Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мультимедійних систем

Розробка клієнт-серверного web-застосунку з використанням Python/Django та Angular

Текстова частина до курсової роботи

за спеціальністю «Інженерія Програмного Забезпечення»- 121

Керівник курсової роботи
м., ст. в. ___Борозенний С. О
(прізвище та ініціали)
_____
(підпис)
“____” _____ 2021 р.
Виконав студент Ляш Д. В.
(прізвище та ініціали)
_____
(підпис)
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,
к.ф.-м. н. О. П. Жежерун

_____

(підпис)

“____” _____ 2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студента Ляша Данила Вячеславовича факультету інформатики 4 курсу
ТЕМА Розробка клієнт-серверного web-застосунку з використанням Python/Django та Angular

Вихідні дані:

Зміст ТЧ до курсової роботи:

Календарний план

Вступ

3 розділи роботи

Висновки

Список використаної літератури

Дата видачі “____” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Розробка клієнт-серверного web-застосунку з використанням Python/Django та Angular

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	15.10.2020	
2.	Пошук тематичної літератури	15.11. 2020	
3.	Ознайомлення з літературою	1.12. 2020	
4.	Вивчення аналогів	20.12. 2020	
5.	Планування веб-сайту	05.01. 2021	
6.	Ознайомлення з мовою Python	12.01. 2021	
7.	Ознайомлення з фреймворком Django	18.01. 2021	
8.	Ознайомлення з технологією Angular	31.01. 2021	
9.	Реалізація першої частини проекту	05.02. 2021	
10.	Реалізація другої частини проекту	25.02. 2021	
11.	Реалізація третьої частини проекту	15.03. 2021	
12.	Оформлення сторінок	25.03. 2021	
13.	Написання текстової частини	29.03. 2021	
14.	Перегляд змісту роботи керівником	08.04. 2021	
15.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	09.04. 2021	
16	Створення презентації	10.04. 2020	

Студент Ляш Д. В.

Керівник Борозенний С. О.

“____” _____

Зміст

Анотація

У роботі розглянута задача аналізу роботи і розробки клієнт-серверного web-застосунку на прикладі створення маркетплейсу з використанням Python, Django та Angular.

В першому розділі розглянуті основні теоретичні відомості про клієнт-серверні застосування. Дається їх визначення і пояснюється як вони побудовані на прикладі створеного застосування маркетплейсу з використанням цих технологій.

В другому розділі розглянуті теоретичні відомості про мову Python, оснований на ній фреймворк Django, використаний для сервера та front-end фреймворк Angular. Вказуються їх переваги та недоліки.

В третьому розділі розглядається створений застосунок. Показана робота з обраними технологіями для клієнтської та серверної частини проекту, оглянуті та проаналізовані їх можливості із частковою демонстрацією роботи програми. У кінці цього розділу розглядаються результати аналізу виконаної роботи.

Вступ

Актуальність

В сучасному світі веб-технологій можливість створювати різноманітні клієнт-серверні застосування дуже актуальна для сучасного суспільства тема, особливо в період пандемії, коли люди виконують більшість справ за допомогою інтернету.[1]

Зараз, більшість людей проглядають кожного дня безліч різноманітних сайтів та використовують сотні додатків на своїх пристроях, починаючи з телефона закінчуючи приставками для телевізорів. Все це побудовано на серверах та клієнтських застосуваннях[1]

Також, кожного дня такі системи підтримуються багатьма людьми: спеціалістами з підтримки користувачів, розробників, аналітиків та іншими працівниками. Тобто це означає створення робочих місць для великої кількості людей, які, навіть зараз у період карантину, можуть працювати не виходячи з дому.[2]

Саме тому розробка та аналіз таких систем є актуальною темою для сьогочасного суспільства, якому потрібні розробники на подібних технологіях.

Мета та завдання курсової роботи

Мета: Показати та дослідити можливості розробки клієнт-серверного застосування на прикладі створення простого маркетплейсу за допомогою Django та Angular. Покращити знання з вказаних технологій. Проаналізувати етапи розробки.

Завдання: Проаналізувати та показати взаємодію клієнт-серверного застосування з демонстрацією можливостей обраних технологій (Angular та Django) . Розробити базове клієнт-серверне веб-застосування маркетплейс для розміщення товарів. Проаналізувати отримані результати роботи.

Розділ 1 Клієнт-серверні застосування

Розділ 1.1 Базові поняття

Клієнт-серверне застосування під собою означає клієнт-серверну архітектуру. Термін «клієнт-сервер», власне, означає програмний комплекс архітектури, в якому описана взаємодія клієнтської та серверної частини. Тобто, клієнт дає запит, сервер відповідає. [3]

Популярність клієнт-серверної архітектури надходить з динамічним розвитком мережі Інтернет та зосередження великої кількості інформації на серверах або в хмарах.[4]

Клієнт-серверна архітектура – це концепція інформаційної мережі, в основі якої сервери з ресурсами, які обслуговують своїх клієнтів. Дана архітектура визначає такі компоненти: [4]

- Сервери, які мають роль зберігання та надання інформації.
- Клієнти, які спілкуються з серверами для надання користувачеві потрібну інформацію.
- Мережа, що тримає на собі взаємодію клієнтів і серверів.

Клієнт-серверна модель взаємодії визначається розподілом обов’язків між клієнтом та сервером, які можна поділити на 3 операції: [4]

- Представлення даних, що по собі в більшості випадків має під собою користувацький інтерфейс і надає дані користувачеві у потрібному представленні, з введенням команд.
- Рівень логіки, який реалізує основну бізнес-логіку застосунку з обробкою інформації.
- Рівень керування даними, в більшості випадків – зв’язок з базою даних

Розділ 1.2 Сервер

Сервер – частина hardware або software, яка надає функціональність іншим частинам, які називають «клієнт».[5]

Роль – це функція сервера, які сервер може відігравати у різній кількості, тобто мати одну або декілька ролей. [4]

Ролі сервера бувають такими:[4]

- Веб-сервер
- Сервер застосунків
- Сервер баз даних
- Файловий сервер

Та інші різновидності.

У даній роботі мова буде йти про основну роль сервера, як веб-сервер.

Веб-сервер – Сервер, що приймає HTTP-запити від клієнтів, та надає HTTP-відповіді, які містять у собі текст, зображення та інше. [4]

В розробленому застосуванні сервер виступає як подання API для отримання інформації, точніше REST API.

Передача репрезентативного стану (англ. Representational State Transfer, REST) - це набір обмежень та рекомендацій, які утворюють надійні, ефективні та масштабовані системи. Система називається RESTful, в якій дотримані такі обмеження.[6]

Основна ідея цього стану полягає у тому, що ресурс передається разом зі станом та зв’язками за допомогою заздалегідь визначених операцій та форматів.

