КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
імені ТАРАСА ШЕВЧЕНКА
Факультет інформаційних технологій
Кафедра прикладних інформаційних систем

122 «Комп'ютерні науки»
(шифр і назва спеціальності)

«Прикладне програмування»
(назва освітньої програми)

Кваліфікаційна робота бакалавра

на тему: «Веб-застосунок з автоматизації роботи працівників готелю»

Виконав _____
(Підпис)
Погребець Сергій Олександрович
(прізвище, ім'я, по батькові)

Керівник _Шолохов Олексій Вікторович
(прізвище, ім'я, по батькові)
_____ _____
(Резолюція «До захисту»)

Попередній захист:

_____
(Висновок: "До захисту в екзаменаційній комісії ")

Завідувач кафедри ___ _____ Плескач В.Л.
(Підпис) (Прізвище, ініціали)
(Дата)

Засвідчую, що у цій дипломній роботі немає запозичень із праць інших авторів без відповідних посилань.
Унікальність тексту роботи - 98 %

Київ – 2022

ЗМІСТ

ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА

Складові частини дипломної роботи	Обсяг, арк.
Титульний аркуш	1
Відомість дипломної роботи	1
Анотація	1
Зміст	2
Перелік скорочень, умовних позначень, термінів	1
Вступ	2
Розділ 1	10
Розділ 2	11
Розділ 3	21
Перелік використаних джерел	2

				ДП ХХХХ 00.000.00		
	ПІБ	Підп.	Дата			
Розробн.	Погребець			Відомість дипломної роботи	Лист	Листів
Керівн.	Шолохов О.В.					
Н/контр.	Базилюк А.М.					
Зав.каф.	Плескач В.Л.					

Анотація

Кваліфікаційна робота: 52 с., 40 рис., 15 джерел.

Ця робота присвячена проектуванню та розробленню веб-застосунку з автоматизації роботи працівників готелю.

Ключові слова: веб-системи, веб-застосунок, розробка, автоматизація роботи.

Метою дипломної роботи є ефективне управління готелем, та автоматизація роботи працівників.

Об'єктом дослідження є процеси обслуговування відвідувачів готелю персоналом.

Предметом дослідження є програмно-технічні підходи до побудови веб-застосунків.

Отримані результати: спроектовано та реалізовано веб-застосунок з автоматизації роботи працівників готелю.

Перелік умовних позначень і скорочень

IT - information technology
HTML – HyperText Markup Language
FTP – File Transfer Protocol
TCP - Transmission Control Protocol
UDP - User Datagram Protocol
EF - Entity Framework
SMTP - Simple Mail Transfer Protocol
DNS - Domain Name System
SOAP - Simple Object Access Protocol
XML - Extensible Markup Language
CSS - Cascading Style Sheets
JS - JavaScript

Вступ

В 21-му столітті майже всі підприємства тримають долю цільової аудиторії покупців в інтернеті. Це може бути як найпростіша фейсбук-сторінка закладу, так і повноцінний сайт в інтернеті. З року в рік методи створення сайтів для підприємств та компаній вдосконалюються, рівень кваліфікації розробників підвищується, а отже й ціна за створення та підтримання такого ресурсу зростає. Через те дедалі частіше замовники звертаються до молодих команд розробників, які не мають за спинами величезного досвіду створення додатків, з надією що молода команда все ж з підказками та уточнення від замовників зможе реалізувати проект так, як його бачить замовник. Головною перевагою таких команд є доступна ціна їх роботи, порівняно з командами досвідчених розробників.

Метою даного дослідження є ефективне управління готелем та втоматизація роботи працівників. Для цього буде проведено знаходження порівняно простого, та ефективного створення веб-застосунків, а завданням побудова веб-застосунку для спрощення роботи працівникам готелю.

Об’єктом дослідження будеуть процеси обслуговування відвідувачів готелю, а предметом дослідження є методи створення архітектури веб-сервісу, та її реалізація.

Для дослідження цієї теми, я оберу декілька вже готових веб-застосунків, та порівняю якість їх реалізації, якість написаного коду, складність написаного коду, та приблизний час, який було затрачено на створення. Різні додатки створюються різними способами, і є сенс їх порівнювати, так як розвиток інформаційних технологій не стоїть на місці, і кожного року з’являються нові методи. Деякі з них є досить зручними, але підходять лише досвідченим розробникам, деякі з них створюються спеціально під потреби конкретної команди розробників та під конкретну задачу.

Розробникам початківцям, які тільки тільки починають свою кар’єру в IT, різні досвідчені розробники будуть радити різні технології для вивчення, і цією

роботою я хочу порівняти різні методи створення веб-сервісів, та поділити які технології підійдуть для початківців, а які для досвідчених програмістів.

