МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інформаційна технологія проектування електронної торгової платформи»

Завідувач
випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Олексієнко Г.А.**

Студента групи ІК.м–01 **Свістельнік А.О.**

СУМИ 2021

Сумський державний університет
(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«122 - Комп'ютерні науки»*

Затверджую:
зав.кафедрою _____
_____
"_____" _____ 20___р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Свістельніку Артему Олександровичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Інформаційна технологія проектування електронної торгової платформи*

затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи)_____

3. Вхідні данні до проекту (роботи) _____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Огляд технологій, що застосовуються для продажу товарів через Інтернет; 2) Постановка завдання й формування завдань дослідження; 3) Огляд технологій, що використовуються під час розробки Api з використання мови програмування Python; 4) Моделювання системи торгової платформи; 5) Розробка додатку; 6) Аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Огляд технологій, що застосовуються для продажу товарів через Інтернет		
2.	Постановка задачі та формування завдань дослідження.		
3.	Опис архітектури додатку		
4.	Розробка додатку з використанням Python та FastApi		
5.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 59 стор., 17 рис., 4 літературні джерела.

Об’єкт дослідження — Інформаційна технологія проектування електронної торгової платформи.

Мета роботи — розробка додатку на python та з використання фреймворку FastApi для продажу та купівлі товарів на базі електронної торгової платформи.

Результати — проведений аналіз літератури, методів та інструментів, які дозволяють проводити замовлення та продаж різних товарів а також , обмін командами і даними між додатками, що написані на різних мовах програмування під різні платформи, вивчені особливості їх застосування у backend, для різних типів користувачів. Після ознайомлення з існуючими рішеннями було розроблено алгоритм програми та її реалізація у вигляді API. Дана розробка дозволяє виконувати купівлю та продаж товарів будь-яким користувачам. Додаток був реалізований за допомогою мови програмування Python та фреймворку FastApi.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ПРОЕКТУВАННЯ ЕЛЕКТРОННОЇ ТОРГОВОЇ ПЛАТФОРМИ, INFORMATION TECHNOLOGY OF DESIGNING AN ELECTRONIC TRADING PLATFORM, PYTHON, FASTAPI, POSTGRESQL

ЗМІСТ

ВСТУП 6

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 7

1.1 Дослідження актуальності проблеми 7

1.2 Аналіз аналогів 8

1.3 Постановка задачі 11

2 ОГЛЯД АРХІТЕКТУРИ ТА ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ 12

2.1 Огляд основних ідей REST архітектури 12

2.2 Вибір мови програмування 13

2.3 Вибір фреймворку 13

2.4 Вибір СУБД 14

2.5 Вибір середовища розробки 14

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ 16

3.1 Інформаційна модель 16

3.2 Програмна реалізація 17

ВИСНОВКИ 57

СПИСОК ЛІТЕРАТУРИ 58

ВСТУП

З кожним днем різноманіття товарів і послуг все збільшується, саме тому, наразі акуальною є проблема їх реалізації, а особливо реалізації через інтернет. Перший карантин 2020-го року досить сильно вплинув на дрібні торгівельні бізнеси, деяким навіть довелось завершити свою діяльність. Через малий поток відвідучів все більшою популярністю стали користуватися інтернет-магазини та особливо, торгові майданчики, які дали змогу продажі будь-чого без прямого контаку продавця і покупця.

Саме тому дана робота присвячена розробці API торгівельного майданчика, який дозволить за домогою веб додатку чи додатку на будь-якій мобільній платформі упростити та пришвидшити процес купівлі/продажу. Система буде вирізнятися своєю простотою і доступністю.

.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Дослідження актуальності проблеми

На сьогодні більшість людей обирають замовляти будь-що через інтернет замість того, для економії часу. Адже час - це найважливіший ресурс сучасної людини.

На рисунках нижче зображена статистика кількості відвідувачів відомих інтернет-магазинів, маркетплейсів та дошок оголошень до початку карантину.

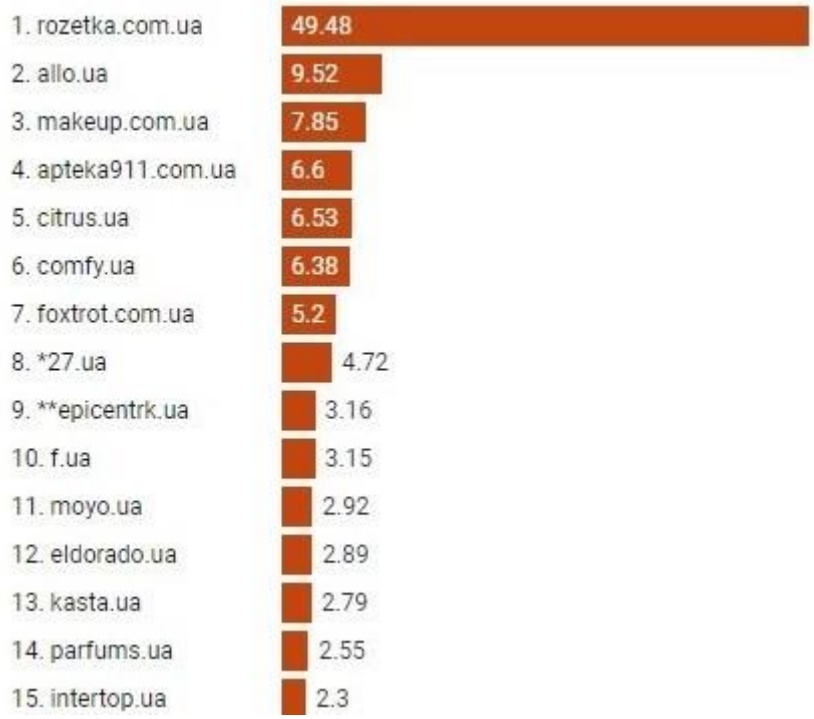

Рисунок 1.1 — Кількість відвідувачів за грудень 2019р (у млн.)

Рисунок 1.2 — Кількість відвідувачів за вересень 2020р (у млн.)

З наведених скріншотів можна дійти висновку, що інтернет торгівля не лише не збавляє своїх обертів, а й навпаки, поступово популяризується. Тому тематика створення власної торгової платформи будь-якого формату є актуальною.

1.2 Аналіз аналогів

Для початку проглянемо статистику відвідувань всіх топ-сайтів в Україні(рисунок 1.3). В даному рейтингі в десятку найпопулярніші входять всім відомі представники надання послуг з онлайн-торгівлі, які продають товари у тому чи іншому вигляді.

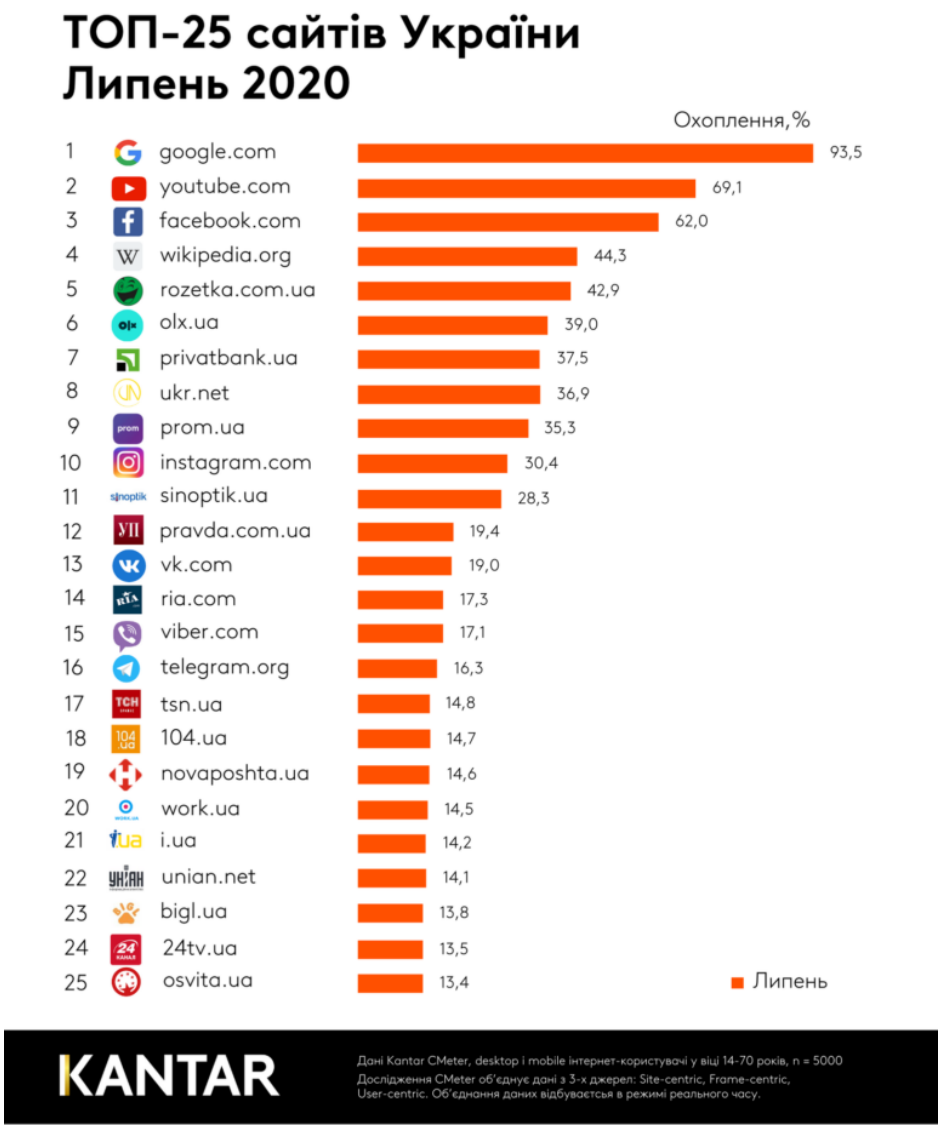

Рисунок 1.3 — Кількість відвідувачів на українських сайтах.

