МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інформаційна технологія проєктування соціальної мережі для підтримки соціально-культурних проектів молоді»

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

Керівник роботи **Д. В. Великодний**

Студент групи ІК.м-01 **П. О. Титов**

СУМИ 2021

Сумський державний університет

(назва вузу)

Факультет ЕлІТ Кафедра Комп'ютерних наук
Спеціальність «122 – Комп'ютерні науки»

Затверджую_____
Зав. кафедри Довбиш А. С.
«_____» _____20___ р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТУ

Титову Павлу Олеговичу
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Інформаційна технологія проєктування соціальної мережі для підтримки соціально-культурних проектів молоді затверджую наказом по університету від «____» _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проєкту (роботи) ___

3. Вхідні дані до проєкту (роботи)_____

_____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Огляд технологій розробки та існуючих рішень, що застосовуються для для підтримки соціально-культурних проєктів молоді. 2) Формування мети та постановки задачі, визначення етапів реалізації проєкту. 3) Вибір технологій та інструментів, що необхідні для розробки, огляд та аналіз баз даних. 4) Проєктування бази даних. 5) Моделювання системи. 6) Розроблення інформаційного та програмного забезпечення. 7) Тестування готового програмного продукту та аналіз результатів.

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

Аналіз предметної області застосування. Структура веб-додатка. Клієнтська частина веб-додатка. Серверна частина веб-додатка. База даних. Реалізований інтерфейс.

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

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання _____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Термін виконання проєкту (роботи)	Примітка
1.	Огляд технологій розробки та існуючих рішень	01.11–07.11	
2.	Формування мети та постановка задачі	08.11–18.11	
3.	Вибір технологій та інструментів для розробки	19.11–29.11	
4.	Проєктування бази даних та моделювання системи	18.11–24.11	
5.	Розробка системи	25.11–02.12	
6.	Тестування готового програмного продукту	03.12-17.12	
7.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи	08.12-12.12	

Студент–дипломник _____
(підпис)

Керівник проєкту _____
(підпис)

РЕФЕРАТ

Записка: 66 стор., 37 рис., 1 додаток, 22 джерел.

Об'єкт дослідження – методи проєктування та технології створення веб-додатків.

Мета роботи – розробка інформаційної технології проєктування соціальної мережі для підтримки соціально-культурних проектів молоді.

Методи дослідження – в процесі розробки веб-сервісу було використано мову розмітки HTML5, таблиці каскадних стилів CSS, мову програмування JavaScript, TypeScript, фреймворк Angular, NestJS, Bootstrap, базу даних MongoDB.

Результати – проведений аналіз предметної області застосування, описана актуальність задачі, визначені наявні проблеми, підготовлена структура веб-додатка для реалізації проєкту та сформульована мета. В результаті виконання кваліфікаційної магістерської дипломної роботи була розроблена інформаційна технологія проєктування соціальної мережі для підтримки соціально-культурних проєктів молоді з простим і зрозумілим у використанні інтерфейсом, головним призначенням якої є пошук людей для спільного проводження дозвілля. Також було проведено тестування роботи веб-сервісу.

ВЕБ-СЕРВІС, ДОЗВІЛЛЯ, JAVASCRIPT, ANGULAR,
TYPESCRIPT, NESTJS, BOOTSTRAP, MONGODB, CSS

Зміст

ВСТУП 6
1 ІНФОРМАЦІЙНИЙ ОГЛЯД 7
1.1 Аналіз предметної області застосування 7
1.2 Дослідження актуальності проблеми 11
1.3 Постановка задачі 13
2 МЕТОДИ РОЗРОБЛЕННЯ ВЕБ-ДОДАТКА 14
2.1 Середовище розробки 14
2.2 Проєктування структури веб-сервісу 27
3 ПРАКТИЧНА РЕАЛІЗАЦІЯ 31
3.1 Розроблення інтерфейсу 31
3.2 Розроблення бази даних 39
3.3 Тестування веб-додатка 42
3.4 Інструкція з використання веб-додатка 47
ВИСНОВКИ 49
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 50
ДОДАТОК А ЛІСТИНГ ПРОГРАМНОГО КОДУ 52

ВСТУП

Інтернет змінив наше повсякденне життя. Він зробив революцію в комунікаціях настільки, що тепер є нашим кращим засобом повсякденного спілкування.

Позитивне використання Інтернету робить наше життя легким і простим. Інтернет надає нам корисні дані, інформацію та знання для особистого, соціального та економічного розвитку, і від нас залежить аби продуктивно використовувати час у всесвітній мережі.

На сьогодні кожний має можливість проходити онлайн-курси та покращувати свої навички письма, спілкування чи ведення бізнесу. Покупки в Інтернеті, соціальні мережі, електронна пошта, спілкування в чаті – це звичайні речі, які ми робимо щодня.

Коли Інтернет почав розвиватися, люди почали більше спілкуватися та ділитися. Вони почали використовувати соціальні мережі та вести блоги.

Сьогодні соціальні мережі стали важливою частиною численного життя через незліченну кількість веб-сайтів і додатків, і люди мають можливість зв’язатися з іншими та поділитися своїми думками, досвідом, знаходити нові корисні знайомства та організовувати соціально-культурні проєкти тощо.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Аналіз предметної області застосування

У наші часи, щоб сприяти стійкому успіху, потрібно постійно залучати нових клієнтів для розвитку і не стояти на місці – необхідно заводити нові корисні знайомства, і один із найкращих способів зробити це – заявити про себе в Google. Причин для використання чи створення сайтів – безліч.

Призначення веб-сайта залежить від потреб власника веб-сайта.

Він може мати багатогранні цілі з різними завданнями веб-сайта, або він може мати одну єдину мету.

Ви можете показати потенційним клієнтам, що вони отримають, працюючи з вами, показуючи високоякісні фотографії на своєму веб-сайті чи просто надавати деяку інформацію кінцевому користувачеві. Вікіпедія є чудовим прикладом цього.

Веб-сайт – це сукупність багатьох веб-сторінок, а веб-сторінки – це цифрові файли, написані за допомогою HTML (мова гіпертекстової розмітки). Щоб ваш веб-сайт був доступним кожній людині у світі, він повинен цілодобово зберігатися або розміщуватися на комп’ютері, підключеному до Інтернету. Такі комп’ютери відомі як веб-сервер.

Компоненти веб-сайта:

− Веб-хост: хостинг - це місце, де веб-сайт фізично розташований. Група веб-сторінок (пов’язаних веб-сторінок), які можуть називатися веб-сайтом лише тоді, коли веб-сторінка розміщена на веб-сервері. Веб-сервер – це набір файлів, які передаються на комп’ютери користувачів, коли вони вказують адресу веб-сайта.

− Адреса: адреса веб-сайта також відома як URL-адреса веб-сайта. Коли користувач хоче відкрити веб-сайт, йому потрібно ввести адресу або URL-адресу веб-сайта у веб-браузер, і веб-сервер доставляє запитуваний веб-сайт.

− Домашня сторінка: домашня сторінка є дуже поширеною та важливою частиною веб-сторінки. Це перша веб-сторінка, яка з'являється, коли відвідувач відвідує веб-сайт. Домашня сторінка веб-сайта дуже важлива, оскільки вона визначає зовнішній вигляд веб-сайта та спрямовує глядачів на інші сторінки веб-сайта.

− Дизайн: це остаточний і загальний вигляд веб-сайта, який є результатом правильного використання та інтеграції елементів, таких як меню навігації, графіка, макет, навігаційні меню тощо.

− Вміст : всі веб-сторінки, що містяться на веб-сайті, разом складають його вміст. Хороший вміст на веб-сторінках робить веб-сайт більш ефективним та привабливим.

− Структура навігації: структура навігації веб-сайта – це порядок сторінок, набір посилань на щось. Зазвичай він утримується принаймні одним навігаційним меню.

Коли ми вводимо певну URL-адресу в рядок пошуку браузера, браузер запитує сторінку у веб-сервера, а веб-сервер повертає браузеру необхідну веб-сторінку та її вміст. Це відрізняє від того, як сервер повертає інформацію, необхідну у випадку статичних і динамічних веб-сайтів. Статичні і динамічні відрізняє те як веб-сайт повертає інформацію.

− Статичний веб-сайт: на статичних веб-сайтах сервер повертає веб-сторінки, які є попередньо створеними файлами вихідного коду, створеними за допомогою простих мов, таких як HTML, CSS або JavaScript. Немає обробки вмісту на сервері (за словами користувача) на статичних веб-сайтах. Веб-сторінки повертаються сервером без змін, тому статичні веб-сайти швидкі. Відсутня взаємодія з базами даних. Крім того, вони менш дорогі, оскільки хост не потребує підтримки обробки на стороні сервера різними мовами.

− Динамічний веб-сайт: у динамічних веб-сайтах веб-сторінки повертаються сервером, який обробляється під час виконання, означає, що вони не є попередньо створеними веб-сторінками, але вони створюються під

час виконання відповідно до вимог користувача за допомогою мов сценаріїв на стороні сервера, таких як PHP. , Node.js, ASP.NET та багато інших, які підтримує сервер. Таким чином, вони повільніші, ніж статичні веб-сайти, але можливі оновлення та взаємодія з базами даних. Динамічні веб-сайти використовуються замість статичних веб-сайтів, оскільки оновлення можна виконувати дуже легко в порівнянні зі статичними веб-сайтами (де потрібно вносити зміни на кожній сторінці), але на динамічних веб-сайтах можна зробити загальну зміну один раз, і це відобразиться у всіх веб-сторінках.

У всьому Інтернеті є різні типи веб-сайтів, коротке уявлення про найпоширеніші категорії наведено нижче.

− Блоги: цими типами веб-сайтів керує окрема особа або невелика група людей, вони можуть висвітлювати будь-які теми – вони можуть давати користувачу поради щодо моди, музичні поради, поради щодо подорожей, фітнесу. Сьогодні професійне ведення блогів стало зовнішнім популярним способом заробітку в Інтернеті.

− Електронна комерція: ці веб-сайти добре відомі як інтернет-магазини. Ці веб-сайти дозволяють нам здійснювати покупку продуктів і онлайн-платежі за продукти та послуги. Магазини можна обробляти як окремі веб-сайти.

− Портфоліо: такі типи веб-сайтів діють як розширення резюме фрілансера. Це надає потенційним клієнтам зручний спосіб перегляду роботи компанії, а також дозволяє їй розширити свої навички чи послуги.

− Брошура: ці типи веб-сайтів в основному використовуються малими підприємствами, ці типи веб-сайтів діють як цифрові візитні картки та використовуються для відображення контактної інформації та реклами послуг лише на кількох сторінках.

− Новини та журнали: ці веб-сайти потребують менше пояснень, головна мета цих типів веб-сайтів – інформувати своїх читачів про поточні події, тоді як журнали зосереджені на розвагах.

– Соціальні мережі: ми всі знаємо про деякі відомі веб-сайти соціальних мереж, як-от Facebook, Twitter, Reddit та багато інших. Ці веб-сайти зазвичай створюються для того, щоб люди могли ділитися своїми думками, зображеннями, відео та іншими корисними компонентами.

– Освітні: освітні веб-сайти досить прості для розуміння, оскільки це пояснює сама назва. Ці веб-сайти призначені для відображення інформації за допомогою аудіо, відео чи зображень.

– Портал: ці типи веб-сайтів використовуються для внутрішніх цілей у школі, інституті чи будь-якому підприємстві. Ці веб-сайти часто містять процес входу, що дозволяє студентам отримати доступ до їх облікової інформації або дозволяє співробітникам отримувати доступ до своїх електронних листів та сповіщень.

Отже, веб-сайт можна розглядати як цифрове середовище, здатне надавати інформацію та рішення та сприяти взаємодії між людьми, місцями та речами для підтримки цілей організації, для якої він був створений.

Схема класифікації веб-сторінок зображена на рис. 1.1.

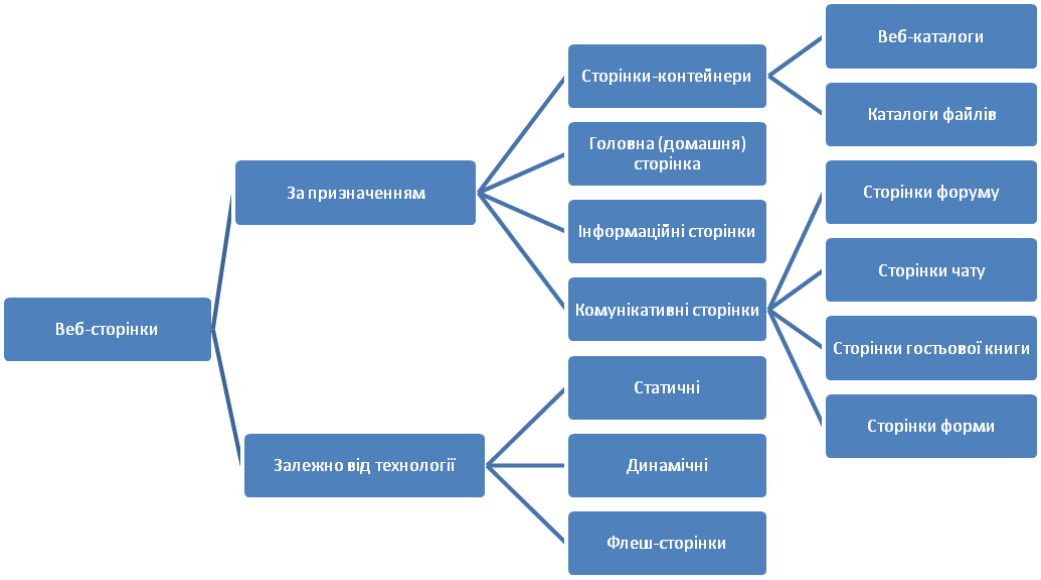

Рисунок 1.1 – Схема класифікації веб-сторінок [1]

Є гарна можливість не тільки створити професійний веб-сервіс, а ще й корисний сайт, який буде допомагати людям.

Такий веб-додаток зробити буває непросто. Важливим моментом є актуальність та цікавість контенту, який буде корисним багатьом користувачам. За наявністю такого контенту, який буде залучати інших людей для відвідування вашого веб-сайта, можна бути впевненими, що бренд є успішним, і лише думати про наступний спосіб зробити бренд більш відомим.

1.2 Дослідження актуальності проблеми

Люди за своєю природою є соціальними істотами. Спілкування майже так само важливе, як їжа та вода для нашого виживання. Протягом останніх років наш спосіб спілкування стрімко розвивався. Саме соціальні мережі дають людям можливість залишатися на зв’язку один з одним, де б вони не були. Не потрібно мати доступ до стаціонарного телефону, щоб зателефонувати комусь із цією технологією сьогодні. Відправлення листа традиційною поштою здається вічністю. Усі вже забули про автовідповідачі або голосову пошту – просто залиште коментар на чиїйсь сторінці, і адресат одразу отримає сповіщення про це.

Завдяки розвитку технологій, Інтернету, обміну миттєвими повідомленнями, смартфонам існує багато каналів і способів взаємодії з іншими. Проте, чи зміг наш еволюційний мозок адаптуватися та йти в ногу з цим наповненням можливостей спілкування, чи можуть ці віртуальні взаємодії замінити спілкування віч-на-віч з точки зору забезпечення задоволеності та покращення загального самопочуття?

Деякі дослідження показують, що використання соціальних медіа призводить до нехтування соціальною діяльністю, принесення в жертву реальних стосунків, самотності та інших подібних негативних наслідків. Негативні результати також виникають у повсякденних життєвих ситуаціях, наприклад роздратування, викликане, коли хтось зайнятий з мобільним телефоном за сімейним обіднім столом. Такі результати можна сприймати як

негативні, але не настільки згубні, як ослаблення існуючих соціальних зв’язків [2].

Хоча Інтернет відкрив нову сферу можливостей щодо зв’язку з людьми по всьому світу, у будь-який час існують невід’ємні фактори онлайн-спілкування, які обмежують його здатність сприяти тому ж рівні задоволення, що й традиційне спілкування віч-на-віч.

Загалом, спілкування віч-на-віч сприяє більш якісній взаємодії, ніж спілкування в Інтернеті. Онлайн-спілкування та соціальні мережі слід використовувати як доповнення до соціального життя. Однак воно не повинно бути невід’ємним або єдиним джерелом спілкування та взаємодії з іншими. Інтернет, технології та смартфони принесли з собою багато переваг. Вони підвищили гнучкість роботи в деяких сферах, дозволили людям залишатися на зв’язку на різних континентах, полегшили життя, об’єднавши всі необхідні інструменти для оплати рахунків, перевірки електронної пошти, спілкування з близькими тощо на одному пристрої. Однак, коли справа доходить до нашої людської потреби спілкуватися та спілкуватися з іншими, спілкування віч-на-віч все одно необхідне.

Щоб покращити наше самопочуття, нам потрібен здоровий баланс між нашим віртуальним і реальним світом. Незважаючи на те, що технології змогли зблизити спільноти та людей, ми повинні підтримувати ці зв’язки та плекати їх через старомодні особисті зустрічі. Для нас, як людського роду, життєво важливо мати можливість продовжувати спілкуватися з іншими, не ховаючись за ширмою [3].

