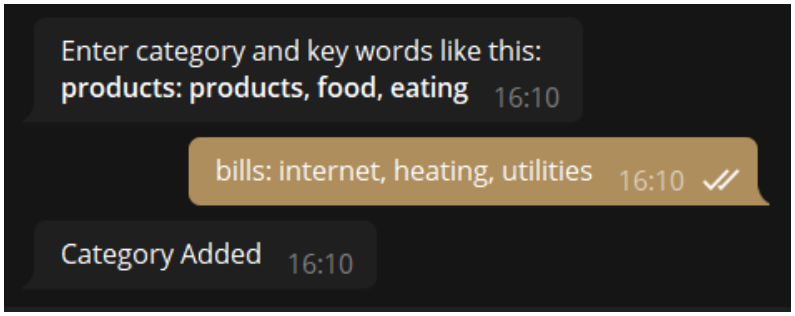

Рис. 4.2.3 – Додавання категорії. Результат «Add category»

Після натискання кнопки «Add expenses» надається можливість додати витрати. Додавання відбувається за назвою категорії, або ж за «альянсами», тобто допоміжними ключовими словами: додавання в категорію «transport» можливо за всіма введеними словами: transport, taxi, metro, bus. Кнопка «Add incomes» працює аналогічно, різниця лише в тому, в яку таблицю додаються дані – expenses чи incomes

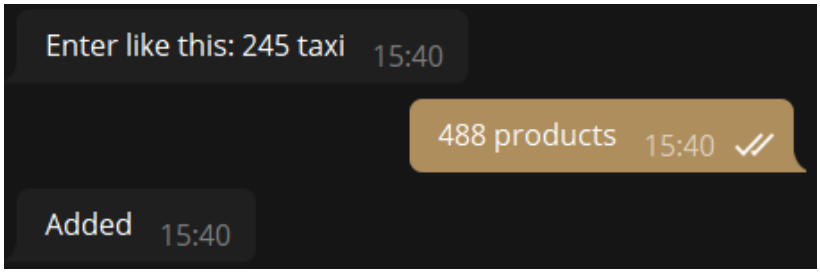

Рис. 4.2.4 – Приклад роботи «Add expenses»

Кнопка «Budget» надає можливість пергелянути свій ліміт, що за замовчуванням встановлюється *Daily: 0* та *Month: 0,* відредагувати, тобто встановити інший ліміт, та повернутись назад до фінансового меню.

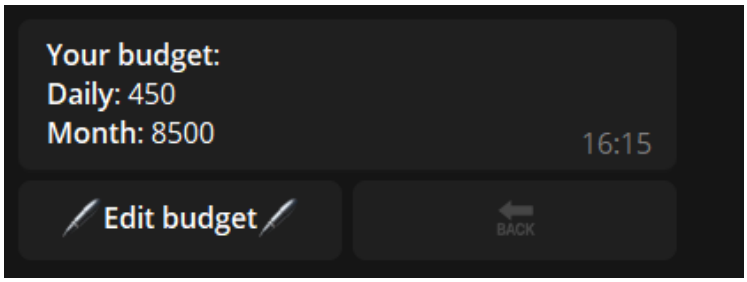

Рис. 4.2.5 – «Budget» меню

Натискання на кнопку «Statistic» переводить юзера в інше меню, де можна вибрати за який період переглянути статистичні дані: поточний тиждень чи місяць.

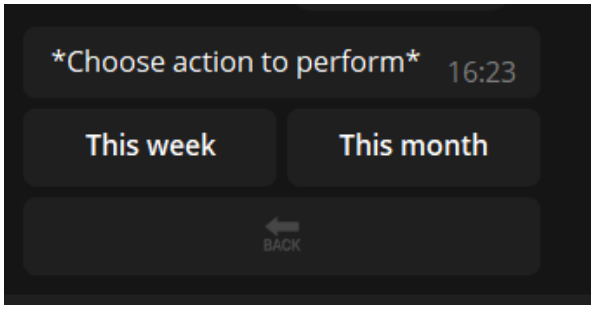

Рис. 4.2.6 – «Statistic» меню

Графіки будуються на основі даних взятих з бази даних, щодо витрат та прибутку користувача, на цих же даних відбувається обчислення.

Рис. 4.2.6 – Приклад роботи статистики («This month»)

Меню «Other» дає можливість переглянути користувачеві свої категорії, витрати чи прибутки за поточний день, тиждень чи місяць, або ж повернутись до фінансового меню.

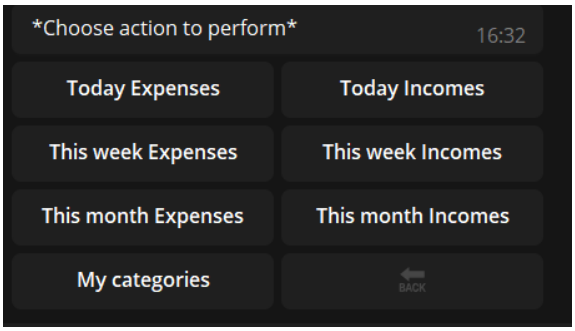

Рис. 4.2.7 – Меню «Other»

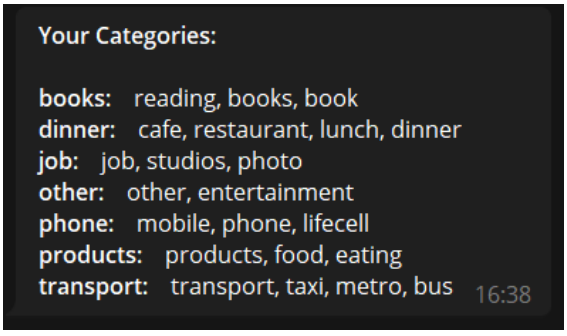

Рис. 4.2.8 – Перегляд власних категорій. Кнопка «My categories»

При перегляді прибутку або витрат надається можливість їх видалити зі своєї бази даних, видалення відбувається дуже просто – в один клік.

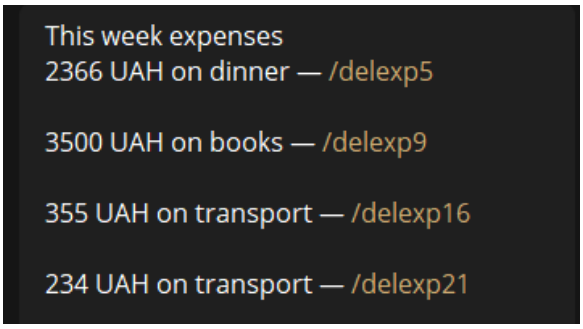

Рис. 4.2.9 – Приклад роботи кнопки «This week expenses»

Натиснувши на команду навпроти витрати чи прибутку можна легко її видалити зі свого списку.

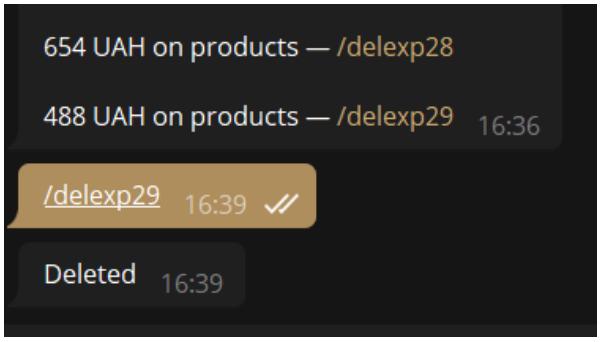

Рис. 4.2.10 – Приклад видалення

ВИСНОВКИ

Під час виконання цієї курсової роботи вдалося набути нові та вдосконалити вже здобуті знання і навички з створення та взаємодії з базами даних за допомогою мови структурованих запитів SQL на основі додатку MySQL. Крім того, здобуто досвід роботи з Framework Aiogram для створення Telegram ботів, здобуто досвід роботи з бібліотекою pymysql.

Метою роботи було створення програмного продукту «Student Bot Assistant», що допомагав би в організації власного розкладу, веденні нотаток та організації коштів, що за допомогою створеного функціоналу давав би можливість додавати, редагувати та видаляти інформацію користувача з бази даних.

Сьогодні в нашому світі, що невпинно швидко рухається й змінюється дуже важливо вміти правильно організувати свій час, розклад, вміти слідкувати за власним добробутом, тому такий, однозначно, буде корисним для будь-кого, простий, інтуїтивно зрозумілий інтерфейс та можливість налаштування – зручним у використані, а доступність в одному із найпопулярніших мессенджерів світу робить такий додаток доступним для кожного.

У ході виконання курсової роботи було реалізовано програму для допомоги у навчанні та в повсякденному житті.

СПИСКИ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Вікіпедія – вільна енциклопедія [Електронний ресурс] https://inlnk.ru/agg1oj
2. Вікіпедія – вільна енциклопедія [Електронний ресурс] https://inlnk.ru/BppY
3. Mysql Documentation [Електронний ресурс] https://dev.mysql.com/doc/
4. Mysql [Електронний ресурс] https://www.mysql.com/
5. https://www.jetbrains.com/ru-ru/pycharm/features/