Дослідження теми є актуальним, так як все більше компаній хочуть тримати свою частину аудиторії в інтернеті. Люди, які мають змогу оплачувати замовлення, використовуючи смартфони та комп’ютери, зменшують кількість роботи для самих працівників. Тому автоматизація таких процесів є дуже важливою частиною бізнесу в наш час.

РОЗДІЛ 1. ОСНОВНІ ПОНЯТТЯ ТА АНАЛІЗ ВЖЕ ІСНУЮЧИХ ВЕБ-СЕРВІСІВ

1.1 Основні поняття

Веб-служба або веб-сервіс (англ. web service) - це програма, яка дозволяє організувати взаємодію різних сайтів та сервісів. Сервіс може зчитувати інформацію з одного порталу, обробляти її, та відправляти на інший портал. Взаємодія можлива за допомогою передачі запитів до сервісу, які основані на ряді протоколів.

Основними протоколами є:

- FTP (англ. File Transfer Protocol) - це стандартний протокол, який використовують для передачі файлів. Часто використовується для завантаження сторінок та інших документів з сервісу на клієнт.
- SMTP (англ. Send Mail Transfer Protocol) - це широко використовуємий простенький протокол відправки електронних листів в мережах IP/TCP.
- DNS (англ. Domain Name System) - це система для отримання інформації про домени. Найчастіше використовується для отримання IP-адреси хосту по його імені.
- SOAP (англ. Simple Object Access Protocol) - протокол, який дозволяє відправляти та отримувати дані з сервісу у вигляді XML-файлу.

1.2 Відмінність веб-сервісу та веб-сайту

Веб-сайти зазвичай мають суто інформативний характер. Вони можуть вміщувати в собі текст, зображення, музику, відео. Веб-сайт складається з декількох веб-сторінок, які з'єднані один з одним в єдиний ресурс. Зазвичай мають елементарну архітектуру написану на HTML-коді з додаванням CSS-стилів.

Сайти не надають користувачам можливості взаємодіяти з програмою. Користувачі максимум можуть мати можливість введення даних в заготовлену форму для отримання підписки на оновлення сайту.

Найбільш популярними прикладами веб-сайтів є:

- Прогноз погоди;
- Онлайн газети;
- Кулінарні статті

Сайт сервіс вже надає функціонал, яким користувачі можуть користуватись. Користувач може вводити дані, отримувати дані, та маніпулювати з даними за допомогою функціоналу, використання якого дозволяє сервер.

Наприклад раніше на сайтах піцерій та кафе ви могли переглянути актуальну інформацію про ціни, час роботи закладу, відкриті вакансії. В тому випадку ви переглядали веб-сайт, а зараз ви маєте можливість зареєструватись, зробити замовлення, оплатити замовлення онлайн, накопичувати бонуси за замовлення, тобто використовувати деякий функціонал веб-сервісу.

Також веб-сервіси можуть взагалі не нести інформативного характеру, а тільки використовуватись для вирішення поставлених задач. Наприклад до веб-сервісів можна віднести:

- переклад тексту на іншу мову;
- перерахунок з однієї валюти на іншу;
- переведення з бінарного коду в десятковий.

Так як веб-сервіси досить часто використовують авторизацію, то користувачам потрібно вводити свої персональні дані, що покладає на плечі розробників веб-сервісів додаткову відповідальність, так як втрата персональних даних часто призводить до небажаних наслідків. Щоб уникнути таких прикрих інцидентів, розробники винаходять все більш складні та надійні методи шифрування та зберігання даних користувачів.

Основні відмінності веб-сервісу від веб-сайту:

- Веб-сайт зазвичай є джерелом інформації, коли веб-сервіс дозволяє працювати в інтерактивному режимі;
- Веб-сервіс може бути частиною сайту, або самостійним ресурсом;
- Функціональність веб-сервісу набагато вища;

- Для розробки веб-сервісу девелопери потребують набагато потужнішу техніку та інтрументи, отже розробляти веб-сайти набагато легше та дешевше ніж веб-сервіси.

1.3 Архітектура веб-сервісів

Забігаючи наперед, можу сказати що існує два основних архітектурних стилі які використовуються при побудові веб-сервісів, це:

- Монолітна архітектура;
- Мікросервісна Архітектура.