Існує безліч платформ і мов, на яких може бути розроблений web-сервіс. У сучасному світі все актульнішою стає проблема пошуку знайомих та друзів. Іноді стає складно знайти собі компанію, наприклад, для катання на велосипедах чи відвідання екскурсії, похід в театр на цікаву виставу чи для організації благодійного заходу. І у випадку, коли людина вже загорілася тією чи іншою ідеєю для проведення часу – хочеться знайти відповідних людей

якомога швидше, і цілком всього за декілька хвилин можна зробити це за допомогою спеціального сервісу.

Тому в першу чергу веб-сервіс створений з метою стимуляції до реального спілкування: користувач матиме можливість знаходити цікаві події та людей в пошуку компанії, чи то цікавий проєкт, до реалізації якого можна буде приєднатися, чи то просто знайомитися з людьми, призначати зустрічі і спілкуватися з ними в реальності.

1.3 Постановка задачі

Мета кваліфікаційної магістерської роботи полягає в розробленні інформаційної технології проєктування соціальної мережі для підтримки соціально-культурних проектів молоді.

Після проведення аналітичного огляду області розробки і визначення актуальності веб-сервісу, були обґрунтовані та поставлені такі задачі:

1) вибір середовища розробки;

2) розробка структури соціальної мережі;

3) створення інтерфейсу;

4) розробка веб-сервісу;

5) після реалізації проєкту провести тестування роботи веб-додатка.

Web-сервіс повинен містити наступні функціональні можливості:

1) реєстрація користувачів;

2) налаштування облікових записів користувачів;

3) перегляд переліку варіантів зустрічей, заходів;

4) створення, бронювання зустрічі.

2 МЕТОДИ РОЗРОБЛЕННЯ ВЕБ-ДОДАТКА

2.1 Середовище розробки

Багато людей думають, що Інтернет і всесвітня мережа – це одне й те саме. Хоча вони тісно пов’язані між собою, це дуже різні системи.

Інтернет – це величезна мережа комп’ютерів, які всі разом з’єднані. Всесвітня мережа (скорочено «www» або «веб») – це сукупність веб-сторінок, знайдених у цій мережі комп’ютерів. Саме веб-сайти містять текстові сторінки, цифрові зображення, аудіо, відео тощо. Користувачі можуть отримати доступ до вмісту цих сайтів з будь-якої точки світу через Інтернет за допомогою своїх пристроїв, таких як комп’ютери, ноутбуки, мобільні телефони тощо. WWW в поєданні з Інтернетом, дозволяє отримувати та відображати текст і медіа на вашому пристрої [4].

Інтернет змінюється і сьогодні. Пошукові системи стали краще читати, розуміти та обробляти інформацію. Вони знайшли розумні способи знайти потрібний нам вміст і навіть можуть показати нам інші речі, які можуть нас зацікавити.

Існує три основні технології, які створені для того, аби всі комп’ютери розуміли один одного (HTML, URL і HTTP). Всі вони використовуються і сьогодні.

HTML (Hyper Text Markup Language) – формат публікації для Інтернету. Він включає в себе можливість форматування документів і посилання на інші документи та ресурси.

URL (Uniform Resource Locator) – URL-адреса є своєрідною «адресою», яка є унікальною для кожного ресурсу в Інтернеті. Це може бути адреса веб-сторінки або файл зображення.

HTTP (Hypertext Transfer Protocol) дозволяє запитувати та передавати документи HTML між браузерами та веб-серверами через Інтернет.

Основою для створення веб-сторінок і веб-додатків є технології HTML (мова розмітки гіпертексту) і CSS (каскадні таблиці стилів). HTML надає структуру сторінки, CSS – (візуальний і звуковий) макет для різних пристроїв.

Визначення HTML – це мова гіпертекстової розмітки.

Гіпертекст – це метод, за допомогою якого ви переміщуєтеся по Інтернету, натискаючи на спеціальний текст, який називається гіперпосиланнями, що переводить вас на наступну сторінку. Той факт, що це гіпер, просто означає, що він не є лінійним – тобто користувач може перейти в будь-яке місце в Інтернеті, натиснувши на посилання, тобто встановленого порядку для виконання завдань немає.

Розмітка – це те, що HTML-теги роблять з текстом всередині них. Вони позначають його як певний тип тексту (наприклад, курсив).

Кожен окремий код розмітки (який розташовується між символами "<" і ">") називається елементом, хоча люди звикли називати його тегом. Деякі елементи представлені парами, які вказують, коли якийсь ефект відображення має початися, а коли закінчиться.

Нижче наведено приклад HTML, який використовується для визначення базової веб-сторінки з заголовком і одним абзацом тексту [5].

```
<!doctype html>
<html>
<head>
<title>TechTerms.com</title>
</head>
<body>
<p>This is an example of a paragraph in HTML.</p>
</body>
</html>
```

Перший рядок визначає, який тип вмісту містить документ. "<!doctype html>" означає, що сторінка написана в HTML5. Правильно відформатовані сторінки HTML повинні включати теги <html>, <head> і <body>, які включені в наведений вище приклад. Заголовок сторінки, метадані та посилання на

посилання на файли розміщуються між тегами <head>. Фактичний вміст сторінки розміщується між тегами <body>.

Тобто HTML є мовою розмітки, що використовується веб-браузерами для інтерпретації та представлення тексту, зображень, відео, аудіо та інших носіїв інформації для людей на веб-сторінках. HTML визначає структуру веб-сайту, у той час як інші технології визначають його зовнішній вигляд (CSS) і поведінку (JavaScript).

CSS - це мова для опису представлення веб-сторінок, включаючи кольори, макет та шрифти. Це дозволяє адаптувати презентацію до різних типів пристроїв, таких як великі екрани, маленькі екрани або принтери. CSS не залежить від HTML і може використовуватися з будь-якою мовою розмітки на основі XML [6].

CSS дозволяє внести ряд нововведень у верстку веб-сторінок, наприклад:

- вказувати шрифти, відмінні від стандартних для браузера;
- задавати колір та розмір тексту та посилань;
- застосовувати кольори до фону;
- укладати елементи веб-сторінки в рамки та переміщувати ці рамки у певні позиції на сторінці.

Веб-сторінка – це серія блоків, кожна з яких містить дискретний елемент. Ці коробки вкладені один в інший.

Наприклад, заголовок сторінки – це поле, і воно містить кілька менших блоків, які містять усі елементи, які утворюють заголовок: логотип, навігацію, кнопки соціальних мереж, кнопки кошика для покупок тощо. Використовуючи CSS, розробник призначає стилі до «заголовок». У цьому прикладі припустимо, що розробник робить текст всередині заголовка фіолетовим, шрифт Arial і п’ятнадцять пунктів.

Ось де в гру вступає «каскадна» частина каскадних таблиць стилів. Стилі шрифтів, застосовані до заголовка, каскадно опускаються до всіх елементів,

що містяться всередині заголовка. Усі елементи, що містять текст, як-от навігація, посилання або заклики до дії, мають фіолетовий колір, шрифт Arial і п’ятнадцять пунктів.

Для покращення HTML-сторінок використовується мова сценаріїв Javascript (JS) і часто зустрічається вбудованим у HTML-код. JavaScript є мовою, що інтерпретується. Таким чином, його не потрібно компілювати. JavaScript відображає веб-сторінки в інтерактивному та динамічному режимі. Це дозволяє сторінкам реагувати на події, демонструвати спеціальні ефекти, приймати змінний текст, перевіряти дані, створювати cookies, визначати браузер користувача тощо.

HTML-сторінки добре підходять для відображення статичного вмісту, наприклад, простого зображення або тексту. Однак більшість сторінок у наш час рідко статичні. Багато сучасних сторінок містять меню, форми, слайд-шоу і навіть зображення, які забезпечують взаємодію з користувачем. Javascript – це мова, яку використовують веб-розробники для забезпечення такої взаємодії. Оскільки JavaScript працює з HTML-сторінками, розробнику необхідно знати HTML, щоб використати весь потенціал цієї мови сценаріїв. Хоча існують інші мови, які можуть бути використані для створення сценаріїв в Інтернеті, практично всі вони в основному використовують Javascript.

Існує два способи використання JavaScript у HTML-файлі. Перший передбачає вбудовування всього коду JavaScript в HTML-код, а другий – використання окремого файлу JavaScript, що викликається з елемента Script, тобто ув’язнений у теги Script. Файли JavaScript позначаються як розширення .js. Хоча JavaScript в основному використовується для взаємодії з об'єктами HTML та бізнес логіки клієнтської частини, його можна використовувати і для взаємодії з іншими не-HTML об’єктами, такими як плагіни браузера, властивості CSS (Cascading Style Sheets), поточна дата або браузер. Для написання коду JavaScript вам знадобиться лише базовий текстовий редактор, такий як Notepad у Windows, Gimp у Linux або BBEdit. Деякі текстові

редактори, наприклад BBEdit, підтримують підсвічування синтаксису JavaScript. Це дозволить вам легко ідентифікувати елементи JavaScript. Останні версії Internet Explorer, Firefox та Opera підтримують JavaScript.

JavaScript має такі властивості [7]:

- JavaScript був створений насамперед для маніпуляцій із DOM. Раніше веб-сайти були переважно статичними, після створення JS з'явилися динамічні веб-сайти.
- Функції в JS – це об'єкти. Вони можуть мати властивості та методи, як та інші об'єкти. Їх можна передавати як аргументи в інші функції.
- Може працювати з датою та часом.
- Перевіряє форми, хоча форми створюються за допомогою HTML.
- Компілятор не потрібний.

Сильно типізовані мови меншою мірою піддаються багам.

Саме TypeScript додає таку функцію до JavaScript – сильну типізацію (вона ввімкнена за замовчуванням, але її можна вимкнути, якщо розробник не хоче її використовувати). Сильно типізована мова не дозволить такі операції, як множення числа на масив і не виконає неявного перетворення об'єкта до рядка [8].

TypeScript – це проєкт з відкритим вихідним кодом, вихідний код якого доступний на GitHub.

TypeScript долає найбільшу проблему з JavaScript, а саме: проблему можна виявити лише під час виконання в JavaScript, що може призвести до того, що програми з помилками надсилаються кінцевому користувачеві. Це негативно вплине на будь-який бізнес, заважаючи роботі користувачів. TypeScript усуває цю проблему, перевіряючи будь-яку проблему під час компіляції [9].

Існує багато відмінностей між Typescript і Javascript. Ось лише деякі з них [10]:

− TypesScript — це об'єктно-орієнтована мова програмування, тоді як JavaScript — мова сценаріїв (з підтримкою об'єктно-орієнтованого програмування).

− TypeScript має статичну типізацію, тоді як JavaScript ні.

− TypeScript використовує типи та інтерфейси для опису використання даних.

− TypeScript має інтерфейси, які є потужним способом визначення контрактів у вашому коді.

− TypeScript підтримує додаткові параметри для функцій, яких не підтримує JavaScript.

Typescript полегшує використання фреймворків та добре з ними інтегрується. Використання Typescript не є обов’язковим фреймворків, але це може підвищити продуктивність і легкість.

Angular – сучасна структура, повністю побудована на TypeScript, і, як наслідок, використання TypeScript з Angular забезпечує безперебійну роботу [11].

Angular розроблено, щоб зробити оновлення максимально простим, тому скористайтеся перевагами останніх розробок з мінімумом зусиль. Найкраще те, що екосистема Angular складається з різноманітної групи з понад 1,7 мільйонів розробників, авторів бібліотек і творців контенту.

Angular framework заснований на компонентах, що починаються в єдиному стилі. Наприклад, кожен компонент розміщує код у класі компонента або визначає декоратор @Component (метадані). Ці компоненти є невеликими елементами інтерфейсу, незалежними один від одного, і пропонують вам кілька переваг, у тому числі:

− Можливість повторного використання: Заснована на компонентах структура Angular робить компоненти дуже багаторазовими в усьому додатку. Ви можете створювати інтерфейс (UI) з рухомими частинами, забезпечуючи при цьому плавний процес розробки для розробників.

− Спрощене модульне тестування. Будучи незалежними один від одного, компоненти значно спрощують модульне тестування.

− Покращена читабельність: Послідовність кодування робить читання коду простою справою для нових розробників у поточному проекті. Це підвищує їхню продуктивність і загальну ефективність проекту.

− Простота обслуговування: розділені компоненти можна замінювати більш досконалими реалізаціями. Простіше кажучи, це дозволяє ефективно обслуговувати та оновлювати код.

− Крім того, тестування в Angular є надзвичайно просто. Модулі Angular.js мають частини програми, якими легко маніпулювати. Завдяки розділенню модулів ви можете завантажувати необхідні сервіси, ефективно виконуючи автоматичне тестування. Вам навіть не потрібно запам'ятовувати порядок завантаження модулів, якщо ви дотримуєтеся принципу «один файл – один модуль».

Для взаємодії з комп’ютером, додатками та веб-сайтами необхідні компоненти інтерфейсу користувача - екран з меню і графічними елементами, клавіатура і миша.

Програмне забезпечення або його елементи не потребують графічного інтерфейсу для взаємодії один з одним. Програмні продукти обмінюються даними та функціональними можливостями через інтерфейси API – інтерфейси прикладного програмування.

API – це набір програмного коду, який забезпечує передачу даних між одним програмним продуктом та іншим. Він також містить умови для цього обміну даними [12].

Інтерфейси прикладного програмування складаються з двох компонентів:

− Технічна специфікація, що описує можливості обміну даними між рішеннями, причому специфікація виконується у вигляді запиту на обробку та протоколів доставки даних.

- Програмний інтерфейс, написаний відповідно до специфікації, що його представляє.

Програмне забезпечення, якому необхідно отримати доступ до інформації (наприклад, Х цін на готельні номери на певні дати) або функціональності (наприклад, маршрут з точки А до точки В на карті на основі розташування користувача) від іншого програмного забезпечення, викликає його API, вказуючи вимоги до того, як дані/функціональність мають бути надані. Інше програмне забезпечення повертає дані/функціональність, запрошені першим додатком.

А інтерфейс, за допомогою якого ці дві програми спілкуються, визначає API.

Кожен API містить та реалізується за допомогою викликів функцій – мовних виразів, які запитують у програмного забезпечення виконання певних дій та послуг. Виклики функцій – це фрази, що складаються з дієслів та іменників, наприклад:

- Почати або завершити сеанс.
- Отримати інформацію про вигоди для одного типу номера.
- Відновити або отримати об'єкт із сервера.
- Виклики функцій описані в документації API.

API служать численним цілям. Загалом, вони можуть спростити та прискорити розробку програмного забезпечення. Розробники можуть додавати функціональність (наприклад, механізм рекомендацій, бронювання житла, розпізнавання зображень, обробка платежів) від інших постачальників до існуючих рішень або створювати нові програми за допомогою послуг сторонніх постачальників. У всіх цих випадках фахівцям не потрібно працювати з вихідним кодом, намагаючись зрозуміти, як інше рішення. Вони просто підключають своє програмне забезпечення до іншого. Іншими словами, API є шаром абстракції між двома системами, приховуючи складність і деталі роботи останньої (рис. 2.1).

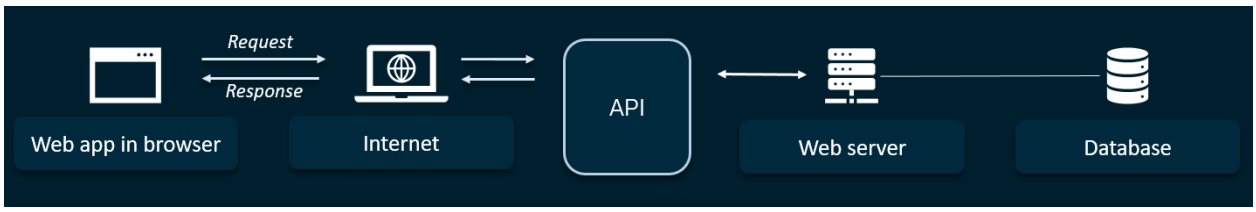

Рисунок 2.1 – Схема роботи API

Для виконання магістерської дипломної роботи було обрано архітектурний стиль інтерфейсу RESTful API.

У REST API є чотири методи HTTP, які використовують для дій з об’єктами на серверах [13]:

- GET (отримання інформації про дані або список об'єктів);
- DELETE (видалення даних);
- POST (додавання або заміна даних);
- PUT (регулярне оновлення даних).

Angular – це фреймворк для створення мобільних і настільних веб-додатків. Подібно до React, його метою є створення інтерактивних інтерфейсів користувача та односторінкових програм.

Завдяки стандартній функціональності, вбудованим функціям безпеки та архітектурі на основі компонентів, Angular є ідеальною платформою для створення програм корпоративного рівня.

На відміну від інших інструментів, таких як React і Vue, які потребують сторонніх бібліотек для створення базової функціональності, Angular є повноцінним фреймворком. Angular дозволяє впорядкувати розробку проекту, оскільки всі необхідні елементи є на місці, щоб ви могли негайно почати роботу.