ДОДАТОК 1

Код програми

НТУУ «КПІ ім. Ігоря Сікорського» ТЕФ АПЕПС ТІ-01

Листів 3

Київ – 2021

Модуль *markup.py*

```
from aiogram.types import InlineKeyboardMarkup, InlineKeyboardButton

"""INLINE KEYBOARD BUTTONS"""

inline_button_notes = InlineKeyboardButton('📝Notes📝', callback_data='📝Notes📝')
inline_button_library = InlineKeyboardButton('🏛Library🏛', callback_data='🏛Library🏛')
inline_button_schedule = InlineKeyboardButton('📆Schedule📆', callback_data='📆Schedule📆')
inline_button_finance = InlineKeyboardButton('💰Finance💰', callback_data='💰Finance💰')
inline_button_back = InlineKeyboardButton('🔙', callback_data='🔙')
inline_keyboard_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_schedule,  inline_button_notes,
inline_button_library, inline_button_finance)

inline_button_check_notes = InlineKeyboardButton("🔎Search note🔎", callback_data="🔎Search note🔎")
inline_button_add_note = InlineKeyboardButton('➕Add note➕', callback_data='➕Add note➕')
inline_button_delete_note = InlineKeyboardButton("🔥Delete note🔥", callback_data="🔥Delete note🔥")
inline_button_edit_note = InlineKeyboardButton("🖊Edit note🖊", callback_data="🖊Edit note🖊")
inline_keyboard_note_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_check_notes,
inline_button_add_note, inline_button_edit_note, inline_button_delete_note, inline_button_back)

inline_button_search_by_name = InlineKeyboardButton("🔎Search by name🔎", callback_data="🔎Search by
name🔎")
inline_button_search_by_theme = InlineKeyboardButton("🔎Search by theme🔎", callback_data="🔎Search by
theme🔎")
inline_button_show_all = InlineKeyboardButton("🔎Show all notes🔎", callback_data="🔎Show all notes🔎")
inline_keyboard_search_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_search_by_name,
                                                    inline_button_search_by_theme,
                                                    inline_button_show_all, inline_button_back)

inline_button_schedule_settings = InlineKeyboardButton("🛠Settings🛠", callback_data="🛠Settings🛠")
inline_button_schedule_currentday = InlineKeyboardButton("🕐Today schedule🕐",
                                                    callback_data="🕐Today schedule🕐")
inline_button_schedule_nextday = InlineKeyboardButton("📆Next day schedule📆", callback_data="📆Next day
schedule📆")
inline_button_schedule_next = InlineKeyboardButton("▫Next pair▫", callback_data="▫Next pair▫")
inline_button_schedule_currentweek = InlineKeyboardButton("▫This week schedule▫", callback_data="▫This week
schedule▫")
inline_button_schedule_nextweek = InlineKeyboardButton("⏺Next week schedule⏺", callback_data="⏺Next week
schedule⏺")
inline_keyboard_schedule_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_schedule_settings,
inline_button_schedule_currentday, inline_button_schedule_nextday, inline_button_schedule_next,
inline_button_schedule_currentweek, inline_button_schedule_nextweek, inline_button_back)

inline_button_add_group = InlineKeyboardButton("➕Add group➕", callback_data="➕Add group➕")
inline_button_delete_group = InlineKeyboardButton("➖Delete group➖", callback_data="➖Delete group➖")
# inline_button_on_notification = InlineKeyboardButton("📨On notification📨", callback_data="📨On
notification📨")
# inline_button_off_notification = InlineKeyboardButton("📵Off notification📵", callback_data="📵Off
notification📵")
inline_button_add_pair = InlineKeyboardButton("➕Add schedule➕", callback_data="➕Add schedule➕")
inline_button_add_monday = InlineKeyboardButton("Monday", callback_data="Monday")
inline_button_add_tuesday = InlineKeyboardButton("Tuesday", callback_data="Tuesday")
```

```
inline_button_add_wednesday = InlineKeyboardButton("Wednesday", callback_data="Wednesday")
inline_button_add_thursday = InlineKeyboardButton("Thursday", callback_data="Thursday")
inline_button_add_friday = InlineKeyboardButton("Friday", callback_data="Friday")
inline_button_add_saturday = InlineKeyboardButton("Saturday", callback_data="Saturday")

inline_keyboard_schedule_settings = InlineKeyboardMarkup(row_width=2).add(inline_button_add_group,
inline_button_delete_group, inline_button_add_pair, inline_button_back)

inline_keyboard_day_of_week = InlineKeyboardMarkup(row_width=2).add(inline_button_add_monday,
inline_button_add_tuesday, inline_button_add_wednesday, inline_button_add_thursday, inline_button_add_friday,
inline_button_add_saturday, inline_button_back)

inline_button_first = InlineKeyboardButton("First", callback_data="First")
inline_button_second = InlineKeyboardButton("Second", callback_data="Second")
inline_button_third = InlineKeyboardButton("Third", callback_data="Third")
inline_button_fourth = InlineKeyboardButton("Fourth", callback_data="Fourth")
inline_button_fifth = InlineKeyboardButton("Fifth", callback_data="Fifth")
inline_button_sixth = InlineKeyboardButton("Sixth", callback_data="Sixth")

inline_keyboard_pair = InlineKeyboardMarkup(row_width=2).add(inline_button_first, inline_button_second,
inline_button_third, inline_button_fourth, inline_button_fifth, inline_button_sixth, inline_button_back)

inline_button_week = InlineKeyboardButton("Week", callback_data="Week")
inline_button_subject_name = InlineKeyboardButton("Subject name", callback_data="Subject name")
inline_button_type_of_class = InlineKeyboardButton("Type of class", callback_data="Type of class")
inline_button_audience = InlineKeyboardButton("Audience", callback_data="Audience")
inline_button_teacher = InlineKeyboardButton("Teacher", callback_data="Teacher")
inline_button_ready = InlineKeyboardButton("✅Ready✅", callback_data="✅Ready✅")

inline_keyboard_add_pair = InlineKeyboardMarkup(row_width=2).add(inline_button_week,
inline_button_subject_name, inline_button_type_of_class, inline_button_audience, inline_button_teacher,
inline_button_ready)

inline_button_even = InlineKeyboardButton("EVEN", callback_data="EVEN")
inline_button_odd = InlineKeyboardButton("ODD", callback_data="ODD")
inline_keyboard_week_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_even, inline_button_odd)

inline_button_lecture = InlineKeyboardButton("Lecture", callback_data="Lecture")
inline_button_lab = InlineKeyboardButton("Lab", callback_data="Lab")
inline_button_practice = InlineKeyboardButton("Practice", callback_data="Practice")
inline_keyboard_type_of_lesson_menu = InlineKeyboardMarkup(row_width=2).add(inline_button_lecture,
inline_button_lab, inline_button_practice, inline_button_back)

inline_button_add_finance_category = InlineKeyboardButton('➕Add category➕', callback_data='➕Add category➕')
inline_button_add_expense = InlineKeyboardButton('💸Add expense💸', callback_data='💸Add expense💸')
inline_button_add_incomes = InlineKeyboardButton('💰Add incomes💰', callback_data='💰Add incomes💰')
inline_button_budget = InlineKeyboardButton('🏦Budget🏦', callback_data='🏦Budget🏦')
inline_button_finance_statistic = InlineKeyboardButton('📈Statistic📈', callback_data='📈Statistic📈')
inline_button_finance_other = InlineKeyboardButton('✨Other✨', callback_data='OTHER_FINANCE_MENU')
inline_button_back_to_finance = InlineKeyboardButton('🔙', callback_data='BACK_TO_FINANCE')

inline_keyboard_finance_menu = InlineKeyboardMarkup(row_width=2).add(
    inline_button_add_incomes, inline_button_add_expense, inline_button_finance_statistic,
    inline_button_add_finance_category, inline_button_budget, inline_button_finance_other,
    inline_button_back
)
```

```
inline_button_statistic_by_week = InlineKeyboardButton('This week', callback_data='WEEK_STATISTIC')
inline_button_statistic_by_month = InlineKeyboardButton('This month', callback_data='MONTH_STATISTIC')

inline_keyboard_statistic_menu = InlineKeyboardMarkup(row_width=2).add(
    inline_button_statistic_by_week, inline_button_statistic_by_month,
    inline_button_back_to_finance
)

inline_button_edit_budget = InlineKeyboardButton('✎Edit budget✎', callback_data='✎Edit budget✎')

inline_keyboard_budget_menu = InlineKeyboardMarkup(row_width=2).add(
    inline_button_edit_budget, inline_button_back_to_finance
)

inline_button_see_today_expenses = InlineKeyboardButton('Today Expenses', callback_data='TODAY_EXPENSES')
inline_button_see_week_expenses = InlineKeyboardButton('This week Expenses',
callback_data='WEEK_EXPENSES')
inline_button_see_month_expenses = InlineKeyboardButton('This month Expenses',
callback_data='MONTH_EXPENSES')

inline_button_see_today_incomes = InlineKeyboardButton('Today Incomes', callback_data='TODAY_INCOMES')
inline_button_see_week_incomes = InlineKeyboardButton('This week Incomes', callback_data='WEEK_INCOMES')
inline_button_see_month_incomes = InlineKeyboardButton('This month Incomes',
callback_data='MONTH_INCOMES')


inline_button_back_to_other_finance = InlineKeyboardButton('🔙', callback_data='BACK_TO_OTHER_FINANCE')
inline_button_see_categories = InlineKeyboardButton('My categories', callback_data='SEE_CATEGORIES')


inline_keyboard_other_menu = InlineKeyboardMarkup(row_width=2).add(
    inline_button_see_today_expenses, inline_button_see_today_incomes,
    inline_button_see_week_expenses, inline_button_see_week_incomes,
    inline_button_see_month_expenses, inline_button_see_month_incomes,
    inline_button_see_categories, inline_button_back_to_finance
)
```