В побудові сучасних веб-сервісів, розробники все частіше й частіше віддають перевагу мікросервісній архітектурі. Цей стиль дозволяє структурувати програму як набір сервісів, кожен з яких відповідає за свою часточку функціоналу. Кожен такий сервіс є абсолютно автономним, і не залежить від інших сервісів, що дозволяє різним командам працювати над створенням різних сервісів. Таким чином, якщо ми захочемо переробити, або навіть прибрати будь який з сервісів, ми можемо це зробити, не завдавши шкоди роботі всіх інших сервісів, та ніяким чином не вплинувши на їх роботу.

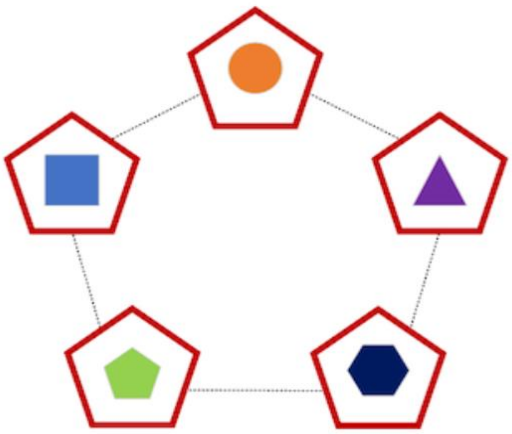

Рисунок 1.1 – Мікросервісна архітектура

Іншим підходом до побудови веб-сервісу є використання монолітної архітектури, яка є вигідна тим, що всі компоненти в такій архітектурі є узгодженими, і тому проблеми при передачі даних від одного компонента до іншого виникають набагато рідше. Така перевага є одночасно і недостатком,

так як при зміні одного компонента, можуть виникнути помилки в інших, і тому тому монолітна архітектура є абсолютно не гнучкою, через що розробники все частіше надають перевагу саме мікросервісній архітектурі.

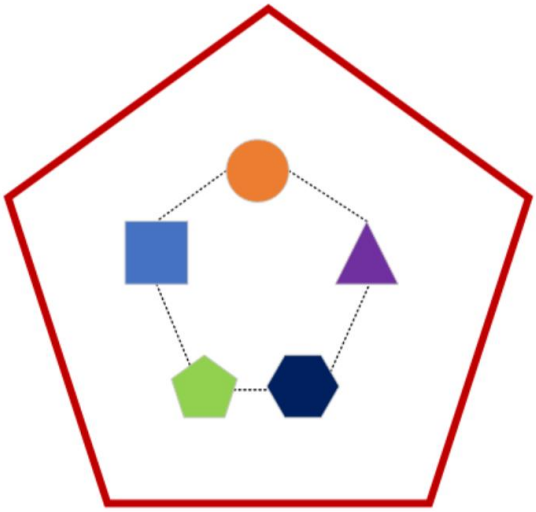

Рисунок 1.2 – Монолітна архітектура

Якщо брати вибірку старіших проектів, то ми побачимо що більшість з них написані використовуючи якраз монолітну архітектуру. Це зумовлено тим, що монолітні проекти зазвичай зберігаються в одному репозиторії, і над такими проектами працює одна команда розробників, яка дуже добре орієнтується в написаному коді, точно знає яка функція за що відповідає, і який файл треба відкрити щоб її змінити. Розширення таких проектів зазвичай довіряють тим самим командам. Цей проект можна передати під управління іншої команди, але в такому випадку буде використано більше часу на освоєння вже написаного коду, і в разі виявлення якогось багу, командою буде затрачено чимало часу щоб його виправити.

Елементи мікросервісної архітектури можуть зберігатися в різних репозиторіях, над ними можуть працювати різні команди розробників. Освоєння вже написаного коду буде займати набагато меншу кількість часу, так як команді розробників доведеться розбиратись в набагато меншій кількості написаного коду. Помилки найчастіше виникають при транспортуванні даних від сервісу до сервісу, але їх дуже легко виявити та знайти в якому саме елементі архітектури відправляється або приймається неправильний тип даних.

Веб-сервіси з монолітною архітектурою часто пишуться на одній, або небагатьох мовах програмування, коли кожен для кожного елементу мікросервісної архітектури можна обрати найбільш підходящу мову, що також спрощує написання коду, та створює додаткові можливості для розширення вже написаного.

Також вагомим аргументом на користь мікросервісної архітектури є те, що при зміні коду з монолітною архітектурою, нам потрібно буде перекомпілювати весь проект, що дуже часто є довгим та ресурсозатратним процесом. Окрім затрат в часі, розробники мають загрозу отримати неочікувану критичну помилку. Такі помилки можуть призводити навіть то пошкодження даних користувачів, тому частота зборки коду з монолітною архітектурою набагато менша ніж з мікросервісною.