Проєкти на основі Angular складаються з компонентів багаторазового використання, незалежних один від одного. Архітектура на основі компонентів у Angular допомагає вам заощадити багато часу на розробку, полегшує модифікацію та підтримку вашої програми, а також забезпечує узгодженість ваших великих програмних програм.

Angular використовується на багатьох платформах. Ми можемо допомогти вам розробити програму Angular, яка працює в Інтернеті, на мобільному пристрої (за допомогою Cordova, Ionic або NativeScript), на комп'ютері (за допомогою Electron). І ми можемо використовувати Angular для розробки PWA, рішення, яке забезпечує користувацький досвід у браузері, схожий на додаток.

Angular написаний на TypeScript, який підтримує статичні типи даних. Статична типізація працює як Grammarly – вона дозволяє веб-розробникам Angular виявляти помилки раніше в процесі розробки, заощаджуючи час на виправлення помилок. Angular також полегшує модульне тестування. Завдяки чистому коду ваша програма працює краще [14].

Angular створено Google і підтримується Google, що означає, що фреймворк залишиться на довгі роки. Angular також створив величезну спільноту розробників, які сприяють удосконаленню цієї технології, додаючи нові бібліотеки. Завжди корисно створити програму з використанням надійних технологій [15].

Програма створена за допомогою Angular, буде безпечною, легкою в обслуговуванні, стабільною по всій базі коду та легко масштабованою.

Для створення форм, кнопок, навігації, спадного списку, модулів, макета та багатьох інших речей без зайвих зусиль використовуюсь Bootstrap.

Bootstrap — це найпопулярніший, безкоштовний фреймворк з відкритим вихідним кодом для створення адаптивного макета на веб-сторінках із набагато меншими зусиллями.

До Bootstrap таким веб-дизайнерам, як ви, доводилося працювати над CSS Media Queries, щоб створити адаптивний веб-дизайн. Bootstrap спростив це, подбавши про медіа-запити самостійно. Це зняло складність і зробило це швидким і легким.

Перша версія Bootstrap була випущена 19 серпня 2011 року командою розробників Twitter [16]. Основною ідеєю було заохочувати послідовність під

час веб-розробки проектів. До цього часу цієї послідовності не було, оскільки під час веб-розробки використовувалися різні бібліотеки. Це також призводило до високих витрат на технічне обслуговування та навантаження.

Bootstrap вирішив цю проблему невідповідності і миттєво став хітом. Сьогодні його використовують мільйони веб-сайтів.

Веб-дизайнери та розробники люблять використовувати Bootstrap у своїх проектах. Вони використовують його для створення адаптивного веб-дизайну, який ідеально виглядає на всіх розмірах екрана (смартфони, планшети, ноутбуки та ПК).

Переваги, які надає Bootstrap:

− Економить ваш час – Швидко створюйте функції за допомогою попередньо визначених класів і шаблонів дизайну, які надаються Bootstrap.

− Адаптивний дизайн – із Bootstrap вам не потрібно застосовувати медіа-запити у файлі CSS. Він динамічно налаштовує веб-сторінки для всіх розмірів екрана.

− Сумісний з усіма браузерами – Вам не доведеться турбуватися про будь-який браузер, оскільки він сумісний з останніми версіями всіх браузерів – Google Chrome, Firefox, Opera, Safari та Edge.

− Легко та просто – це дуже легко та просто використовувати у веб-дизайні. Якщо у вас є базові знання HTML і CSS, вам слід продовжити.

− Послідовність – це дає вам узгодженість між вашими проектами та іншими розробниками.

− Безкоштовний і відкритий вихідний код – немає жодних обмежень. Bootstrap доступний на GitHub, де розробники можуть внести свої зусилля.

Для створення ефективних і масштабованих додатків на стороні сервера з TypeScript використовують NestJS - фреймворк Node.js. Він побудований на чіткому дизайні з кількома простими компонентами (контролерами, модулями та провайдерами). Це робить поділ програм на мікросервіси легким.

Основні компоненти Nest JS [17]:

− Контролери несуть відповідальність за обробку вхідних запитів і відповідь на стороні клієнта програми.

− Модулі використовуються для структурування коду та розділення функціональних можливостей на логічні фрагменти, які можна повторно використовувати.

− Постачальники або послуги, які абстрагують складність і логіку від користувача. Можна впровадити службу в контролери або інші служби.

Кілька чудових функцій NestJS:

− Nest.js був створений, щоб допомогти розробникам створювати моноліти та мікросервіси.

− Він простий у використанні, швидкий в освоєнні та легкий у застосуванні

− Він використовує TypeScript — строго типізовану мову, яка є наднабором JavaScript

− Потужний інтерфейс командного рядка (CLI) для підвищення продуктивності та простоти розробки

− Підтримка десятків специфічних для гнізд модулів, які допомагають легко інтегруватися із звичайними технологіями та концепціями, такими як Type ORM, Mongoose, GraphQL, Logging, Validation, Caching, WebSockets та багато іншого

− Прості програми модульного тестування

− Чудова документація.

− Створено для великомасштабних корпоративних додатків [18].

Розробники, які використовують NestJS, мають перевагу в конкурентній боротьбі. Вам просто потрібно створити нову програму, щоб почати, а потім ви готові почати розробку. Ця структура дає вам значний старт, а також допомагає вам розвивати програму, правильно її організувавши.

Для підвищення надійності важливих даних було обрано сховище документів MongoDB.

MongoDB — це нереляційна база даних документів, яка має гнучку модель даних, що дозволяє зберігати неструктуровані дані, а також забезпечує повну підтримку індексації та реплікації за допомогою багатих та інтуїтивно зрозумілих API.

Реляційні бази даних зберігають інформацію в строго регламентованих таблицях і стовпцях. MongoDB — це сховище документів, яке зберігає інформацію в колекціях і документах. Основна відмінність тут полягає в тому, що колекції та документи є неструктурованими, їх іноді називають безсхемними. Це означає, що структура екземпляра MongoDB (колекції та документи) не визначена заздалегідь і згинається, щоб вмістити будь-які дані, які в ньому розміщені (рис. 2.2).

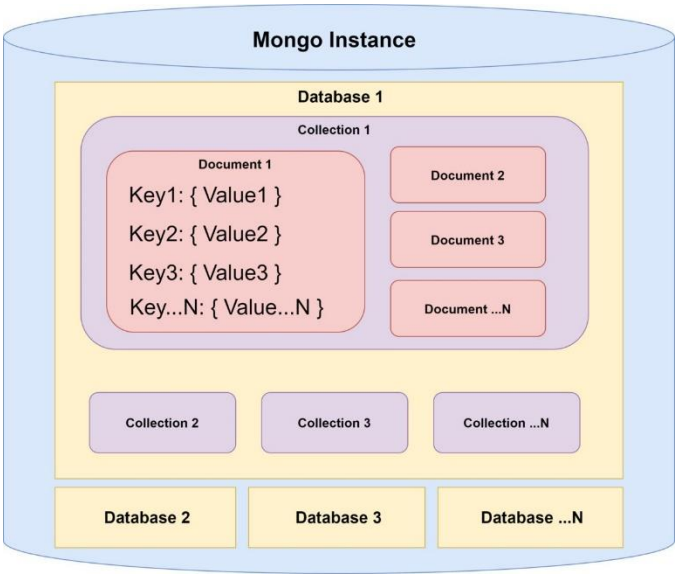

Рисунок 2.2 – Сховище документів MongoDB [19]

Гнучкість, успадкована в цьому типі моделювання даних, означає, що дані можна обробляти за принципом використання і потреби, забезпечуючи переваги продуктивності, як описано тут.

Щоб отримати конкретне розуміння цієї різниці, порівняйте два наступні способи досягнення одного і того ж завдання (створення запису, а потім додавання поля з програми), спочатку в реляційній базі даних, а потім у MongoDB [20].

2.2 Проєктування структури веб-сервісу

Структура сайта – це різні сторінки сайта, що пов’язані одна з одною за допомогою внутрішніх посилань та їхньої ієрархії. Це те, як інформація на сайті організована та представлена, аби алгоритм добре читав її контекст. Хороша структура веб-сайта полегшує навігацію як для користувачів, так і для сканерів, що покращує рейтинг SEO веб-сайта в пошукових системах.

Для доступності і простоті у використанні необхідно створити веб-сайт із чудовим UX («user experience»), адже чудова структура веб-сайта покращує зручність використання веб-сайта, полегшуючи користувачам пошук того, що вони шукають. Щоб створити структуру веб-сайта, потрібно намітити, як буде організований вміст на сайті (головна сторінка, категорії, окрема сторінка, дописи в блозі).

Ось чому структурування веб-сайта має бути першим кроком у будь-якому проєкті веб-дизайну.

Більшість веб-сайтів у верхній частині кожної сторінки містять інформацію про весь сайт, таку як логотип веб-сайта, функція пошуку та параметри навігації [21]. HTML5 містить елемент <header>, який можна використовувати для визначення такої області (рис. 2.3).

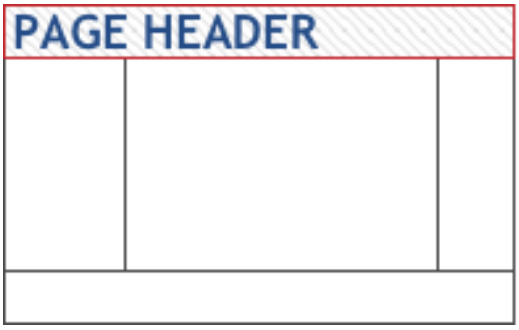

Рисунок 2.3 – Блок заголовка (header)

Подібно до заголовка сторінки, більшість веб-сайтів також мають область внизу кожної сторінки, яка містить інформацію для всього сайта, таку як інформація про авторські права, заяви про конфіденційність або

застереження. HTML5 містить елемент <footer>, який можна використовувати для визначення такої області (рис. 2.4).

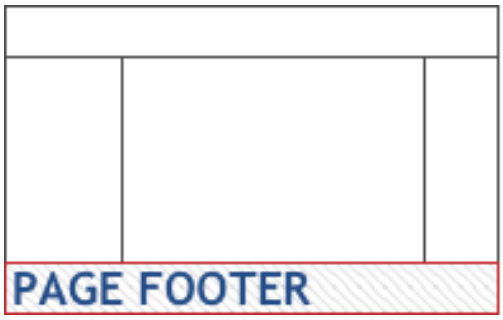

Рисунок 2.4 – Підвал (footer) сайта

Веб-сторінка може мати будь-яку кількість навігаційних меню (рис. 2.5).

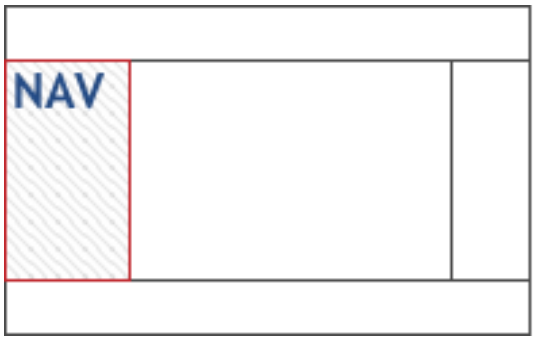

Рисунок 2.5 – Навігаційне меню

Для визначення області основного вмісту веб-сторінки або програми використовують елемент HTML5 <main> (рис. 2.6).

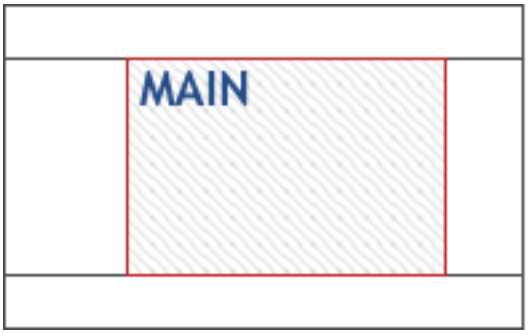

Рисунок 2.6 – Інформаційний блок

Зовнішня структура сайта показана на рис. 2.7.

Рисунок 2.7 – Приклад схеми зовнішньої структури сайта

Основним принципом відмінної структури веб-сайту є інформаційна архітектура (ІА). ІА гарантує, що вміст організовано, структуровано та позначено ефективно та послідовно. Щоб створити найкращу інформаційну архітектуру для свого веб-сайту, потрібно враховувати наступні фактори:

− Шлях користувача: оскільки веб-сайти створюються для обслуговування користувачів, важливо враховувати, як вони можуть взаємодіяти з ним, а також їхні очікування щодо того, як він повинен працювати. Ви можете визначити шлях ваших користувачів, опитуючи їх або виконуючи вправу на сортування карток.

− Вміст: структура вашого веб-сайту також значною мірою визначатиметься типом та обсягом вмісту на вашому сайті. Структура сайту електронної комерції буде відрізнятися від структури академічного сайту.

− Контекст: Контекст веб-сайту визначається його бізнес-цілями, культурним контекстом, в якому він існує, і доступними ресурсами. Важливо враховувати цей факт, структуруючи свій веб-сайт.

Найпоширенішою структурою веб-сайта є ієрархічна структура, яка була обрана для реалізації магістерської дипломної роботи (рис. 2.8), і котра базується на одній батьківській (головній сторінці) і дочірніх сторінках (категоріях і підкатегоріях), які випливають з головної сторінки.

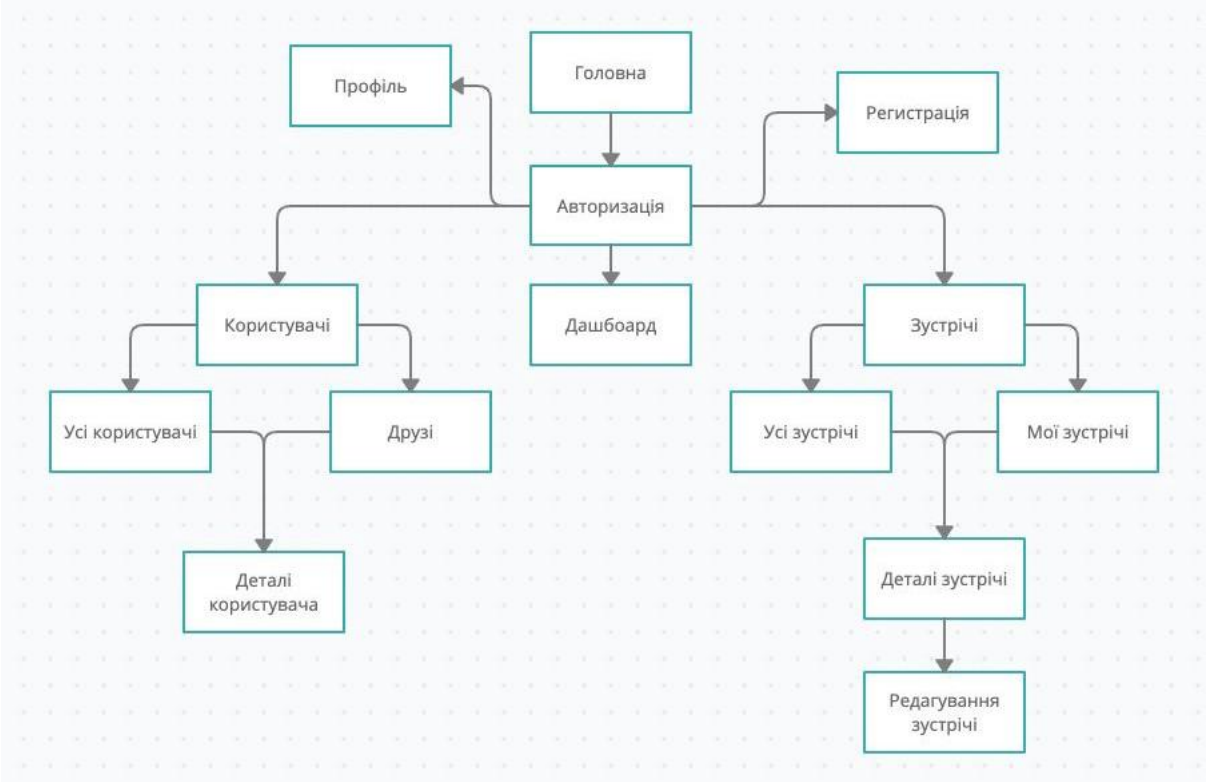

Рисунок 2.8 – Логічна структура веб-додатку

Кожен з елементів структури веб-сервісу можна оптимізувати під час процесу проєктування:

− Домашня сторінка – це верхня сторінка в ієрархії веб-сайта та центральне місце, звідки користувачі переходять на веб-сайт. Важливо, щоб усі важливі сторінки веб-сайта містили посилання з цієї сторінки. Взаємозв’язок між домашньою сторінкою та сторінками основної категорії відображається в меню веб-сайта або головною навігацією.