Розділ 1.3 Клієнт

Невід’ємною частиною клієнт-серверної архітектури, звісно, є клієнтська частина.[7]

Сам по собі клієнт, це також комп’ютер, який отримує інформацію з сервера. Наприклад, коли хтось перевіряє електронну пошту через сайт Gmail, то цей сайт стає клієнтською частиною між користувачем та сервером. Так само як сайт, клієнтською частиною можуть бути: [7]

- Застосунки на персональному комп’ютері
- Застосунки на мобільному телефоні(смартфоні)
- Сайти, які відображають браузери

Є два основні типи як використовують клієнт: [7]

- Модель тонкого клієнта – коли основна логіка застосунку зосереджена на серверній частині, а клієнтська частина забезпечує функції рівня представлення
- Модель товстого клієнта – коли сервер керує передачею даних, а обробка усієї інформації базується на стороні клієнта. Товстими клієнти частіше за все називають пристрої з обмежено потужністю

Розділ 2 Технології використані для розробки

Розділ 2.1 Мова Python

Мова Python – інтерпретована об’єктно-орієнтовна високорівнева мова програмування, створена Гвідо ван Россумом, випущена у 1991 році. Назву «Пайтон» запозичено з британського шоу “Monty Python”. Філософія дизайну цієї мови базована на читабельності коду з великою кількістю відступів.[8]

Python підтримує та пакети модулів, що дозволяє повторно використовувати код і робить цю мову досить зручним у використанні. Інтерпретатор та стандартні бібліотеки доступні у вихідній та скомпільованій формі на основних платформах. Мова підтримує декілька парадигм: об’єктно-орієнтовна, функціональна, процедурна та аспектно-орієнтовна.[8]

Переваги Python:

- Базовий стереотип має багато корисних модулів.
- Переносимість програм.
- Можливість використання у діалоговому режимі.
- Відкритий код.
- Просте і потужне середовище розробки у базовому дистрибутиві.
- Чистий синтаксис.

Недоліки Python:

- Швидкість відтворення
- Обмеження дизайну
- Порівняно гірший доступ до баз даних

У мови також є своя філософія про прості речі у мові, якими можна описати основні переваги Python. Основний текст цієї філософії з’явився у коді у 2004 році, його можна отримати за допомогою команди import this у інтерпретаторі. [8]

Саме цю мову бере за основу фреймворк Django, який використано при розробці веб-застосунку для серверної частини.

Розділ 2.2 Фреймворк Django

Django – це безкоштовний веб фреймворк базований на мові Python з відкритим кодом. Підтримується некомерційною організацією Django Software Foundation. [9]

Основною задачею цього фреймворку є легке створення комплексних веб-сайтів з використанням баз даних. Django базується на перевикористанні компонентів, зменшення коду (в більшості повторюваного), швидка розробка.[9]

Основні можливості та переваги використання: [9]

- Швидке створення Web API
- Легкий та автономний веб-сервер
- ORM для доступу до БД
- Розширювана система шаблонів з наслідуванням
- Шаблони контролерів
- Вбудована система авторизації та аутентифікації
- Вбудований інтерфейс адміністратора
- Система серіалізації для зчитування або створення XML або JSON відтворення моделей.

Багато з відомих сайтів використовують Django. Наприклад, Instagram, Mozilla, Bitbucket та інші.

Цей фреймворк використаний як серверна частина застосування та приймає запити від клієнтської

Розділ 2.3 Angular

Angular – це платформа для розробки, що написана мовою TypeScript, яка розширює можливості JavaScript. Ця платформа розроблюється командою компанії Google та спільнотою приватних розробників. Angular – це, принципово новий, переписаний фреймворк AngularJS.[10]

Angular, як платформа, включає у себе: [10]

- Фреймворк, що базується на компонентах
- Колекцію гарно інтегрованих бібліотек, які покривають велику кількість потреб розробників.
- Набір інструментів для розробки, будування, тестування та оновлення коду.

Основи, на яких будується Angular застосування: [10]

- Ін'єкція залежностей
- Компоненти
- Шаблони
- Модулі

Ця платформа використана як клієнтська частина для відображення користувачу.

Розділ 3 Розробка та застосунок

Розділ 3.1 Сервер

Розділ 3.1.1 Основне

Для розробки сервера, який буде спілкуватися з клієнтом, як зазначено вище, був обраний фреймворк Django з пакетом Django REST framework для будування API.

Дивлячись на назву пакету Django REST framework, можна зрозуміти, що для створення API була використана архітектура REST, про яку згадано у теоретичних відомостях.

За цією архітектурою кожна одиниця інформації однозначно визначається за URL. Тобто, якщо взяти для прикладу маркетплейс де є товари, то вони визначені таким форматом – /goods, а якийсь певний товар – /goods/{id}, де {id} – унікальний ідентифікатор товару [11], в випадку реалізації проектної частини – UUID, що означає Universally Unique Identifier, в перекладі – універсальний унікальний ідентифікатор.

Для керування інформацією використані 5 методів: GET, POST, PATCH, PUT, DELETE – вони реалізують дії зчитування, додавання, часткової заміни, повної заміни та видалення відповідно [11].

Отже, даний фреймворк призначений для того щоб створити обгортку над інформацією, в моєму випадку – реляційна база даних, яка керується SQLite – системою керування базами даних, яка йде разом з проектом при створенні.

Розділ 3.1.2 Створення проекту та основні файли

Тож як почати створювати такий проект? Для цього був використане IDE PyCharm (інтегроване середовище розробки), оскільки це дуже зручний інструмент для розробника мовою Python.

При створенні проекту PyCharm пропонує обрати базу для проекту: чи то буде чистий Python, чи фреймворк Django, чи якісь інші можливі розширення. Також одразу пропонується створити Virtual Environment, який буде тримати наші пакети.

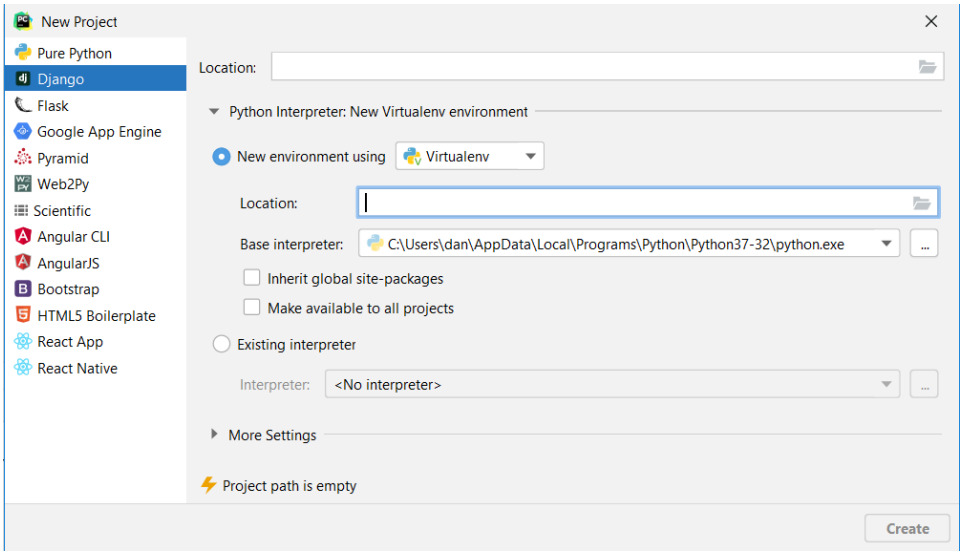

Рис. 1 Створення проекту

Усе це необхідно було би створювати вручну через консоль вручну, але таким чином це перший раз, коли можна зберегти час на розробці проекту.