1.4 Принципи SOLID

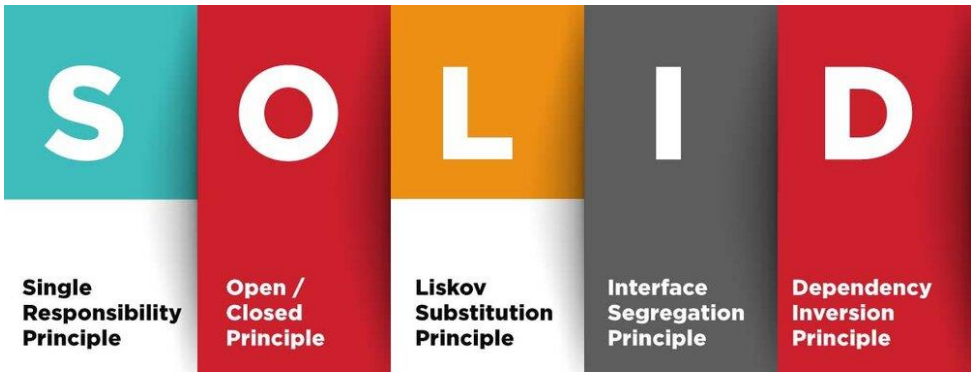

Рисунок 1.3 – SOLID

З появою об’єктно-орієнтованого програмування підходи до побудови структури додатків почали змінюватись. Почало з’явилось багато патернів проектування, була виділена функціональність, яка широко використовувалась у багатьох проектах. Виділення такої функціональності дало змогу розробити стандартні класи, які є доступними для розробника з самого початку створення програми.

З розширенням проекту кількість написаного коду накопичується, і розробникам, стає важко орієнтуватись в коді. Саме для того, щоб спростити розуміння коду було створено патерни програмування. Дотримання патернів дозволяло розробникам орієнтуватись в написаному іншими розробниками коді,

а від того витрати на розробку зменшувалися. Програмісти витрачали менше часу та сил на дописування коду та знаходження помилок.

Власне дотримання структури патерна не гарантувало що код буде читабельним та легко зрозумілим. Паттерн дозволяє побудувати загальну структуру, але читабельність коду, в першу чергу, залежить від майстерства програміста. Саме з метою допомогти всім бажаючим розробляти якісне програмне забезпечення, Роберт Мартін (більш відомий серед програмістів як Uncle Bob) виділив п’ять основних принципів об’єктно-орієнтованого програмування та проектування, перші букви назв яких утворюють акронім SOLID.

Акронім SOLID розшифровується так:

- S - Single Responsibility Principle
- O - Open-Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Якщо при проектуванні та розробці структури розробник дотримується цих принципів, то код виходить малозв’язаний, читабельний та стійкий.

1.4.1 Single Responsibility Principle

Перший принцип - це принцип єдиної відповідальності. Він полягає в тому, що один клас має відповідати за виконання одного завдання. Якщо клас відповідає за виконання декількох задач, то його нащадки, які будуть його реалізовувати, виявляться зв’язаними. І зміна головного класу призведе до необхідності змін в дочірніх класах. Але якщо розділяти функціональність так, щоб для кожної задачі був виділений спеціальний клас, ми уникнемо таких проблем.

Уявимо що ми маємо клас Car, який дозволяє зберігати машину в базі даних, виводити про неї інформацію, заправляти її і тд. В такому вигляді клас Car порушує перший принцип, бо має функціональність як для роботи з базою даних(зберігання машини), так і функціональність виводу інформації. Для того щоб виконувався перший принцип, ми маємо розділити цей клас на два класи.

Перший відповідатиме за роботу з базою даних, а другий за вивід та обробку інформації.

1.4.2 Open-Closed Principle

Принцип відкритості-закритості формулюється так: “Об’єкти(функції, класи, модулі) мають бути відкриті для розширення, але закриті для модифікації”.

На рисунку 1.4 ми бачимо клас Human та функцію TellSomething. В такому вигляді написаний код порушує принцип відкритості-закритості, так як якщо будуть створені об’єкти класу Human, імена яких відрізнятимуться від “Serhii” або “Helga” нам доведеться восити зміни в функцію TellSomething. Щоб принцип виконувався, нам варто зберігати текст повідомлення в самому класі Human, а в функції TellSomething лише його виводити. Код з дотриманими умовами виконання принципу показаний на рисунку 1.5.

```
public class Human
{
    2 references
    public string Name { get; set; }
}

0 references
public static void TellSomething(Human human)
{
    if(human.Name == "Serhii")
    {
        Console.WriteLine("I`m programer!");
    }
    if(human.Name == "Helga")
    {
        Console.WriteLine("I love game God of War!");
    }
}
```

Рисунок 1.4 – Клас Human та функція TellSomething

```
public class Human
{
    0 references
    public string Name { get; set; }
    2 references
    public string Message { get; set; }
    0 references
    public Human(string text)
    {
        this.Message = text;
    }
}

0 references
public static void TellSomething(Human human)
{
    Console.WriteLine(human.Message);
}
```

Рисунок 1.5 – Переписаний код, який відповідає принципу відкритості-закритості

1.4.3 Liskov Substitution Principle

Принцип полягає в тому, що дочірні класи мають доповнювати функціонал батьківського класа, а не заміщати його. Дотримання цього принципу робить можливим використання дочірнього класу замість батьківського, та попереджує появу небажаних помилок при заміщенні.