− Навігація/меню. Відвідувачі сайта використовуватимуть навігацію, щоб зрозуміти, як структурована інформація на веб-сайті, і знайти те, що вони шукають. Отже, усі основні сторінки категорій мають бути представлені у меню або головній навігації. Крім того, під час створення навігації потрібно дотримуватись такого правила: використовувати короткі фрази або навіть одне слово для кожного елемента та просту мову, зрозумілу користувачам.

Основна навігація Apple дотримується цих правил, щоб створити просте, але надзвичайно корисне меню.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Розроблення інтерфейсу

Інтерфейс користувача (User interface) – це точка взаємодії між людиною та комп’ютером. Мета інтерфейсу користувача – дати можливість користувачеві ефективно керувати комп’ютером або машиною, з якою вони взаємодіють, і отримувати зворотний зв’язок, щоб повідомляти про ефективне виконання завдань. Успішний інтерфейс користувача має бути інтуїтивно зрозумілим (не вимагати навчання для роботи), ефективним (не створюючи додаткових або непотрібних зусиль) і зручним для користувачів (бути приємним використанні).

Найкращий спосіб конкурувати в Інтернеті – це розробити привабливий та ефективний інтерфейс користувача (UI), який оптимізує користувацький досвід (UX).

Найважливіші загальні елементи гарного інтерфейсу користувача:

− Інформаційна архітектура: логічно структурувати та організувати вміст веб-сайту важливо, щоб користувачі могли переміщатися по сайту з мінімальними зусиллями. Компоненти ІА включають три основні типи організаційних структур: ієрархічну (рівень важливості), послідовну (логічний порядок кроків) і матричну (в якій користувач вибирає організацію контенту, який бачить).

Приклад: елементи навігації (кнопки, вкладки, значки), мітки (термінологія), функції пошуку (рядок пошуку) та системи організації (категорії).

− Інтерактивний дизайн: елементи ідентифікатора мають на меті перетворити пасивних читачів на активних учасників, представляючи приклади введення користувачів. Треба пам’ятати про користувача під час створення інтерфейсу користувача, що допоможе покращити інтерактивність і виконання конкретних способів поведінки, які задовольняють потреби

користувачів. Крім того, ефективно розроблені інтерактивні інтерфейси можуть «навчитися» передбачати та усувати будь-які проблеми, які можуть виникнути, перш ніж вони негативно вплинуть на роботу користувача.

Приклад: функції соціального доступу, перемикачі, кнопки.

– Візуальний дизайн: важливість естетичної цінності вашого сайту не можна недооцінювати. Ефективний дизайн використовує колір, контраст, шрифт, відео та фотоелементи, щоб залучити відвідувачів і полегшити їм читання та роботу з вмістом, щоб створити логічний, інтуїтивно зрозумілий потік функціональних можливостей.

Приклад: контраст, колір, пробіл, типографіка, оптимізація для мобільних пристроїв [22].

Усі сторінки сайта розбиті на такі основні частини як верхня (header), основна (контент) та нижня (footer), що були розроблені за допомогою мови розмітки HTML, каскадної таблиці стилів CSS та фреймворку Angular.

Верхня частина веб-сервісу, так звана «Шапка» є одним з основних візуальних елементів веб-сайта, заголовок є цінним елементом цифрової нерухомості.

«Шапка» веб-додатка містить такі пункти меню, як назва веб-додатка та кнопка з іконкою людини, натиснувши на яку з’являється кнопка «Login», що дає можливість авторизації користувачеві (рис. 3.1).

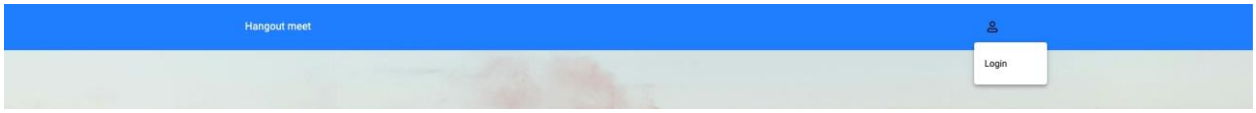

Рисунок 3.1 – «Шапка» веб-сайта

При авторизації у верхній частині веб-додатка з’явиться можливість переглянути свій профіль (Profile), повернутися на головну сторінку (Introduction) та вийти зі свого акаунту (Logout) (рис. 3.2).

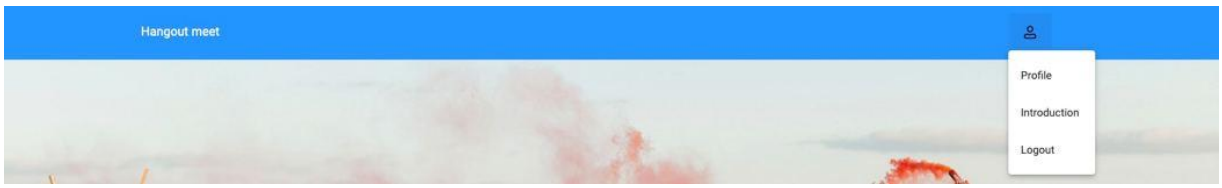

Рисунок 3.2 – «Шапка» веб-сайта при виконанні авторизації

При переході за посиланням «Logout» користувач матиме можливість авторизуватись чи натиснувши на клікабельний текст «Don’t have an account», при потрібності, зареєструватися (рис. 3.3).

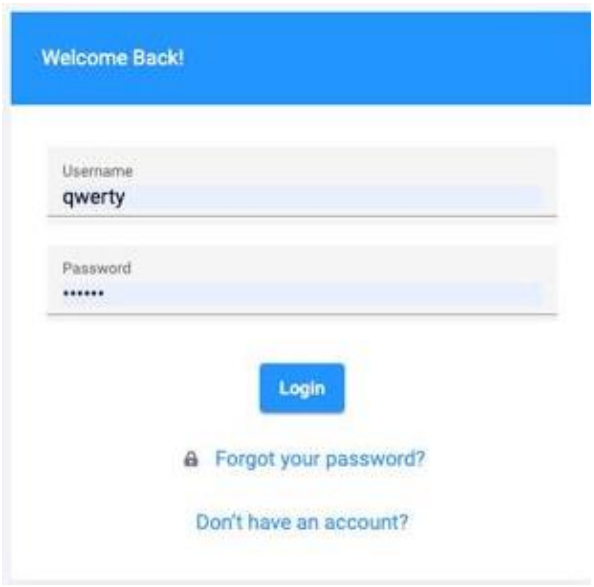

Рисунок 3.3 – Панель авторизації

Процес реєстрації складається с чотирьох кроків: заповнення особистих даних (рис. 3.4), введення електронної пошти (рис. 3.5), потім паролю (рис. 3.6) і останній крок – додавання посилань на соціальні мережі (Instagram/Telegram/Twitter) (рис. 3.7).

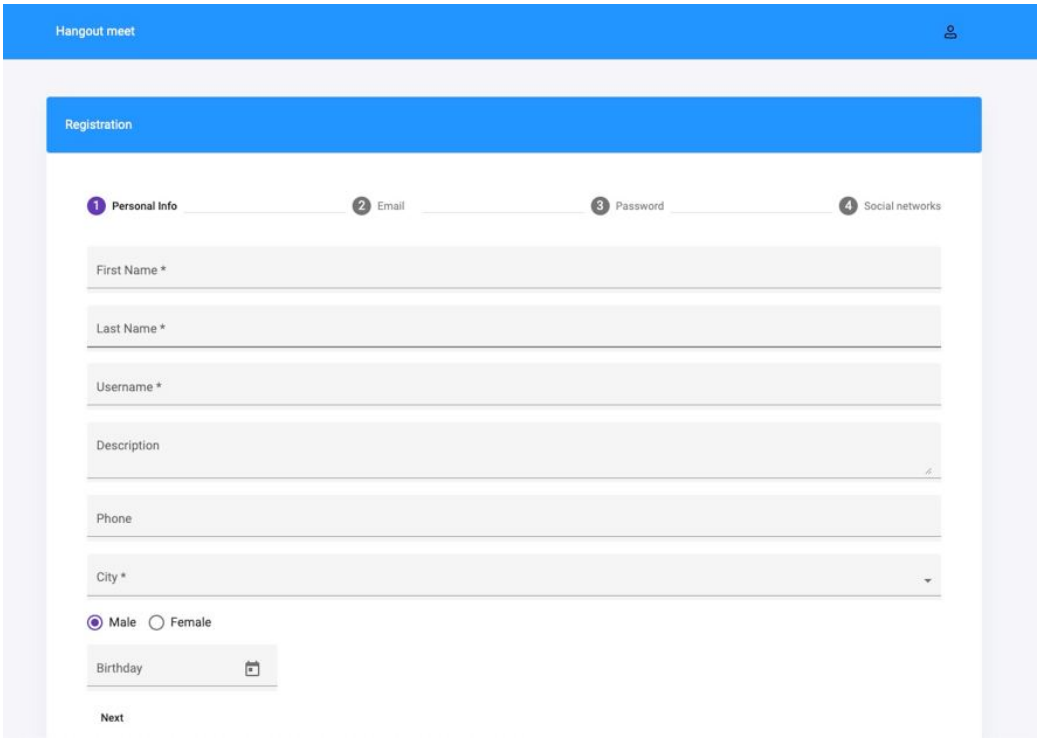

Рисунок 3.4 – Форма для заповнення особистих даних

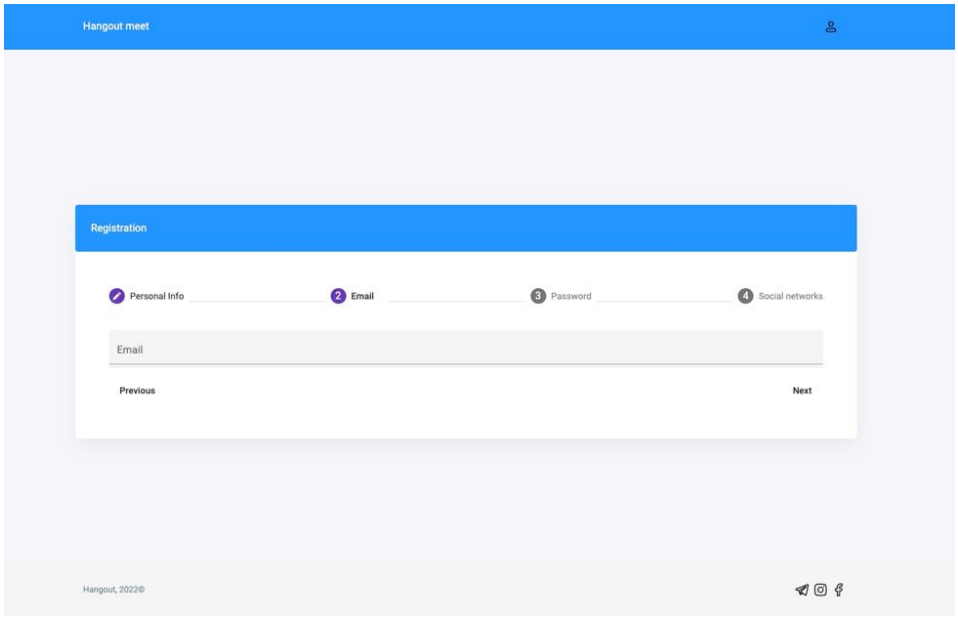

Рисунок 3.5 – Форма для введення електронної пошти

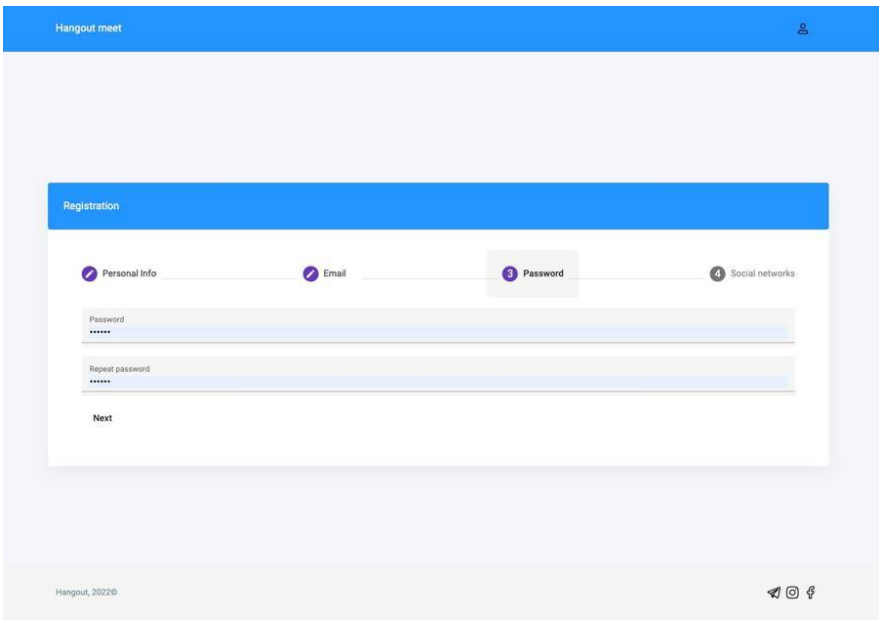

Рисунок 3.6 – Форма для введення нового паролю

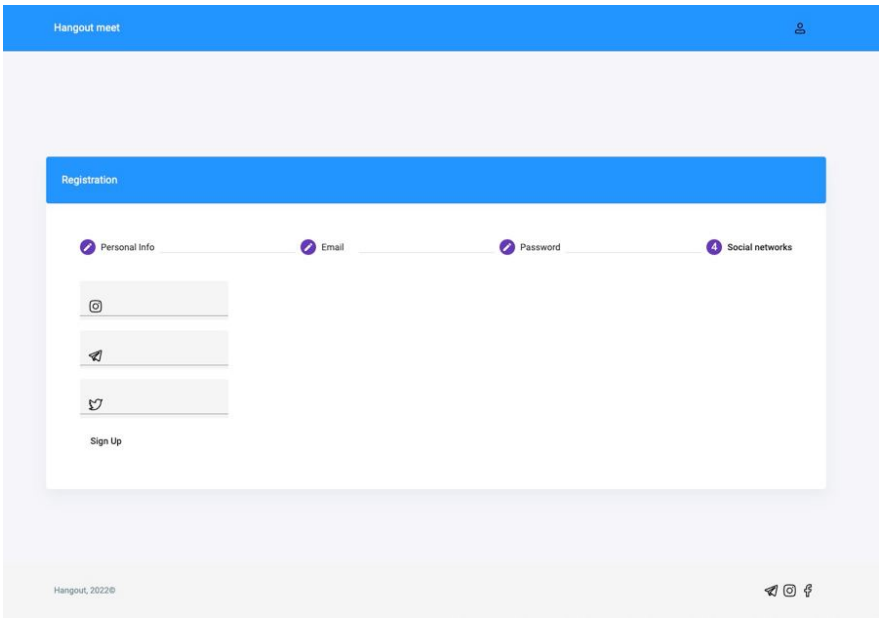

Рисунок 3.7 – Форма для додавання посилань на соціальні мережі

Контентна частина головної сторінки веб-сервісу містить фонове зображення, блок з перевагами сайта для заохочення користування веб-сервісом користувачів та новинами сайта (рис. 3.8).

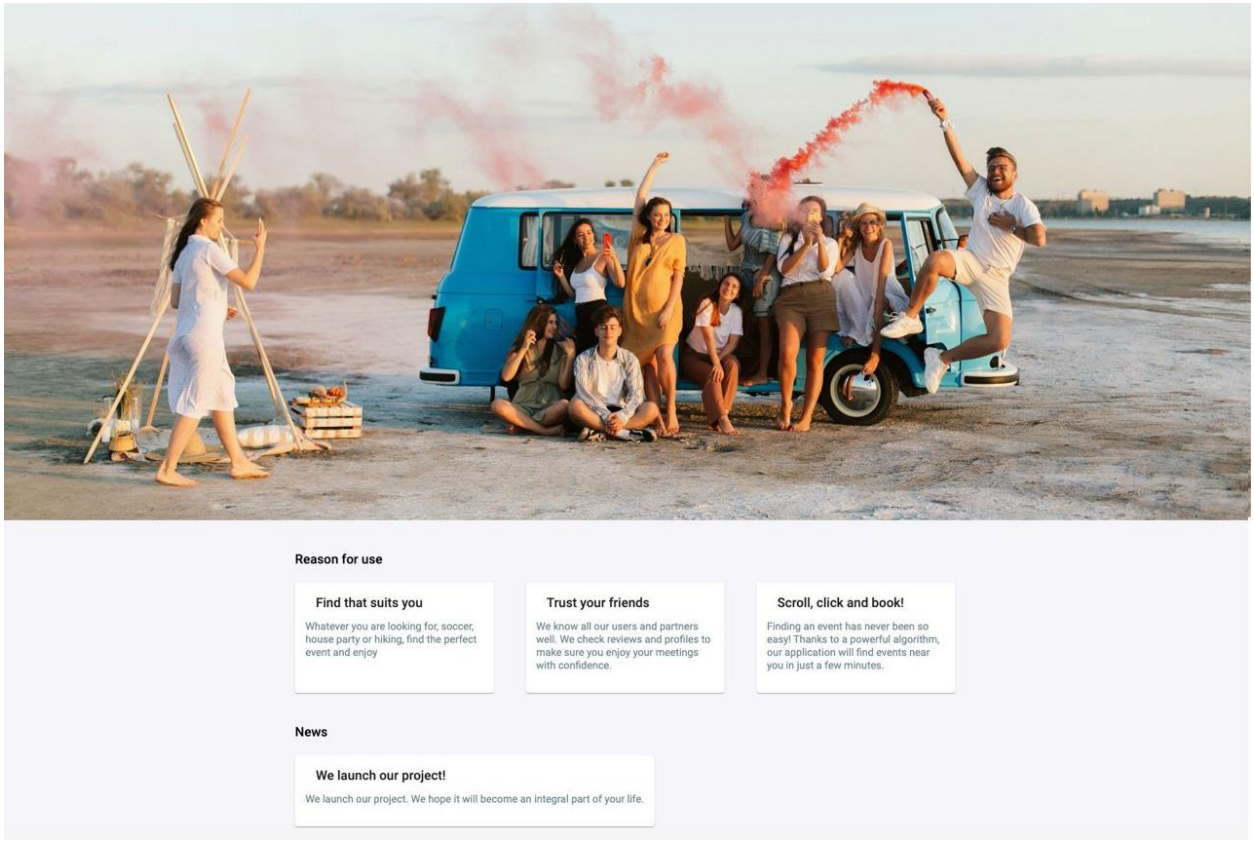

Рисунок 3.8 – Контента частина веб-сервісу

На всіх сторінках на контентній частині сайту після авторизації з’явиться навігаційне меню з посиланнями, котрі дають можливість пересуватися по різним сторінкам сайту, таким як: Dashboard, Meets, Users, Profile.