Перший представник - маркетплейс Rozetka(рисунок 1.4), який займає 4 місце. Цей колишній інтернет магазин - лише додав у кількості користувачів створивши можливість продавати товари від власного імені різним продавцям під доменом цього відомого веб-ресурсу. Цей сайт високонавантажений, і його швидкість роботи залишає бажати кращого.

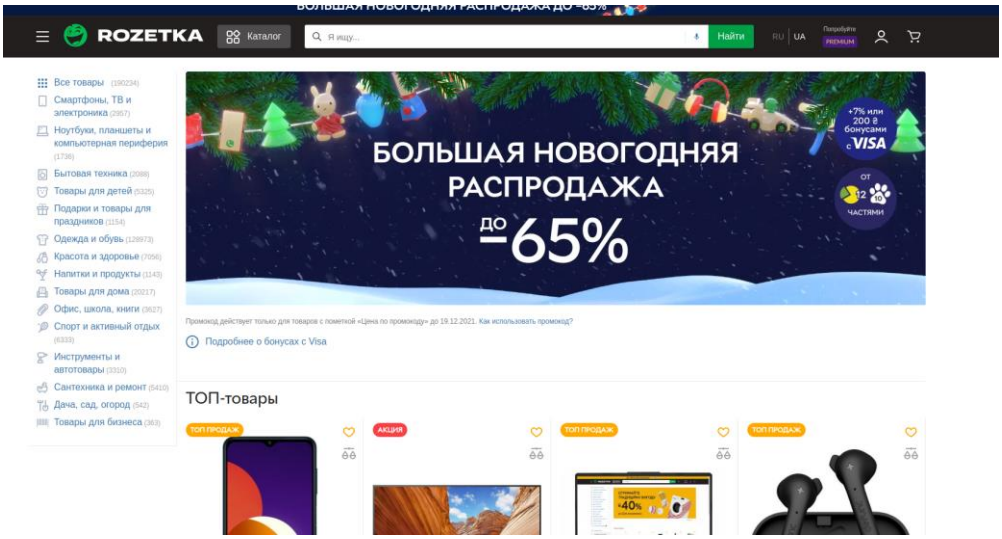

Рисунок 1.4 — rozetka.com.ua

Другий представник - дошка оголошень OLX (рисунок 1.5). Він популярний тим, що дозволяє досить просто створити власне оголошення і розмістити його, а також придбати що-небудь. З мінусів даного ресурсу можна зазначити, невеликий функціонал для продавців великої кількості різних за категоріями товарів.

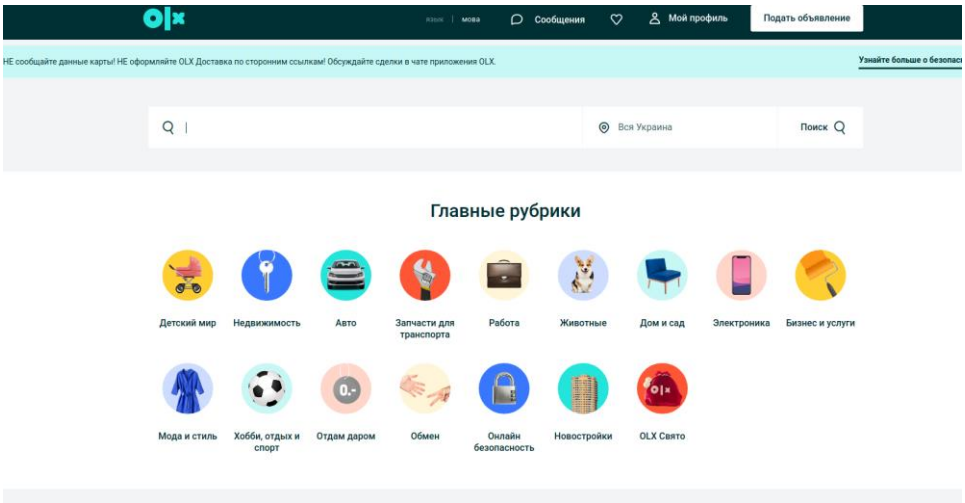

Рисунок 1.5 — olx.ua

Третій, останній представник лідерів - це Prom.ua(Рисунок 1.6). Цей ресурс має достатній функціонал для продавця, навіть можливість створення свого

сайту на піддомені prom-у з невеликими змінами у кольоровій схемі. Але - так як продавці ніяк не перевіряються то сайт переповнений немалою кількістю недобросовісних представників.

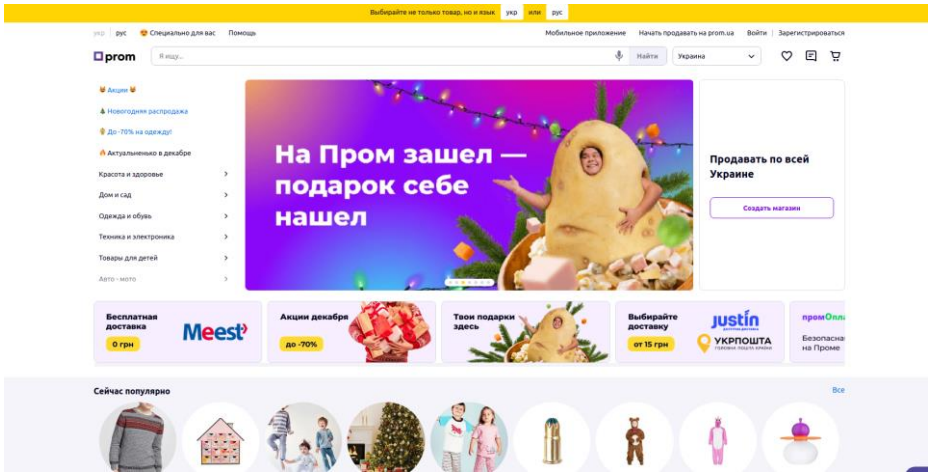

Рисунок 1.6 — prom.ua

1.3 Постановка задачі

Метою роботи є розробка програмної системи, котра буде слугувати для:

1. Продавці, як сервіс реалізації товарів чи послуг;
2. Клієнтів, як надійний, швидкий сервіс для пошуку необхіндних товарів чи послуг.

Для досягнення поставленої мети сформульовані наступні задачі:

1) вибрати відповідні методи та технології для:
   1.1) авторизації та автентифікації користувачів;
   1.2) збереження всіїх необхіної інформації;
   1.3) створення API на базі фреймворку FastApi;
2) провести аналіз конкурентів;
3) спроектувати передачу даних між компонентами системи;
4) спроектувати програмну систему та її частини;
5) розробка та тестування додатку.

2 ОГЛЯД АРХІТЕКТУРИ ТА ВИБІР МЕТОДІВ РЕАЛІЗАЦІЇ

2.1 Огляд основних ідей REST архітектури

Абривіатура REST розшифровується як REpresentational State Transfer, а API -Application Program Interface. REST - це стиль побуди програмного забезпечення, який визначає набір правил, які використовуються для створення веб-сервісів. Ті веб-служби, які наслідують правила REST - називають RESTful. Він дозволяє запитувати у систем доступ до веб-ресурсів і використовувати їх за домогою певного набору правил. Вся взаємодія в системах на основі REST відбувається через HTTP - протоколо передачі гіпертексту в Інтернеті.

Restful система складається з:

1) Клієнту, який робить запити на ресурс
2) Серверу, який має ресурси і відповідає на запити

Архітектурні обмеження які має RestfulAPI :

1) Єдиний інтерфейс : це ключове обмеження яке розділяє на rest api і не rest api. Мається на увазі те, що має бути лише один спосіб взаємодії з даним сервером незалежно від типу додатку та пристрою.
2) Без збереження стану : це означає, що стан, необхідний для обробки запиту - буде міститися в самому запиті і не буди необхідності серверу зберігати дані, пов’язані з сенасом.
3) Кеш : кожна відповідь має вказати, чи буде відповідь кешована чи ні і на який час.
4) Зв’язок між клієнтом і сервером : Додаток має мати архітектуру клієнт-сервер. Клієнт - лише надсилає запити, і не зберігає даних. Сервер - зберігає дані і відповідає на запити.
5) Багаторівнена система : Архітектура програми має складати з кількох рівнів. Кожен рівень нічого не знає про інші, окрім безпосереднього рівня.