1.4.4 Interface Segregation Principle

Принцип розділення інтерфейсів говорить про те, що розробник має створювати вузьконаправлені інтерфейси призначені для виконання лише однієї конкретної дії. При написанні сервісів дуже часто виникає ситуація, що різні класи можуть мати спільний функціонал. Щоб двічі не прописувати одні і ті ж функції, розробники мають змогу створити клас зі спільними функціями, але потім виявляється що окрім необхідного функціоналу, об’єкт може використовувати функції, які абсолютно не відносяться до нього по логіці. Наприклад розробник може створити інтерфейс IAnimal, який буде мати функції Run, Swim та Fly. Якщо ми створюємо клас Duck, який наслідує інтерфейс IAnimal, то все виглядає логічно, так як качка може і плавати, і літати, і ходити, вле якщо ми створимо клас Dog та також унаслідуємо інтерфейс IAnimal до

нього, то ми побачимо, що деякі функції не підходять для собаки. Якщо функції Run та Swim ніяк не суперечать логіці, то функція Fly для собаки абсолютно не підходить. Тому на самому початку розробнику краще розділити інтерфейс IAnimal на IRunable, ISwimable та IFlyable, і вже при створенні конкретних класів таких як Duck та Dog наслідувати необхідні інтерфейси.

1.4.5 Dependency Inversion Principle

Принцип інверсії залежностей говорить про те, що об’єктом залежності має бути абстракція, а не якийсь конкретний об’єкт. Також є правило, що реалізовувати клас потрібно залежно від інтерфейсу, а не підлаштовувати функції інтерфейсу під конкретну реалізацію. Цей принцип дуже доцільно використовувати в сервісах, які складаються з декількох модулей, так як він буде попереджувати проблеми залежностей високорівневих модулів від низькорівневих. Якщо все буде базуватись на абстракції, проблем з заміщенням класів та заміною функціоналу не буде.

ВИСНОВОК ДО РОЗДІЛУ 1

Сфера IT дуже швидко розвивається, так само як і росте рівень знань програмістів. Інструменти для розробки стають все досконалішими, а архітектури все більш стійкими. Дотримання обраної заздалегідь структури проекту дозволяє команді дуже легко орієнтуватись в коді, і навіть при зміні членів команди дозволяє досить швидко увійти в курс справи новачкам. Також важливо дотримуватись принципів SOLID, які допоможуть команді підтримувати малозв’язаність коду та його тримати його якість на високому рівні.

Порівнявши ознаки монолітної архітектури з ознаками мікросервісної архітектури можу сказати, що в сучасних проектах все частіше й частіше розробники будуть віддавати перевагу саме мікросервісній архітектурі, так як вона не тільки дозволяє одразу декільком командам працювати над проектом, але й є більш надійною.

РОЗДІЛ 2. ОПИС ОБРАНИХ ТЕХНОЛОГІЙ ТА ІНСТРУМЕНТІВ

2.1 C#

C# це об'єктно-орієнтована мова програмування, яка була створена командою розробників з компанії Microsoft. В основі створення цієї мови програмування лежить всім відома мова C. Головною метою команди розробників було створення універсальної мови програмування, яка підходила б для вирішення будь яких задач та підходила б для всіх операційних систем та пристроїв.

Багато програмістів стверджують що C# схожий на комбінацію двох інших мов програмування: Java та C++. Якщо трошки поглибитись в нього то можна й справді виділити деякі особливості, які C# запозичив у Java. Головними особливостями, які можна виділити, є:

- Підтримка поліморфізму;
- Підтримка перевантаження операторів;
- Використання атрибутів, подій та делегацій;
- Використання анонімних функцій;
- Статична типізація.

Досить довго розробники вважали що мову C# можна використовувати лише для створення десктопних додатків, для операційної системи Windows, але насправді C# давно вийшов за ті рамки, і тепер розробники мають змогу створювати додатки для різних операційних систем, ігри, веб-сервіси, програмне забезпечення для захисту та навіть мобільні додатки.