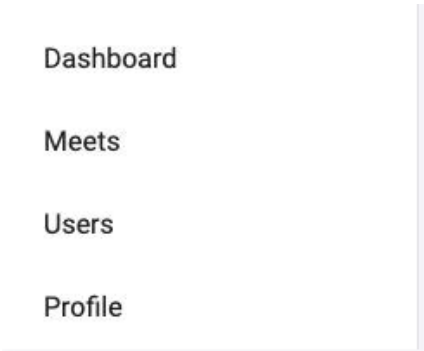

Рисунок 3.9 – Навігаційне меню

Скориставшись навігаційним меню і перейшовши за посиланням «Users» можна побачити таблицю зареєстрованих користувачів. (рис. 3.10)

Далі є можливість перейти на картку конкретного зареєстрованого користувача, де буде розміщена деяка інформація про нього. (рис. 3.11)

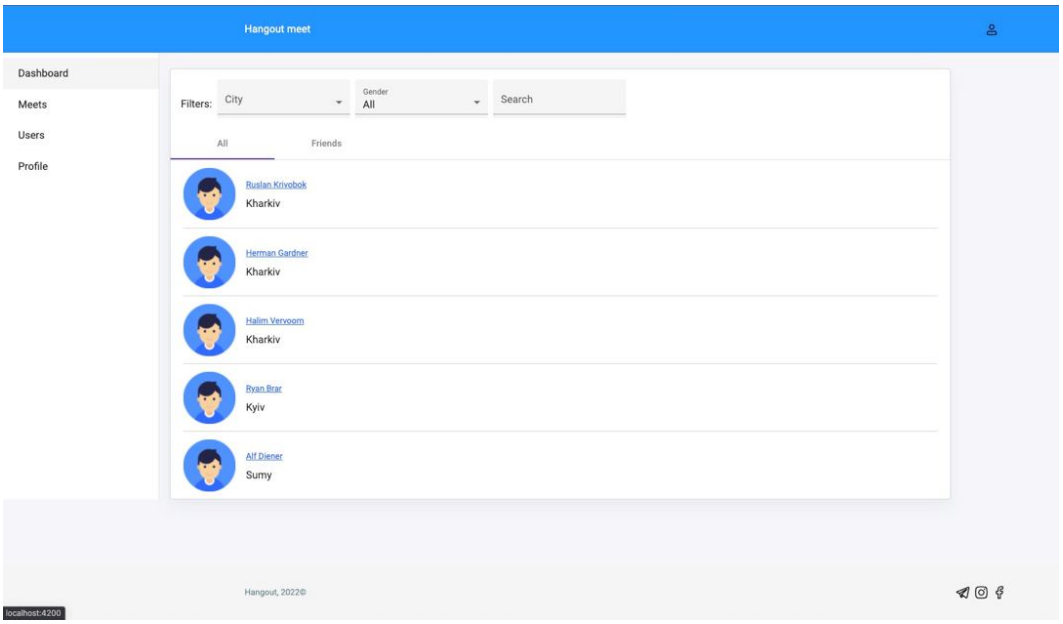

Рисунок 3.10 – Сторінка зареєстрованих користувачів

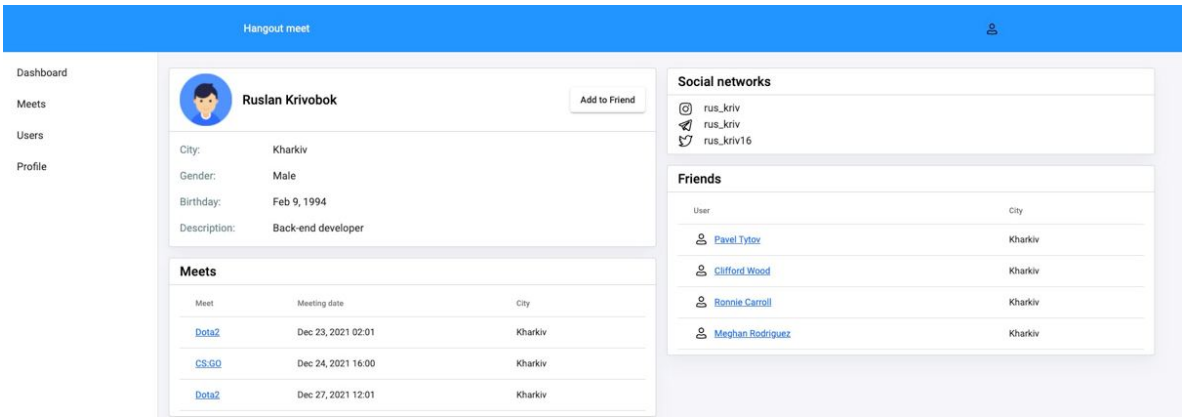

Рисунок 3.11 – Картка зареєстрованого користувача

Вже створені зустрічі можна переглянути скориставшись навігаційним меню за допомогою посилання під назвою «Meets» (рис. 3.12), а перейшовши за посиланням «Create meet» є можливість створити власну зустріч (рис. 3.13).

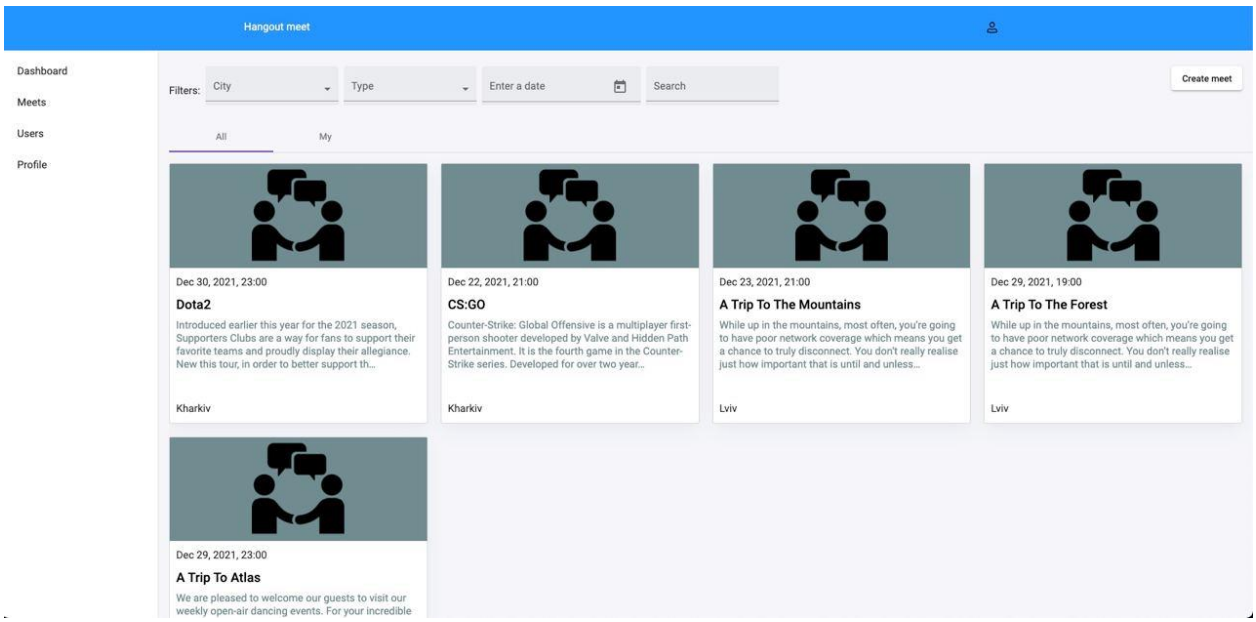

Рисунок 3.12 – Картка з можливими для перегляду вже створеними кимось подіями

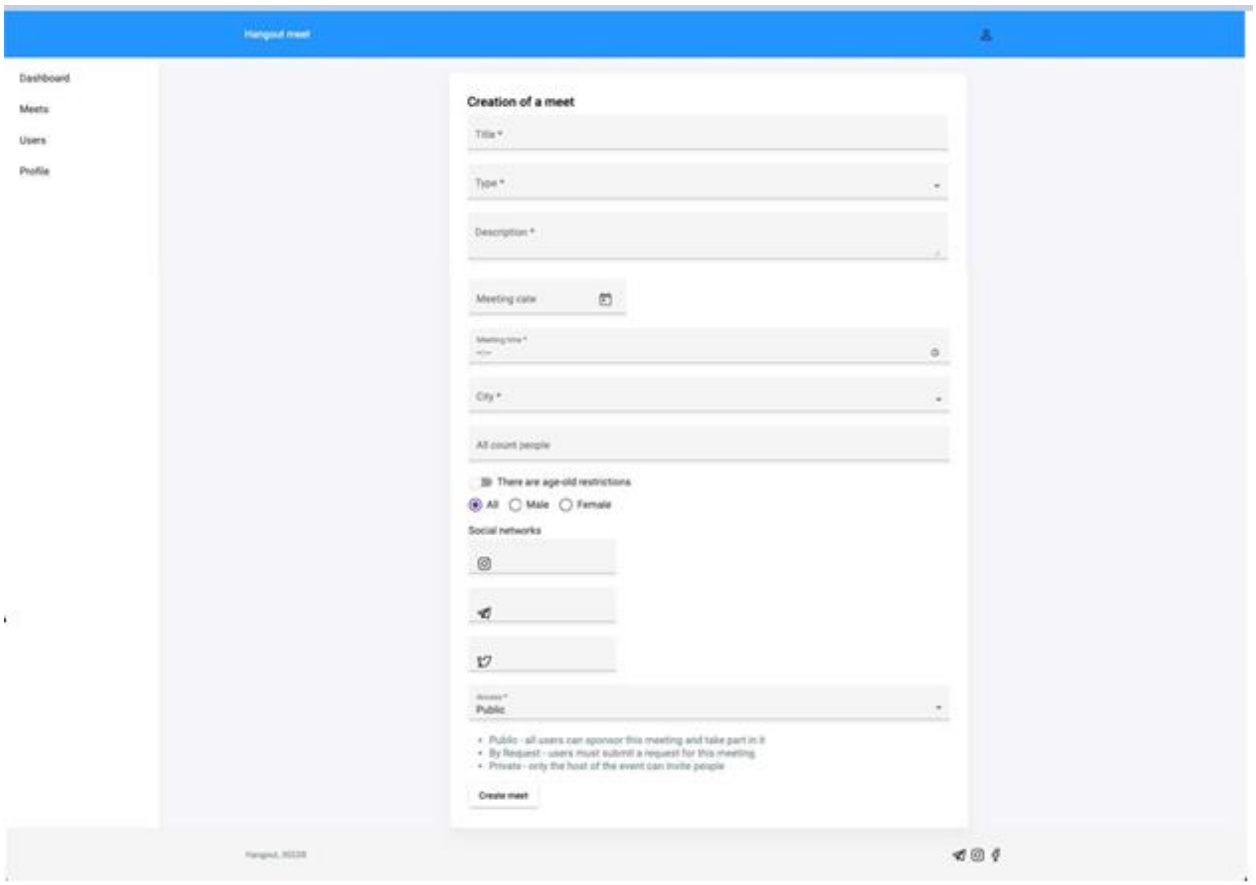

Рисунок 3.13 – Картка для створення власної зустрічі

Також є можливість дізнатися більше деталей про подію натиснувши на будь-яку відображених на сторінці. (рис.3.14)

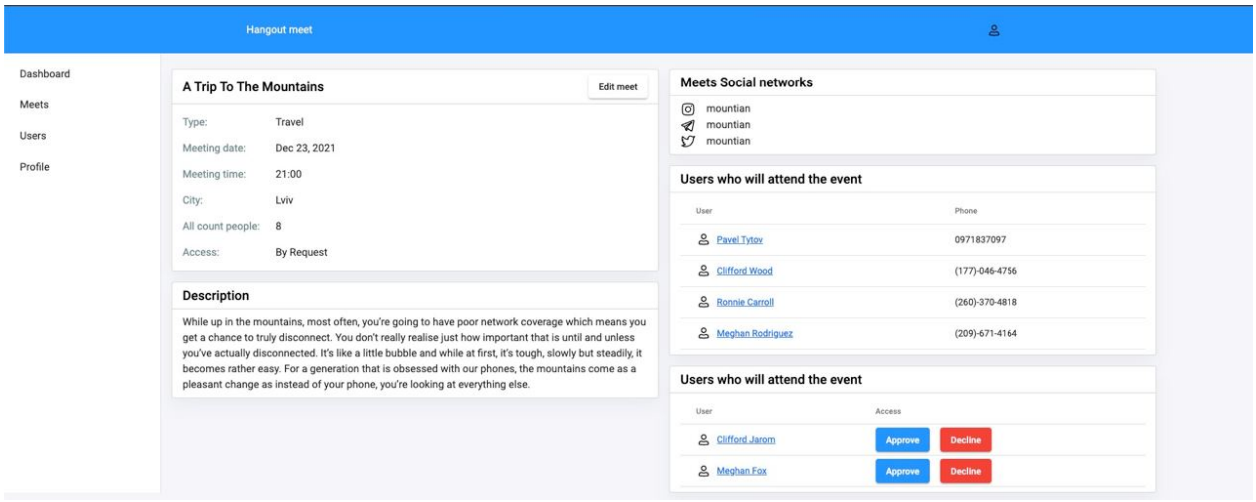

Рисунок 3.14 – Картка з деталями про заплановану подію

Нижня частина сайту (footer) вміщує в собі службову інформацію та посилання на соціальні мережі веб-додатка. (рис. 3.15)

Рисунок 3.15 – Footer веб-сервісу

3.2 Розроблення бази даних

У веб-додатку використовується провідна сучасна платформа баз даних загального призначення - MongoDB, яка розроблена, щоб розкрити потужність програмного забезпечення та даних для розробників і програм, які вони створюють.

«App» є основною базою даних веб-додатка, яка включає в себе дві колекції даних: meets та users (рис. 3.16–3.18).

Ці колекції вміщують в собі основну інформацію про користувачів (ім’я, місто, номер телефону, день народження, соціальні мережі) та про

зустрічі/події (назва зустрічі, її тематика, опис, місце та час проведення, список користувачів, які будуть на зустрічі).