6) Код по запиту : Додаткова функція. У відповідності з цим, сервери можуть надавати виконуємий код клієнту.

2.2 Вибір мови програмування

В 2020 році для створення сайту можна використати велику кількість мов програмування. Чому саме для створення торгового майданчика був обраний Python? Python - це сучасна, універсальна, високорівнева мова програмування. До переваг Python можна віднести його ефективність і добре сконструйований легко читаємий код. У python простий для читання і написання синтаксис, що легко дозволяє вичити його навіть новачку у програмуванні. Python був написаний на мові програмування C (сі), що дозволяє створювати доповнення і бібліотеки до нього на самому C . Такий спосіб доповнення використовується у випадках коли необхідна критична швидкодія. Python підтримує багато стилів розробки, до яких входять популярні: ООП (об'єктно-орієнтоване програмування) і функціональне програмування.

Для чого використовують python? Python підходить для розробки будь-яких проектів. Також він підтримується майже усіма платформами. З операційнми системами на базі лінукс python буде встановлений по замовчуванням . Найчастіше python використовують у сферах роботи з Big Data, нейромережами, штучним інтелектом, машинним навчанням і у web-розрозбці . Для веб-розробки часто використовуються такі фреймворки як Flask , Pylons і найпопулярніший серед всіх - Django. Також python часто використовуються для розробки різноманітних парсерів, які збирають різноманітну інформацію з мережі інтернет.

2.3 Вибір фреймворку

Для розробки API було вирішено використовувати FastApi. Цей врйемворк створений на мові програмування Python спеціалізовано для створення REST проектів. Вибір пав на цього тому, що FastApi не має зайвих налаштувань і бібліотек. Це сповільнює розгортання проекту на перших етапах, але дає повну свободу до підходу реалізації проекту, навідміну від Django. Також цей фреймворк асинхронний, завдяки чому і забезпечується його швидкодія.

2.4 Вибір СУБД

Для побудови і підтримки бази даних було використано PostgreSQL СУБД. PostgreSQL — об'єктно-реляційна система керування базами даних (СКБД). Є альтернативою як комерційним СКБД (Oracle Database, Microsoft SQL Server, IBM DB2 та інші), так і СКБД з відкритим кодом (MySQL, Firebird, SQLite).

Порівняно з іншими проектами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

PPostgreSQL це найбільш продвинута база даних, яка орієнтована на відповідність стандартам і масштабованість. Ця база створена на технології Posgres і чудово справляється з обробкою декількох запитів одночасно , що добре підходить для такого веб-сервісу як торговий майданчик.

2.5 Вибір середовища розробки

Інтегроване середовище розробки (IDE) (англ. Integrated Development Environment) - програмний комплекс, який призначений для ефективної розробки різної складності систем. IDE складається з:

1) Редактора текста і ресурсів. Текстовий редактор призначений для зручного і швидкого створення і редагування тексту (коду). Часто містить автодоповнення і підказки.
2) Компілятора. Компілятор - це спеціалізоване пз (програмне забезпечення) яке перетворює програмний текст, який написаний на

одній з мов програмування в набір машинних команд, які будуть зрозумілі цій машині.

3) Відладчика. Відладчик - це спеціалізована програма вбудована в IDE для пошука помилок в коді програми, чи інших системних помилок.
4) Засобів управління проектом. Цей пункт зображує аналог вбудованого файловго менеджера.
5) Стандартних заготовок, спрощуючик розробку стандартних задач.

Як IDE для створення торгового майданчик було обрано PyCharm IDE. Це IDE від компанії Jetbrains, яке підходить для роботи на мові програмування Python, а також для використання популярних фреймворків таких як Django. PyCharm містить в собі всі пункти вказані вище які допомагають у розробці будь-яких програм чи сервісів

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Інформаційна модель

Першим кроком у створенні проекту - було прийнято рішення створити систему з наступною структурою: Python, FastApi, PostrgeSQL

Наступним кроком було побудовано ERD діаграму (Рисунок 3.1), котра містить в себе всі основні сутності сервісу, а також відношення між ними на рівні бази даних.

Основні сутності:

1) Users (Користувачі)
2) Shops(Магазини)
3) Categories (Категорії товарів)
4) Products (Товари)
5) Images (Галерея зображень для моделі Products)
6) Orders (Замовлення)
7) Order_Items (елемент замовлення)

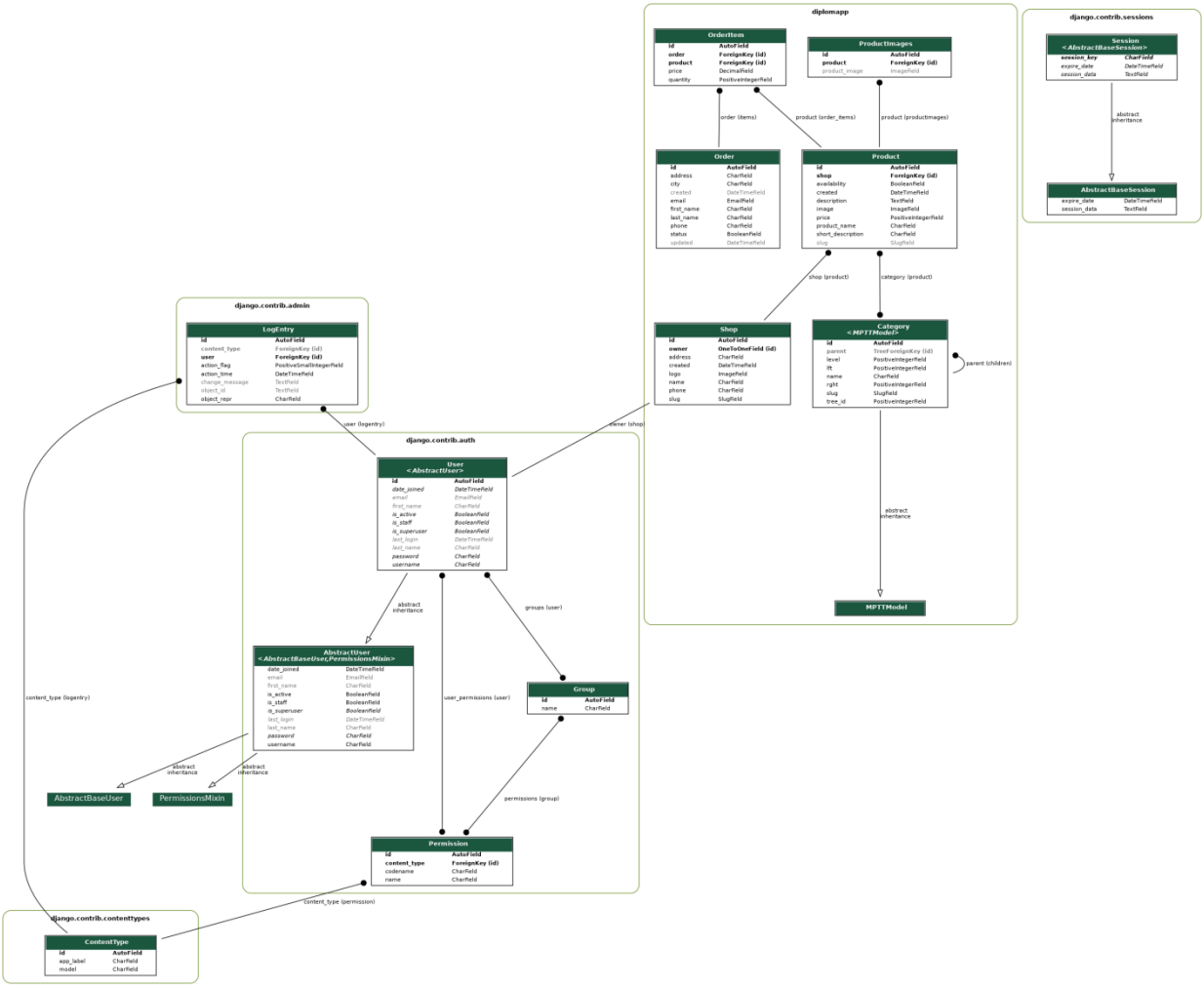

Рисунок 3.1 – ERD діаграма

3.2 Програмна реалізація

Розробка бекенду почалася з створення схем які описують взаємодію між бекендом та сервісами проекту.

За функціонал API торгової платформи, в головній мірі, відповідають 2 види файлів:

1) Файли з директорії repositories - це файли, які містять підключення до бази данних і реалізують функціонал CRUD (Create, Read,Update,Delete), а саме створення об’єктів, їх зміна, перегляд та видалення.
2) Файли з директорії endpoints – це файли, які за заданими роутами(url) – отримують параметри запиту та дані які були передані і передають на

відповідну функцію в директорії repositories.