Модуль *Bot_Assistant.py*

```
from aiogram.dispatcher import FSMContext
from aiogram.types import InlineKeyboardMarkup
import Finances
import Statistic
import bd
import exceptions
from imports import *
import config
import logging
from datetime import datetime, timedelta as tmd
from config import connection
from aiogram import Bot, Dispatcher, executor, types

from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.dispatcher.filters.state import State, StatesGroup
import asyncio
import markup

logging.basicConfig(level=logging.INFO)
bot = Bot(token=config.TOKEN)
dp = Dispatcher(bot, storage=MemoryStorage())


class GroupAdd(StatesGroup):
    AddGroup = State()
    SelectGroup = State()


class NoteSearch(StatesGroup):
    SearchNote = State()


class NoteSearchByTitle(StatesGroup):
    SearchByTitle = State()


class NoteEdit(StatesGroup):
    DeleteNote = State()
    DeleteTopic = State()


class EditNote(StatesGroup):
    SearchThemes = State()
    SearchTitle = State()
    EnterText = State()


class Note(StatesGroup):
    AddTheme = State()
    AddName = State()
    AddText = State()


class AddPair(StatesGroup):
    AddSubject = State()
```

```
class AddAudiences(StatesGroup):
    AddAudience = State()


class AddTeachers(StatesGroup):
    AddTeacher = State()


class HandlerIncomes(StatesGroup):
    AddIncomesState = State()


class HandlerExpenses(StatesGroup):
    AddExpensesState = State()


class HandlerStatistic(StatesGroup):
    StatisticState = State()


class HandlerCategory(StatesGroup):
    CategoriesState = State()


class HandlerBudget(StatesGroup):
    BudgetState = State()


class HandlerOtherFinances(StatesGroup):
    OtherState = State()


list_of_themes = []
selected_day = 0
selected_pair = 0
selected_week = 0
selected_type_of_class = 0
selected_subject = 0
selected_audience = 0
selected_teacher = 0
selected_audience_id = 0
selected_subject_id = 0
selected_teacher_id = 0
update_week = False


@dp.message_handler(commands=['start'])
async def send_welcome_message(message: types.Message):
    await message.delete()
    if not Finances.check_user_exists(str(message.from_user.id)):
        Finances.add_user(message['from'])
        Finances.set_default_budget(str(message.from_user.id))
    await bot.send_message(
        message.from_user.id, f"👤*Hi! {message.from_user.first_name if message.from_user.first_name else "} "
                              f"{message.from_user.last_name if message.from_user.last_name else "}\n "
```

```
                    f"I'm bot Student Assistant.*", parse_mode="HTML",
        reply_markup=markup.inline_keyboard_menu
    )


@dp.callback_query_handler(text="📝Notes📝")
async def note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>Choose action to perform</b>", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_note_menu)


@dp.callback_query_handler(text="➕Add note➕")
async def add_note(call: types.CallbackQuery):
    await call.message.delete()
    selected_theme_id = 0
    selected_note_id = 0

    with connection.cursor() as cursor:

        select_id = f"""SELECT themes FROM themes WHERE user_id={call.from_user.id}"""
        cursor.execute(select_id)
        result = cursor.fetchall()
        global list_of_themes
        list_of_themes = []
        if result:
            for themes in result:
                for theme in themes:
                    list_of_themes.append(theme)
            await bot.send_message(call.from_user.id, "<b>Available themes:</b>\n" + '\n'.join(list_of_themes),
                                   parse_mode="HTML")
        await bot.send_message(call.from_user.id, "<b>📝Enter THEME of note📝</b>", parse_mode="HTML")
        await Note.AddTheme.set()

    @dp.message_handler(state=Note.AddTheme)
    async def add_theme(message: types.Message):
        nonlocal selected_theme_id
        global list_of_themes

        if message.text in list_of_themes:
            with connection.cursor() as curs:
                search_theme = f"SELECT id FROM themes WHERE user_id = {message.from_user.id} " \
                               f"AND themes = '{message.text}'"
                curs.execute(search_theme)
                selected_theme_id = curs.fetchone()[0]
        else:
            with connection.cursor() as curs:
                adding_theme = f"INSERT INTO themes(user_id, themes) VALUES({message.from_user.id},
'{message.text}');"
                curs.execute(adding_theme)
                connection.commit()
                search_theme = f"SELECT id FROM themes WHERE user_id = {message.from_user.id} " \
                               f"AND themes = '{message.text}'"
                curs.execute(search_theme)
                selected_theme_id = curs.fetchone()[0]
        await bot.send_message(message.from_user.id, "<b>📝Enter NAME of note📝</b>", parse_mode="HTML")
```

```
        await Note.AddName.set()

    @dp.message_handler(state=Note.AddName)
    async def add_name(message: types.Message):
        nonlocal selected_theme_id
        nonlocal selected_note_id
        with connection.cursor() as cur:

            this_title = f"""SELECT title FROM notatics WHERE themes_id={selected_theme_id}"""
            cur.execute(this_title)
            results = cur.fetchall()
            list_of_title = []
            if results:
                for titles in results:
                    for title in titles:
                        list_of_title.append(title)
        if message.text in list_of_title:
            await bot.send_message(message.from_user.id, "This name is already used on this topic, try another")
        else:

            with connection.cursor() as curs:
                adding_name = f"""INSERT INTO notatics(themes_id, title)
                VALUES({selected_theme_id}, "{message.text}")"""
                curs.execute(adding_name)
                connection.commit()
                search_note = f"""SELECT id FROM notatics WHERE themes_id = {selected_theme_id} """ \
                              f"""AND title = "{message.text}" """
                curs.execute(search_note)
                selected_note_id = curs.fetchone()[0]
                await Note.AddText.set()
                await bot.send_message(message.from_user.id, "<b>📝Enter TEXT of note📝</b>", parse_mode="HTML")

    @dp.message_handler(state=Note.AddText)
    async def add_text(message: types.Message):
        nonlocal selected_note_id
        with connection.cursor() as curs:
            adding_name = f"""UPDATE notatics SET text_notatics = "{message.text}" WHERE id =
{selected_note_id};"""
            curs.execute(adding_name)
            connection.commit()
        await Note.next()
        mes = await bot.send_message(message.from_user.id, "<b>✅Note added✅</b>", parse_mode="HTML")
        await asyncio.sleep(3)
        await mes.edit_text(f"👤<b>Hi! {call.from_user.first_name if call.from_user.first_name else ''} "
                            f"{call.from_user.last_name if call.from_user.last_name else ''}\n I'm "
                            f"bot Student Assistant.</b>", parse_mode="HTML",
                            reply_markup=markup.inline_keyboard_menu)


@dp.callback_query_handler(text="🔥Delete note🔥")
async def send_themes(call: types.CallbackQuery):
    await call.message.delete()
    topic_names_list = []
    theme_id = 0
    with connection.cursor() as cursor:
        select_id = f"""SELECT themes FROM themes WHERE user_id={call.from_user.id}"""
```

```
        cursor.execute(select_id)
        result = cursor.fetchall()
        list_themes = []
        if result:
            for themes in result:
                for theme in themes:
                    list_themes.append(theme)
            await bot.send_message(call.from_user.id, "<b>Available themes:</b>\n" + '\n'.join(list_themes),
                                   parse_mode="HTML")
        if len(list_themes):
            await bot.send_message(call.from_user.id, "<b>📝Enter THEME of note to delete it📝</b>",
parse_mode="HTML")
            await NoteEdit.DeleteNote.set()
        else:
            note = await bot.send_message(call.from_user.id, "<b>❌THERE IS NO NOTES TO DELETE❌</b>",
parse_mode="HTML")
            await asyncio.sleep(3)
            await note.edit_text(f"👤<b>Hi! {call.from_user.first_name if call.from_user.first_name else ''} "
                                 f"{call.from_user.last_name if call.from_user.last_name else ''}\n I'm "
                                 f"bot Student Assistant.</b>", parse_mode="HTML",
                                 reply_markup=markup.inline_keyboard_menu)

    @dp.message_handler(state=NoteEdit.DeleteNote)
    async def delete_note(message: types.Message):
        nonlocal list_themes

        if message.text in list_themes:
            with connection.cursor() as curs:
                nonlocal topic_names_list
                nonlocal theme_id
                select_theme_id = f"""SELECT id FROM themes WHERE themes="{message.text}" AND
user_id={message.from_user.id};"""
                curs.execute(select_theme_id)
                theme_id = curs.fetchone()[0]
                select_topics = f"""SELECT title FROM notatics WHERE themes_id= {theme_id}"""
                curs.execute(select_topics)
                topics = curs.fetchall()
                for topic in topics:
                    topic_names_list.append(topic[0])
                await bot.send_message(call.from_user.id,
                                       "<b>Available notes on this theme:</b>\n" + '\n'.join(topic_names_list),
                                       parse_mode="HTML")
                await bot.send_message(call.from_user.id, "<b>📝Enter TITLE of note to delete it📝</b>",
                                       parse_mode="HTML")
                await NoteEdit.DeleteTopic.set()

    @dp.message_handler(state=NoteEdit.DeleteTopic)
    async def delete_topic(message: types.Message):
        if message.text in topic_names_list:
            with connection.cursor() as curs:
                note_delete = f"""DELETE
                                  FROM notatics
                                  WHERE themes_id={theme_id}
                                  AND title = '{message.text}';"""
                curs.execute(note_delete)
                connection.commit()
```

```
        mes = await bot.send_message(message.from_user.id, f"<b>✅Note was successfully deleted✅</b>",
                                     parse_mode="HTML")
        await asyncio.sleep(3)
        await mes.edit_text(f"👤<b>Hi! {message.from_user.first_name if message.from_user.first_name else "} "
                            f"{message.from_user.last_name if message.from_user.last_name else "}\n I'm "
                            f"bot Student Assistant.</b>", parse_mode="HTML",
                            reply_markup=markup.inline_keyboard_menu)
        await NoteEdit.next()
        topic_names_list.remove(message.text)
        if not topic_names_list:
            with connection.cursor() as curs:
                themes_delete = f"""DELETE FROM themes WHERE id={theme_id};"""
                curs.execute(themes_delete)
                connection.commit()
    else:
        await bot.send_message(message.from_user.id, "<b>❌THERE IS NO NAMES LIKE THIS❌</b>\nTRY
AGAIN.",
                               parse_mode="HTML")


@dp.callback_query_handler(text="🔎Search note🔎")
async def search_note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>Choose action to perform</b>", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_search_menu)

    @dp.callback_query_handler(text="🔎Show all notes🔎")
    async def show_all_notes(c: types.CallbackQuery):
        tuple_list = await find_tuple_list(c)
        output_list = []
        for i in tuple_list:
            notes_text = f"Title: {i[0]}\nText: {i[1]}"
            output_list.append(notes_text)
        await bot.send_message(c.from_user.id, "<b>Notes:</b>\n" + "\n\n".join(output_list), parse_mode="HTML")

    @dp.callback_query_handler(text="🔎Search by theme🔎")
    async def search_by_theme(c: types.CallbackQuery):
        list_themes = []
        tuple_list = await find_tuple_list(c)
        with connection.cursor() as cursor:
            select_id = f"""SELECT themes FROM themes WHERE user_id={c.from_user.id}"""
            cursor.execute(select_id)
            result = cursor.fetchall()
            if result:
                for themes in result:
                    for theme in themes:
                        list_themes.append(theme)
                await bot.send_message(c.from_user.id, "<b>Available themes:</b>\n" + '\n'.join(list_themes),
                                       parse_mode="HTML")
                await bot.send_message(c.from_user.id, "<b>📝Enter THEME of note📝</b>", parse_mode="HTML")
                await NoteSearch.SearchNote.set()
            else:
                note = await bot.send_message(c.from_user.id, "<b>❌THERE IS NO THEMES❌</b>",
parse_mode="HTML")
                await asyncio.sleep(3)
                await note.edit_text(f"👤<b>Hi! {c.from_user.first_name if c.from_user.first_name else "} "
```

```
                    f"{c.from_user.last_name if c.from_user.last_name else "}\n I'm "
                    f"bot Student Assistant.</b>", parse_mode="HTML",
                    reply_markup=markup.inline_keyboard_menu)

    @dp.message_handler(state=NoteSearch.SearchNote)
    async def search_notes_for_themes(message: types.Message):
        nonlocal tuple_list
        output_list = []
        with connection.cursor() as curs:
            curs.execute(f"""SELECT id FROM themes WHERE themes = '{message.text}'""")
            this_theme_id = curs.fetchone()[0]
        for i in tuple_list:
            if this_theme_id == i[2]:
                notes_text = f"Title: {i[0]}\nText: {i[1]}"
                output_list.append(notes_text)
        await bot.send_message(message.from_user.id, "<b>Notes:</b>\n" + "\n\n".join(output_list),
                               parse_mode="HTML")
        await NoteSearch.next()

@dp.callback_query_handler(text="🔎Search by name🔎")
async def search_by_name(call_this: types.CallbackQuery):
    tuple_list = await find_tuple_list(call_this)
    this_title = []
    iteration_list = []
    with connection.cursor() as curs:

        for i in tuple_list:
            curs.execute(f"""SELECT title FROM notatics WHERE themes_id = {i[2]}""")
            temp = curs.fetchall()[iteration_list.count(i[2])][0]
            if temp not in this_title:
                this_title.append(temp)
            iteration_list.append(i[2])
    if this_title:
        await bot.send_message(call_this.from_user.id, "<b>Titles:</b>\n" + "\n".join(this_title),
                               parse_mode="HTML")
        await bot.send_message(call_this.from_user.id, "<b>📝Enter TITLE of note📝</b>", parse_mode="HTML")
        await NoteSearchByTitle.SearchByTitle.set()
    else:
        note = await bot.send_message(call_this.from_user.id, "<b>❌THERE IS NO TITLE❌</b>",
parse_mode="HTML")
        await asyncio.sleep(3)
        await note.edit_text(f"👤<b>Hi! {call_this.from_user.first_name if call_this.from_user.first_name else "} "
                             f"{call_this.from_user.last_name if call_this.from_user.last_name else "}\n I'm "
                             f"bot Student Assistant.</b>", parse_mode="HTML",
                             reply_markup=markup.inline_keyboard_menu)

    @dp.message_handler(state=NoteSearchByTitle.SearchByTitle)
    async def search_notes_for_title(message: types.Message):
        output_list = []

        for j in tuple_list:
            if j[0] == message.text:
                notes_text = f"Title: {j[0]}\nText: {j[1]}"
                output_list.append(notes_text)

        await bot.send_message(message.from_user.id, "<b>Notes:</b>\n" + "\n\n".join(output_list),
```

```
                        parse_mode="HTML")
        await NoteSearchByTitle.next()


async def find_tuple_list(c):
    await c.message.delete()
    id_themes_list = []
    result_list = []
    with connection.cursor() as cursor:
        cursor.execute(f"""SELECT id FROM themes WHERE user_id = {c.from_user.id}""")
        result = cursor.fetchall()
    for id_themes in result:
        id_themes_list.append(id_themes[0])

    with connection.cursor() as cursor:
        for id_themes in id_themes_list:
            cursor.execute(f"""SELECT title, text_notatics, themes_id FROM notatics WHERE themes_id =
{id_themes}""")
            result = cursor.fetchall()
            result_list.append(result)
    tuple_list = []
    for i in result_list:
        for j in i:
            tuple_list.append(j)
    return tuple_list


@dp.callback_query_handler(text="🖉Edit note🖉")
async def edit_note(call: types.CallbackQuery):
    await call.message.delete()
    topic_names_list = []
    theme_id = 0
    title_this = 0
    with connection.cursor() as cursor:
        select_id = f"""SELECT themes FROM themes WHERE user_id={call.from_user.id}"""
        cursor.execute(select_id)
        result = cursor.fetchall()
        list_themes = []
        if result:
            for themes in result:
                for theme in themes:
                    list_themes.append(theme)
            await bot.send_message(call.from_user.id, "<b>Available themes:</b>\n" + '\n'.join(list_themes),
                                   parse_mode="HTML")
        if len(list_themes):
            await bot.send_message(call.from_user.id, "<b>📝Enter THEME of note to delete it📝</b>",
parse_mode="HTML")
            await EditNote.SearchThemes.set()
        else:
            note = await bot.send_message(call.from_user.id, "<b>❌YOU HAVE NO THEMES❌</b>",
parse_mode="HTML")
            await asyncio.sleep(3)
            await note.edit_text(f"👤<b>Hi! {call.from_user.first_name if call.from_user.first_name else ''} "
                                 f"{call.from_user.last_name if call.from_user.last_name else ''}\n I'm "
                                 f"bot Student Assistant.</b>", parse_mode="HTML",
                                 reply_markup=markup.inline_keyboard_menu)
```

```
@dp.message_handler(state=EditNote.SearchThemes)
async def delete_note(message: types.Message):
    nonlocal list_themes

    if message.text in list_themes:
        with connection.cursor() as curs:
            nonlocal topic_names_list
            nonlocal theme_id
            select_theme_id = f"""SELECT id FROM themes WHERE themes="{message.text}" AND
user_id={message.from_user.id};"""
            curs.execute(select_theme_id)
            theme_id = curs.fetchone()[0]
            select_topics = f"""SELECT title FROM notatics WHERE themes_id= {theme_id}"""
            curs.execute(select_topics)
            topics = curs.fetchall()
            for topic in topics:
                topic_names_list.append(topic[0])
            await bot.send_message(call.from_user.id,
                                   "<b>Available notes on this theme:</b>\n" + '\n'.join(topic_names_list),
                                   parse_mode="HTML")
            await bot.send_message(call.from_user.id, "<b>📝Enter TITLE of note to delete it📝</b>",
                                   parse_mode="HTML")
            await EditNote.SearchTitle.set()

@dp.message_handler(state=EditNote.SearchTitle)
async def delete_topic(message: types.Message):
    nonlocal title_this
    if message.text in topic_names_list:
        title_this = message.text
        await bot.send_message(message.from_user.id, "<b>Enter new note text</b>", parse_mode="HTML")
        await EditNote.EnterText.set()
    else:
        await bot.send_message(message.from_user.id, "<b>❌THERE IS NO NAMES LIKE THIS❌</b>\nTRY
AGAIN.",
                               parse_mode="HTML")

@dp.message_handler(state=EditNote.EnterText)
async def edit_text(message: types.Message):
    with connection.cursor() as curs:
        curs.execute(f"""UPDATE notatics
                    SET text_notatics = '{message.text}'
                    WHERE themes_id={theme_id}
                    AND title = '{title_this}';""")
        connection.commit()
        this_note = await bot.send_message(message.from_user.id, "<b>✅NOTE SUCCESSFULLY
EDITED✅</b>",
                                           parse_mode="HTML")

        await asyncio.sleep(3)
        await EditNote.next()
        await this_note.edit_text(
            f"👤<b>Hi! {message.from_user.first_name if message.from_user.first_name else "} "
            f"{message.from_user.last_name if message.from_user.last_name else "}\n I'm "
            f"bot Student Assistant.</b>", parse_mode="HTML",
            reply_markup=markup.inline_keyboard_menu)
```

```
@dp.callback_query_handler(text="📆Schedule📆")
async def schedule_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>Choose action to perform</b>", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_schedule_menu)


@dp.callback_query_handler(text="🔧Settings🔧")
async def schedule_settings(call: types.CallbackQuery):
    await call.message.delete()
    with connection.cursor() as cursor:
        cursor.execute(f"""SELECT group_id FROM users WHERE id = '{call.from_user.id}';""")
        group_id = cursor.fetchone()[0]
        if group_id:
            cursor.execute(f"""SELECT group_name FROM group_name WHERE id = {group_id}""")
            group_name = cursor.fetchone()[0]
        else:
            group_name = "❌"
    await bot.send_message(call.from_user.id, f"<b>Current settings</b>:\nGroup: {group_name}",
parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_schedule_settings)


@dp.callback_query_handler(text="➕Add group➕")
async def add_group(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>Enter your group</b>", parse_mode="HTML")
    founded_groups = []
    await GroupAdd.AddGroup.set()

    @dp.message_handler(state=GroupAdd.AddGroup)
    async def find_group(message: types.Message):
        nonlocal founded_groups
        groups_name = []
        groups_correct_name = []
        with connection.cursor() as cursor:
            cursor.execute("SELECT group_name FROM group_name")
            groups_name.append(cursor.fetchall())
        for group in groups_name[0]:
            groups_correct_name.append(group[0])
        for group_name in filter(re.compile(message.text.lower()).match, groups_correct_name):
            founded_groups.append(group_name)
        if not founded_groups or not message.text[:1].isalpha():
            await bot.send_message(message.from_user.id, "<b>Invalid name (at least two letters), try again</b>",
                                   parse_mode="HTML")
        else:
            await bot.send_message(message.from_user.id,
                                   "<b>Founded group:\n</b>" + '\n'.join(founded_groups),
                                   parse_mode="HTML")
            await bot.send_message(message.from_user.id, "<b>Choose group from this list, and write it</b>",
                                   parse_mode="HTML")
            await GroupAdd.SelectGroup.set()

    @dp.message_handler(state=GroupAdd.SelectGroup)
```

```
    async def select_group(message: types.Message):
        nonlocal founded_groups
        if message.text.lower() in founded_groups:
            with connection.cursor() as curs:
                curs.execute(f"""SELECT id FROM group_name WHERE group_name = "{message.text.lower()}";""")
                group_id = curs.fetchone()[0]
                curs.execute("SET FOREIGN_KEY_CHECKS=OFF;")
                curs.execute(f""" UPDATE users SET group_id = {group_id} WHERE id="{message.from_user.id}";""")
            connection.commit()
            note = await bot.send_message(message.from_user.id, "<b>✅GROUP SUCCESSFULLY ADDED✅</b>",
                                          parse_mode="HTML")
            await GroupAdd.next()
            await asyncio.sleep(3)
            await note.edit_text(f"👤<b>Hi! {message.from_user.first_name if message.from_user.first_name else "} "
                                 f"{message.from_user.last_name if message.from_user.last_name else "}\n I'm "
                                 f"bot Student Assistant.</b>", parse_mode="HTML",
                                 reply_markup=markup.inline_keyboard_menu)
        else:
            await bot.send_message(message.from_user.id,
                                   "<b>❌Invalid name, there is no such name in groups!❌\nTry again</b>",
                                   parse_mode="HTML")


@dp.callback_query_handler(text="➖Delete group➖")
async def delete_group(call: types.CallbackQuery):
    await call.message.delete()
    with connection.cursor() as cursor:
        cursor.execute("SET FOREIGN_KEY_CHECKS=OFF;")
        cursor.execute(
            f"""UPDATE users SET group_id = NULL WHERE id = '{call.from_user.id}';""")
    connection.commit()
    note = await bot.send_message(call.from_user.id, "<b>✅GROUP SUCCESSFULLY DELETED✅</b>",
                                  parse_mode="HTML")
    await asyncio.sleep(3)
    await note.edit_text(f"👤<b>Hi! {call.from_user.first_name if call.from_user.first_name else "} "
                         f"{call.from_user.last_name if call.from_user.last_name else "}\n I'm "
                         f"bot Student Assistant.</b>", parse_mode="HTML",
                         reply_markup=markup.inline_keyboard_menu)


@dp.callback_query_handler(text="➕Add schedule➕")
async def add_lesson_menu(call: types.CallbackQuery):
    with connection.cursor() as curs:
        curs.execute(f"""SELECT group_id FROM users WHERE id = '{call.from_user.id}';""")
        result = curs.fetchone()[0]
    await call.message.delete()
    if result:
        await bot.send_message(call.from_user.id, "<b>Choose day of weekday</b>",
                               reply_markup=markup.inline_keyboard_day_of_week, parse_mode="HTML")
    else:
        note = await bot.send_message(call.from_user.id, "<b>❌You can not use schedule before you choose
group❌</b>",
                                      parse_mode="HTML")
        await asyncio.sleep(3)
        await note.edit_text(f"<b>Current settings</b>:\nGroup: ❌", parse_mode="HTML",
                             reply_markup=markup.inline_keyboard_schedule_settings)
```

```
@dp.callback_query_handler(text="Monday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 1
    await add_lessons_day_of_week(call)


@dp.callback_query_handler(text="Tuesday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 2
    await add_lessons_day_of_week(call)


@dp.callback_query_handler(text="Wednesday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 3
    await add_lessons_day_of_week(call)


@dp.callback_query_handler(text="Thursday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 4
    await add_lessons_day_of_week(call)


@dp.callback_query_handler(text="Friday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 5
    await add_lessons_day_of_week(call)


@dp.callback_query_handler(text="Saturday")
async def add_monday(call: types.CallbackQuery):
    global selected_day
    selected_day = 6
    await add_lessons_day_of_week(call)


async def add_lessons_day_of_week(call):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>Add lesson</b>", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_pair)


@dp.callback_query_handler(text="First")
async def first(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 1
    await bot.send_message(call.from_user.id, "<b>ADD FIRST PAIR INFO</b>",
                           reply_markup=markup.inline_keyboard_add_pair,
```

```
                    parse_mode="HTML")


@dp.callback_query_handler(text="Second")
async def second(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 2
    await bot.send_message(call.from_user.id, "<b>ADD SECOND PAIR INFO</b>",
                    reply_markup=markup.inline_keyboard_add_pair,
                    parse_mode="HTML")


@dp.callback_query_handler(text="Third")
async def third(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 3
    await bot.send_message(call.from_user.id, "<b>ADD THIRD PAIR INFO</b>",
                    reply_markup=markup.inline_keyboard_add_pair,
                    parse_mode="HTML")


@dp.callback_query_handler(text="Fourth")
async def fourth(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 4
    await bot.send_message(call.from_user.id, "<b>ADD FOURTH PAIR INFO</b>",
                    reply_markup=markup.inline_keyboard_add_pair,
                    parse_mode="HTML")


@dp.callback_query_handler(text="Fifth")
async def fifth(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 5
    await bot.send_message(call.from_user.id, "<b>ADD FIFTH PAIR INFO</b>",
                    reply_markup=markup.inline_keyboard_add_pair,
                    parse_mode="HTML")


@dp.callback_query_handler(text="Sixth")
async def sixth(call: types.CallbackQuery):
    await call.message.delete()
    global selected_pair
    selected_pair = 6
    await bot.send_message(call.from_user.id, "<b>ADD SIXTH PAIR INFO</b>",
                    reply_markup=markup.inline_keyboard_add_pair,
                    parse_mode="HTML")


@dp.callback_query_handler(text="Week")
async def week(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, f"<b>SELECT WEEK</b>", parse_mode="HTML",
```

```
                    reply_markup=markup.inline_keyboard_week_menu)


@dp.callback_query_handler(text="EVEN")
async def even(call: types.CallbackQuery):
    await choose_week(call, 1, "EVEN")


@dp.callback_query_handler(text="ODD")
async def odd(call: types.CallbackQuery):
    await choose_week(call, 2, "ODD")


async def choose_week(call, select_week, string_week):
    await call.message.delete()
    global selected_week
    global update_week

    with connection.cursor() as curs:
        curs.execute(
            f"""SELECT week_number FROM schedules
            WHERE user_id='{call.from_user.id}' AND day_of_week = {selected_day} AND class_id =
{selected_pair};""")
        information = curs.fetchall()
        weeks = [x[0] for x in information]
    if select_week in weeks:
        update_week = True
    selected_week = select_week
    note = await bot.send_message(call.from_user.id, f"<b>✅YOU SELECT {string_week} WEEK✅</b>",
parse_mode="HTML")
    await asyncio.sleep(3)
    await note.edit_text("<b>ADD PAIR INFO</b>",
                         reply_markup=markup.inline_keyboard_add_pair,
                         parse_mode="HTML")


@dp.callback_query_handler(text="Type of class")
async def type_of_class(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, f"<b>SELECT TYPE OF LESSON</b>", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_type_of_lesson_menu)


@dp.callback_query_handler(text="Lecture")
async def lecture(call: types.CallbackQuery):
    await select_type(call, 2, "LECTURE")


@dp.callback_query_handler(text="Lab")
async def lab(call: types.CallbackQuery):
    await select_type(call, 3, "LAB")


@dp.callback_query_handler(text="Practice")
async def practice(call: types.CallbackQuery):
    await select_type(call, 1, "PRACTICE")
```

```
async def select_type(call, select_type_class, string_type_of_class):
    await call.message.delete()
    global selected_type_of_class
    selected_type_of_class = select_type_class
    note = await bot.send_message(call.from_user.id, f"<b>✅YOU SELECT {string_type_of_class}✅</b>",
                                  parse_mode="HTML")
    await asyncio.sleep(3)
    await note.edit_text("<b>ADD PAIR INFO</b>",
                         reply_markup=markup.inline_keyboard_add_pair,
                         parse_mode="HTML")


@dp.callback_query_handler(text="Subject name")
async def subject_name(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>🖊Enter subject🖊</b>", parse_mode="HTML")
    await AddPair.AddSubject.set()

    @dp.message_handler(state=AddPair.AddSubject)
    async def add_subject(message: types.Message):
        global selected_subject
        selected_subject = message.text
        await AddPair.next()
        note = await bot.send_message(call.from_user.id, f"<b>✅YOU SELECT SUBJECT✅</b>",
                                      parse_mode="HTML")
        await asyncio.sleep(3)
        await note.edit_text("<b>ADD PAIR INFO</b>",
                             reply_markup=markup.inline_keyboard_add_pair,
                             parse_mode="HTML")


@dp.callback_query_handler(text="Audience")
async def audience(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>🖊Enter audience🖊</b>", parse_mode="HTML")
    await AddAudiences.AddAudience.set()

    @dp.message_handler(state=AddAudiences.AddAudience)
    async def add_audience(message: types.Message):
        global selected_audience
        selected_audience = message.text
        await AddAudiences.next()

        note = await bot.send_message(call.from_user.id, f"<b>✅YOU SELECT AUDIENCE✅</b>",
                                      parse_mode="HTML")
        await asyncio.sleep(3)
        await note.edit_text("<b>ADD PAIR INFO</b>",
                             reply_markup=markup.inline_keyboard_add_pair,
                             parse_mode="HTML")


@dp.callback_query_handler(text="Teacher")
async def teacher(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "<b>🖊Enter teacher🖊</b>", parse_mode="HTML")
```

```
    await AddTeachers.AddTeacher.set()

    @dp.message_handler(state=AddTeachers.AddTeacher)
    async def add_audience(message: types.Message):
        global selected_teacher
        selected_teacher = message.text
        await AddTeachers.next()

        note = await bot.send_message(call.from_user.id, f"<b>✅YOU SELECT TEACHER✅</b>",
                                      parse_mode="HTML")
        await asyncio.sleep(3)
        await note.edit_text("<b>ADD PAIR INFO</b>",
                             reply_markup=markup.inline_keyboard_add_pair,
                             parse_mode="HTML")


@dp.callback_query_handler(text="✅Ready✅")
async def ready(call: types.CallbackQuery):
    await call.message.delete()
    global selected_day, selected_pair, selected_type_of_class, selected_week, selected_audience, selected_subject, \
        selected_teacher, selected_audience_id, selected_subject_id, selected_teacher_id
    if selected_subject:
        selected_subject_id = await push_info('subjects', 'subject_name', selected_subject)
    if selected_audience:
        selected_audience_id = await push_info('audiences', 'audience', selected_audience)
    if selected_audience:
        selected_teacher_id = await push_info('teacher', 'teacher_name', selected_teacher)
    with connection.cursor() as cursor:
        cursor.execute(f"""SELECT group_id FROM users WHERE id = {call.from_user.id}""")
        group_id = cursor.fetchone()
    note1 = 0
    note2 = 0
    note3 = 0
    note4 = 0
    note5 = 0

    if not selected_type_of_class:
        note1 = await bot.send_message(call.from_user.id,
                                       "<b>You must enter all fields. Please, select type of class. </b>",
                                       parse_mode="HTML")

    if not selected_week:
        note2 = await bot.send_message(call.from_user.id,
                                       "<b>You must enter all fields. Please, select week, odd or even.</b>",
                                       parse_mode="HTML")

    if not selected_audience_id:
        note3 = await bot.send_message(call.from_user.id,
                                       "<b>You must enter all fields. Please, write down audience.</b>",
                                       parse_mode="HTML")

    if not selected_subject_id:
        note4 = await bot.send_message(call.from_user.id,
                                       "<b>You must enter all fields. Please, write down subject.</b>",
                                       parse_mode="HTML")

    if not selected_teacher_id:
```

```
        note5 = await bot.send_message(call.from_user.id,
                                       "<b>You must enter all fields. Please, write down teacher.</b>",
                                       parse_mode="HTML")

    if not selected_audience_id or not selected_week or not selected_audience_id or not selected_subject_id \
            or not selected_teacher_id:
        await asyncio.sleep(3)
        if not isinstance(note1, int):
            await note1.delete()
        if not isinstance(note2, int):
            await note2.delete()
        if not isinstance(note3, int):
            await note3.delete()
        if not isinstance(note4, int):
            await note4.delete()
        if not isinstance(note5, int):
            await note5.delete()

        await bot.send_message(call.from_user.id, "<b>ADD PAIR INFO</b>",
reply_markup=markup.inline_keyboard_add_pair,
                               parse_mode="HTML")
    else:
        await add_schedules(call, selected_day, selected_pair, selected_type_of_class, selected_week,
                            selected_audience_id, selected_subject_id, call.from_user.id, group_id[0],
                            selected_teacher_id)

        selected_subject = 0
        selected_audience = 0
        selected_day = 0
        selected_pair = 0
        selected_type_of_class = 0
        selected_week = 0
        selected_teacher = 0

        await bot.send_message(call.from_user.id, "<b>Choose day of weekday</b>",
                               reply_markup=markup.inline_keyboard_day_of_week, parse_mode="HTML")


async def push_info(table, line, selected):
    with connection.cursor() as cursor:
        cursor.execute(f"""SELECT id FROM {table} WHERE {line}='{selected}'""")
        result = cursor.fetchone()
    if result:
        return result[0]
    else:
        with connection.cursor() as cur:
            cur.execute(f"""INSERT INTO {table}({line}) VALUES("{selected}");""")
            connection.commit()
        with connection.cursor() as c:
            c.execute(f"""SELECT id FROM {table} WHERE {line} = '{selected}';""")
            return c.fetchone()[0]


async def add_schedules(call, day, pair, type_class, weeks, audiences, subject, user_id, group_id, teacher_id):
    global update_week
    if update_week:
        with connection.cursor() as cur:
```

```
            cur.execute(
                f"""UPDATE schedules SET group_id = {group_id}, subject_name = {subject}, type_of_class =
{type_class},
                audience = {audiences}, teacher = {teacher_id} WHERE user_id = '{call.from_user.id}'
                AND day_of_week = {selected_day} AND class_id = {selected_pair} AND week_number =
{selected_week}""")
            connection.commit()
        update_week = False
    else:
        with connection.cursor() as curs:
            curs.execute(
                f"""INSERT INTO schedules(
                day_of_week, class_id, type_of_class, week_number ,audience, subject_name, user_id, group_id, teacher)
                VALUES({day}, {pair}, {type_class}, {weeks}, {audiences}, {subject},
                '{user_id}', {group_id}, {teacher_id});"""
            )
            connection.commit()


@dp.callback_query_handler(text="⏲Today schedule⏲")
async def today_schedule(call: types.CallbackQuery):
    await call.message.delete()
    today = datetime.today().isoweekday()
    today_week = await even_or_odd()
    group_id_number = 0
    today_schedules = []
    await day_schedule(call, today, today_week, group_id_number, today_schedules)


async def even_or_odd():
    now = datetime.now()
    sep = datetime(now.year if now.month >= 9 else now.year - 1, 9, 1)
    return 2 if not ((((now - tmd(days=now.weekday())) -
                       (sep - tmd(days=sep.weekday()))).days // 7) % 2) else 1


async def day_schedule(call, day, weeks, group_id, today_schedules):
    with connection.cursor() as curs:
        curs.execute(f"""SELECT group_id FROM users WHERE id = '{call.from_user.id}';""")
        group_id = curs.fetchone()[0]
    if day == 7:
        await bot.send_message(call.from_user.id, f"<b>Неділя\nВихідний\nLesson: ❌</b>")
    else:
        with connection.cursor() as cursor:
            cursor.execute(
                f"""SELECT class_id, type_of_class,
                audience, subject_name, teacher FROM schedules
                WHERE day_of_week = {day} AND user_id = "{call.from_user.id}" AND group_id = {group_id}
                AND week_number = {weeks};""")
            today_info = cursor.fetchall()
        with connection.cursor() as cur:
            cur.execute(f"""SELECT name_day_of_week FROM days_of_week WHERE id = {day};""")
            day = cur.fetchone()[0]
            today_schedules.append(day)
            for today_information in today_info:
                cur.execute(f"""SELECT id, time_of_start, time_of_end FROM classes WHERE id =
{today_information[0]}""")
```

```
            time_of_lesson = cur.fetchall()
            cur.execute(f"""SELECT subject_name FROM subjects WHERE id = {today_information[3]}""")
            subject_now = cur.fetchone()[0]
            cur.execute(f"""SELECT type_of_class FROM type_of_classes WHERE id = {today_information[1]}""")
            type_of_class_now = cur.fetchone()[0]
            cur.execute(f"""SELECT teacher_name FROM teacher WHERE id = {today_information[4]}""")
            teacher_now = cur.fetchone()[0]
            cur.execute(f"""SELECT audience FROM audiences WHERE id = {today_information[2]}""")
            audience_now = cur.fetchone()[0]
            for pair_time in time_of_lesson:
                today_schedules.append(f"Lesson: {pair_time[0]}")
                today_schedules.append("Time of start: " + str(pair_time[1]))
                today_schedules.append("Time of end: " + str(pair_time[2]))
            today_schedules.append(f"Subject: {subject_now}")
            today_schedules.append(f"Type of lesson: {type_of_class_now}")
            today_schedules.append(f"Teacher: {teacher_now}")
            today_schedules.append(f"Audience: {audience_now}\n")
        await bot.send_message(call.from_user.id, "<b>" + '\n'.join(today_schedules) + "</b>", parse_mode="HTML")


@dp.callback_query_handler(text="Next day schedule")
async def back(call: types.CallbackQuery):
    await call.message.delete()
    next_day = datetime.today().isoweekday() + 1
    next_week = await even_or_odd()
    if next_day == 8:
        next_day = 1
        if next_week == 1:
            next_week = 2
        else:
            next_week = 1
    group_id_number = 0
    next_day_schedules = []
    await day_schedule(call, next_day, next_week, group_id_number, next_day_schedules)


@dp.callback_query_handler(text="Next pair")
async def next_pair(call: types.CallbackQuery):
    await call.message.delete()
    next_pair_day = datetime.today().isoweekday()
    next_pair_weeks = await even_or_odd()