Мова C# стала популярна для створення ігор саме завдяки платформі Unity. Саме мова програмування C# найбільше адаптована для роботи з нею, тому програмісти при виборі мови для роботи з Unity одразу обирають саме її.

Ключевою перевагою C# є її простота. Саме через простоту синтаксису багато програмістів віддають перевагу цій мові, а новачкам радять першою мовою для програмування обирати саме C#. Популярність мови програмування залежить від кількості програмістів, які на ній пишуть. А так як матеріалу для

вивчення C# більш ніж достатньо, це гарантує що мова C# буде популярною ще досить довго.

Написаний код зберігають в файлах з розширенням .cs. Якщо ми створимо консольну програму, то по дефолту в нас буде створений один єдиний файл Program.cs в якому буде створена функція Main.

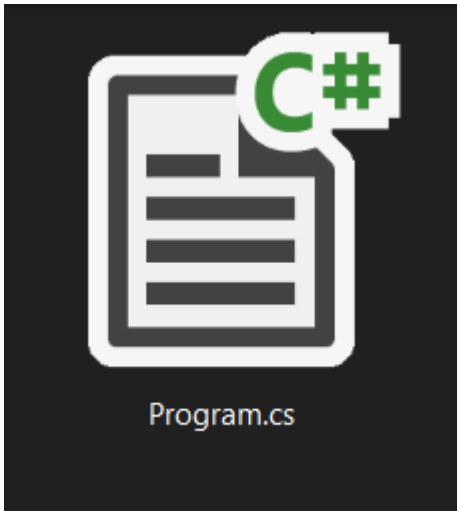

Рисунок 2.1 – Файл Program.cs

Як і у всіх мовах програмування програміст може створювати змінні. На відміну від Java, де ми маємо в розпорядженні 8 примітивних типів даних, C# дозволяє використовувати 15 примітивних типів даних:

- bool;
- byte;
- sbyte;
- int;
- uint;
- float;
- decimal;
- char;
- short;
- ushort;
- long;
- ulong;
- string;

- object;

Окрім примітивних типів даних, стандартні бібліотеки C# дозволяють створювати колекції, такі як стек, черга, словник. Також варто відзначити що мова є реєстрозалежною, що досить часто викликає помилки. Якщо назва функції починається з великої букви, а при виклику її написати назву з маленької, то програма не буде скомпільована, і розробник отримає повідомлення що при виклику функція не була знайдена.

Є також деякі можливості спільні для мови C# та для Java, наприклад автоматизоване вивільнення ресурсів, одиночне наслідування, можливість створювати інтерфейси. Особливої уваги заслуговує автоматизоване вивільнення ресурсів, бо запобігає витоку пам’яті. Таких механізмів не мають низькорівневі мови програмування, такі як C++.

2.2 ASP.NET

З часу виходу першої версії .NET пройшло вже багато років, і з’явилося чимало нових рішень для різних типів задач. Спеціально для розробки веб-застосунків було створено платформу ASP.NET. Незважаючи на те, що ASP.NET не приніс з собою якоїсь революції в створенні веб-сервісів та веб-застосунків, він досить швидко завоював популярність серед розробників.

Однією з причин популярності стала інтеграція з .NET Framework. Через цю інтеграцію ASP.NET має доступ до величезної кількості вже написаних класів. Кожен клас розміщується в спеціальному ієрархічному контейнері. Тані контейнери мають назву простори імен. Ці контейнери мають логічну структуру, тому навіть розробники які не користувались платформою раніше зможуть дуже швидко освоїти використання функціоналу запропоноване .NET Framework.

Різні простори імен зберігають класи з різною функціональністю, але в цілому вони мають вже готові розв’язки для майже будь якої задачі. Хоча й наявність готових розв'язків ніяк не забороняє розробникам розробити своє власне рішення. Всі разом ці розв’язки мають назву бібліотека класів.

Думаю пару слів треба сказати про CLR. Common Language Runtime - це середовище виконання коду яке підходить для споріднених з .NET мов програмування. Для CLR підходять такі мови як:

- C#;
- F#;
- Visual Basic;
- Managed C++.

ASP.NET підтримує багато мов програмування. Це стає можливим через те, що написаний код спочатку компілюється в MSIL або просто IL. MSIL розшифровується як Microsoft Intermediate Language, і можна сказати що саме IL є єдиною мовою .NET, так як CLR розпізнає лише його.