Також в проекті знаходиться папка Models, яка містить інформацію(відповідні класи) з типізацією данних, які необхідно віддавати чи отримувати.

```python
import datetime
from typing import Optional
from pydantic import BaseModel, EmailStr, validator, constr


class BaseUser(BaseModel):
    name: str
    email: EmailStr
    is_company: Optional[bool] = False

class User(BaseUser):
    id: Optional[str] = None
    hashed_password: str
    created_at: datetime.datetime
    updated_at: datetime.datetime

class UserOutput(BaseUser):
    id: Optional[str] = None
    created_at: datetime.datetime
    updated_at: datetime.datetime

class UserIn(BaseUser):
    password: constr(min_length=8)
    password2: str

    @validator("password2")
    def password_match(cls, v, values, **kwargs):
        if 'password' in values and v != values["password"]:
            raise ValueError("passwords don't match")
        return v
```

Рисунок 3.2 – Схеми для моделі User

```python
class Token(BaseModel):
    access_token: str
    token_type: str


class Login(BaseModel):
    email: EmailStr
    password: str
```

Рисунок 3.3 – Схеми для моделі Token

В даній схемі описані основні поля і їх типи для моделі користувача, а також схема токена, який необхідний для аутентифікації користуача. Доступні методи можна переглянути на рисунку 3.4, на якому також відображено дані, необхідні для отримання токену доступу до решти функцій.

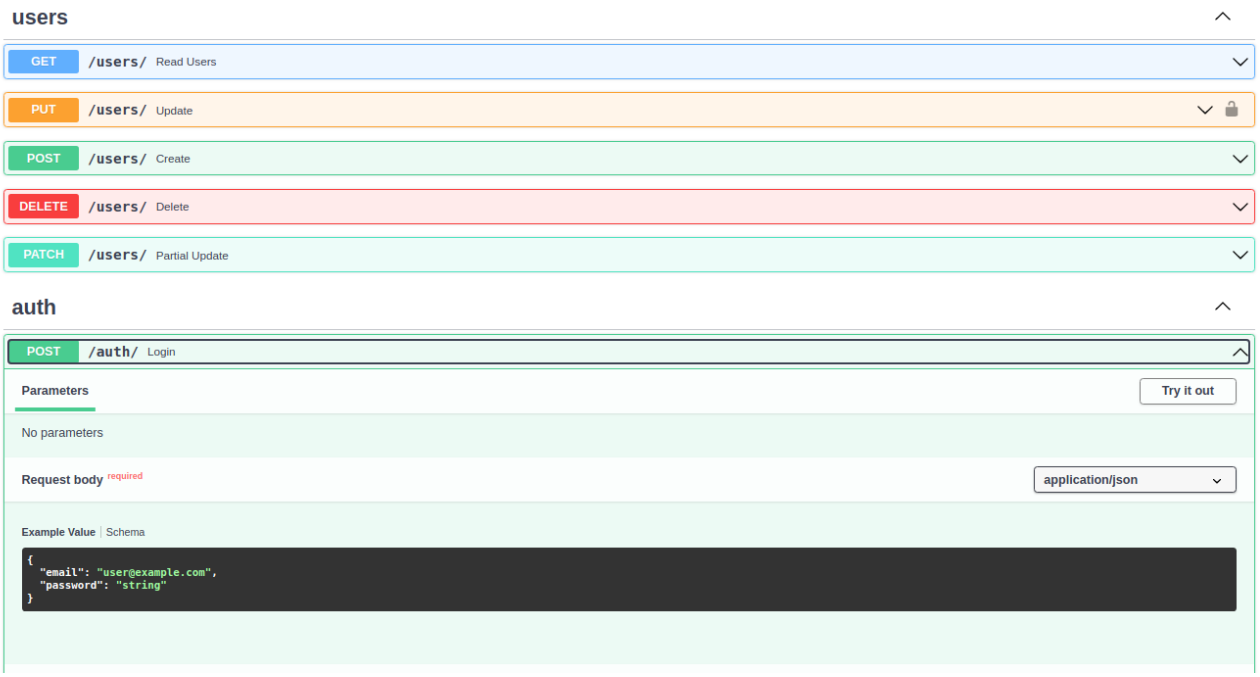

Рисунок 3.4 – Методи доступні для моделі User

Щоб дані можна було завантажити та отримати за даною схемою, необхідно було написати самі функції роботи за даними, а також реалізувати endpoint-и

Методи роботи з моделлю User:

```
import datetime
from typing import List, Optional
```

```
from db.users import users
from models.user import User, UserIn,UserOutput,BaseUser
from core.security import hash_password
from .base import BaseRepository

class UserRepository(BaseRepository):
    async def get_all(self, limit: int = 100, skip: int = 100)
-> List[User]:
        query = users.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def get_by_id(self, id: int) -> Optional[User]:
        query = users.select().where(users.c.id == id)
        user = await self.database.fetch_one(query=query)
        if user is None:
            return None
        return User.parse_obj(user)

    async def create(self, u: UserIn) -> User:
        user = User(
            name=u.name,
            email=u.email,
            hashed_password=hash_password(u.password),
            is_company=u.is_company,
            created_at=datetime.datetime.utcnow(),
            updated_at=datetime.datetime.utcnow(),
        )
```

```
        values = {**user.dict()}
        values.pop("id", None)
        query = users.insert().values(**values)
        user.id = await self.database.execute(query)
        return user

    async def update(self, id: int, u: UserIn) -> User:
        user = User(
            id=id,
            name=u.name,
            email=u.email,
            hashed_password=hash_password(u.password2),
            is_company=u.is_company,
            created_at=datetime.datetime.utcnow(),
            updated_at=datetime.datetime.utcnow(),
        )
        values = {**user.dict()}
        values.pop("created_at", None)
        values.pop("id", None)
        query = users.update().where(users.c.id ==
id).values(**values)
        await self.database.execute(query)
        return user

    async def partial_update(self, id: int, u: BaseUser) ->
Optional[User]:
        values={**u.dict()}
        values = {k:v for k, v in values.items() if v is not
```

```
None}
        query = users.select().where(users.c.id == id)
        user = await self.database.fetch_one(query=query)
        query = users.update().where(users.c.id ==
id).values(values)
        await self.database.execute(query)
        return user

    async def delete(self,id:int)-> Optional[User]:
        query = users.delete().where(users.c.id == id)
        user = await self.database.fetch_one(query=query)
        if user is None:
            return None
        return User.parse_obj(user)

    async def get_by_email(self, email: str) ->
Optional[User]:
        query = users.select().where(users.c.email == email)
        user = await self.database.fetch_one(query=query)
        if user is None:
            return None
        return User.parse_obj(user)
```

Endpoints.users.(Функції API для моделі user):

```
from typing import List

from fastapi import APIRouter,
```

```
Depends,HTTPException,status

    from models.user import User, UserIn, UserOutput,
BaseUser
    from repositories.users import UserRepository
    from .depends import get_user_repository,get_current_user

    router = APIRouter()


    @router.get("/", response_model=List[UserOutput])
    async def read_users(
            users: UserRepository =
Depends(get_user_repository),
            limit: int = 100,
            skip: int = 100):
        return await users.get_all(limit=limit, skip=skip)


    @router.post("/", response_model=UserOutput)
    async def create(user: UserIn,
                     users: UserRepository =
Depends(get_user_repository)):
        return await users.create(u=user)


    @router.put("/", response_model=UserOutput)
    async def update(id: int, user: UserIn,
```

```
                    users: UserRepository =
Depends(get_user_repository),

current_user:User=Depends(get_current_user)):
        old_user=await users.get_by_id(id=id)
        if old_user is None or
old_user.email!=current_user.email:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Not found user")
        return await users.update(id=id, u=user)


    @router.patch("/", response_model=UserOutput)
    async def partial_update(id: int, user: BaseUser,
                             users: UserRepository =
Depends(get_user_repository)):
        return await users.partial_update(id=id, u=user)


    @router.delete("/", response_model=User)
    async def delete(id: int,
                     users: UserRepository =
Depends(get_user_repository)):
       return await users.delete(id=id)
```