    next_pair_this_time = datetime.now().time()
    next_pair_group_id = 0
    today_schedules = []
    with connection.cursor() as curs:
        curs.execute(f"""SELECT group_id FROM users WHERE id = '{call.from_user.id}';""")
        next_pair_group_id = curs.fetchone()[0]

    today_schedules = await next_pair_schedule(call, next_pair_day, next_pair_weeks, next_pair_group_id,
                                               next_pair_this_time)
    while not today_schedules:
        next_pair_day += 1
        next_pair_this_time = datetime.strptime("00:00:00", "%H:%M:%S").time()
        if next_pair_day == 8:
            next_pair_day = 1
            if next_pair_weeks == 1:
                next_pair_weeks = 2
```

```
            else:
                next_pair_weeks = 1
        today_schedules = await next_pair_schedule(call, next_pair_day, next_pair_weeks, next_pair_group_id,
                                                   next_pair_this_time)
    await bot.send_message(call.from_user.id, "<b>" + '\n'.join(today_schedules) + "</b>", parse_mode="HTML")


async def next_pair_schedule(call, day, weeks, group_id, this_time):
    today_schedules = []
    with connection.cursor() as cursor:
        cursor.execute(
            f"""SELECT class_id, type_of_class,
            audience, subject_name, teacher FROM schedules
            WHERE day_of_week = {day} AND user_id = "{call.from_user.id}" AND group_id = {group_id}
            AND week_number = {weeks};""")
        today_info = cursor.fetchall()
    with connection.cursor() as cur:
        cur.execute(f"""SELECT name_day_of_week FROM days_of_week WHERE id = {day};""")
        day = cur.fetchone()[0]
        for today_information in today_info:
            cur.execute(f"""SELECT id, time_of_start, time_of_end FROM classes WHERE id =
{today_information[0]}""")
            time_of_lesson = cur.fetchall()
            cur.execute(f"""SELECT subject_name FROM subjects WHERE id = {today_information[3]}""")
            subject_now = cur.fetchone()[0]
            cur.execute(f"""SELECT type_of_class FROM type_of_classes WHERE id = {today_information[1]}""")
            type_of_class_now = cur.fetchone()[0]
            cur.execute(f"""SELECT teacher_name FROM teacher WHERE id = {today_information[4]}""")
            teacher_now = cur.fetchone()[0]
            cur.execute(f"""SELECT audience FROM audiences WHERE id = {today_information[2]}""")
            audience_now = cur.fetchone()[0]
            for pair_time in time_of_lesson:
                lesson_time = str(pair_time[1])
                if this_time < datetime.strptime(lesson_time, "%H:%M:%S").time():
                    today_schedules.append(day)
                    today_schedules.append(f"Lesson: {pair_time[0]}")
                    today_schedules.append("Time of start: " + str(pair_time[1]))
                    today_schedules.append("Time of end: " + str(pair_time[2]))
                    today_schedules.append(f"Subject: {subject_now}")
                    today_schedules.append(f"Type of lesson: {type_of_class_now}")
                    today_schedules.append(f"Teacher: {teacher_now}")
                    today_schedules.append(f"Audience: {audience_now}\n")
                    return today_schedules


@dp.callback_query_handler(text="□This week schedule□")
async def this_week_schedule(call: types.CallbackQuery):
    await call.message.delete()
    await week_schedule(call, await even_or_odd())


@dp.callback_query_handler(text="●Next week schedule●")
async def this_week_schedule(call: types.CallbackQuery):
    await call.message.delete()
    current_week = await even_or_odd()
    if current_week == 1:
        current_week = 2
```

```
    else:
        current_week = 1
    await week_schedule(call, current_week)


async def week_schedule(call, this_week_week):
    today_schedules = []
    with connection.cursor() as curs:
        curs.execute(f"""SELECT group_id FROM users WHERE id = '{call.from_user.id}';""")
        group_id = curs.fetchone()[0]
    with connection.cursor() as cursor:
        cursor.execute(
            f"""SELECT  day_of_week, class_id, type_of_class,
            audience, subject_name, teacher FROM schedules
            WHERE user_id = "{call.from_user.id}" AND group_id = {group_id} AND week_number =
{this_week_week};""")
        this_week_info = cursor.fetchall()
    with connection.cursor() as cur:

        for today_information in this_week_info:
            cur.execute(f"""SELECT name_day_of_week FROM days_of_week WHERE id =
{today_information[0]};""")
            day = cur.fetchone()[0]
            cur.execute(f"""SELECT id, time_of_start, time_of_end FROM classes WHERE id =
{today_information[1]}""")
            time_of_lesson = cur.fetchall()
            cur.execute(f"""SELECT subject_name FROM subjects WHERE id = {today_information[4]}""")
            subject_now = cur.fetchone()[0]
            cur.execute(f"""SELECT type_of_class FROM type_of_classes WHERE id = {today_information[2]}""")
            type_of_class_now = cur.fetchone()[0]
            cur.execute(f"""SELECT teacher_name FROM teacher WHERE id = {today_information[5]}""")
            teacher_now = cur.fetchone()[0]
            cur.execute(f"""SELECT audience FROM audiences WHERE id = {today_information[3]}""")
            audience_now = cur.fetchone()[0]
            if day not in today_schedules:
                today_schedules.append(day)
            for pair_time in time_of_lesson:
                today_schedules.append(f"Lesson: {pair_time[0]}")
                today_schedules.append("Time of start: " + str(pair_time[1]))
                today_schedules.append("Time of end: " + str(pair_time[2]))
            today_schedules.append(f"Subject: {subject_now}")
            today_schedules.append(f"Type of lesson: {type_of_class_now}")
            today_schedules.append(f"Teacher: {teacher_now}")
            today_schedules.append(f"Audience: {audience_now}\n")
    await bot.send_message(call.from_user.id, "<b>" + '\n'.join(today_schedules) + "</b>", parse_mode="HTML")


@dp.callback_query_handler(text="🔙")
async def back(call: types.CallbackQuery):
    """ back to main menu """
    await call.message.delete()
    await bot.send_message(call.from_user.id,
                           f"👤<b>Hi! {call.from_user.first_name if call.from_user.first_name else "} "
                           f"{call.from_user.last_name if call.from_user.last_name else "}\n I'm "
                           f"bot Student Assistant.</b>", parse_mode="HTML", reply_markup=markup.inline_keyboard_menu)
```

```
""" Finance handlers  """


@dp.callback_query_handler(text="BACK_TO_FINANCE")
async def back(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(
        call.from_user.id, "*Choose action to perform*", parse_mode="HTML",
        reply_markup=markup.inline_keyboard_finance_menu
    )


""" Finance handlers  """


@dp.callback_query_handler(text='💰Finance💰')
async def note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "*Choose action to perform*", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_finance_menu)


@dp.callback_query_handler(text='🏛Budget🏛')
async def note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(
        call.from_user.id, f'<b>Your budget</b>:\n<b>Daily</b>:
{Finances.get_budget_daily_limit(call.from_user.id)}\n'
                           f'<b>Month</b>: {Finances.get_budget_month_limit(call.from_user.id)}', parse_mode='HTML',
        reply_markup=markup.inline_keyboard_budget_menu
    )


@dp.callback_query_handler(text='📈Statistic📈')
async def note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "*Choose action to perform*", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_statistic_menu)


@dp.callback_query_handler(text='OTHER_FINANCE_MENU')
async def note_menu(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "*Choose action to perform*", parse_mode="HTML",
                           reply_markup=markup.inline_keyboard_other_menu)


@dp.callback_query_handler(text='💸Add expense💸')
async def add_expense_(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "Enter like this: 245 taxi")
    await HandlerExpenses.AddExpensesState.set()

    @dp.message_handler(state=HandlerExpenses.AddExpensesState)
    async def adding_expense(message: types.Message):
        try:
```

```
            Finances.add_expense(message['text'], message.from_user.id)
        except exceptions.AddExpenseError as exp:
            await message.answer(str(exp))
        await bot.send_message(message.from_user.id, 'ADDED', parse_mode='HTML')
        await HandlerExpenses.next()


@dp.callback_query_handler(text='💰Add incomes💰')
async def add_incomes(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.from_user.id, "Enter like this: 245 job")
    await HandlerIncomes.AddIncomesState.set()

    @dp.message_handler(state=HandlerIncomes.AddIncomesState)
    async def adding_incomes(message: types.Message):
        try:
            Finances.add_incomes(message['text'], message.from_user.id)
        except exceptions.AddIncomeError(str(message)) as exp:
            await message.answer(str(exp))
        await bot.send_message(message.from_user.id, 'ADDED', parse_mode='HTML')
        await HandlerIncomes.next()


@dp.callback_query_handler(text='✏Edit budget✏')
async def edit_budget(call: types.CallbackQuery):
    await call.message.delete()
    await HandlerBudget.BudgetState.set()
    await bot.send_message(call.from_user.id, "daily <i>number</i> month <i>number</i>", parse_mode="HTML")

    @dp.message_handler(state=HandlerBudget.BudgetState)
    async def editing_budget(message: types.Message):
        try:
            Finances.edit_budget(message['text'], message.from_user.id)
        except exceptions.ChangeBudgetError(str(message)) as exp:
            await message.answer(str(exp))
        await HandlerBudget.next()


@dp.callback_query_handler(text='➕Add category➕')
async def add_category(call: types.CallbackQuery):
    await call.message.delete()
    await HandlerCategory.CategoriesState.set()
    await bot.send_message(
        call.from_user.id, "Enter category and key words like this:\n<b>products: products, food, eating</b>",
        parse_mode="HTML"
    )

    @dp.message_handler(state=HandlerCategory.CategoriesState)
    async def creating_finance_category(message: types.Message):
        try:
            Finances.create_category_finance(message['text'], message.from_user.id)
        except exceptions.AddCategoryError as exp:
            await message.answer(str(exp))
            return
        await bot.send_message(message.from_user.id, 'Edited', parse_mode='HTML')
        await HandlerCategory.next()
```

```
@dp.message_handler(lambda message: message.text.startswith('/delexp'))
async def del_expense(message: types.Message):
    try:
        Finances.delete_expense(int(message.text[7:]), message.from_user.id)
    except exceptions.DeleteError(str(message)) as exp:
        await bot.send_message(message.from_user.id, f'{exp}', parse_mode='HTML')
    await bot.send_message(message.from_user.id, 'Deleted')


@dp.message_handler(lambda message: message.text.startswith('/delinc'))
async def del_expense(message: types.Message):
    try:
        Finances.delete_expense(int(message.text[7:]), message.from_user.id)
    except exceptions.DeleteError(str(message)) as exp:
        await bot.send_message(message.from_user.id, f'{exp}', parse_mode='HTML')
    await bot.send_message(message.from_user.id, 'Deleted')


@dp.callback_query_handler(text='SEE_CATEGORIES')
async def categories_viewing_handler(call: types.CallbackQuery):
    await call.message.delete()
    categories_data = Finances.see_categories(call.from_user.id)
    if not categories_data:
        await bot.send_message(call.from_user.id, 'You haven`t any category yet', parse_mode='HTML')
        return
    categories_ = [f'<b>{category.name_}:</b>    {category.category_text}' for category in categories_data]
    await bot.send_message(call.from_user.id, '<b>Your Categories:</b>\n\n' + '\n'.join(categories_),
parse_mode='HTML')


@dp.callback_query_handler(text='TODAY_EXPENSES')
async def today_expenses_handler(call: types.CallbackQuery):
    await call.message.delete()
    today_expenses_ = Finances.today_expenses(call.from_user.id)
    if not today_expenses_:
        await bot.send_message(call.from_user.id, 'Today expenses were not added', parse_mode='HTML')
        return
    today_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delexp{expense.id}'
        for expense in today_expenses_
    ]
    await bot.send_message(call.from_user.id, 'Today expenses\n' + '\n\n'.join(today_expenses_rows),
parse_mode='HTML')


@dp.callback_query_handler(text='WEEK_EXPENSES')
async def week_expenses_handler(call: types.CallbackQuery):
    await call.message.delete()
    this_week_expenses_ = Finances.this_week_expenses(call.from_user.id)
    if not this_week_expenses_:
        await bot.send_message(call.from_user.id, 'This week expenses were not added', parse_mode='HTML')
        return
    this_week_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delexp{expense.id}' for expense in
this_week_expenses_
    ]
```

```
    await bot.send_message(
        call.from_user.id, 'This week expenses\n' + '\n\n'.join(this_week_expenses_rows), parse_mode='HTML'
    )


@dp.callback_query_handler(text='MONTH_EXPENSES')
async def month_expenses_handler(call: types.CallbackQuery):
    await call.message.delete()
    this_month_expenses_ = Finances.this_month_expenses(call.from_user.id)
    if not this_month_expenses_:
        await bot.send_message(call.from_user.id, 'This month expenses were not added', parse_mode='HTML')
        return
    this_month_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delexp{expense.id}'
        for expense in this_month_expenses_
    ]
    await bot.send_message(
        call.from_user.id, 'This month expenses\n' + '\n\n'.join(this_month_expenses_rows), parse_mode='HTML'
    )


@dp.callback_query_handler(text='TODAY_INCOMES')
async def today_incomes_handler(call: types.CallbackQuery):
    await call.message.delete()
    today_expenses_ = Finances.today_incomes(call.from_user.id)
    if not today_expenses_:
        await bot.send_message(call.from_user.id, 'Today incomes were not added', parse_mode='HTML')
        return
    today_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delinc{expense.id}'
        for expense in today_expenses_
    ]
    await bot.send_message(
        call.from_user.id, 'Today incomes\n' + '\n\n'.join(today_expenses_rows), parse_mode='HTML'
    )


@dp.callback_query_handler(text='WEEK_INCOMES')
async def week_incomes_handler(call: types.CallbackQuery):
    await call.message.delete()
    this_week_expenses_ = Finances.this_week_incomes(call.from_user.id)
    if not this_week_expenses_:
        await bot.send_message(call.from_user.id, 'This week incomes were not added', parse_mode='HTML')
        return
    this_week_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delinc{expense.id}'
        for expense in this_week_expenses_
    ]
    await bot.send_message(
        call.from_user.id, 'This week incomes\n' + '\n\n'.join(this_week_expenses_rows), parse_mode='HTML'
    )


@dp.callback_query_handler(text='MONTH_INCOMES')
async def month_incomes_handler(call: types.CallbackQuery):
    await call.message.delete()
    this_month_expenses_ = Finances.this_month_incomes(call.from_user.id)
```

```
    if not this_month_expenses_:
        await bot.send_message(call.from_user.id, 'This month incomes were not added', parse_mode='HTML')
        return
    this_month_expenses_rows = [
        f'{expense.amount} UAH on {expense.category_name} — /delinc{expense.id}'
        for expense in this_month_expenses_
    ]
    await bot.send_message(
        call.from_user.id, 'This month incomes\n' + '\n\n'.join(this_month_expenses_rows), parse_mode='HTML'
    )


@dp.callback_query_handler(text='WEEK_STATISTIC')
async def this_week_statistic_handler(call: types.CallbackQuery):
    await call.message.delete()
    file_name_ = Statistic.stats_for_current_week(call.from_user.id)
    result_ = Statistic.resulting_for_the_current_week(call.from_user.id)
    await bot.send_photo(
        call.from_user.id, open(f'{file_name_}.png', 'rb'),
        caption=f'<b>Total expenses:</b> {result_[0]}\n<b>Total incomes:</b> {result_[1] - result_[2]}\n'
                f'<b>Pure profit:</b> {result_[2]}\n',
        # f'<b>Of Budget:</b> {Finances.get_budget_month_limit(call.from_user.id)}'
        parse_mode='HTML'
    )
    await asyncio.sleep(10)
    Statistic.delete_stats_image(file_name_)


@dp.callback_query_handler(text='MONTH_STATISTIC')
async def this_month_statistic_handler(call: types.CallbackQuery):
    await call.message.delete()
    file_name_ = Statistic.stats_for_current_month(call.from_user.id)
    result_ = Statistic.resulting_for_the_current_month(call.from_user.id)
    await bot.send_photo(
        call.from_user.id, open(f'{file_name_}.png', 'rb'),
        caption=f'<b>Total expenses:</b> {result_[0]}\n<b>Total incomes:</b> '
                f'{result_[1]}\n<b>Pure profit: </b>{result_[2]}\n',
        # f'<b>Of Budget:</b> {Finances.get_budget_month_limit(call.from_user.id) - result_[2]}',
        parse_mode='HTML'
    )
    await asyncio.sleep(10)
    Statistic.delete_stats_image(file_name_)


@dp.callback_query_handler(text='BACK_TO_OTHER_FINANCE')
async def back(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(
        call.from_user.id, "*Choose action to perform*", parse_mode="HTML",
        reply_markup=markup.inline_keyboard_other_menu
    )

lib = types.InlineKeyboardMarkup(row_width=1)
add = types.InlineKeyboardButton(text="Add book to list", callback_data="add")
stats = types.InlineKeyboardButton(text="View your top", callback_data="stats")
full = types.InlineKeyboardButton(text="View full list", callback_data="full")
lib.add(add, stats, full, markup.inline_button_back)
```

```
class FSMBook(StatesGroup):
    write_search = State()
    write_delete = State()


@dp.message_handler(content_types=['text', 'document', 'audio', 'photo', 'sticker', 'video', 'voice', 'unknown'],
                    state=FSMBook.write_search)
async def search(message: types.Message, state: FSMContext):
    if message.text:
        info = bd.get_info(message.text)
        await state.finish()
        if info:
            if bd.add_book(info, message.from_user.id):
                await bot.send_message(message.chat.id, "Книга-" + info[0]['Название'] + '\nЗа авторством-' + info[0][
                    'Автор'] + '\nОт издания-' + info[0]['Издательство'] + '\nБыла добавлена в ваш список')
            else:
                await bot.send_message(message.chat.id, "Это произведение уже есть у вас ,или поиск выдал не книгу")
        else:
            await bot.send_message(message.chat.id, "Я не смог найти книгу по даному запросу")
    else:
        await bot.send_message(message.chat.id, "Введите пожалуйста текст")
        await message.delete()


@dp.message_handler(content_types=['text', 'document', 'audio', 'photo', 'sticker', 'video', 'voice', 'unknown'],
                    state=FSMBook.write_delete)
async def deletess(message: types.Message, state: FSMContext):
    if message.text and not any((numb not in '1234567890') for numb in message.text):
        await state.finish()
        if bd.delete_book(int(message.text), message.from_user.id):
            await bot.send_message(message.chat.id, "Книга удалена")
        else:
            await bot.send_message(message.chat.id, "Я не смог найти книгу по даному запросу")
    else:
        await bot.send_message(message.chat.id, "Введите пожалуйста код")
        await message.delete()


@dp.callback_query_handler(text="📚Library📚")
async def library(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.message.chat.id, "Выбирайте,что именно вы хотите сделать с книжным списком
желаний:",
                           reply_markup=lib)


@dp.callback_query_handler(text="add")
async def add_(call: types.CallbackQuery):
    await call.message.delete()
    await bot.send_message(call.message.chat.id, "Напишите запрос,по которому я буду искать книгу:")
    await FSMBook.write_search.set()


@dp.callback_query_handler(text="full")
async def full_(call: types.CallbackQuery):
    await call.message.delete()
    delete = types.InlineKeyboardButton(text="Удалить книгу со списка", callback_data="delete")
```

```
    await bot.send_message(call.message.chat.id, "Список ваших интересов:\n" + bd.get_list(
        call.from_user.id) + '\nВведи код книги ,если хочешь убрать её',
                           reply_markup=InlineKeyboardMarkup(row_width=1).add(delete))


@dp.callback_query_handler(text="delete")
async def delete_(call: types.CallbackQuery):

    await bot.send_message(call.message.chat.id, "Введи код книги для удаления:")
    await FSMBook.write_delete.set()


@dp.callback_query_handler(text="stats")
async def stats_(call: types.CallbackQuery):
    await call.message.delete()
    data = bd.get_max_info(call.from_user.id)
    await bot.send_message(call.message.chat.id, "Информация:\n" + '<><><><><><><><><>\n'.join(
        '\n'.join(str(item) for item in group) for group in data.items()))


executor.start_polling(dp, skip_updates=True)

if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```

Модуль *bd.py*