При компіляції написаний код проходить через два етапи. Перший етап - це перетворення написаного на C# коду, в код на мові IL. Другий етап компілювання полягає вже в низькорівневий машинний код. Після другого етапу компіляції програма вже готова до запуску. Так як процес компіляції складається з двох етапів, компілятор, перед другим етапом, може перевірити в якій операційній системі та на якому пристрої буде запускатись програма. Це є причиною того, що .NET підходить не тільки для програмування для операційної системи Windows.

Так як ASP.NET обслуговується середовищем CLR, то вона має доступ до всіх переваг .NET. Автоматичне очищення пам’яті не є виключенням. При створення об’єкту класу CLR автоматично виділяє необхідну кількість пам’яті, і так само, коли об’єкт зникає з зони видимості, він попадає під вивільнення пам’яті. Це звільняє програмістів від необхідності ручної очистки пам’яті. Також за допомогою бібліотеки класів ми можемо легко створювати асинхронні функції, які досить потрібні при розробці веб-застосунків.

ASP.NET є чудовим прикладом використання об’єкто-орієнтованого підходу. Вона дозволяє використовувати абсолютно всі концепції об'єктно-орієнтованого програмування. ASP.NET працює з абсолютно всіма браузерами. При розробці веб-сервісів перед розробником постає справжнє випробування,

йому потрібно продумати абсолютно всі варіанти налаштувань таким чином, щоб інформація правильно відображалась на всіх пристроях. Один із шляхів вирішення цієї задачі, це перебирати такі варіанти, та підлаштовуватись під кожну конфігурацію окремо. В ASP.NET це вирішено. Розмітка для браузерів генерується автоматично, опираючить на конфігурацію та беручи до уваги всі особливості клієнта.

І нарешті найголовнішою перевагою ASP.NET є те, що платформа дуже спрощує процес налаштування та запуску готового веб-сервісу. Розгортання готового додатку є мабуть найважчим процесом для розробника. Доводиться переносити багато файлів, знову налаштовувати бази даних, змінювати багато конфігурацій. Цей процес потребує досить таки багато часу та сил, але якщо на сервері встановлений .NET Framework, то робити якісь надскладні речі розробникам не доведеться.

2.3 HTML

HTML є невід’ємною частиною створення будь яких ресурсів в інтернеті. HTML це мова створення гіпертекстових документів, а не мова програмування, як часто думають новачки в веб-розробці. Якщо ви відкриєте будь який файл з розширенням .html в текстовому редакторі, то побачити що це просто текст, але деякі слова інколи заключаються в квадратні дужки. Саме слова, заключені в квадратні дужки, являють собою керуючі елементи. Такі елементи називаються теги.

Теги дозволяють браузеру зрозуміти яким чином йому потрібно показувати інформацію. Де потрібно розділити текст, де вставити фотографію, а де додати таблицю. HTML-документ має дуже просту структуру. Файл починається тегом <html>, а закінчується тегом </html>. Вся інформація, яка знаходитиметься за межами цих двох тегів, буде просто проігнорована браузером. Код всередині тегів <html> та </html> поділяється на дві частини:

- Заголовок;
- Тіло.

Заголовок починається тегом <head>, а закінчується тегом </head>. Всередині цієї секції, як правило, підключаються файли стилів та зберігається мета-інформація.

Тіло починається тегом <body>, а закінчується тегом </body>. В тілі знаходиться весь контент, який буде відображати браузер. Варто також зазначити, що тіло відповідає саме за інформацію, а за те яким чином буде відображена ця інформація, відповідають файли стилів.

2.4. CSS

CSS або каскадні таблиці стилів - це мова, яка дозволяє описувати те, яким чином браузер має відображати контент, описаний в html. Ціллю створення мови CSS було розділення описання вмісту веб-сторінки, та опис зовнішнього вигляду .веб-сторінки.Відділення файлу стилів від сторінки html дозволяє розділити стилі на декілька файлів та дати їм різні відповідні назви. Розробникам буде набагато простіше орієнтуватись в декількох файлах, коли навіть по назвах можна зрозуміти, що описують відповідні стилі.

Існує декілька варіантів підкллючення css-файлів. Якщо стилі описані в окремому файлі, то в файлі html, всередині елементу <head> ми можемо прописати тег <link>, вказавши всередині <link> що це css-файл, та шлях до нього. Другий спосіб полягає в тому, що ми можемо додати тег <style> до тіла елементу <head>, і всередині нього описувати стилі. Якщо ми вже маємо готовий файл CSS, і в нас є тег <style>, то ми можемо використати інструкцію @import, вказавши шлях до файлу. Стилі також можуть описуватись всередині кожного тегу окремо. Наприклад для кожного параграфу ми можемо прямо в файлі html додати свій стиль, просто додавши атрибут <style>.