Endpoints.auth.(Функція аутентифікації):

```
    @router.post("/", response_model=Token)
    async def login(login: Login, users: UserRepository =
Depends(get_user_repository)):
        user = await users.get_by_email(login.email)
        if user is None or not
verify_password(login.password, user.hashed_password):
            raise
HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Incorrect username or password")
        return Token(
            access_token=create_access_token({"sub":
user.email}),
            token_type="Bearer"
        )
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

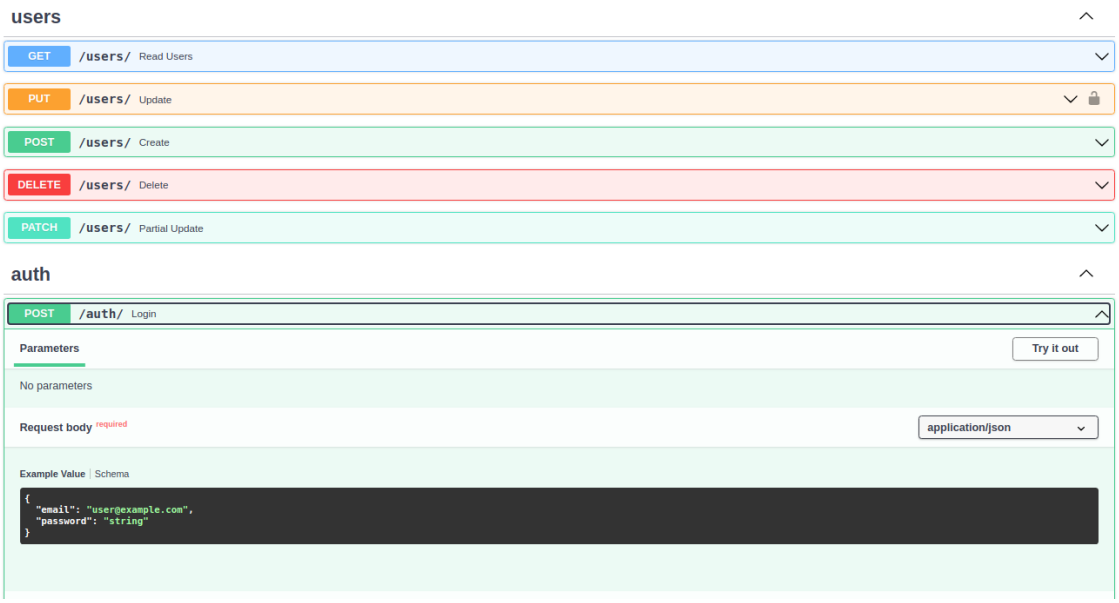

Рисунок 3.5 – FastApi docs для моделей User та Auth

Методи роботи з моделлю Shops:

```
   class ShopRepository(BaseRepository):
    async def create(self, user_id: int, sh: BaseShop) ->
Shop:
        shop = Shop(
            id=0,
            owner_id=user_id,
            shop_name=sh.shop_name,
            phone=sh.phone,
            address=sh.address
        )
        values = {**shop.dict()}
        values.pop("id", None)
        query = shops.insert().values(**values)
        shop.id = await self.database.execute(query=query)
        user_query = users.select().where(users.c.id ==
user_id)
        await self.database.fetch_one(query=user_query)
        user_update_query = users.update().where(users.c.id ==
user_id).values(
            {"is_company": True})
        await self.database.execute(user_update_query)
        return shop

    async def update(self, shop_id: int, user_id: int, sh:
BaseShop) -> Shop:
        shop = Shop(
```

```
            id=shop_id,
            owner_id=user_id,
            shop_name=sh.shop_name,
            phone=sh.phone,
            address=sh.address
        )
        values = {**shop.dict()}
        values.pop("id", None)
        query = shops.update().where(shops.c.id ==
shop_id).values(**values)
        await self.database.execute(query=query)
        return shop

    async def partial_update(self, shop_id: int, user_id: int,
sh: ShopPartialUpdateIn)->Optional[Shop]:
        values = {**sh.dict()}
        values = {k: v for k, v in values.items() if v is not
None}
        query = shops.update().where(shops.c.id ==
shop_id).values(values)
        await self.database.execute(query)
        query = shops.select().where(shops.c.id == shop_id)
        shop = await self.database.fetch_one(query=query)
        return shop

    async def delete(self,id:int)->Optional[Shop]:
        query = shops.delete().where(shops.c.id == id)
        shop = await self.database.fetch_one(query=query)
```

```
        if shop is None:
            return None
        return Shop.parse_obj(shop)


    async def get_all(self, limit: int = 100, skip: int = 0)
-> List[Shop]:
        query = shops.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def get_by_id(self, id: int) -> Optional[Shop]:
        query = shops.select().where(shops.c.id == id)
        shop = await  self.database.fetch_one(query=query)
        if shop is None:
            return None
        return Shop.parse_obj(shop)
```

Endpoints.shops.(Функції API для моделі shops):

```
router = APIRouter()


@router.get("/", response_model=List[Shop])
async def read_shops(
        limit: int = 100,
        skip: int = 0,
        shops: ShopRepository =
Depends(get_shop_repository)):
    return await shops.get_all(limit=limit, skip=skip)
```

```
    @router.post("/", response_model=Shop)
    async def create_shop(
            sh: BaseShop,
            shops: ShopRepository =
Depends(get_shop_repository),
            current_user: User = Depends(get_current_user)):
        return await
shops.create(user_id=int(current_user.id), sh=sh)


    @router.put("/", response_model=Shop)
    async def update_shop(
            id: int,
            sh: BaseShop,
            shops: ShopRepository =
Depends(get_shop_repository),
            current_user: User = Depends(get_current_user)):
        shop = await shops.get_by_id(id=id)
        user_id = current_user.id
        shop_owner_id = shop.owner_id
        if shop is None or int(shop_owner_id) !=
int(user_id):
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Shop not found")
        return await shops.update(shop_id=id,
```

```
user_id=int(current_user.id), sh=sh)


    @router.patch("/", response_model=Shop)
    async def partial_update(
            id: int,
            sh: ShopPartialUpdateIn,
            shops: ShopRepository =
Depends(get_shop_repository),
            current_user: User = Depends(get_current_user)):
        shop = await shops.get_by_id(id=id)
        if shop is None or shop.owner_id !=
int(current_user.id):
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Shop not found")
        return await shops.partial_update(shop_id=id,
user_id=int(current_user.id), sh=sh)

    @router.delete("/", response_model=Shop)
    async def delete(id:int,

shops:ShopRepository=Depends(get_shop_repository),
                     current_user: User =
Depends(get_current_user)):
        shop = await shops.get_by_id(id=id)
        if shop is None or shop.owner_id !=
int(current_user.id):
```

```
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Shop not found")

        return await shops.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

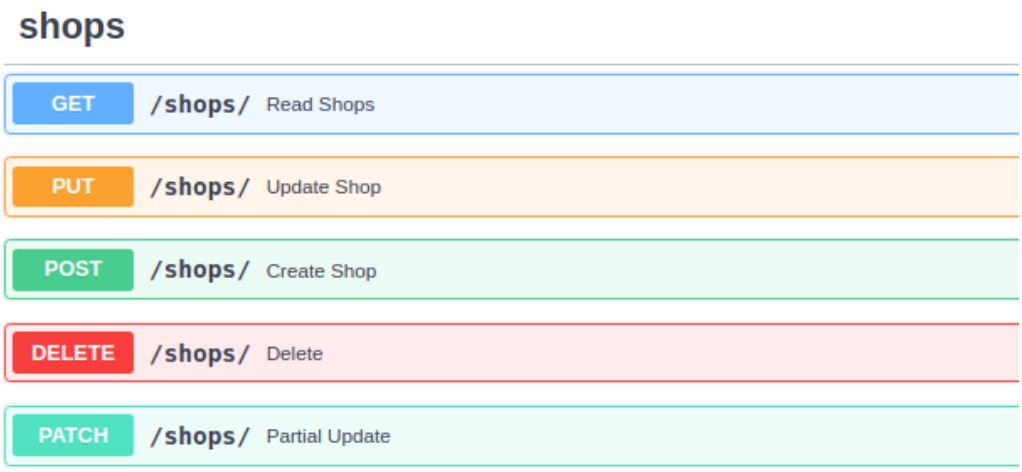

Рисунок 3.6 – FastApi docs для моделі Shops

Методи роботи з моделлю Categories:

```
   class CategoryRepository(BaseRepository):
    async def get_all(self, limit: int = 100, skip: int = 0)
-> List[Category]:
        query = categories.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)
```

```
    async def get_by_id(self, id: int) -> Optional[Category]:
        query = categories.select().where(categories.c.id ==
id)
        category = await self.database.fetch_one(query=query)
        if category is None:
            return None
        return Category.parse_obj(category)


    async def create(self,cat:BaseCategory)->Category:
        category=Category(
            id=0,
            title=cat.title,
            description=cat.description
        )
        values = {**category.dict()}
        values.pop("id", None)
        query = categories.insert().values(**values)
        category.id=await self.database.execute(query=query)
        return category

    async def update(self, id: int, cat: BaseCategory) ->
Category:
        category=Category(
            id=id,
            title=cat.title,
            description=cat.description
        )
```

```
        values = {**category.dict()}
        values.pop("id", None)
        query = categories.update().where(categories.c.id ==
id).values(**values)
        await self.database.execute(query)
        return category

    async def partial_update(self, id:int,
cat:CategoryPartialUpdateIn)->Optional[Category]:
        values={**cat.dict()}
        values = {k:v for k, v in values.items() if v is not
None}
        query = categories.update().where(categories.c.id ==
id).values(values)
        await self.database.execute(query)
        query = categories.select().where(categories.c.id ==
id)
        category = await self.database.fetch_one(query=query)
        return category

    async def delete(self,id:int)->Optional[Category]:
        query = categories.delete().where(categories.c.id ==
id)
        category = await self.database.fetch_one(query=query)
        if category is None:
            return None
        return Category.parse_obj(category)
```