```
import requests, pymysql
from bs4 import BeautifulSoup
from config import connection


def get_info(book_info):
    url = []
    data = {}
    r = requests.get('https://www.yakaboo.ua/search/?multi=0&cat=&q=' + book_info)
    html_main = BeautifulSoup(r.text, features="html.parser")
    quote_main = html_main.find('ul',
                                class_="products-grid thumbnails thumbnails_horizontal thumbnails_middle data-list")
    if quote_main:
        r = requests.get(quote_main.find_all('li')[0].find('a').get('href'))
        html_main = BeautifulSoup(r.text, features="html.parser")
    quote_book = html_main.find('table', id="product-attribute-specs-table")
    if not quote_book:
        return
    keys = ['Автор', 'Издательство', 'Язык', 'Год издания', 'Количество страниц']
    key = ''
    data['Название'] = html_main.find('div', class_="big-description block translate").find('span').text
    data['Код'] = int(html_main.find('div', class_="product-sku").find('span', itemprop="sku").text)
    ind = False
    for q in quote_book.find_all('td'):
        if (ind):
            if q.find('a'):
                url.append(q.find('a').get('href'))
            data[key] = q.text.removeprefix(' ')
            ind = False
        if (q.text in keys):
            key = q.text
            ind = True
    for check in keys:
        if check not in data:
            data[check] = 'Нет информации'
    return [data, url]