Для звернення від клієнтської частини додатку до серверної використовуються контролери, які визначають, які функції будуть викликані на серверній частині веб-додатка. (рис. 3.19, 3.20)

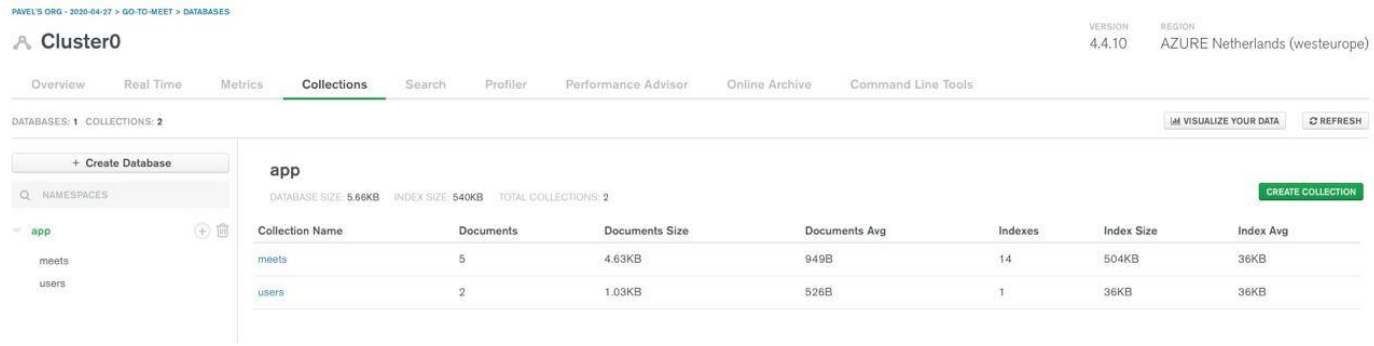

Рисунок 3.16 – Адміністративна панель бази даних «APP»

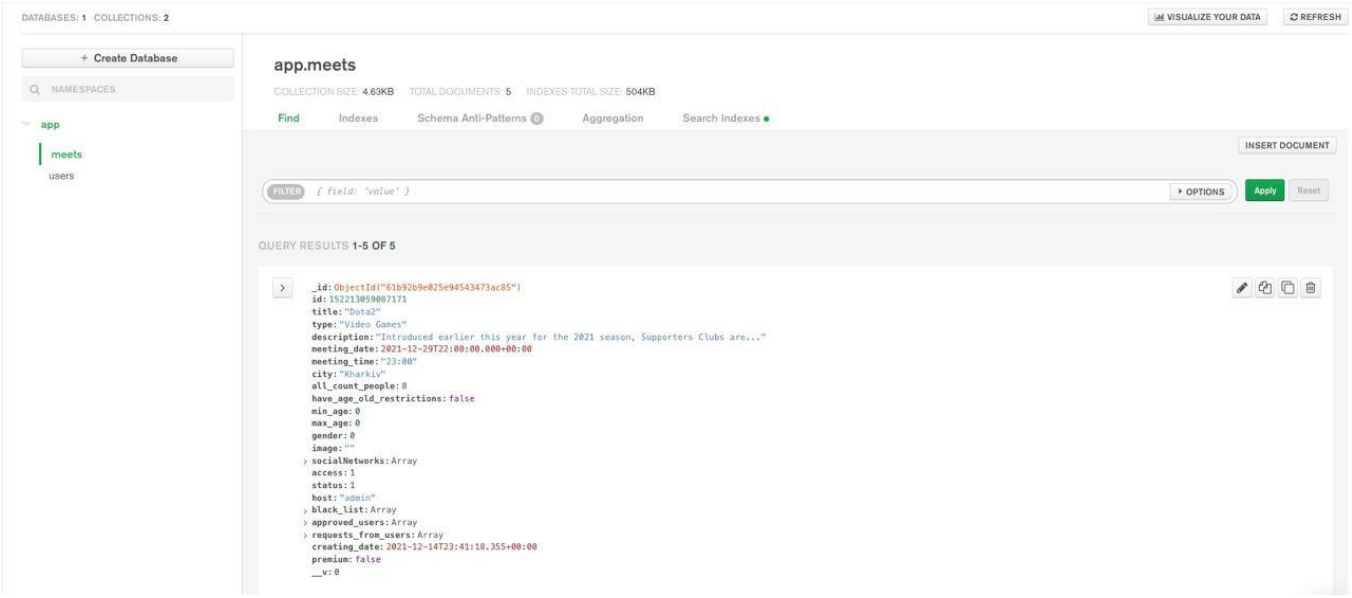

Рисунок 3.17 – Приклад даних в колекції «meets»

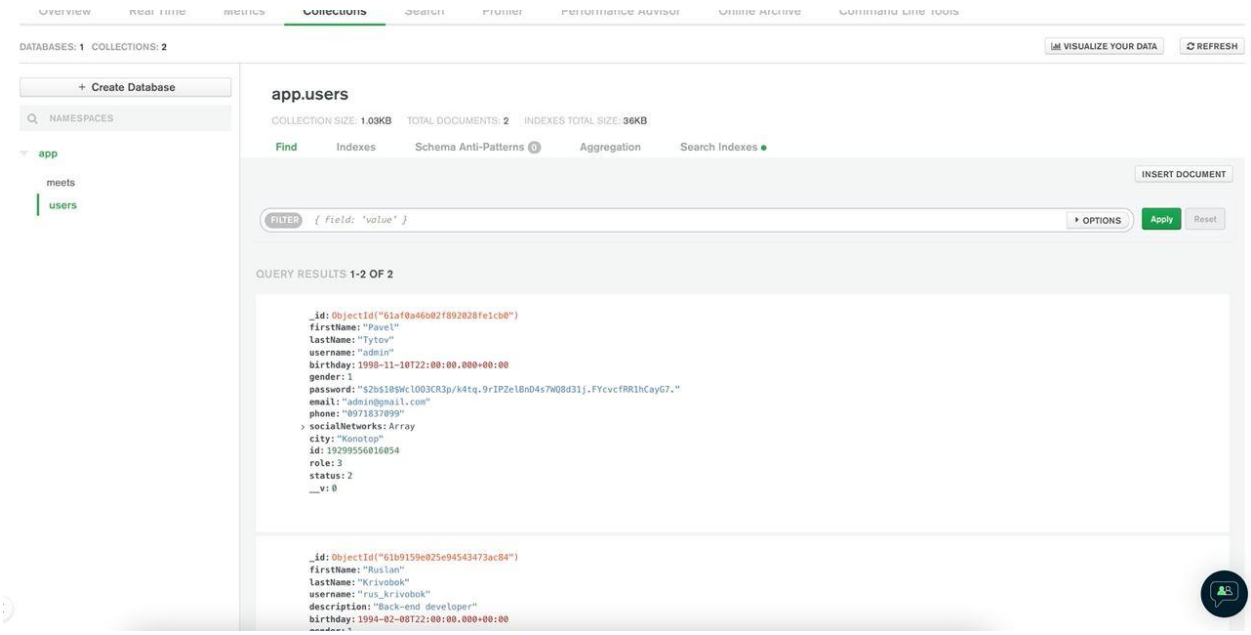

Рисунок 3.18 – Приклад даних в колекції «users»

```
import ...

@Controller( prefix: 'users')
export class UsersController {
    constructor(private readonly userService: UsersService) {}

    @UseGuards(JwtGuard)
    @Get()
    getAll(@Req() req): Observable<Users[]> {
        const currentUsername = req.user.username;
        return this.userService.getAll(currentUsername);
    }

    @Get( path: ':username')
    getByUsername(@Param( property: 'username') username: string): Promise<Users> {
        return this.userService.getByUserName(username);
    }

    @UseGuards(JwtGuard)
    @Post()
    @HttpCode(HttpStatus.CREATED)
    async create(@Body() createUser: CreateUserDto): Promise<Users> {
        return await this.userService.create(createUser).catch((error: ErrorMessageEnum) => {
            throw new HttpException(error, HttpStatus.CONFLICT);
        });
    }

    @UseGuards(JwtGuard)
    @Delete( path: ':id')
    async delete(@Param( property: 'id') id: number): Promise<Users> {
        return this.userService.delete(id);
    }

    @Put( path: ':username')
    @HttpCode(HttpStatus.OK)
    @Roles(RoleEnum.Admin)
    @UseGuards(JwtGuard, RolesGuard)
    async update(@Body() createUser: UpdateUserDto, @Param( property: 'username') username: string): Promise<Users> {
        return this.userService.update(createUser, username);
    }
}
```

Рисунок 3.19 – Зміст контролера для користувачів

```
@Controller( prefix: 'meets')
export class MeetsController {
    constructor(private readonly service: MeetsService) {
    }

    @UseGuards(JwtGuard)
    @Get()
    @HttpCode(HttpStatus.OK)
    getAll(): Observable<MeetViewModel[]> {
        return this.service.getAll();
    }

    @Get( path: ':id')
    @HttpCode(HttpStatus.OK)
    @UseGuards(JwtGuard)
    getByUsername(@Param( property: 'id') id: number): Promise<MeetViewModel> {
        return this.service.getById(id);
    }

    @UseGuards(JwtGuard)
    @Post()
    @HttpCode(HttpStatus.CREATED)
    async create(@Body() createMeet: MeetCreateDto, @Req() req): Promise<MeetViewModel> {
        return await this.service.create(createMeet, req.user.username).catch((error: ErrorMessageEnum) => {
            throw new HttpException(error, HttpStatus.CONFLICT);
        });
    }

    @UseGuards(JwtGuard)
    @Delete( path: ':id')
    @HttpCode(HttpStatus.OK)
    async delete(@Param( property: 'id') id: number): Promise<MeetViewModel> {
        return this.service.delete(id);
    }

    @Put( path: ':id')
    @HttpCode(HttpStatus.OK)
    @UseGuards(JwtGuard)
    async update(@Body() createUser: MeetUpdateDto, @Param( property: 'id') id: string): Promise<MeetViewModel> {
        return this.service.update(createUser, id);
    }
}
```

Рисунок 3.20 – Зміст контролера для подій

3.3 Тестування веб-додатка

Тестування є важливою частиною будь-якого веб-сервісу або веб-додатка. Дуже важливо зробити тестування додатків на потенційні помилки, використовуючи різні методики перед запуском або розгортанням.

Під час тестування сайта робиться декілька перевірок, як-от безпека, функціональність веб-сайта, доступність для користувачів. Однак деякі організації чи окремі особи іноді думають, що не потрібно тестувати веб-сайт

після його запуску, оскільки вони чудово створили його та протестували перед релізом. Але це неправильне рішення. Для успішного ведення онлайн-бізнесу і створення чудових пропозицій клієнтам, необхідно перевірити веб-додаток.

Під час тестування веб-сервісу була проведена перевірена перевірка на коректність його роботи. Здійснювалися такі перевірки:

1) Валідація поля на порожнечу. Якщо поле не заповнено – показує помилку після відправки форми (рис. 3.21).

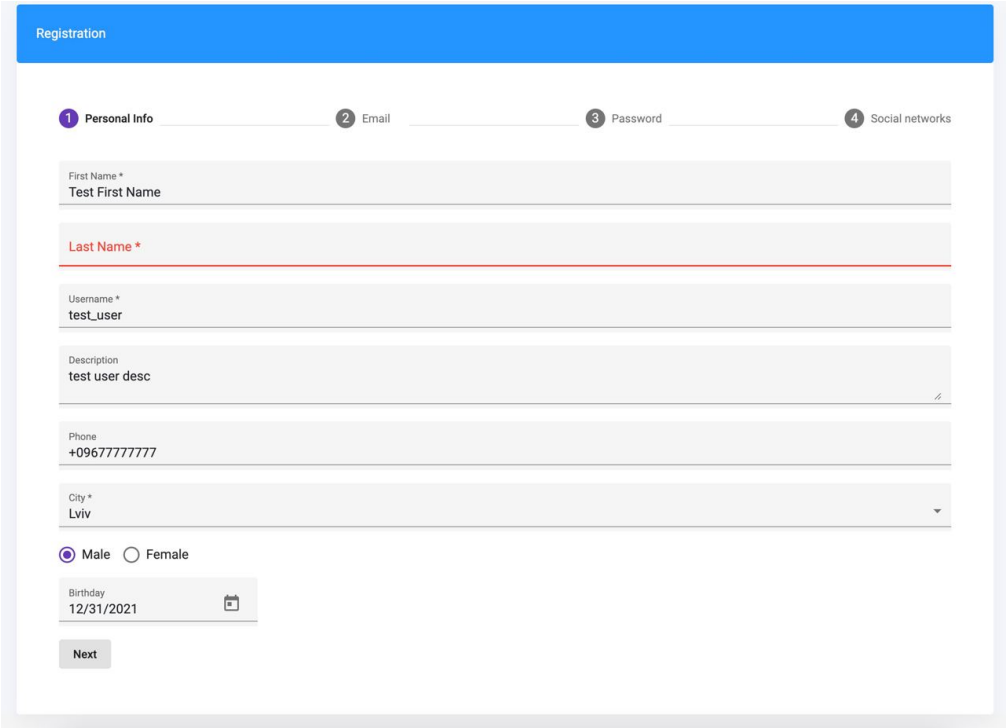

Рисунок 3.21 – Валідація поля на порожнечу

2. Проведена перевірка коректності запису даних в базу при реєстрації. (рис. 3.22–3.23).

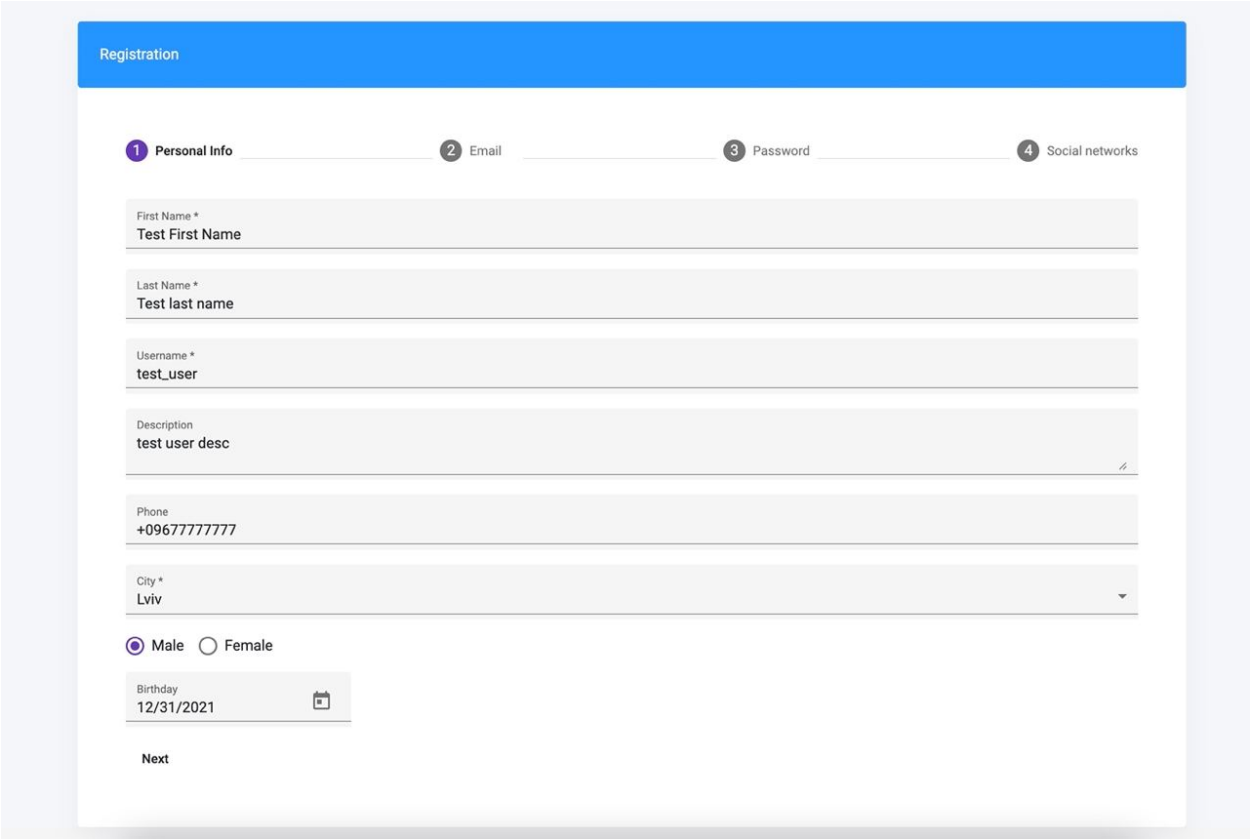

Рисунок 3.22 – Виконання реєстрації, заповнення даних

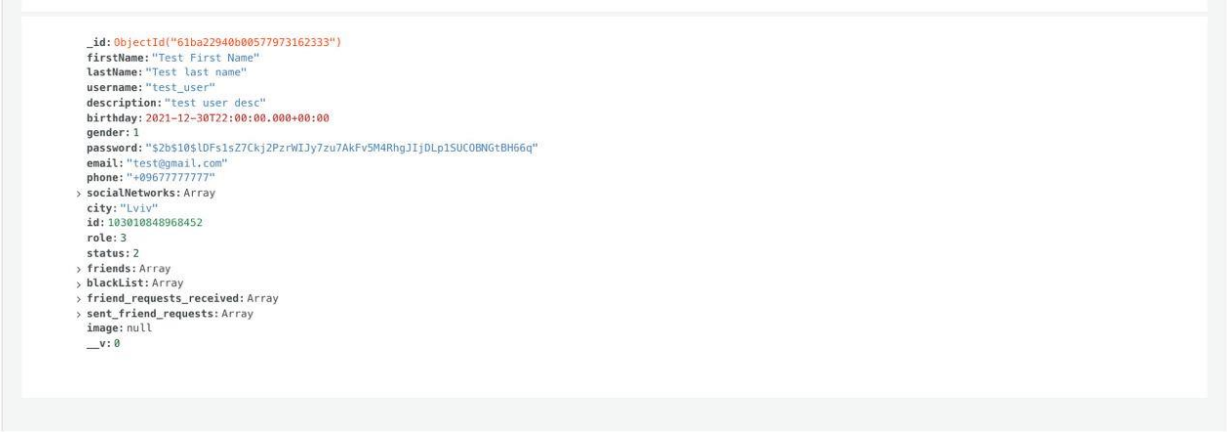

Рисунок 3.23 – Коректне відображення даних в БД після виконання реєстрації користувачем

3. Перевірка на захищеність засобів авторизації (рис. 3.24).