Endpoints.categories.(Функції API для моделі categories):

```
router = APIRouter()

@router.get("/", response_model=List[Category])
async def read_categories(
        limit: int = 100,
        skip: int = 0,
        shops: CategoryRepository =
Depends(get_category_repository)):
    return await shops.get_all(limit=limit, skip=skip)

@router.post("/", response_model=Category)
async def create_category(
        cat:BaseCategory,

categories:CategoryRepository=Depends(get_category_repository)
):
    return await categories.create(cat=cat)

@router.put("/",response_model=Category)
async def update_category(
        id:int,
        cat:BaseCategory,

categories:CategoryRepository=Depends(get_category_repository)
):
    category = await categories.get_by_id(id=id)
```

```
        if category is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                detail="Category not found")
        return await categories.update(id=id,cat=cat)

    @router.patch("/", response_model=Category)
    async def partial_update(
            id:int,
            cat:CategoryPartialUpdateIn,

categories:CategoryRepository=Depends(get_category_repository)
):
        category=await categories.get_by_id(id=id)
        if category is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                detail="Category not found")
        return await categories.partial_update(id=id,cat=cat)

    @router.delete("/", response_model=Category)
    async def delete(
            id:int,
            categories: CategoryRepository =
Depends(get_category_repository)):
        return await categories.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі

FastAPI Docs

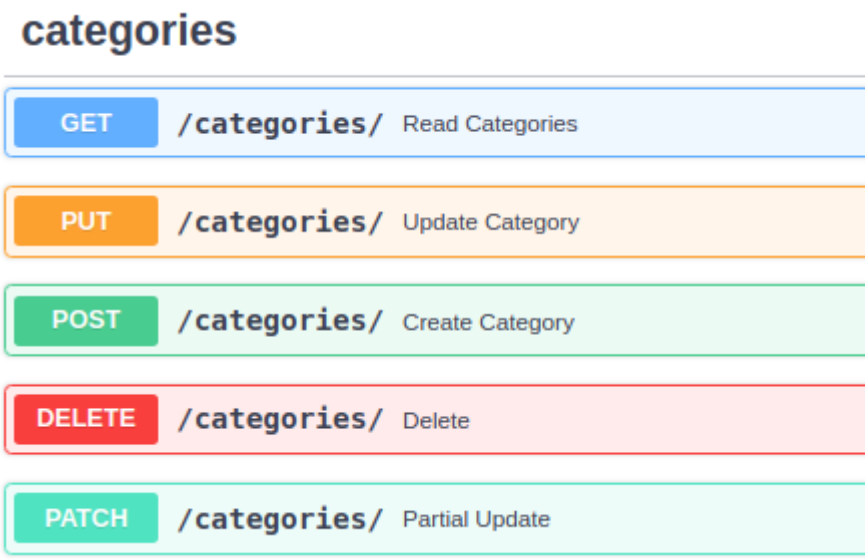

Рисунок 3.7 – FastApi docs для моделі Categories

Методи роботи з моделлю Products:

```
class ProductRepository(BaseRepository):
    async def get_all(self, limit: int = 100, skip: int = 0)
-> List[Product]:
        query = products.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def get_by_id(self, id: int) -> Optional[Product]:
        query = products.select().where(products.c.id == id)
        product = await self.database.fetch_one(query=query)
        if product is None:
            return None
        return Product.parse_obj(product)

    async def get_all_by_category_id(self, category_id: int,
                                     limit: int = 100,
```

```
                                    skip: int = 0) ->
List[Product]:
        query = products.select().where(products.c.category_id
== category_id).limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)


    async def create(self,p:BaseProduct)->Product:
        product=Product(
            id=0,
            shop_id=p.shop_id,
            category_id=p.category_id,
            title=p.title,
            short_description=p.short_description,
            description=p.description,
            price=p.price,
            created_at=datetime.datetime.utcnow(),
            head_image_url=p.head_image_url,
            is_availability=p.is_availability
        )
        values = {**product.dict()}
        values.pop("id", None)
        query = products.insert().values(**values)
        product.id = await self.database.execute(query=query)
        return product

    async def update(self,id:int,p:ProductUpdateIn):
        product = Product(
```

```
            id=id,
            shop_id=p.shop_id,
            category_id=p.category_id,
            title=p.title,
            short_description=p.short_description,
            description=p.description,
            created_at=datetime.datetime.utcnow(),
            price=p.price,
            head_image_url=p.head_image_url,
            is_availability=p.is_availability
        )
        values = {**product.dict()}
        values.pop("id", None)
        values.pop("created_at", None)
        query = products.update().where(products.c.id ==
id).values(**values)
        await self.database.execute(query)
        return product

    async def
partial_update(self,id:int,p:ProductPartialUpdateIn)->Optional
[Product]:
        values = {**p.dict()}
        values = {k: v for k, v in values.items() if v is not
None}
        query = products.update().where(products.c.id ==
id).values(values)
        await self.database.execute(query)
```

```
        query = products.select().where(products.c.id == id)
        product = await self.database.fetch_one(query=query)
        return product

    async def delete(self,id:int)->Optional[Product]:
        query = products.delete().where(products.c.id == id)
        product = await self.database.fetch_one(query=query)
        if product is None:
            return None
        return Product.parse_obj(product)
```

Endpoints.products.(Функції API для моделі products):

```
router = APIRouter()


@router.get("/", response_model=List[Product])
async def read_products(
        limit: int = 100,
        skip: int = 0,
        products: ProductRepository =
Depends(get_product_repository)):
    return await products.get_all(limit=limit, skip=skip)

@router.get("/{order_id}", response_model=List[Product])
async def read_products_by_category(
        category_id:int,
        limit: int = 100,
        skip: int = 0,
```

```
            products: ProductRepository =
Depends(get_product_repository)):
        return await
products.get_all_by_category_id(category_id=category_id,limit=
limit, skip=skip)

    @router.post("/", response_model=Product)
    async def create_product(
            product: BaseProduct,
            products: ProductRepository =
Depends(get_product_repository)):
        return await products.create(p=product)



    @router.put("/", response_model=Product)
    async def update_product(
            id: int,
            p: ProductUpdateIn,
            products: ProductRepository =
Depends(get_product_repository)):
        product = await products.get_by_id(id=id)
        if product is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
        return await products.update(id=id, p=p)
```

```
    @router.patch("/", response_model=Product)
    async def partial_update_product(
            id: int,
            p: ProductPartialUpdateIn,
            products: ProductRepository =
Depends(get_product_repository)):
        product = await products.get_by_id(id=id)
        if product is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
        return await products.partial_update(id=id, p=p)


    @router.delete("/", response_model=Product)
    async def delete(
            id: int,
            products: ProductRepository =
Depends(get_product_repository)):
        return await products.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

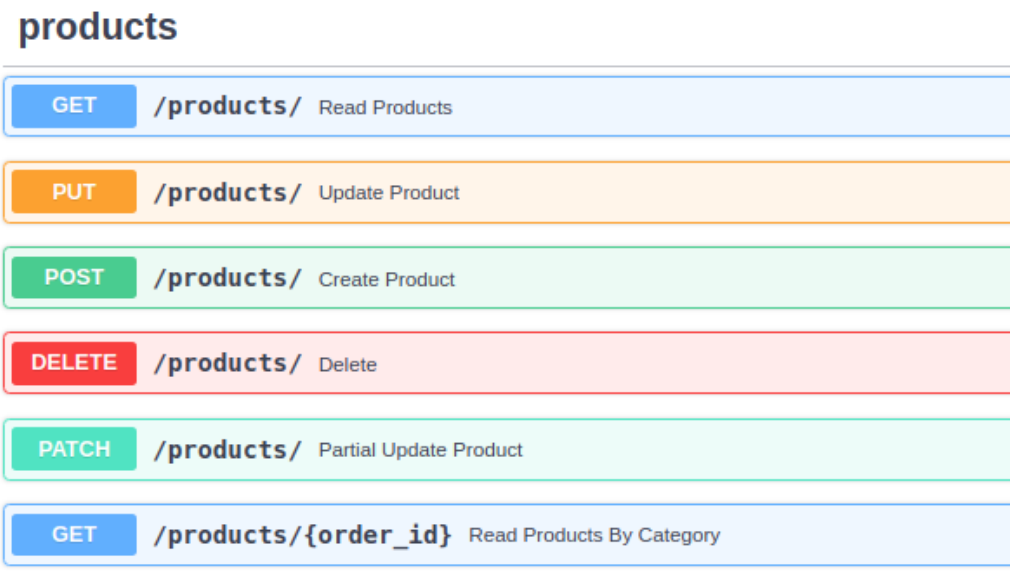

Рисунок 3.8 – FastApi docs для моделі Products

Методи роботи з моделлю Images:

```
class ImageRepository(BaseRepository):
 async def get_all(self, limit: int = 100, skip: int = 0)
-> List[Image]:
        query = images.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

 async def get_by_id(self, id: int) -> Optional[Image]:
        query = images.select().where(images.c.id == id)
        image = await self.database.fetch_one(query=query)
        if image is None:
            return None
        return Image.parse_obj(image)

 async def get_all_by_product_id(self, product_id: int,
                                 limit: int = 100,
```

```
                                        skip: int = 0) ->
List[Image]:
        query = images.select().where(images.c.product_id ==
product_id).limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def create(self, i: BaseImage) -> Image:
        image=Image(
            id=0,
            product_id=i.product_id,
            image_url=i.image_url
        )
        values = {**image.dict()}
        values.pop("id", None)
        query = images.insert().values(**values)
        image.id = await self.database.execute(query=query)
        return image

    async def update(self, id: int, i: ImageUpdateIn) ->
Image:
        image = Image(
            id=0,
            product_id=0,
            image_url=i.image_url
        )
        values = {**image.dict()}
        values.pop("id", None)
        values.pop("product_id", None)
```

```
        query = images.update().where(images.c.id ==
id).values(**values)
        await self.database.execute(query)
        query = images.select().where(images.c.id == id)
        image = await self.database.fetch_one(query=query)
        return image

    async def delete(self,id:int)->Optional[Image]:
        query = images.delete().where(images.c.id == id)
        image = await self.database.fetch_one(query=query)
        if image is None:
            return None
        return Image.parse_obj(image)
```