def add_book(information, telegram_id):
    data = information[0]
    url = information[1]
    if data['Издательство'] == 'Нет информации':
        return False
    with connection.cursor() as cursor:
        if len(url) == 1:
            url.append(url[0])
        else:
            html_author = BeautifulSoup(requests.get(url[0]).text, features="html.parser")
            psevdo = html_author.find('div', class_="page-title category-title").find('h1').text.removesuffix(
                ' — книги и биография').removeprefix('    ')
            create_query = f"""SELECT psevdonim FROM Author WHERE psevdonim='{psevdo}';"""
            cursor.execute(create_query)
            if not cursor.fetchall():
                create_query = f"""INSERT INTO Author (psevdonim) VALUES('{psevdo}');"""
                cursor.execute(create_query)
```

```
            quote_author = html_author.find('div', class_="product-shop span6")
            if quote_author:
                if len(quote_author.find_all('td')) > 2:
                    create_query = f"""UPDATE Author SET date_of_birth='{quote_author.find_all('td')[3].text}'
                    WHERE psevdonim='{psevdo}';"""
                    cursor.execute(create_query)
                create_query = f"""UPDATE Author SET author_description='{quote_author.find('p').text}'
                WHERE psevdonim='{psevdo}';"""
                cursor.execute(create_query)
        html_publisher = BeautifulSoup(requests.get(url[1]).text, features="html.parser")
        publisher_name = html_publisher.find('div', class_="page-title category-title").find('h1').text.removeprefix(
            '    Издательство книг ')
        create_query = f"""SELECT publisher_name FROM Publisher WHERE publisher_name='{publisher_name}';"""
        cursor.execute(create_query)
        if not cursor.fetchall():
            create_query = f"""INSERT INTO Publisher (publisher_name) VALUES('{publisher_name}');"""
            cursor.execute(create_query)
            quote_publisher = html_publisher.find('div', class_="product-shop span9")
            if quote_publisher:
                create_query = f"""UPDATE Publisher SET publisher_description='{quote_publisher.find('p').text}'
                WHERE publisher_name='{publisher_name}';"""
                cursor.execute(create_query)
        connection.commit()
        create_query = f"""SELECT book_id FROM Books WHERE book_id={data['Код']};"""
        cursor.execute(create_query)
        if not cursor.fetchall():
            create_query = \
                f"""INSERT INTO Books
(book_id,title,author,publisher,book_language,year_of_publishing,amount_of_pages)
                VALUES({data['Код']},'{data['Название']}','{data['Автор']}','{data['Издательство']}','{data['Язык']}',
{data['Год издания']},'{data['Количество страниц']}');"""
            cursor.execute(create_query)
        connection.commit()
        create_query = f"""SELECT user_id FROM LibraryUser WHERE telegram_id={telegram_id} AND
book={data['Код']};"""
        cursor.execute(create_query)
        if cursor.fetchall():
            return False
        create_query = f"""INSERT INTO LibraryUser (book,telegram_id) VALUES({data['Код']},{telegram_id});"""
        cursor.execute(create_query)
        connection.commit()
    return True


def delete_book(book_id, telegram_id):
    with connection.cursor() as cursor:
        create_query = f"""SELECT book FROM LibraryUser WHERE book={book_id} AND
telegram_id={telegram_id};"""
        cursor.execute(create_query)
        if not cursor.fetchall():
            return False
        create_query = f"""DELETE FROM LibraryUser WHERE book={book_id} AND
telegram_id={telegram_id};"""
        cursor.execute(create_query)
        connection.commit()
        return True
```

```
def get_max_info(telegram_id):
    data = {}
    keys = [['Псевдоним', 'Дата рождения', 'Описание', 'Количество'], ['Название', 'Описание', 'Количество']]
    with connection.cursor() as cursor:
        create_query = f"""SELECT psevdonim,date_of_birth,author_description,COUNT(telegram_id) AS amount
        FROM LibraryUser LEFT OUTER JOIN (SELECT book_id,psevdonim,date_of_birth,author_description
        FROM Books LEFT OUTER JOIN Author ON psevdonim=author) AS Psevdo ON book_id=book
        WHERE telegram_id={telegram_id} GROUP BY psevdonim ORDER BY amount DESC LIMIT 3;"""
        cursor.execute(create_query)
        rows = cursor.fetchall()
        if not rows:
            data['Топ 3 ваших автора'] = 'Твой список пуст'
        else:
            string = ''
            for row in rows:
                string += '--' * 29 + '\nАвтор:\n'
                author_info = '\n'.join(str(item) for item in row).split('\n')
                for i in range(len(author_info)):
                    string += keys[0][i] + ': ' + (
                        '-' if author_info[i] == 'None' or author_info[i] == 'Нет информации' else author_info[i]) + '\n'
            data['Топ 3 ваших автора'] = string

        create_query = f"""SELECT publisher_name,publisher_description,COUNT(telegram_id) AS amount
        FROM LibraryUser LEFT OUTER JOIN (SELECT book_id,publisher_name,publisher_description
        FROM Books LEFT OUTER JOIN Publisher ON publisher_name=publisher) AS Publ ON book_id=book
        WHERE telegram_id={telegram_id} GROUP BY publisher_name ORDER BY amount DESC LIMIT 3;"""
        cursor.execute(create_query)
        rows = cursor.fetchall()
        if not rows:
            data['Топ 3 ваших издательства'] = 'Твой список пуст'
        else:
            string = ''
            for row in rows:
                string += '--' * 29 + '\nИздательство:\n'
                publisher_info = '\n'.join(str(item) for item in row).split('\n')
                for i in range(len(publisher_info)):
                    string += keys[1][i] + ': ' + (
                        '--' if publisher_info[i] == 'None' or publisher_info[i] == 'Нет информации' else
                        publisher_info[i]) + '\n'
            data['Топ 3 ваших издательтва'] = string
        return data


def get_list(telegram_id):
    string = ''
    keys = ['Код', 'Название', 'Автор', 'Издательство', 'Язык', 'Год издания', 'Количество страниц']
    with connection.cursor() as cursor:
        create_query = f"""SELECT book FROM LibraryUser WHERE telegram_id={telegram_id};"""
        cursor.execute(create_query)
        if not cursor.fetchall():
            # connection.close()
            return 'Твой список пуст'
        create_query = f"""SELECT book_id,title,author,publisher,book_language,year_of_publishing,amount_of_pages
        FROM LibraryUser LEFT OUTER JOIN Books ON book_id=book WHERE telegram_id={telegram_id};"""
        cursor.execute(create_query)
        rows = cursor.fetchall()
```

```
    for row in rows:
        string += '--' * 29 + '\nКнига:\n'
        book_info = '\n'.join(str(item) for item in row).split('\n')
        for i in range(len(keys)):
            string += keys[i] + ': ' + book_info[i] + '\n'
return string
```

Модуль *Finances.py*

```
from datetime import datetime
from typing import NamedTuple, List, Optional, re
from config import connection

from imports import *
from exceptions import *
import databases as db


class Categories:
    def __init__(self):
        self._categories = self._load_categories()

    def _load_categories(self):
        return self._fill_aliases(db.fetchall_("category", "code_name category_name aliases_".split()))

    def _fill_aliases(self, categories):
        categories_result = []
        for index, category in enumerate(categories):
            aliases = category["aliases_"].split(",")
            aliases = list(filter(None, map(str.strip, aliases)))
            aliases.append(category["code_name"])
            aliases.append(category["category_name"])
            categories_result.append(
                Category(codename=category['code_name'], name=category['category_name'], aliases=aliases)
            )
        return categories_result

    @property
    def get_all_categories(self):
        return self._categories

    def get_category(self, category_name):
        category_finded = None
        others = None
        for category in self._categories:
            if category.codename == "other_":
                others = category
            for alias in category.aliases:
                if category_name in alias:
                    category_finded = category
        if not category_finded:
            category_finded = others
        return category_finded


class CreateCategory(NamedTuple):
    code_name_: str
    category_name_: str
    aliases_text_: str


class CategoryMessage(NamedTuple):
    name_: str
    category_text: str
```

```
class Category(NamedTuple):
    codename: str
    name: str
    aliases: List[str]


class Message(NamedTuple):
    amount: int
    category_text: str


class BudgetMessage(NamedTuple):
    daily_amount: int
    month_amount: int


class IncomeExpense(NamedTuple):
    id: Optional[int]
    amount: int
    category_name: str


class UserData(NamedTuple):
    id: str
    first_name: Optional[str]
    last_name: Optional[str]
    username: Optional[str]


def _parse_user_data(message):
    return UserData(
        id=str(message['id']), first_name=message['first_name'], last_name=message['last_name'],
        username=message['username']
    )


def _parse_category(message):
    regexp_ = re.match(r'^([a-z]+: )([a-z]+, ){1,7}([a-z]+)', str(message))
    if not regexp_ or not regexp_.group(0):
        raise AddCategoryError(str(message))
    reg_str = str(regexp_.group(0)).split(': ')
    return CategoryMessage(name_=reg_str[0], category_text=reg_str[1])


def _parse_message(message):
    regexp = re.match(r"([\d ]+) (.*)", message)
    if not regexp or not regexp.group(0) \
            or not regexp.group(1) or not regexp.group(2):
        raise AddCategoryError(str(message))
    return Message(amount=int(regexp.group(1).replace(" ", "")), category_text=regexp.group(2).strip().lower())


def _parse_budget_message(message):
    regexp_ = re.match(r'(daily) ([1-9]([0-9]){0,10}) (month) ([1-9]([0-9]){0,10})',
                       str(message).lower().replace("  ", " "))
```