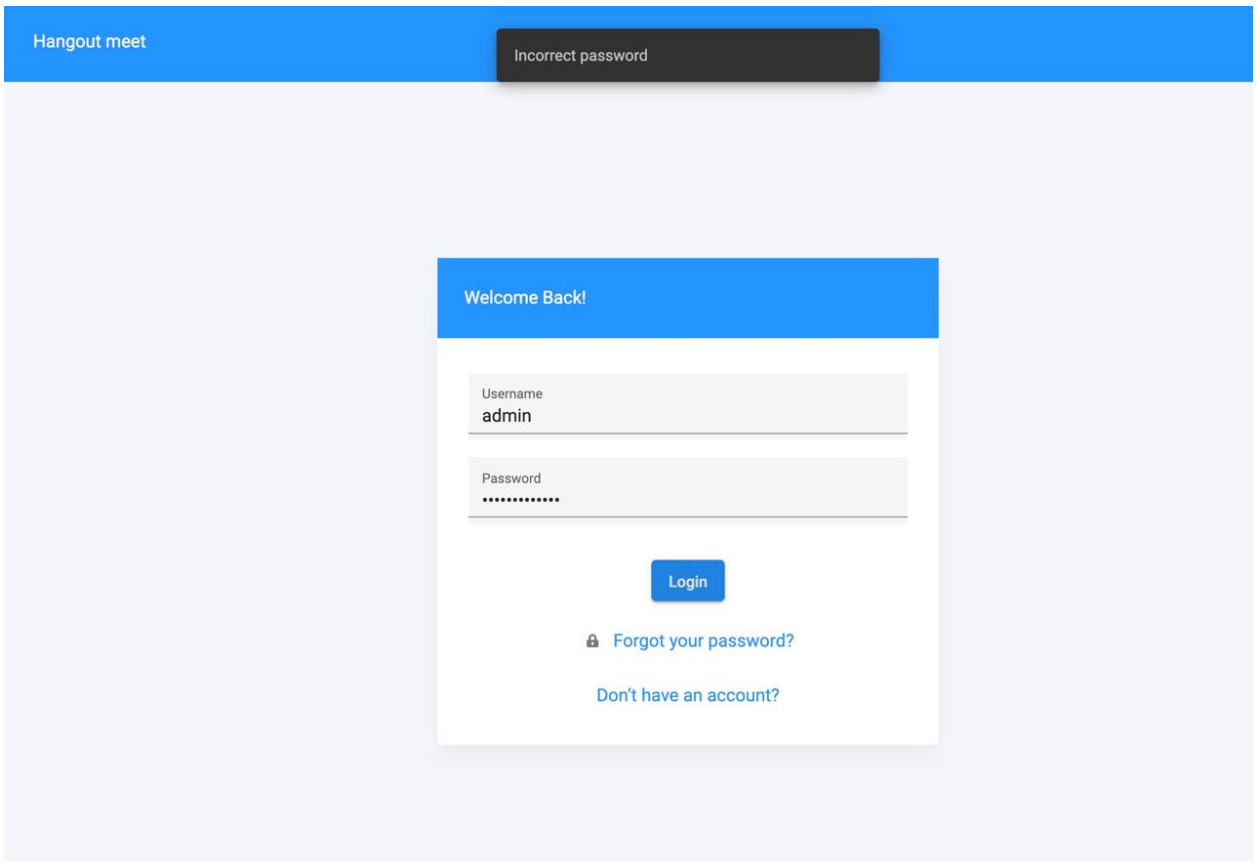

Рисунок 3.24 – Виклик помилки «Incorrect password»

4. Перевірка кросбраузерності веб-додатка. Було перевірено у браузерах: Chrome, Firefox, Safari (рис. 3.25–3.27).

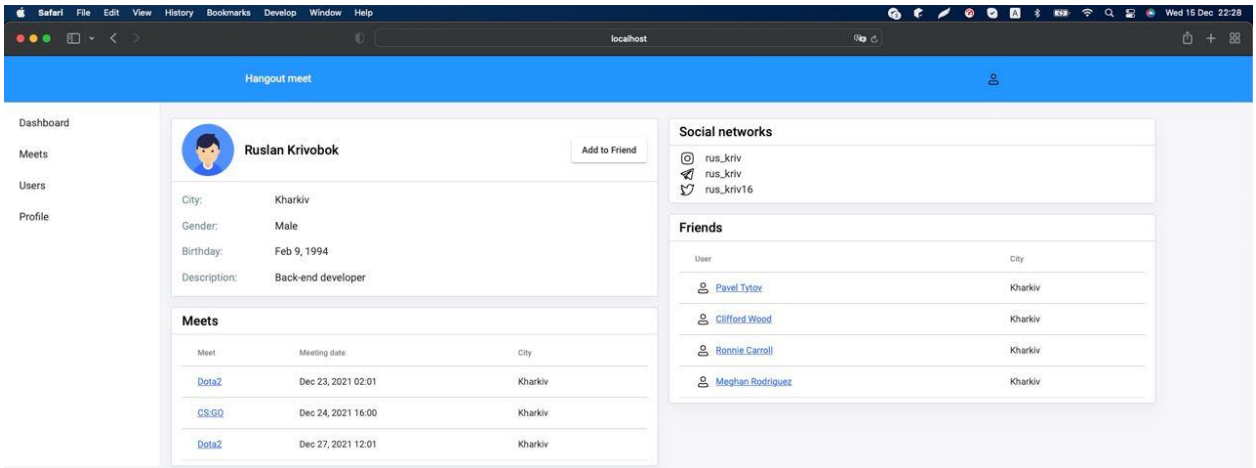

Рисунок 3.25 – Перегляд сайта в інтернет-браузері Safari

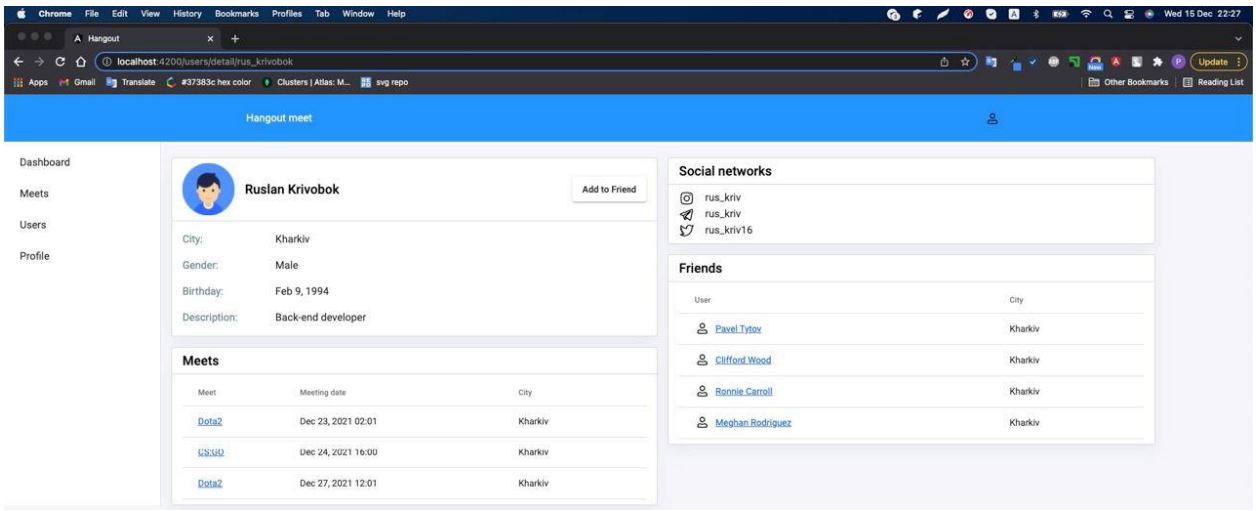

Рисунок 3.26 – Перегляд сайта в інтернет-браузері Chrome

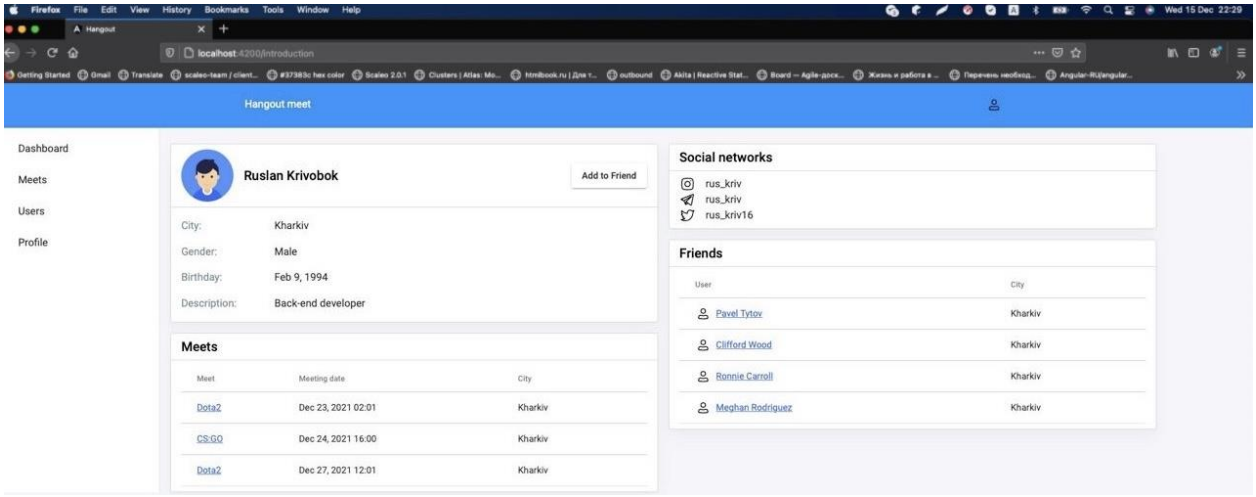

Рисунок 3.27 – Перегляд сайта в інтернет-браузері FireFox

Перевірка на коректність заповнення даних. Якщо паролі не співпадають – викликається помилка «Repeat password» (рис. 3.28).

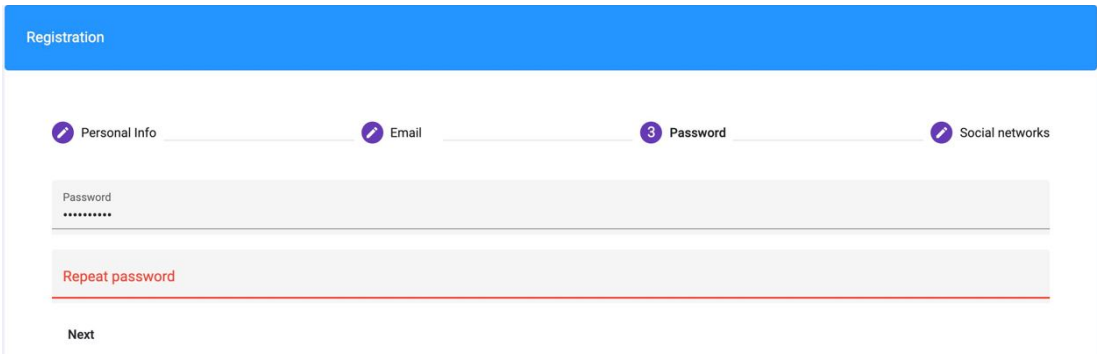

Рисунок 3.28 – Перевірка на коректність заповнення даних

Також було перевірено стабільну і безпомилкову роботу сайта на різних операційних системах, таких як Windows та MacOs.

3.4 Інструкція з використання веб-додатка

На стартовій сторінці веб-додатка є такі пункти меню як назва веб-додатка та кнопка з іконкою людини, що дає можливість користувачу авторизуватися (рис. 3.1).

У випадку, якщо користувач ще не зареєстрований, на панелі авторизації потрібно натиснути на клікабельний текст «Don’t have an account» (рис. 3.3) та пройти чотири кроки реєстрації: заповнення особистих даних, введення електронної пошти, потім паролю і останній крок – додавання посилань на соціальні мережі (рис. 3.4-3.7).

Щоб переглянути свій особистий профіль потрібно авторизуватися і перейти за посиланням (Profile), яке знаходиться на навігаційному меню, що розташоване зліва на контентній частині сторінки або у випадаючому списку натиснувши на іконку людини на «шапці» профілю (рис. 3.9).

Щоб повернутися на головну сторінку потрібно перейти за посиланням (Introduction) у випадаючому списку натиснувши на іконку людини на «шапці» профілю (рис. 3.2).

Щоб вийти зі свого акаунту потрібно перейти за посиланням (Logout), що знаходиться поряд с посиланням (Introduction) (рис. 3.2).

Для того щоб мати можливість пересуватися по різним сторінкам сайта, таким як Dashboard (де буде розміщено топові події у місті та події які були нещодавно додані, тобто найактуальніші), Meets, Users, Profile (де користувач може змінювати інформацію про себе, налаштовувати мову сайт, а так само там буде чорний список користувачів, який користувач додав раніше) – потрібно авторизуватися після чого на всіх сторінках на контентній частині сайту з’явиться навігаційне меню с потрібними посиланнями.

Скористувавшись навігаційним меню і перейшовши за посиланням «Users» можна побачити таблицю зареєстрованих користувачів (рис. 3.10).

Щоб дізнатися більше інформації про конкретного користувача потрібно буде лише перейти на його картку, натиснувши на неї (рис. 3.11).

Вже створені зустрічі можна переглянути скориставшись навігаційним меню за допомогою посилання під назвою «Meets» (рис. 3.12), а перейшовши за посиланням «Create meet» є можливість створити власну зустріч (рис. 3.13).

Аби дізнатися більше деталей про подію, що зацікавила – потрібно лише натиснути на будь-яку відображених на сторінці (рис.3.14).

ВИСНОВКИ

У сучасному світі не рідко можна зіткнутися з проблемою пошуку компанії для проведення дозвілля, а за допомогою спеціального сервісу цілком можна буде зробити це всього за декілька хвилин.

Веб-сервіс повинен бути створений в першу чергу для реального спілкування: користувач знаходить цікаві події та нові заклади, знайомиться з людьми, призначає зустріч і спілкується в реальності.

Кваліфікаційна магістерська робота присвячена аналізу актуальності проблеми з метою поліпшення взаємодії людей і інформаційних систем один з одним і забезпечити взаємне проникнення різних систем і процесів.

Проведений аналіз предметної області застосування, описана актуальність задачі, визначені наявні проблеми, підготовлена структура веб-додатку для реалізації проєкту та сформульована мета.

В результаті виконання кваліфікаційної магістерської дипломної роботи була розроблена інформаційна технологія проєктування соціальної мережі для підтримки соціально-культурних проектів молоді з простим і зрозумілим у використанні інтерфейсом, також було проведено тестування роботи веб-сервісу.

Веб-сервіс виконаний в повному обсязі та відповідно до усіх заданих вимог і норм. Усі поставлені завдання реалізовані автором.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Класифікація веб-сайтів [Електронний ресурс]. – Режим доступу : https://sites.google.com/site/yaremusinform/home

2. Negative outcomes of Internet use: A qualitative analysis in the homes of families with different educational backgrounds [Електронний ресурс]. – Режим доступу : https://www.tandfonline.com/doi/full/10.1080/01972243.2019.1649774

3. Communication: Online vs. Face-to-Face Interactions [Електронний ресурс]. – Режим доступу : https://psychminds.com/communication-online-vs-face-to-face-interactions/

4. What is the world wide web? [Електронний ресурс]. – Режим доступу : https://www.bbc.co.uk/bitesize/topics/zkcqn39/articles/z2nbgk7

5. What is HTML? [Електронний ресурс]. – Режим доступу : https://www.yourhtmlsource.com/starthere/whatishtml.html

6. What is CSS, and why is it important? [Електронний ресурс]. – Режим доступу : https://www.bigcommerce.com/ecommerce-answers/what-css-and-why-it-important/

7. JavaScript (JS) [Електронний ресурс]. – Режим доступу : https://www.techopedia.com/definition/3929/javascript-js

8. Javascript [Електронний ресурс]. – Режим доступу : https://scriptdev.ru/ts/007/

9. What is TypeScript and why should you use it? [Електронний ресурс]. – Режим доступу : https://learn.coderslang.com/0056-what-is-typescript-and-why-should-you-use-it/

10. Angular [Електронний ресурс]. – Режим доступу : https://www.typescriptlang.org/docs/handbook/angular.html

11. 8 Proven Reasons You Need Angular for Your Next Development Project [Електронний ресурс]. – Режим доступу : https://www.grazitti.com/blog/8-proven-reasons-you-need-angular-for-your-next-development-project/

12. What is API: Definition, Types, Specifications, Documentation [Електронний ресурс]. – Режим доступу : https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/

13. REST API [Електронний ресурс]. – Режим доступу : https://blog.skillfactory.ru/glossary/rest-api/

14. Murray N., Coury F., Lerner A., Taborda C. ng-book: The Complete Guide to Angular, 5th edition. Technical Editor: Frode Fikke, 2018. 626 p.