Після генерації проекту можна побачити таку структуру:

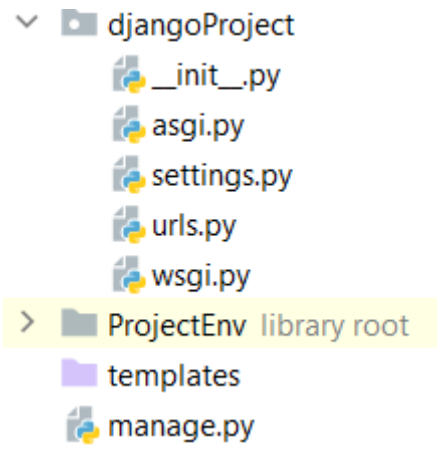

Рис. 2 Структура створеного проекту

Тут можна побачити три директорії: djangoProject, ProjectEnv, templates – це папка для додатків , папка Virtual Environment та папка шаблонів, які не використовуються у даному проекті. В головній папці, djangoProject, знаходяться ще декілька важливих основних для проекту файлів: settings.py та urls.py.

Settings.py – це папки в якій знаходяться основні конфігурації проекту, це можуть бути конфігурація бази даних, конфігурація посторінкового виводу, конфігурація авторизації та аутентифікації та багато іншого. Наприклад, тут визначаються додатки які використовуються у проекті:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',  # REST
    'drf_yasg',  # SWAGGER
    'django_filters',  # FILTERS

    'Project.app.authentication',
    'Project.app.user',
    'Project.app.goods',
    'Project.app.images',
]
```

Рис. 3 Частина налаштувань settings.py

Тут у коментарях вказані різноманітні пакети, які були завантажені через pip та використані у різних частинах проекту. Трохи нижче знаходяться додатки згенеровані через manage.py, про них мова піде далі.

Що до urls.py, то подібна папка буде генеруватися при створенні кожного додатку всередині нашого проекту. В цих файлах знаходиться конфігурація URL, за якими буде знаходитися ресурси для запитів.

Також у кінці списку згенерованих файлів можна знайти файл manage.py. Це файл з яким треба працювати для обробки різних частин проекту та його запуску. Наприклад, використовувати для міграцій, створення додатків і, власне, запуску сервера.

Розділ 3.1.3 Базова структура

У Django кожний проект складається з додатків, які ми генеруємо за допомогою команди python manage.py startapp {назва_додатку}. Кожен додаток є пакетом Python та має своє поле відповідальності. В проекті маркетплейс я створив таку структуру додатків:

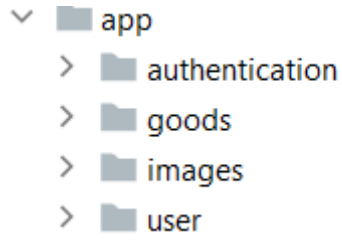

Рис. 4 Структура додатків

Додаток authentication відповідає за авторизацію та аутентифікацію у проекті, goods – основний додаток, в якому знаходиться основна частина проекту з товарами. Зображення інша структура для передачі і знаходиться у додатку images. В додатку user знаходиться інформація про користувачів.

Кожен з таких додатків генерується з вище зазначеним файлом для конфігурації URL і також налаштування моделей, тестів, представлень та іншого.

Розділ 3.1.4 Моделі

У кожній Об’єктно-орієнтованій мові сутності відображуються об’єктами, а об’єкти в Django керуються через ORM і мають назву «моделі» і зберігаються в файлі models.py для кожного додатку[9]. У нашому випадку фреймворк надає прикладний програмний інтерфейс доступу до даних. Якщо простими словами, то ми керуємо моделями через код і потреби в використанні SQL-запитів немає.

Отже, якщо навести приклад такої моделі – перейдемо в код і на прикладі побачимо як це відображено. На наступному зображенні можна побачити модель товарів:

```
class Goods(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False, unique=True)
    title = models.CharField(max_length=70)
    description = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=9, decimal_places=2)

    STATUS_TYPES = (
        ('CHECKING', 'Checking'),
        ('ACTIVE', 'Active'),
        ('DISABLED', 'Disabled')
    )

    status = models.CharField(max_length=15, choices=STATUS_TYPES, default='CHECKING')

    department = models.ForeignKey(Department, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)