```
    # if not (regexp_.group(1) and regexp_.group(2)) or (regexp_.group(4) and regexp_.group(5)):
    if not regexp_.group(0):
        raise ChangeBudgetError(str(message))
    return BudgetMessage(daily_amount=int(regexp_.group(2)), month_amount=int(regexp_.group(5)))


def check_user_exists(user_id):
    with connection.cursor() as cursor:
        cursor.execute(f"SELECT id FROM users WHERE id = {user_id}")
    return bool(cursor.rowcount)


def add_user(message):
    user_info = _parse_user_data(message)
    db.insert(
        "users",
        {
            "id": user_info.id,
            "first_name": user_info.first_name,
            "last_name": user_info.last_name,
            "username": user_info.username
        }
    )


def add_expense(message, user_id):
    parsed_message = _parse_message(message)
    category = Categories().get_category(
        parsed_message.category_text)
    db.insert(
        "expenses",
        {
            "amount": parsed_message.amount,
            "date_time": _get_now_formatted(),
            "category": category.codename,
            "user_id": str(user_id)
        }
    )
    return IncomeExpense(id=None, amount=parsed_message.amount, category_name=category.name)


def edit_budget(message, user_id):
    parsed_message = _parse_budget_message(message)
    db.update_(
        "budget", {
            'daily_limit': parsed_message.daily_amount,
            'month_limit': parsed_message.month_amount,
            'user_id': str(user_id)
        },
        f"code_name = 'general' AND user_id = {user_id}"
    )


def add_incomes(message, user_id):
    parsed_message = _parse_message(message)
    category = Categories().get_category(
        parsed_message.category_text)
```

```
    db.insert(
        "incomes",
        {
            "amount": parsed_message.amount,
            "date_time": _get_now_formatted(),
            "category": category.codename,
            "user_id": str(user_id)
        }
    )
    return IncomeExpense(id=None, amount=parsed_message.amount, category_name=category.name)


def create_category_finance(message, user_id):
    parsed_data = _parse_category(message)
    db.insert(
        "category", {
            'code_name': parsed_data.name_ + '_',
            'category_name': parsed_data.name_,
            'aliases_': parsed_data.category_text,
            'user_id': str(user_id)
        }
    )
    return CreateCategory(code_name_=parsed_data[0] + '_', category_name_=parsed_data[0],
aliases_text_=parsed_data[1])


def see_categories(user_id):
    with connection.cursor() as cursor:
        cursor.execute(f'SELECT category_name, aliases_ FROM category WHERE user_id = {str(user_id)}')
        rows = cursor.fetchall()
    return [CategoryMessage(name_=row[0], category_text=row[1]) for row in rows]


def delete_expense(row_id, user_id):
    with connection.cursor() as cursor:
        row_id = int(row_id)
        cursor.execute(f"DELETE FROM expenses WHERE expense_id = {row_id} AND user_id = {user_id}")
        connection.commit()


def delete_income(row_id, user_id):
    with connection.cursor() as cursor:
        row_id = int(row_id)
        cursor.execute(f"DELETE FROM incomes WHERE income_id = {row_id} AND user_id = {user_id}")
        connection.commit()


def _get_now_formatted():
    return _get_now_datetime().strftime('%Y-%m-%d %H:%M:%S')


def _get_now_datetime():
    return datetime.now()


def set_default_budget(user_id):
    with connection.cursor() as cursor:
```

```
        cursor.execute(f'INSERT INTO budget (daily_limit, month_limit, user_id) VALUES (0, 0, {str(user_id)})')
        connection.commit()


def get_budget_month_limit(user_id):
    return db.fetchone_for_budget('budget', f"month_limit".split(), f'user_id = {user_id}')


def get_budget_daily_limit(user_id):
    return db.fetchone_for_budget('budget', f"daily_limit".split(), f'user_id = {user_id}')


def today_expenses(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.expense_id, a.amount, b.category_name '
            f'FROM expenses a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (CAST(date_time AS DATE) = CAST(CURDATE() AS DATE) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]


def this_week_expenses(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.expense_id, a.amount, b.category_name '
            f'FROM expenses a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (YEARWEEK(CURDATE()) = YEARWEEK(date_time) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]


def this_month_expenses(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.expense_id, a.amount, b.category_name '
            f'FROM expenses a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (MONTH(date_time) = MONTH(CURDATE()) AND '
            f'YEAR(date_time) = YEAR(CURDATE()) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]


def today_incomes(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.income_id, a.amount, b.category_name '
            f'FROM incomes a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (CAST(date_time AS DATE) = CAST(CURDATE() AS DATE) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]
```

```
def this_week_incomes(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.income_id, a.amount, b.category_name '
            f'FROM incomes a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (YEARWEEK(CURDATE()) = YEARWEEK(date_time) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]


def this_month_incomes(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT a.income_id, a.amount, b.category_name '
            f'FROM incomes a LEFT JOIN category b ON b.code_name=a.category AND a.user_id = b.user_id '
            f'WHERE (MONTH(date_time)=MONTH(CURDATE()) AND '
            f'YEAR(date_time)=YEAR(CURDATE()) AND a.user_id = {str(user_id)})'
        )
        rows = cursor.fetchall()
    return [IncomeExpense(id=row[0], amount=row[1], category_name=row[2]) for row in rows]
```

Модуль *Statistic.py*

```
import os

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.dates import DayLocator
from matplotlib.ticker import AutoMinorLocator
from random import shuffle
from config import connection


def _get_formatted(date):
    return date.strftime('%Y-%m-%d')


def merging_list(arr1, arr2):
    return [i for i in arr2 if i not in arr1] + arr1


def delete_stats_image(name_):
    path_ = os.getcwd().replace(f'\\', '/')
    os.remove(f"{path_}/{name_}.png")


def get_week_expenses_for_stats(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT SUM(amount), CAST(date_time AS DATE) AS Date_, date_time FROM expenses '
            f'WHERE yearweek(date_time, 1) = yearweek(CURDATE(), 1) AND user_id = {user_id} '
            f'GROUP BY CAST(date_time AS DATE) '
            f'ORDER BY date_time ASC'
        )
        rows = cursor.fetchall()
        return [i[0] for i in rows], [_get_formatted(j[1]) for j in rows]


def get_week_incomes_for_stats(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT SUM(amount), CAST(date_time AS DATE) AS Date_, date_time FROM incomes '
            f'WHERE yearweek(date_time, 1)  = yearweek(CURDATE(), 1)  AND user_id = {user_id} '
            f'GROUP BY CAST(date_time AS DATE) '
            f'ORDER BY date_time ASC'
        )
        rows = cursor.fetchall()
        return [i[0] for i in rows], [_get_formatted(j[1]) for j in rows]


def get_month_expenses_for_stats(user_id):
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT SUM(amount), CAST(date_time AS DATE) AS Date_, date_time FROM expenses '
            f'WHERE MONTH(date_time) = MONTH(CURDATE()) '
            f'AND YEAR(date_time) = YEAR(CURDATE()) AND user_id = {user_id} '
            f'GROUP BY CAST(date_time AS DATE) '
            f'ORDER BY date_time ASC'
```

```
        )
        rows = cursor.fetchall()
        return [i[0] for i in rows], [_get_formatted(j[1]) for j in rows]


def get_month_incomes_for_stats(user_id):
    print('get_month_incomes_for_stats')
    with connection.cursor() as cursor:
        cursor.execute(
            f'SELECT SUM(amount), CAST(date_time AS DATE) AS Date_, date_time FROM incomes '
            f'WHERE MONTH(date_time) = MONTH(CURDATE()) '
            f'AND YEAR(date_time) = YEAR(CURDATE()) AND user_id = {user_id} '
            f'GROUP BY CAST(date_time AS DATE) '
            f'ORDER BY date_time ASC'
        )
        rows = cursor.fetchall()
        print(rows)
        # rows.sort(key=lambda x: x.count, reverse=True)
        return [i[0] for i in rows], [_get_formatted(j[1]) for j in rows]


def week_data_for_stats(user_id):
    expenses_ = get_week_expenses_for_stats(user_id)
    incomes_ = get_week_incomes_for_stats(user_id)
    return expenses_[0], expenses_[1], incomes_[0], incomes_[1]


def month_data_for_stats(user_id):
    expenses_ = get_month_expenses_for_stats(user_id)
    incomes_ = get_month_incomes_for_stats(user_id)
    return expenses_[0], expenses_[1], incomes_[0], incomes_[1]


def stats_for_current_week(user_id):
    current_week_data_ = week_data_for_stats(user_id)
    return create_diagram_for_stats(current_week_data_[0], current_week_data_[2],
                                    current_week_data_[1], current_week_data_[3], 'Week Statistic')


def stats_for_current_month(user_id):
    current_month_data_ = month_data_for_stats(user_id)
    return create_diagram_for_stats(current_month_data_[0], current_month_data_[2],
                                    current_month_data_[1], current_month_data_[3], 'Month Statistic')


def create_diagram_for_stats(values_expenses, values_incomes, dates_expenses, dates_incomes, type_):
    print('create_diagram_for_stats')
    x = np.arange(len(dates_expenses))
    y = np.array(values_expenses)
    y2 = np.array(values_incomes)
    x2 = np.arange(len(dates_incomes))
    fig, ax = plt.subplots(figsize=(10, 6))
    plt.title(type_)
    plt.xlabel("Date")
    plt.ylabel("Amount")
    ax.xaxis.set_major_locator(DayLocator())
    plt.plot(x, y, 'o-', label='expenses')
```

```
    plt.plot(x2, y2, 'o-', label='incomes')
    plt.legend(loc="upper left")
    print(dates_expenses)
    print(merging_list(dates_expenses, dates_incomes))
    ax.set_xticklabels(merging_list(dates_expenses, dates_incomes), fontsize=10)
    ax.grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
    ax.xaxis.set_minor_locator(AutoMinorLocator())
    fig.autofmt_xdate()
    alphabet_ = list('abcdefghhijklmnopqrsruwxyz')
    shuffle(alphabet_)
    name_ = ''.join(i for i in alphabet_[:10])
    plt.savefig(name_)
    return name_


def calculating_results(values_expenses, values_incomes):
    total_expenses = sum(values_expenses)
    total_incomes = sum(values_incomes)
    pure_profit = total_incomes - total_expenses
    return total_expenses, total_incomes, pure_profit


def resulting_for_the_current_week(user_id):
    current_week_data_ = week_data_for_stats(user_id)
    return calculating_results(current_week_data_[0], current_week_data_[2])


def resulting_for_the_current_month(user_id):
    current_month_data_ = month_data_for_stats(user_id)
    return calculating_results(current_month_data_[0], current_month_data_[2])
```

Модуль *exceptions.py*

```
class AddCategoryError(Exception):
    def __init__(self, category, error_message='Incorrect category input'):
        self._category = category
        self._error_message = error_message
        super().__init__(self._error_message)

    def __str__(self):
        return f'{self._error_message}:\n{self._category} does not much the format:\n' \
               f'`products: products, food, eat`'


class AddExpenseError(Exception):
    def __init__(self, expense, error_message='Incorrect expense input'):
        self._expense = expense
        self._error_message = error_message
        super().__init__(self._error_message)

    def __str__(self):
        return f'{self._error_message}:\n{self._expense} does not much the format: `125 taxi`'


class AddIncomeError(Exception):
    def __init__(self, income, error_message='Incorrect income input'):
        self._income = income
        self._error_message = error_message
        super().__init__(self._error_message)

    def __str__(self):
        return f'{self._error_message}:\n{self._income} does not much the format: `125 job`'


class ChangeBudgetError(Exception):
    def __init__(self, budget, error_message='Incorrect budget input'):
        self._budget = budget
        self._error_message = error_message
        super().__init__(self._error_message)

    def __str__(self):
        return f'{self._error_message}:\n{self._budget} does not much the format:\n' \
               f'`daily 150 month 8500`\n' \
               f'daily 360\n' \
               f'month 4520'


class DeleteError(Exception):
    def __init__(self, delete, error_message='Deleting error'):
        self._delete = delete
        self._error_message = error_message
        super().__init__(self._error_message)

    def __str__(self):
        return f'{self._error_message}:\n{self._delete}\n'
```

Модуль *databases.py*

```
from config import connection


def insert(table, column_values):
    columns = ', '.join(column_values.keys())
    values = [tuple(i for i in column_values.values())]
    placeholders = '%s' + ', %s' * (len(column_values) - 1)
    with connection.cursor() as cursor:
        cursor.executemany(f'INSERT INTO {table} ({columns}) VALUES ({placeholders})', values)
        connection.commit()


def fetchall_(table, columns):
    columns_joined = ", ".join(columns)
    with connection.cursor() as cursor:
        cursor.execute(f"SELECT {columns_joined} FROM {table}")
        rows = cursor.fetchall()
    result_ = []
    for row in rows:
        dict_row = {}
        for index, column in enumerate(columns):
            dict_row[column] = row[index]
        result_.append(dict_row)
    return result_


def fetchone_for_budget(table, columns, condition):
    columns_joined = ", ".join(columns)
    with connection.cursor() as cursor:
        cursor.execute(f"SELECT {columns_joined} FROM {table} WHERE {condition}")
        z = cursor.fetchone()
        return z[0]


def delete(table, row, value):
    with connection.cursor() as cursor:
        cursor.execute(f"DELETE FROM {table} WHERE {row}='{value}'")
        connection.commit()


def update_(table, data, condition):
    columns = ', '.join([f'{i} = %s' for i in data])
    values = [[i] for i in data.values()]
    with connection.cursor() as cursor:
        cursor.execute(f'UPDATE {table} SET {columns} WHERE {condition}', values)
        connection.commit()
```