Endpoints.images.(Функції API для моделі images):

```
router = APIRouter()

@router.get("/", response_model=List[Image])
async def read_images(
        limit: int = 100,
        skip: int = 0,
        images: ImageRepository =
Depends(get_image_repository)):
    return await images.get_all(limit=limit, skip=skip)

@router.get("/{product_id}", response_model=List[Image])
async def read_images_by_product(
```

```
            product_id:int,
            limit: int = 100,
            skip: int = 0,
            images: ImageRepository =
Depends(get_image_repository)):
        return await
images.get_all_by_product_id(product_id=product_id,limit=limit
, skip=skip)

    @router.post("/", response_model=Image)
    async def create_image(
            image: BaseImage,
            images: ImageRepository =
Depends(get_image_repository)):
        return await images.create(i=image)

    @router.put("/", response_model=Image)
    @router.patch("/", response_model=Image)
    async def update_image(
            id: int,
            i: ImageUpdateIn,
            images: ImageRepository =
Depends(get_image_repository)):
        image = await images.get_by_id(id=id)
        if image is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
```

```
        return await images.update(id=id, i=i)

    @router.delete("/", response_model=Image)
    async def delete(
            id: int,
            images: ImageRepository =
Depends(get_image_repository)):
        return await images.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

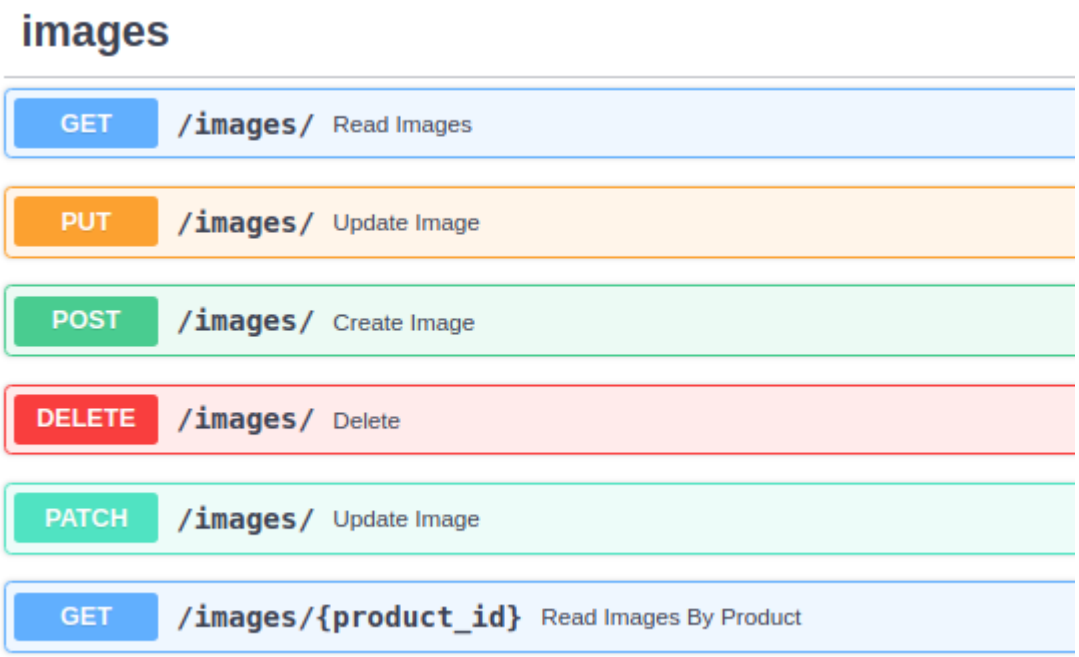

Рисунок 3.10 – FastApi docs для моделі Images

Методи роботи з моделлю Orders:

```
class OrderRepository(BaseRepository):
```

```
    async def get_all(self, limit: int = 100, skip: int = 0)
-> List[Order]:
        query = orders.select().limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def get_by_id(self, id: int) -> Optional[Order]:
        query = orders.select().where(orders.c.id == id)
        order = await self.database.fetch_one(query=query)
        if order is None:
            return None
        return Order.parse_obj(order)

    async def create(self, o: BaseOrder) -> Order:
        order=Order(
            id=0,
            customer_id=o.customer_id,
            recipient_name=o.recipient_name,
            contact_phone_number=o.contact_phone_number,
            city=o.city,
            address=o.address,
            created_at=datetime.datetime.utcnow(),
            updated_at=datetime.datetime.utcnow(),
            status=o.status,
            )
        values = {**order.dict()}
        values.pop("id", None)
        query = orders.insert().values(**values)
        order.id = await self.database.execute(query=query)
```

```
        return order

    async def update(self, id: int, o:OrderUpdateIn)->Order:
        order = Order(
            id=id,
            customer_id=o.customer_id,
            recipient_name=o.recipient_name,
            contact_phone_number=o.contact_phone_number,
            city=o.city,
            address=o.address,
            created_at=datetime.datetime.utcnow(),
            updated_at=datetime.datetime.utcnow(),
            status=o.status,
        )
        values = {**order.dict()}
        values.pop("id", None)
        values.pop("created_at", None)
        query = orders.update().where(orders.c.id ==
id).values(**values)
        await self.database.execute(query)
        return order

    async def partial_update(self, id: int, o:
OrderPartialUpdateIn) -> Optional[Order]:
        values = {**o.dict()}
        values = {k: v for k, v in values.items() if v is not
None}
        query = orders.update().where(orders.c.id ==
```

```
id).values(values)
        await self.database.execute(query)
        query = orders.select().where(orders.c.id == id)
        order = await self.database.fetch_one(query=query)
        return order

    async def delete(self,id:int)->Optional[Order]:
        query = orders.delete().where(orders.c.id == id)
        order = await self.database.fetch_one(query=query)
        if order is None:
            return None
        return Order.parse_obj(order)
```

Endpoints.orders.(Функції API для моделі orders):

```
router = APIRouter()

@router.get("/", response_model=List[Order])
async def read_orders(
        limit: int = 100,
        skip: int = 0,
        orders: OrderRepository =
Depends(get_order_repository)):
        return await orders.get_all(limit=limit, skip=skip)

@router.post("/", response_model=Order)
async def create_product(
        order: BaseOrder,
```

```
            orders: OrderRepository =
Depends(get_order_repository)):
        return await orders.create(o=order)

    @router.put("/", response_model=Order)
    async def update_product(
            id: int,
            o: OrderUpdateIn,
            orders: OrderRepository =
Depends(get_order_repository)):
        order = await orders.get_by_id(id=id)
        if order is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
        return await orders.update(id=id, o=o)

    @router.patch("/", response_model=Order)
    async def update_product(
            id: int,
            o: OrderPartialUpdateIn,
            orders: OrderRepository =
Depends(get_order_repository)):
        order = await orders.get_by_id(id=id)
        if order is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
```

```
        return await orders.partial_update(id=id, o=o)


    @router.delete("/", response_model=Order)
    async def delete(
            id: int,
            orders: OrderRepository =
Depends(get_order_repository)):
        return await orders.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