    class Meta:
        db_table = "goods"
```

Рис. 5 Модель товарів

Так, на минулому зображенні, можна побачити, що клас Goods розширює базовий клас моделі і дає фреймворку зрозуміти, що наш клас – модель. Далі прописані властивості товару: унікальний ідентифікатор, найменування, опис, ціна, статус із можливими його варіантами, відділ товарів і користувач який володіє товаром. Також, усередині класу є мета клас, в якому вказана назва таблиці у яку буде зберігатися модель.

Такі моделі, як описано раніше, треба створювати для кожної сутності чи то користувач, чи то картинки, чи то розділи тощо.

Після створення усіх моделей треба викликати команду python manage.py makemigrations для того, щоб створити міграції, якщо просто – зміни, у базі даних, а потім застосувати їх командою python manage.py migrate. Після цього у під’єднаній баз з’являться таблиці.

Для мого проекту вийшла така схема бази даних:

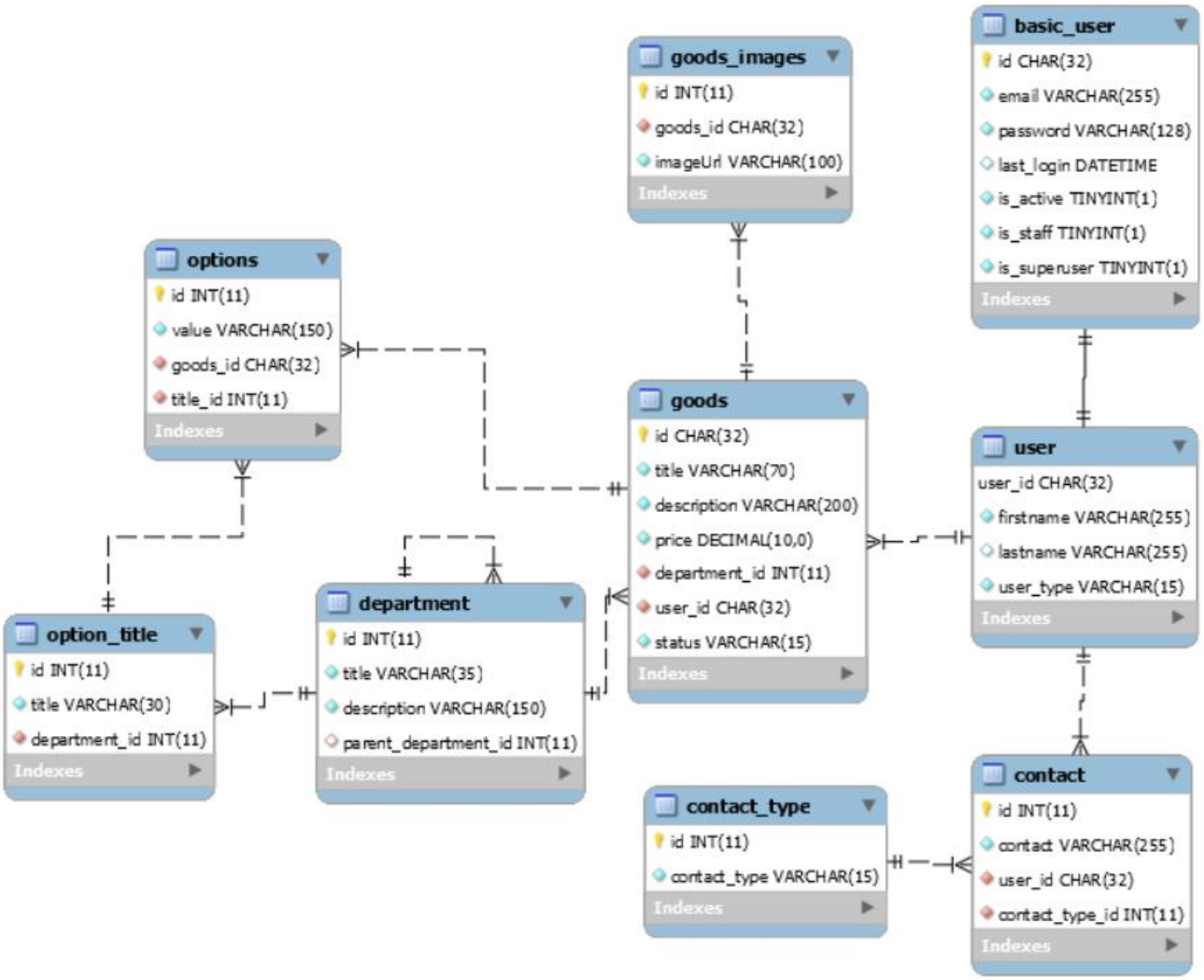

Рис. 6 Автоматично створена модель бази даних

Розділ 3.1.5 Представлення та внутрішній склад

Для зв’язку з клієнтською частиною за REST потрібні представлення або доступ до ресурсів, які надає сервер. Вони описуються у файлі views.py.

Існують декілька способів створення таких представлень, наприклад: Views та ViewSet. Різноманітні реалізації ViewSet дозволяють швидко створювати стандартні представлення та взаємодію з даними у застосуванні. Також такі реалізації допомагають запобігти плутанині із застарілими визначеннями URL через regex і зменшити розмір коду. Тому у реалізації маркетплейсу я зупинився саме на них для основної частини проекту.

Наприклад, так виглядає основне таке представлення:

```
class GoodsViewSet(viewsets.ModelViewSet):

    queryset = Goods.objects.all()

    filter_backends = [filters.SearchFilter, DjangoFilterBackend]
    search_fields = ['title']
    filter_class = GoodsFilterSet

    default_serializer_class = GoodsSerializer
    serializer_classes = {
        'list': GoodsReadSerializer,
        'retrieve': GoodsReadSerializer,
    }

    def get_serializer_class(self):
        return self.serializer_classes.get(self.action, self.default_serializer_class)

    def get_permissions(self):
        if self.action == 'create':
            return (CreateAuthenticated(),)
        elif self.action == 'update' or self.action == 'partial_update' or self.action == 'destroy':
            return (UpdateOwner(),)
        else:
            return (permissions.AllowAny(), )
```

Рис. 7 Представлення для товарів

Спочатку коду можна побачити, що створюється клас, який наслідує ModelViewSet, представлення яке генерує нам методи GET, POST, PUT, PATCH, DELETE, через які ми і взаємодіємо з клієнтом.

Далі йде queryset, це аргумент, за яким ми будемо отримувати та записувати дані нашого застосування на серверній стороні. Таким чином ми фактично обираємо основну модель для представлення.

Після цього прописані три поля: filter_backends, search_fields, filter_class. Їх визначають для того, щоб якимось чином фільтрувати отримані дані. SearchFilter використовує search_fields для того, щоб у формі запиту з'явилось поле title, за яким буде виконуватись пошук за найменуванням. DjangoFilterBackend використовує вже filter_class, у якому більш детально вказуються поля для різноманітних пошукових операцій.

Так виглядає реалізація фільтрів класу товарів:

```
class CharInFilter(BaseInFilter, CharFilter):
    pass


class NumberInFilter(BaseInFilter, NumberFilter):
    pass


class GoodsFilterSet(FilterSet):
    option__value1 = CharInFilter(lookup_expr='in', field_name='option__value')
    option__value2 = CharInFilter(lookup_expr='in', field_name='option__value')

    department_id = NumberInFilter(lookup_expr='in', field_name='department__id')

    class Meta:
        model = Goods
        fields = {
            'price': ['gte', 'lte'],
            'user_id': ['exact'],
            'status': ['exact']
        }
```

Рис. 8 Фільтри для товарів

За основу береться FilterSet, який дозволяє фреймворку зрозуміти як саме ми фільтруємо дані. У середині класу є мета клас де вказується модель для фільтрування та поля з методами цих фільтрувань.

В даному прикладі визначених полів наявні 4 методи: gte, lte, exact, in – більше або дорівнює, менше або дорівнює, точний пошук або значення мають знаходитися у масиві поданому у запиті.

У наведеному класі також є додаткові поля, які реалізують класи для фільтрації із числовими та рядковими типами. Наприклад, для фільтрації по характеристиках використані рядкові фільтри за значенням характеристики, а для фільтрації по розділах використаний числовий фільтр який дістає товари за унікальним ідентифікатором.

Наступною частиною є визначення серіалізаторів, які використовуються для того, щоб перетворювати складні дані у власні типи даних Python, які легко перетворюються у JSON для передачі між клієнтом та сервером. Реалізація як на рисунку представлення, не є стандартною, вона таким чином налаштована, для того щоб надати для читання та вписування різні реалізації з різними обмеженнями полів у запитах.

Наприклад, основний серіалізатор для зчитування товарів виглядає так:

```
class GoodsReadSerializer(serializers.ModelSerializer):
    user = GoodsUsersSerializer(required=True)
    department = DepartmentsSerializer(required=True)
    images = GoodsImageReadSerializer(source='goodsimage_set', many=True, read_only=True)
    options = GoodsOptionSerializer(source='option_set', many=True)

    class Meta:
        model = Goods
        fields = '__all__'
```

На цьому серіалізаторі ми вказуємо що він є нащадком ModelSerializer – основний для серілізації моделей. Всередині цього класу є ще один мета клас, який слугує для того, щоб вказати поля для репрезентації та модель для якої визначається серіалізатор.

Також, просто у класі існують додаткові поля, які існують для того, щоб з'єднувати моделі: user, department, images, options – власник товару, розділ, картинки товару та характеристики товару відповідно, які використовуються для відображення для клієнтів. В свою чергу кожне з цих полів використовує інші серіалізатори з опціями та джерелами які приєднуються по формату {назваМоделі}_set, самі ж серіалізатори мають таку ж саму структуру:

```
class GoodsImageReadSerializer(GoodsImageSerializer):

    class Meta(GoodsImageSerializer.Meta):
        exclude = ('goods', 'imageUrl', )
        fields = None


class GoodsUsersSerializer(serializers.ModelSerializer):
    contacts = UserContactsSerializer(source='contact_set', many=True)

    class Meta:
        model = User
        fields = '__all__'
```

наприклад картинки як основу використовують інший серіалізатор з розширенням виключень полів. Серіалізатор для користувачів також використовує серіалізатор, з цього можна зробити висновок, що серіалізатори пишуться як ієрархія моделей з вказуванням використовуваних, або ж виключених. полів

За визначенням серіалізаторів йде визначення доступу до даних. Тут відповідно до різних методів йде різний доступ, чи то запис доступний тільки для власників товарів та адміністраторів. Чи то просто для аутентифікованих користувачів.

Самі класи ж доступів виглядають так:

```
from rest_framework import permissions


class CreateAuthenticated(permissions.IsAuthenticated):

    def has_permission(self, request, view):
        return not request.user.is_staff


class UpdateOwner(permissions.IsAuthenticated):

    def has_object_permission(self, request, view, obj):
        return request.user.id == obj.user.user_id or request.user.is_staff
```

Рис. 9 Доступи для товарів

Тут клас CreateAuthenticated дає доступ на створення тим, хто не є адміністратором, але є аутентифікованим.

Наступний клас UpdateOwner дозволяє змінювати властивості товарів тільки для адміністраторів та власників

Усі ці доступи обумовлюють безпеку запитів, щоб ніхто окрім вас не зміг доступитися до ваших товарів без доступу.

Описані вище речі є частиною конфігурації представлень з усіма необхідними інструментами, в яких і посідає логіка серверного застосування.

Щоб підключити представлення, необхідно додати URL до файлу, який був описаний вище urls.py

Для додавання ми використовуємо DefaultRouter, який дозволяє додатку згенерувати за поданою назвою посилання ресурсів.

У проектному прикладі для вище описаного представлення це зроблено наступним чином:

```
from django.urls import path, include

from rest_framework import routers

from Project.app.goods import views

router = routers.DefaultRouter()
router.register(r'goods', views.GoodsViewSet)
router.register(r'departments', views.DepartmentsViewSet)

urlpatterns = [
    path('', include(router.urls))
]
```

Створюється роутер у якому реєструються назви наших ресурсів, тут їх дві оскільки у додатку може бути декілька представлень, в цьому випадку це товари та розділи.

І у кінці ми додаємо налаштований роутер до шляхів наших посилань. Наступним кроком після додання посилань у основному файлі посилань:

```
schema_view = get_schema_view(
    openapi.Info(
        title="Snippets API",
        default_version='v1',
        description="Test description",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="contact@snippets.local"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', schema_view.with_ui('swagger', cache_timeout=0), name='schema-swagger-ui'),
    path('redoc', schema_view.with_ui('redoc', cache_timeout=0), name='schema-redoc'),
    path('api/', include('Project.app.user.urls')),
    path('api/', include('Project.app.authentication.urls')),
    path('api/', include('Project.app.goods.urls')),
    path('api/', include('Project.app.images.urls')),