15. What Are Angular Services and Why Should You Use them? [Електронний ресурс]. – Режим доступу : https://chudovo.com/what-are-angular-services-and-why-should-you-use-them/

16. What is Bootstrap and Why you should use it in Web Development [Електронний ресурс]. – Режим доступу : https://www.yogihosting.com/what-is-bootstrap/

17. Why You Should Use NestJS for Backend Development? [Електронний ресурс]. – Режим доступу : https://enlear.academy/why-you-should-use-nestjs-as-your-backend-framework-bd1ff1acce5d

18. Magolan G., Bell J., Guijarro D., Peretti A., Housley P. Nest.js: A Progressive Node.js Framework, Kindle Edition. Bleeding Edge Press, 2018. 350 p.

19. What is MongoDB? A quick guide for developers [Електронний ресурс]. – Режим доступу : https://www.infoworld.com/article/3623357/what-is-mongodb-a-quick-guide-for-developers.html

20. Bradshaw S., Brazil E., Chodorow K. MongoDB: The Definitive Guide, 3rd Edition. O'Reilly Media, 2019. 514 p.

21. Page Regions [Електронний ресурс]. – Режим доступу : https://www.w3.org/WAI/tutorials/page-structure/regions/

22. What Is a User Interface? (Definition, Types and Examples) [Електронний ресурс]. – Режим доступу : https://www.indeed.com/career-advice/career-development/user-interface

ДОДАТОК А

ЛІСТИНГ ПРОГРАМНОГО КОДУ

```
import { SetMetadata } from '@nestjs/common';
import { RoleEnum } from '../../../shared/enums/role.enum';

export const Roles = (...roles: RoleEnum[]) => SetMetadata( metadataKey: 'roles', roles);
```

```
export class JwtGuard extends AuthGuard( type: 'jwt'){
    constructor() {
        super();
    }
}
```

```
@Controller( prefix: 'auth')
export class AuthController {
    constructor(private readonly authService: AuthService, private readonly usersService: UsersService) {}

    @Post( path: 'login')
    @HttpCode(HttpStatus.OK)
    async login(@Body() authData: AuthModel): Promise<{ access_token: string }> {
        return await from(this.authService.validateUser(authData)).pipe(switchMap( project: (username : string ) => this.authService.login(username))).toPromise();
    }

    @Post( path: 'sign-up')
    @HttpCode(HttpStatus.OK)
    signUp(@Body() user: CreateUserDto): Observable<{ access_token: string }> {
        return from(this.usersService.create(user)).pipe(switchMap( project: ({ username }: Users) => this.login( authData: { username, password: user.password })));
    }

    @Post( path: 'logout')
    @HttpCode(HttpStatus.OK)
    logout(@Req() req): void {
        req.logout();
    }
}
```

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { UsersModule } from '../users/users.module';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { jwtConstants } from './constants';
import { JwtStrategy } from './strategies/jwt.strategy';

@Module( metadata: {
    imports: [
        UsersModule,
        PassportModule,
        JwtModule.register( options: {
            secret: jwtConstants.secret,
            signOptions: { expiresIn: '30d' },
        }),
    ],
    controllers: [AuthController],
    providers: [AuthService, JwtStrategy],
    exports: [JwtStrategy, PassportModule]
})
export class AuthModule {}
```

```
@Injectable()
export class AuthService {
    constructor(private usersService: UsersService, private jwtService: JwtService) {}

    async login(username: string): Promise<{ access_token: string }> {
        const payload = { username };
        return {
            access_token: this.jwtService.sign(payload)
        };
    }

    async validateUser(authData: AuthModel): Promise<string> {
        const { username, password } = authData;
        return await from(this.usersService.getByUserName(username)).pipe(
            switchMap( project: (user: Users) => {
                if (!user) {
                    throw new HttpException(ErrorMessageEnum.UserNotFound, HttpStatus.FORBIDDEN);
                }
                return forkJoin( sources: [from(PasswordHashUtil.unHashPasswordIsMatch(user.password, password)), of(user)])
            }),
            switchMap( project: ([unHashPasswordIsMatch, user]: [boolean, Users]) => {
                if (!unHashPasswordIsMatch) {
                    throw new HttpException(ErrorMessageEnum.IncorrectPassword, HttpStatus.FORBIDDEN);
                }

                return of(user.username);
            })
        ).toPromise();
    }

    verify(token: string): Observable<boolean> {
        return this.jwtService.verify(token);
    }


}
```

```
@Controller( prefix: 'meets')
export class MeetsController {
    constructor(private readonly service: MeetsService) {
    }

    @UseGuards(JwtGuard)
    @Get()
    @HttpCode(HttpStatus.OK)
    getAll(): Observable<MeetViewModel[]> {
        return this.service.getAll();
    }

    @Get( path: ':id')
    @HttpCode(HttpStatus.OK)
    @UseGuards(JwtGuard)
    getByUsername(@Param( property: 'id') id: number): Promise<MeetViewModel> {
        return this.service.getById(id);
    }

    @UseGuards(JwtGuard)
    @Post()
    @HttpCode(HttpStatus.CREATED)
    async create(@Body() createMeet: MeetCreateDto, @Req() req): Promise<MeetViewModel> {
        return await this.service.create(createMeet, req.user.username).catch((error: ErrorMessageEnum) => {
            throw new HttpException(error, HttpStatus.CONFLICT);
        });
    }

    @UseGuards(JwtGuard)
    @Delete( path: ':id')
    @HttpCode(HttpStatus.OK)
    async delete(@Param( property: 'id') id: number): Promise<MeetViewModel> {
        return this.service.delete(id);
    }

    @Put( path: ':id')
    @HttpCode(HttpStatus.OK)
    @UseGuards(JwtGuard)
    async update(@Body() createUser: MeetUpdateDto, @Param( property: 'id') id: string): Promise<MeetViewModel> {
        return this.service.update(createUser, id);
    }
}
```

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { AuthModule } from '../auth/auth.module';
import { MeetsService } from './meets.service';
import { MeetsController } from './meets.controller';
import { Meets, MeetsSchema } from './schemas/meets.schema';

@Module( metadata: {
    imports: [
        MongooseModule.forFeature( models: [{ name: Meets.name, schema: MeetsSchema }]),
        AuthModule
    ],
    providers: [MeetsService],
    controllers: [MeetsController]
})
export class MeetsModule {}
```

```
@Injectable()
export class MeetsService {
    constructor(@InjectModel(Meets.name) private readonly meetsModel: Model<MeetsDocument>) {}

    public getAll(): Observable<MeetViewModel[]> {
        return from(this.meetsModel.find());
    }

    public async getById(id: number): Promise<MeetViewModel> {
        return this.meetsModel.findOne(({ id: id }));
    }

    public async delete(id: number): Promise<MeetViewModel> {
        return this.meetsModel.findByIdAndRemove(id);
    }

    public async update(body: MeetUpdateDto, id: string): Promise<MeetViewModel> {
        return this.meetsModel.findOneAndUpdate( filter: { id }, body).catch( onrejected: () => {
            throw new HttpException(ErrorMessageEnum.Forbidden, HttpStatus.FORBIDDEN)
        });
    }

    public async create(meet: MeetCreateDto, userHost: string): Promise<MeetViewModel> {
        return new this.meetsModel(this.transformMeet(meet, userHost)).save();
    }
```

```
    private transformMeet(meet: MeetCreateDto, userHost: string): Meets {
        const { max_age, min_age} = meet;
        return {
            id: UuidUtil.generateNumberId(),
            ...meet,
            status: StatusEnum.Active,
            host: userHost,
            black_list: [],
            approved_users: [],
            requests_from_users: [],
            creating_date: new Date(),
            premium: false,
            max_age: max_age || 0,
            min_age: min_age || 0
        }
    }
}
```

```ts
export type MeetsDocument = Meets & Document;


@Schema()
export class Meets {
    @Prop( options: { type: Number, required: true, unique: true })
    id: number;

    @Prop( options: { type: String, required: true, unique: false })
    description: string;

    @Prop( options: { type: String, required: true, unique: false })
    host: string;

    @Prop( options: { type: Number })
    all_count_people: number;

    @Prop( options: { type: Date, required: true, unique: false })
    meeting_date: Date;

    @Prop( options: { type: String, required: true, unique: false })
    meeting_time: string;

    @Prop( options: { type: Boolean, required: true, unique: false })
    have_age_old_restrictions: boolean;

    @Prop( options: { type: Number })
    min_age: number;

    @Prop( options: { type: Number })
    max_age: number;

    @Prop( options: { type: Number })
    gender: SexEnum;

    @Prop( options: { type: String, required: true, unique: false })
    title: string;
```

```
    @Prop( options: { type: Array })
    black_list: string[] = []

    @Prop( options: { type: Array })
    approved_users: string[] = []

    @Prop( options: { type: () => Buffer })
    image: Buffer;

    @Prop( options: { type: Number, required: true, unique: false })
    status: StatusEnum;

    @Prop( options: { type: Array })
    socialNetworks: SocialNetworkModel[] = []

    @Prop( options: { type: String, required: true, unique: false })
    city: CityEnum;

    @Prop( options: { type: Array })
    requests_from_users: string[] = []

    @Prop( options: { type: String, required: true, unique: false })
    type: MeetTypeEnum;

    @Prop( options: { type: Number, required: true, unique: false })
    access: MeetAccessEnum;

    @Prop( options: { type: Date, required: true, unique: false })
    creating_date: Date;

    @Prop( options: { type: Boolean, required: true, unique: false })
    premium: boolean;
}

export const MeetsSchema = SchemaFactory.createForClass(Meets);
```

```
@Controller( prefix: 'profile')
export class ProfileController {
    constructor(private readonly service: ProfileService) {}

    @Get()
    @UseGuards(JwtGuard)
    @HttpCode(HttpStatus.OK)
    getProfile(@Req() req: Request): Observable<ProfileModel> {
        return this.service.getProfile(req['user']);
    }
}
```

```
import { Module } from '@nestjs/common';
import { AuthModule } from '../auth/auth.module';
import { UsersModule } from '../users/users.module';
import { ProfileController } from './profile.controller';
import { ProfileService } from './profile.service';

@Module( metadata: {
    imports: [UsersModule, AuthModule],
    controllers: [ProfileController],
    providers: [ProfileService]
})
export class ProfileModule {}
```

```
import { Injectable } from '@nestjs/common';
import { Observable, of } from 'rxjs';
import { Users } from '../users/schemas/users.schema';
import { ProfileModel } from './models/profile.model';

@Injectable()
export class ProfileService {
    getProfile(user: Users): Observable<ProfileModel> {
        return of(this.transformUserToProfile(user));
    }

    private transformUserToProfile(user: Users): ProfileModel {
        return user;
    }
}
```

```
export type UsersDocument = Users & Document;

@Schema()
export class Users {
    @Prop( options: { type: Number })
    id: number;

    @Prop( options: { type: String, required: true })
    firstName: string;

    @Prop( options: { type: String, required: true })
    lastName: string;

    @Prop( options: { type: Date, required: true })
    birthday: Date;

    @Prop( options: { type: Number, required: true })
    gender: number;

    @Prop( options: { type: String, required: false })
    description: string;

    @Prop( options: { type: String, required: true })
    password: string;

    @Prop( options: { type: String, required: true })
    email: string;

    @Prop( options: { type: String, required: true })
    phone: string;

    @Prop( options: { type: String, required: true })
    city: string;

    @Prop( options: { type: String, required: true })
    username: string;

    @Prop( options: { type: Number, required: true })
    role: RoleEnum;
```

```
    @Prop( options: { type: Number, required: true })
    status: StatusEnum;

    @Prop( options: { type: Array })
    socialNetworks: SocialNetworkModel[] = []

    @Prop( options: { type: Array })
    friends: string[] = []

    @Prop( options: { type: Array })
    friend_requests_received: string[] = []

    @Prop( options: { type: Array })
    sent_friend_requests: string[] = []

    @Prop( options: { type: () => Buffer })
    image: Buffer;

    @Prop( options: { type: Array })
    blackList: string[] = []
}

export const UsersSchema = SchemaFactory.createForClass(Users);
```

```typescript
@Controller( prefix: 'users')
export class UsersController {
    constructor(private readonly userService: UsersService) {}

    @UseGuards(JwtGuard)
    @Get()
    getAll(@Req() req): Observable<Users[]> {
        const currentUsername = req.user.username;
        return this.userService.getAll(currentUsername);
    }

    @Get( path: ':username')
    getByUsername(@Param( property: 'username') username: string): Promise<Users> {
        return this.userService.getByUserName(username);
    }

    @UseGuards(JwtGuard)
    @Post()
    @HttpCode(HttpStatus.CREATED)
    async create(@Body() createUser: CreateUserDto): Promise<Users> {
        return await this.userService.create(createUser).catch((error: ErrorMessageEnum) => {
            throw new HttpException(error, HttpStatus.CONFLICT);
        });
    }

    @UseGuards(JwtGuard)
    @Delete( path: ':id')
    async delete(@Param( property: 'id') id: number): Promise<Users> {
        return this.userService.delete(id);
    }

    @Put( path: ':username')
    @HttpCode(HttpStatus.OK)
    @Roles(RoleEnum.Admin)
    @UseGuards(JwtGuard, RolesGuard)
    async update(@Body() createUser: UpdateUserDto, @Param( property: 'username') username: string): Promise<Users> {
        return this.userService.update(createUser, username);
    }
}
```

```typescript
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { Users, UsersSchema } from './schemas/users.schema';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module( metadata: {
    imports: [MongooseModule.forFeature( models: [{ name: Users.name, schema: UsersSchema }])],
    controllers: [UsersController],
    providers: [UsersService],
    exports: [UsersService]
})
export class UsersModule {}
```

```
@Injectable()
export class UsersService {
    constructor(@InjectModel(Users.name) private readonly userModel: Model<UsersDocument>) {}

    public getAll(currentUsername: string): Observable<Users[]> {
        return from(this.userModel.find()).pipe(map( project: (users : UsersDocument[] ) => users.filter(({ username : string }) => username !== currentUsername)));
    }

    public async getById(id: number): Promise<Users> {
        return this.userModel.findById(id);
    }

    public async create(user: CreateUserDto): Promise<Users> {
        return this.checkUserToDuplicateEmailPhoneUsername(user).pipe(
            switchMap( project: (errorMessage: ErrorMessageEnum) => {
                if (errorMessage) {
                    return throwError(errorMessage)
                }
                return from(this.transformUser(user))
            }),
            switchMap( project: (user: Users) => {
                return new this.userModel(user).save();
            })
        ).toPromise()
    }

    public async delete(id: number): Promise<Users> {
        return this.userModel.findByIdAndRemove(id);
    }

    public async update(body: UpdateUserDto, username: string): Promise<Users> {
        return this.userModel.findOneAndUpdate( filter: { username }, body).catch( onrejected: () => {
            throw new HttpException(ErrorMessageEnum.UserNotFound, HttpStatus.FORBIDDEN)
        });
    }

    public async getByUserName(username: string): Promise<Users> {
        return this.userModel.findOne( filter: {username: username}).exec();
    }
```

```
    private async transformUser(user: CreateUserDto): Promise<Users> {
        const password = await PasswordHashUtil.hashPassword(user.password);
        return {
            ...user,
            id: UuidUtil.generateNumberId(),
            role: RoleEnum.User,
            status: StatusEnum.Pending,
            password,
            friends: [],
            blackList: [],
            friend_requests_received: [],
            sent_friend_requests: [],
            image: user.image || null
        }
    }

    private checkUserToDuplicateEmailPhoneUsername(user: CreateUserDto): Observable<ErrorMessageEnum> {
        const { phone, email, username } = user;
        return from(this.userModel.find( filter: { $or: [ {phone}, { email }, {username} ]})).pipe(
            map( project: (users: Users[]) => {
                if (users.some((user : Users ) => user.username === username)) {
                    return ErrorMessageEnum.UsernameIsUse;
                }

                if (users.some((user : Users ) => user.email === email)) {
                    return ErrorMessageEnum.EmailIsUse;
                }

                if (users.some((user : Users ) => user.phone === phone)) {
                    return ErrorMessageEnum.PhoneIsUse;
                }

                return null;
            })
        )
    }
}
```

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { mongodb } from './configs/mongodb.config';
import { AuthModule } from './modules/auth/auth.module';
import { MeetsModule } from './modules/meets/meets.module';
import { PlatformListModule } from './modules/platformlist/platformlist.module';
import { ProfileModule } from './modules/profile/profile.module';
import { UsersModule } from './modules/users/users.module';

@Module( metadata: {
    imports: [
        AuthModule,
        UsersModule,
        MongooseModule.forRoot(mongodb.url),
        ProfileModule,
        PlatformListModule,
        MeetsModule
    ],
    controllers: [AppController],
    providers: [AppService],
})
export class AppModule {}
```