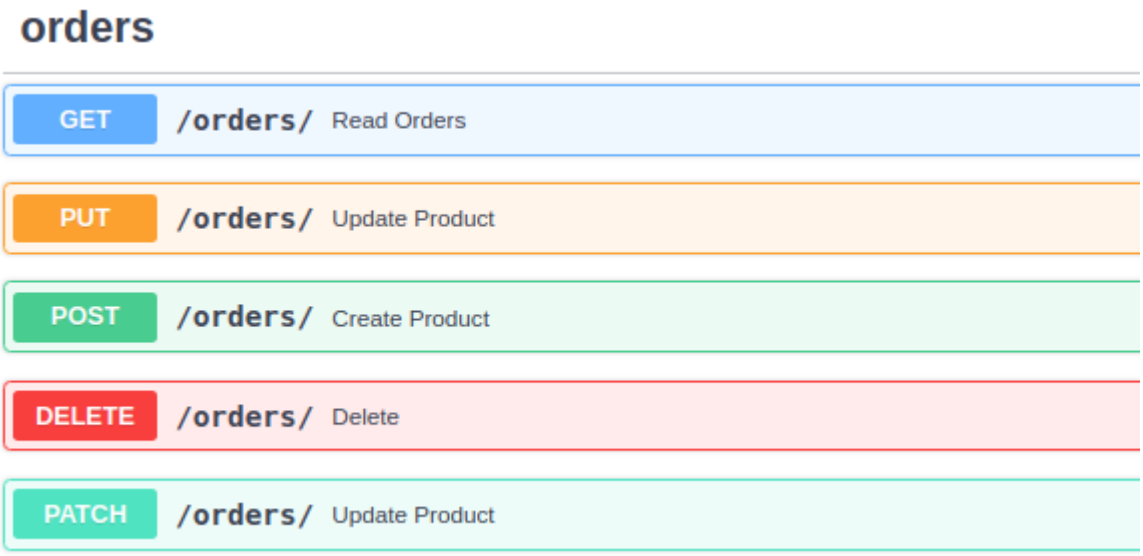

Рисунок 3.11 – FastApi docs для моделі Orders

Методи роботи з моделлю Order_items:

```
  class OrderItemRepository(BaseRepository):


    async def get_all(self, limit: int = 100, skip: int = 0)
-> List[OrderItem]:
        query = order_items.select().limit(limit).offset(skip)
```

```
        return await self.database.fetch_all(query=query)

    async def get_by_id(self, id: int) -> Optional[OrderItem]:
        query = order_items.select().where(order_items.c.id ==
id)
        order_item = await
self.database.fetch_one(query=query)
        if order_item is None:
            return None
        return OrderItem.parse_obj(order_item)

    async def get_all_by_order_id(self, order_id: int,
                                  limit: int = 100,
                                  skip: int = 0) ->
List[OrderItem]:
        query =
order_items.select().where(order_items.c.order_id ==
order_id).limit(limit).offset(skip)
        return await self.database.fetch_all(query=query)

    async def create(self, o_i: BaseOrderItem) -> OrderItem:
        order_item = OrderItem(
            id=0,
            order_id=o_i.order_id,
            product_id=o_i.product_id,
            quantity=o_i.quantity
        )
        values = {**order_item.dict()}
```

```
        values.pop("id", None)
        query = order_items.insert().values(**values)
        order_item.id = await
self.database.execute(query=query)
        return order_item

    async def update(self, id: int, o_i: OrderItemUpdateIn) ->
OrderItem:
        order_item = OrderItem(
            id=0,
            order_id=0,
            product_id=0,
            quantity=o_i.quantity
        )
        values = {**order_item.dict()}
        values.pop("id", None)
        values.pop("order_id", None)
        values.pop("product_id", None)
        query = order_items.update().where(order_items.c.id ==
id).values(**values)
        await self.database.execute(query)
        query = order_items.select().where(order_items.c.id ==
id)
        order_item = await
self.database.fetch_one(query=query)
        return order_item

    async def delete(self,id:int)->Optional[OrderItem]:
```

```
        query = order_items.delete().where(order_items.c.id ==
id)
        order_item = await
self.database.fetch_one(query=query)
        if order_item is None:
            return None
        return OrderItem.parse_obj(order_item)
```

Endpoints.order_items.(Функції API для моделі order_items):

```
router = APIRouter()

@router.get("/", response_model=List[OrderItem])
async def read_order_items(
        limit: int = 100,
        skip: int = 0,
        order_items: OrderItemRepository =
Depends(get_order_item_repository)):
         return await order_items.get_all(limit=limit,
skip=skip)

@router.get("/{order_id}",
response_model=List[OrderItem])
async def read_order_items_by_order(
        order_id:int,
        limit: int = 100,
        skip: int = 0,
        order_items: OrderItemRepository =
Depends(get_order_item_repository)):
```

```
        return await
order_items.get_all_by_order_id(order_id=order_id,limit=limit,
skip=skip)

    @router.post("/", response_model=OrderItem)
    async def create_order_item(
            order_item: BaseOrderItem,
            order_items: OrderItemRepository =
Depends(get_order_item_repository)):
        return await order_items.create(o_i=order_item)

    @router.put("/", response_model=OrderItem)
    @router.patch("/", response_model=OrderItem)
    async def update_order_item(
            id: int,
            o_i: OrderItemUpdateIn,
            order_items: OrderItemRepository =
Depends(get_order_item_repository)):
        order_item = await order_items.get_by_id(id=id)
        if order_item is None:
            raise
HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                                  detail="Product not found")
        return await order_items.update(id=id, o_i=o_i)

    @router.delete("/", response_model=OrderItem)
    async def delete(
            id: int,
```

```
            order_items: OrderItemRepository =
Depends(get_order_item_repository)):
        return await order_items.delete(id=id)
```

Превірити роботу функціоналу можна у вбудованому в FastApi механізмі FastAPI Docs

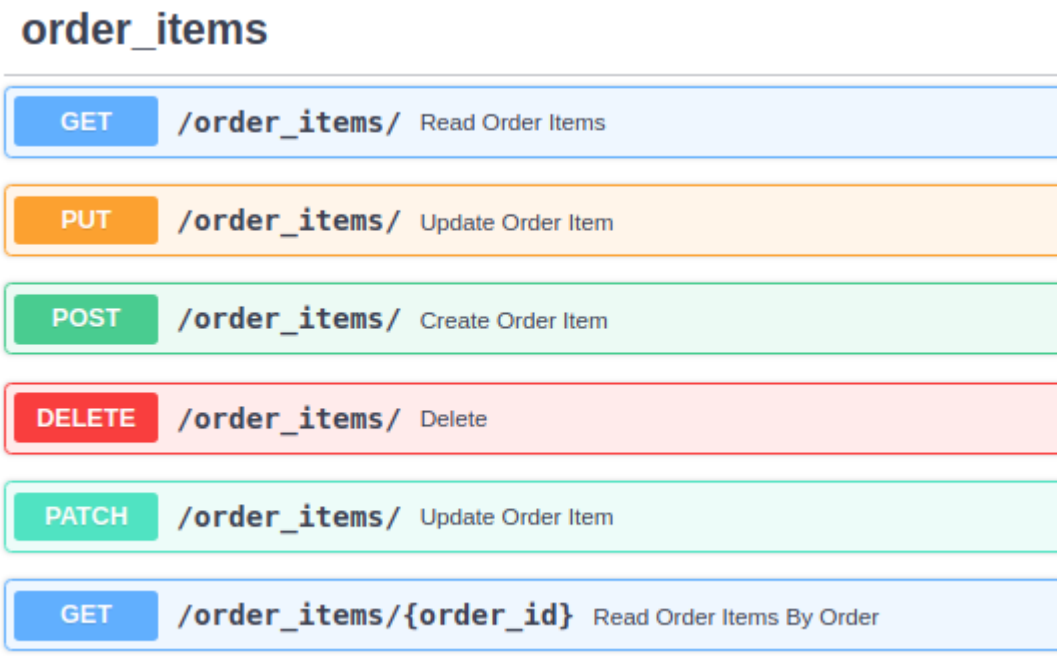

Рисунок 3.12 – FastApi docs для моделі Order_Items

ВИСНОВКИ

За результатами аналізу наявних програмних продуктів, було сформовано ряд вимог до створення власного торгового майданчика.

Також, проаналізовано та вибрано авторизацію та аутентифікацію для користувачів в виді токенів.

Під розробки системи керування були використанні різноманітні технології та знання, зокрема

- мова Python;
- Фреймворк FastAPI
- СУБД PostgreSQL
- підтримка програмного забезпечення: використання системи контролю версій GIT.

Розроблений додаток задовольняє всім вимогам, поставленим на етапі постановки завдання.

СПИСОК ЛІТЕРАТУРИ

1. Свістельнік А.О. Інформаційна система онлайн-магазину з використанням фреймворку Django [Текст]: робота на здобуття кваліфікаційного рівня бакалавр; спец.: 122 - комп`ютерні науки А.О. Свістельнік; кер. Г.А. Олексієнко. - Суми: СумДУ, 2020. - 12-16 с.

2. Свістельнік А.О. Інформаційна система онлайн-магазину з використанням фреймворку Django [Текст]: робота на здобуття кваліфікаційного рівня бакалавр; спец.: 122 - комп`ютерні науки А.О. Свістельнік; кер. Г.А. Олексієнко. - Суми: СумДУ, 2020. - 12-16 с. [Електронний ресурс] – Режим доступу: https://essuir.sumdu.edu.ua/bitstream-download/123456789/79717/1/Svistelnik_bac_rob.pdf

3. REST API Architectural Constraints [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/rest-api-architectural-constraints/

4. Документація Fast api [Електронний ресурс] – Режим доступу: https://fastapi.tiangolo.com/

5. Стаття про концепцію розробки MVC https://rtfm.co.ua/django-book-model-razrabotki-mtc-model-view-controller/

6. Learning python Mark Lutz

7. Python practice book Anand Chitipothu

8. Администрирование PostgreSQL 9. Книга рецептов

9. Practical PostgreSQL Джошуа Д. Дрейк

10. REST API Design Mark H. Massé