]
```

Рис. 10 Налаштовані посилання для ресурсів сервера

Тут ситуація схожа, є підключення усіх можливих посилань проекту. Також налаштований Swagger – автоматична документація API для кращого розуміння чи правильно сконфігурований додаток, а також для більш зручного розуміння використання розробнику клієнтської частини застосування.

Розділ 3.1.6 Підсумок по серверу

Так все описане реалізовується індивідуально для кожної моделі. Якщо покроково, то створюється модель, для неї представлення, представлення конфігурується потрібною аутентифікацією, фільтрами ресурсів та видом запитів через серіалізатори.

Такі сервери легко створюються, дійсно можна за декілька днів зробити великий серверний застосунок, швидко конфігурувати його та розширяти потрібними додатками або доповненнями.

Також треба сказати що у поєднанні з, достатньо простою для розуміння мовою Python все опрацьовується і розроблюється досить швидко, також IDE PyCharm допомагає в швидкості розробки.

В результаті розробки маємо ресурси, які використовуються клієнтом через POST, GET ,PUT, DELETE запити і отримують результати у JSON форматі.

Що ж до самої технології, то в період розробки були визначені основні переваги та недоліки для розробки.

Що ж до самої технології, то в період розробки були визначені основні переваги та недоліки для розробки.

Основні переваги для розробки:

- Можливість дуже швидкої розробки застосувань. Наприклад, через low coupling можна легко паралельно працювати з частинами коду і швидко інтегрувати їх.
- Застосунки на Django добре розширювані. Кожен додаток можна легко доповнювати, розширювати, змінювати та наповнювати при необхідності.

- Багато можливостей одразу вшиті при генерації проекту. Наприклад, всередині одразу інтегрована база даних SQLite з моделями аутентифікації та авторизації

Також можна виділити і недоліки для розробки:

- Django застосунки –монолітні. Фреймворк має певний спосіб визначати та виконувати завдання, що потребує додаткового вивчення.
- Також, зрозуміло з першого недоліку, що розробник повинен вивчати деякі способи розробки. Наприклад, спочатку не одразу стає зрозуміло як саме краще описувати API з усіма потрібними фільтрами та дозволами.

Розділ 3.2 Клієнт

Розділ 3.2.1 Основне

Після створеного сервера з базою даних, від якої надані посилання керуванням ресурсів, треба описати відображення цих ресурсів користувачу через клієнтське застосування.

Таким клієнтським застосуванням є додаток, який написаний на платформі Angular у проектній частині роботи.

У минулому підрозділі було зрозуміло, що дані між клієнтом та сервером будуть передаватися через відповідь сервера у форматі JSON. Що ж слугує для саме відображення даних користувачу?

Все просто: відображення даних обслуговується за допомогою HTML та CSS – мовою розмітки гіпертексту та каскадних таблиць стилів, для яких в більшій частині проекту використана бібліотека Bootstrap. А дані які ми отримуємо через сервер ми під’єднаємо за допомогою TypeScript сервісів.

Розділ 3.2.2 Структура клієнту

Усі додатки Angular з модулів з компонентами, сервісами та різноманітними утилітами, наприклад, interceptor або guard. Тобто структура базується на модулях як глобальній структурі, всередині якої знаходиться уся логіка застосунку [10]

Сама базова структура додатку маркетплейс виглядає так:

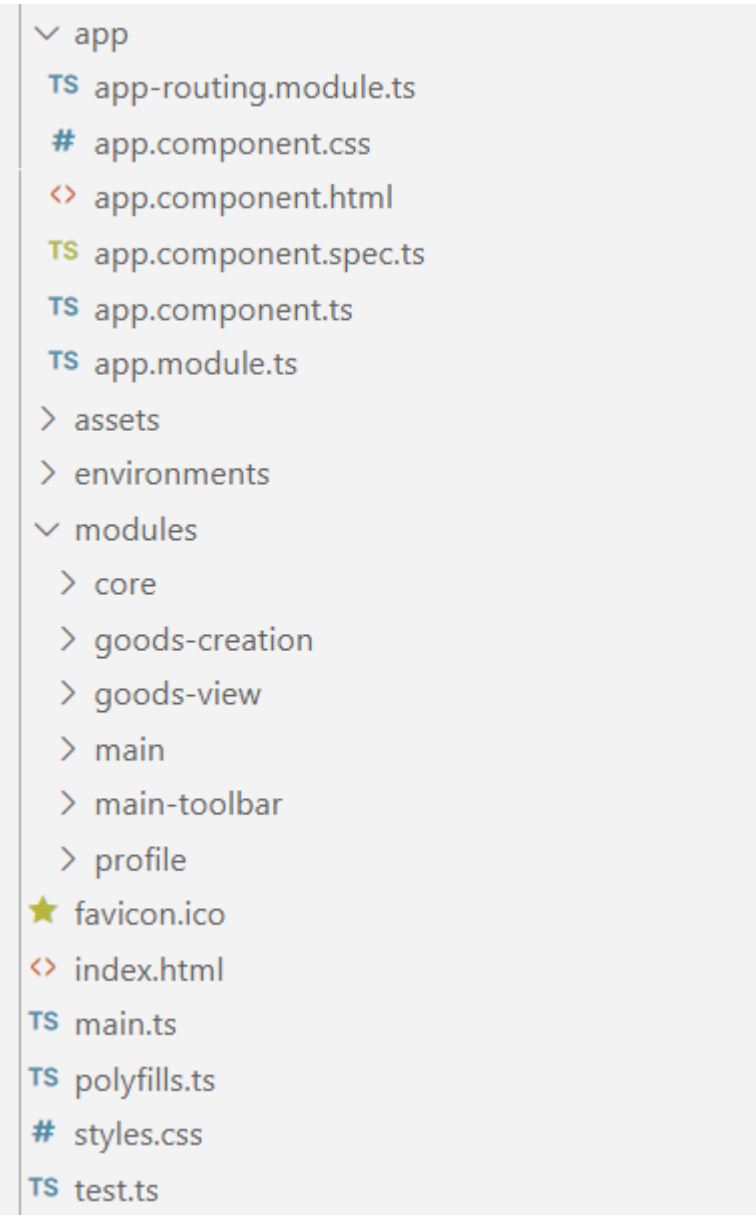

Рис. 11 Структура клієнтського застосування

Тут можна побачити 4 основні папки: app, assets, environments, modules – основний модуль додатку, ресурси додатку, оточення і усі модулі відповідно. Також є додаткові файли, серед них: іконка сайту, сторінка в якій знаходиться корінь додатку та файл глобальних стилів.

Кожний модуль схожий на модуль app, з деякими відмінностями. Кожний модуль містить в собі файл module.ts, де імпортуються зовнішні та внутрішні модулі для використання, також кожний модуль може містити в собі компоненти, сервіси та утиліти.

Кожен файл module.ts виглядає так:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    CommonModule,
    BrowserModule,
    HttpClientModule,
    MainModule,
    AppRoutingModule,
    MainToolbarModule,
    GoodsViewModule,
    GoodsCreationModule,
    ProfileModule,
    NgbModule,
  ],
  providers: [
    {provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor, multi: true},
    CreateGuard, ValidationGuard, SettingsGuard
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Рис. 12 Вигляд основного файлу модулів

Тут є об’єкт в якому конфігуруються декларації компонентів, імпорти інших модулів, провайдери, кореневий компонент для створення для вставки в хостову сторінку.

Усі одиниці структури Angular додатку можуть бути легко згенерованими за допомогою команди ng generate {структура} {назва}, з встановленим Angular[10]. Після виклику даної команди генерується папка за назвою з файлом модуля, якщо генерується модуль, а якщо розробник генерує, наприклад, компонент, то отримується папка з назвою компоненту та файлами component.ts, component.css та component.html.

В додатку маркетплейс при розробці був створений основний core модуль, який відповідає за сервіси з логікою, моделі, утиліти та основні імпорти зовнішніх модулів, наприклад FormsModule.

Також всередині проекту знаходяться інші модулі, в яких зберігаються компоненти для відображення, в ці модулі також імпортується core модуль.

Розділ 3.2.3 Компоненти додатку

Як описано вище, кожен додаток складається з модулів і компонентів всередині. Компонент – структурна одиниця Angular, яка використовується для відображення різноманітної інформації. [10]

Будь-який елемент додатку розбивається на модулі. Наприклад, в додатку маркетплейс є модуль для навігаційної панелі, для відображення товарів, для налаштувань користувача – усі ці модулі містять або використовують ще менші модулі, наприклад, для фільтрації товарів або компонент модального вікна для авторизації.

Всередині компоненту усі файли пов’язані, ts з html і з css. Через TypeScript файл прописується зв’язка з різноманітними вхідними даними, які вводяться через інтерфейс користувача визначений у html.

Для прикладу візьмемо налаштування користувача у проектному додатку. У налаштуваннях контактів користувач повинен додавати свої різноманітні контакти, чи то контактний телефон чи то профіль телеграм. У коді TypeScript це виглядає так:

```
public contacts: Contact[];
```

Рис. 13 Масив контактів

Це масив контактів, які заповнені через інтерфейс. Сама модель виглядає так:

```
export interface Contact{
    id: number;
    contact: string;
    user: string;
    contact_type: ContactType;
}
```

Рис. 14 Модель контактів

Ця модель використовується для приймання та надсилання запитів на сервер у сервісах та використана як модель прив’язки у компонентах.

Виведення створених контактів виглядає так:

```
<div *ngFor="let contact of contacts; let i = index">
    <div class="p-2 form-row">
        <div class="form-group input-group col-md-6 col-sm-6">
            <button *ngIf="!canBeRemoved(contact)" class="input-group-prepend btn btn-danger mr-1" (click)="removeContact(i)">
            </button>
            <select [disabled]="contact.id" [(ngModel)]="contact.contact_type" class="form-control">
                <option [ngValue]="1">Telegram</option>
                <option [ngValue]="2">Телефон</option>
             </select>
        </div>
        <div class="form-group col-md-6 col-sm-6">
            <input [(ngModel)]="contact.contact" type="text" class="form-control" placeholder="Дані">
        </div>
    </div>
</div>
```

Рис. 15 Частина HTML коду компоненту налаштування контактів

Тут за допомогою атрибута [(ngModel)] ми під’єднуємо у циклі *ngFor вивід елементів у html тегах[10]. Через select обирається тип контакту і у поле поряд вводиться власне контактні дані.

Відображення цього у інтерфейсі користувача виглядає так:

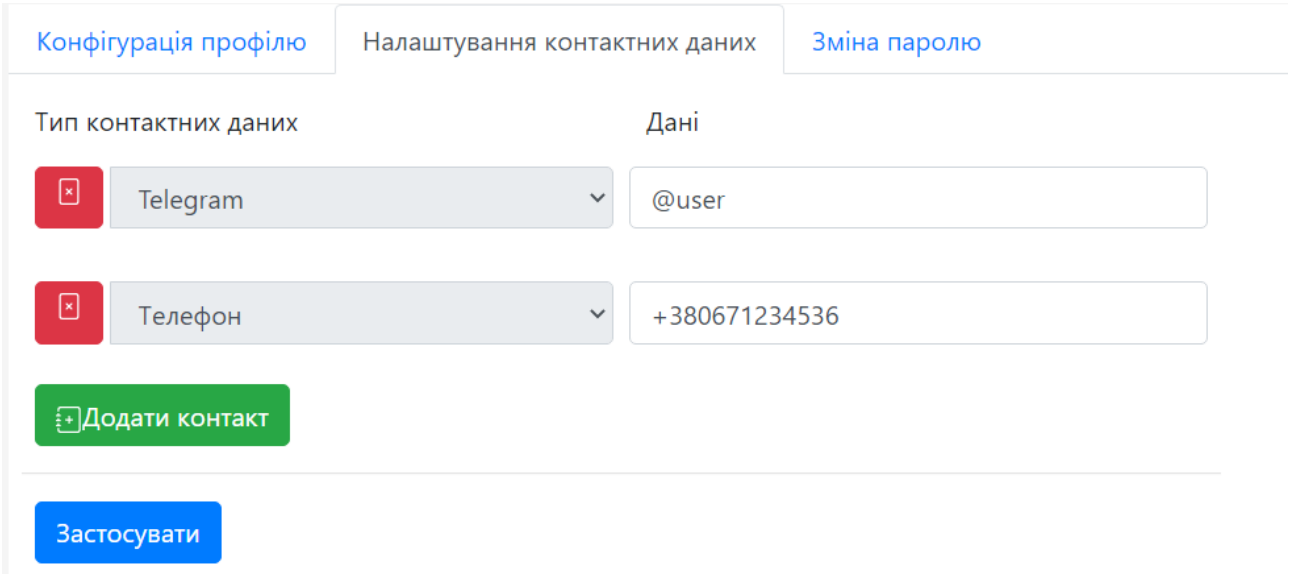

Рис. 16 Відображення налаштувань контактних даних

На цьому інтерфейсі можна побачити і кнопки для різних дій, вони з’єднані також через html атрибути (click) у цьому компоненті таким чином:

```
<button class="btn btn-success ml-2" (click)="addContact()">···
  </svg>Додати контакт</button>
<hr/>
<button class="ml-2 btn btn-primary" (click)="save()">Застосувати</button>
```

Рис. 17 Частина HTML коду кнопок налаштування контактних даних

У файлі з кодом є методи, які вказані у цих атрибутах:

```
public addContact(): void{
  if(this.contacts.length <=4){
    this.contacts.push({contact: "", contact_type: 1, id: null, user: null});
  }

}

public save(): void{
  this.subscriptions.add(this.usersService.saveContacts(this.contacts).subscribe(
    () => {
      this.alert = {type: 'success', message: "Налаштування застосовані!" }
      this.init();
    },
    () => {
      this.alert = {type: 'danger', message: "Невірно введений старий пароль" }
    }
  ));
}
```

Рис. 18 Методи компоненту налаштувань контактних даних

Тут перший метод відповідає за додавання нового контакту: створюється новий пустий об’єкт контакту та додається у масив про який мова йшла вище, і Angular автоматично додає елемент, який з’являється у цьому масиві до інтерфейсу користувача.

Другий метод застосовує налаштовані контакти за допомогою ін’єктованого сервісу користувачів.

Розділ 3.2.4 Сервіси

Уся бізнес-логіка застосувань, що стосується клієнтської частини у Angular, знаходиться у сервісах. В них саме і завантажуються та відправляються запити до створеного серверу.[10]

Показаний у минулому розділі метод для збереження налаштувань контактних даних використовував сервіс користувачів. Розглянемо цей метод збереження у обраному сервісі saveContacts:

```
public saveContacts(contacts: Contact[]): Observable<any>{

  return this.http.post<any>(this.url+'-contacts/', JSON.stringify(
    {
      user: this.authenticationService.currentUserValue.user_id,
      contacts
    }
  ), this.httpOptions).pipe(
    map(data => {
      console.log(data);
      return data;
    })
  );
}
```

Рис. 19 Метод сервісу для надсилання POST запиту

.

Тут повертається Observable з відповіддю від сервера, який викликається http методом POST як видно з прикладу. У цей метод подається посилання ресурсу, за яким змінюються контакти користувача, у тіло метода подається наш об’єкт контактів, який надходить з компоненту налаштувань і з сервісу аутентифікації ми отримуємо унікальний ідентифікатор користувача, в якого і змінюються контакти. Далі через ріре виводимо у консоль отриману від сервера відповідь, також, якщо треба, на неї налаштовують підписку у компоненті і виводять відповідні повідомлення користувачу про те, що збереження вдале або невдале.

Такі методи – основа сервісів, в них оброблюються усі можливі запити до сервера, отримуються дані і передаються до відображення у компоненти.

Взагалі сервіси можуть не тільки обробляти якісь запити а й мати якусь власну логіку потрібну для можливого використання. Але у випадку реалізації простого маркетплейсу вони не знадобилися.

Розділ 3.2.5 Утиліти

У Angular існують різні структурні одиниці утиліт, які допомагають розробнику в різних реалізаціях. Основні утиліти використані у додатку – це interceptor та guard.[10]

Interceptor'и потрібні для того, щоб робити якісь дії перед відправленням запиту. [10] У цій роботі він використаний як автоматизація додавання токену авторизації до headers запиту. Розглянемо його детальніше:

```
@Injectable()
export class JwtInterceptor implements HttpInterceptor {
    constructor(private authenticationService: AuthenticationService) {
    }

    intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        const currentUser = this.authenticationService.currentUserValue;

        if(currentUser)
        request = request.clone({
            setHeaders: {
                Authorization: `Bearer ${localStorage.getItem('userData')}`
            }
        });

        return next.handle(request);
    }
}
```

Рис. 20 JwtInterceptor

Тут ми використовуємо сервіс аутентифікації для того, щоб отримати збережений токен у сховищі та додати його у header Authorization з початком Bearer. Цей токен використовується для різноманітних доступів описаних у сервері.

Інша утиліта, guard, використовується для того, щоб заборонити деяким користувачам доступатись до певних даних[10]. Для прикладу візьмемо Validation guard, який забороняє звичайним користувачам отримати доступ до

можливостей адміністратора перевіряти товари. Сам guard – це так само імплементація одного метода:

```
@Injectable({
    providedIn: 'root'
})
export class ValidationGuard implements CanActivate {
    constructor(private authenticationService: AuthenticationService, private router: Router) {
    }
    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | UrlTree | Observable<boolean | UrlTree>
        const user = this.authenticationService.currentUserValue;

        if(user){
            const res = user.is_staff;

            if(!res)
            this.router.navigate(['/']);

            return res
        }else{
            this.router.navigate(['/']);
            return false;
        }

    }
}
```

Рис. 21 Validation Guard

Тут ми перевіряємо чи користувач є персоналом і може отримати доступ до сторінки на сайті. Якщо ж користувач не є членом персоналу, то його перенаправляє на головну сторінку і повертає у методі false, а навпаки, якщо користувач належить до групи персоналу, то повертає true і йому надається дозвіл перейти до бажаної сторінки.

Розділ 3.2.6 Підсумок по клієнту

У результаті розробки на платформі Angular з’ясувалося, що це дуже потужна платформа з великою купою можливостей і інструментів для розробника.

Сама розробка сильно відрізняється від розробки на чистому JavaScript з HTML та CSS. Логіка додатку інша і спосіб описання всієї взаємодії з одного боку складніша, але набагато зручніша, оскільки Angular надає розробнику усе потрібне та структурує застосунок певним чином.

Також ця платформа пропонує усі можливості одразу з початку розробки, які можна досить швидко опанувати, знаючи основи JavaScript.

Що ж до самої технології, то в період розробки були визначені основні переваги та недоліки для розробки.

Основні переваги, які були виявлені:

- Базована на компонентах архітектура, що покращує якість коду. Покращується повторна використовність, читабельність та можливість швидко змінювати код на краще.
- Використання TypeScript допомагає зробити код чистішим та зрозумілішим, додається можливість статичної типізації.
- Використання RxJS з усіма її можливостями ефективної асинхронності

Також присутні у Angular є один основний недолік: складність у вивчанні. Усі перераховані переваги становлять деяку складність для опанування. Спочатку треба зрозуміти структуру, вивчити можливості, особливо основу для асинхронності RxJS.

Розділ 3.3 Загальні висновки

У результаті цієї роботи було розроблене застосування, яке складається з серверної частини побудованої за допомогою фреймворку Django з використанням бази даних SQLite і клієнтської частини розроблене на платформі Angular.

В результаті розробки були покращені знання з даних технологій, помічені та проаналізовані етапи розробки таких систем з використанням можливостей, які надаються обраними технологіями і оцінки їх.

Під час розробки системи були виявлені та проаналізовані недоліки та переваги можливостей обраних технологій для створення веб-застосунку маркетплейс.

Django, який був використаний для розробки серверної частини виявився швидким, розширюваним та мав у собі одразу деякі корисні інструменти, як-от аутентифікація з панеллю адміністратора. З іншого боку, для початку розробки треба було опанувати деякі особливості розробки які потребує фреймворк і не одразу зрозумілі для розробника, який робив аналогічні застосування іншими мовами, наприклад, на Java.

Клієнтська частина на Angular також мала певні переваги та недоліки. Платформа надає багато можливостей для якісної розробки зі своєю структурою, шаблонами та асинхронністю. Але, при цьому, сама платформа може бути складною для опанування з її можливостями. Особливо, використання асинхронності RxJS потребує деякого часу для вивчення.

Список літератури

1. Укрінформ. В Україні кількість інтернет-користувачів зросла до 23 мільйонів. 10 жовтня 2019. [Електронний ресурс] – URL: https://www.ukrinform.ua/rubric-technology/2797152-v-ukraini-kilkist-internetkoristuvaciv-zrosla-do-23-miljoniv.html
2. .Редакція DOU. Ринок праці 2019: ріст 20% й ажіотаж навколо податків.12 грудня 2019. [Електронний ресурс] – URL: https://dou.ua/lenta/articles/jobs-and-trends-2019/
3. Снопок А. І. Дипломний проект «Багатопотоковий клієнт-серверний додаток». червень 2020. [Електронний ресурс] – URL: https://ela.kpi.ua/bitstream/123456789/34806/1/Snopok-A-I_bakalavr.pdf
4. Іван Змерзлий. Клієнт-серверна архітектура та ролі серверів. 1 лютого 2017. [Електронний ресурс] – URL: https://medium.com/@IvanZmerzlyi/клієнт-серверна-архітектура-та-ролі-серверів-9893d8048229
5. Wikipedia. Server(Computing). [Електронний ресурс] – URL: https://en.wikipedia.org/wiki/Server_(computing) (Дата звернення 04.04.2021)
6. MDN Web Docs. REST. . [Електронний ресурс] – URL: https://developer.mozilla.org/uk/docs/Glossary/REST (Дата звернення 04.04.2021)
7. Joshnunez. The client-serve relationship. 25 травня 2020. [Електронний ресурс] – URL: https://medium.com/@joshjnunez09/the-client-server-relationship-9ac90fadb3d2
8. Python Software Foundation. Python documentation. [Електронний ресурс] – URL: https://wiki.python.org/ (Дата звернення 04.04.2021)

9. Django Software Foundation. Django documentation. [Електронний ресурс] – URL: https://docs.djangoproject.com/en/3.1/
(Дата звернення 04.04.2021)
10. Google. Angular documentation. [Електронний ресурс] – URL: https://angular.io/docs
(Дата звернення 06.04.2021)
11. Користувач enartemy. Архитектура REST. 2 вересня 2008. [Електронний ресурс] – URL: https://habr.com/ru/post/38730/