Переваги використання CSS:

- Так як всі стилі знаходяться в окремому файлі, розробнику дуже легко редагувати дизайн. Немає потреби змінювати кожну сторінку окремо, достатньо лише змінити файл CSS;

- Швидке завантаження сторінок. Через те, що стилі знаходяться окремо, браузер завантажує файл зі стилями також окремо. Окрім того файл CSS може бути закешованим, та використовуватись браузером повторно;
- Використання CSS дозволяє створити різні дизайни веб-сторінок для різних пристроїв. Користувач може відвідати сайт як зі смартфону, так і з комп’ютера. Без використання CSS дуже часто виникають помилки відображення інформації.

Недоліки використання CSS:

- Розділення контенту та дизайну на різні файли призводить до того, що коли розробники змінюють файл html, вони вимушені так само змінювати й файл CSS. Інколи, працюючи з великим проектом, розробнику треба витратити досить багато часу для того, щоб знайти який саме файл css потрібно редагувати;
- Деякі старі браузери підтримують не всей функціонал CSS, через те різні браузери можуть показувати інформацію по різному.

2.5 ASP.NET MVC

MVC - це архітектурний шаблон. Взаємодія користувача та сервера відбувається за допомогою надсилання http запитів, та повернення відповідей користувачу. Користувач заповнює форму, або виконує якусь дію, яка відправляє запит за сервер, сервер опрацьовує запит, змінює модель даних на самому сервері, а в відповідь відправляю вже готове представлення, яке відображається в браузері.

Шаблон MVC складається з 3 частин:

- Model;
- View;
- Controller.

Контроллер відповідає за шляхи зв’язку між користувачем та сервісом, модель відповідає за набір даних, з яким користувач може взаємодіяти, а представлення відповідає за те, яким чином інформація буде відображена користувачу.

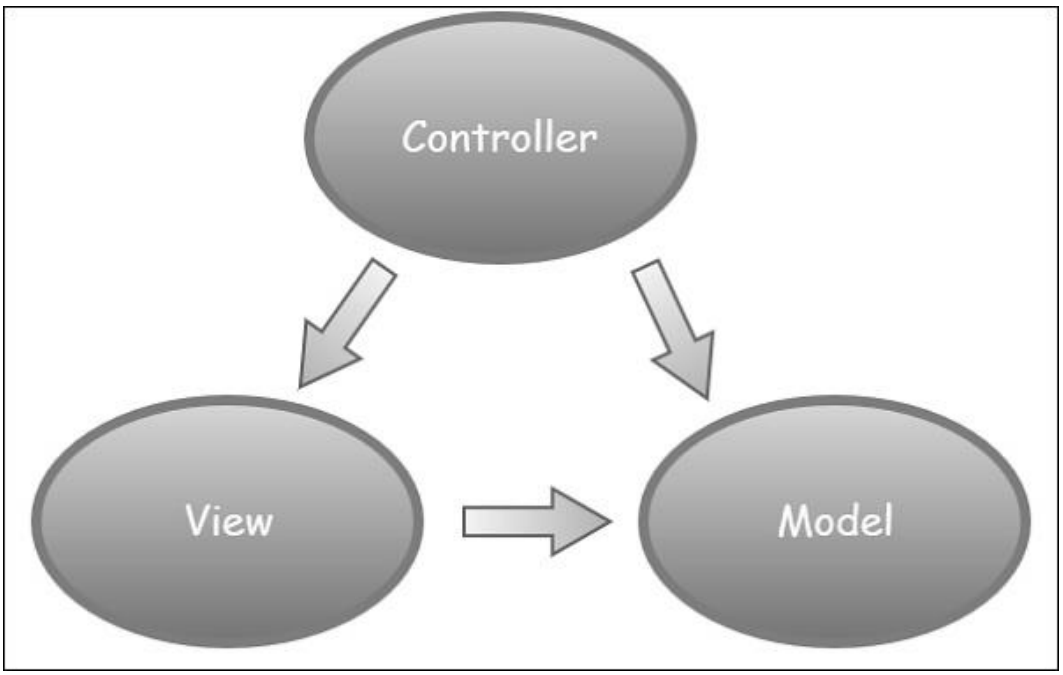

Рисунок 2.2 – MVC

Платформа розробки ASP.NET MVC реалізує шаблон MVC, а також дозволяє розділити відповідальність за різні процеси на різні рівні програми. Один рівень може відповідати за роботу з базою даних, другий за відправку запитів та отримання відповідей з інших сервісів, третій за те, яке саме представлення відправляти користувачу.

ASP.NET MVC передбачає підключення бібліотек для створення інтерфейсу користувача. Такі бібліотеки як Bootstrap або jQuery. Підтримка jQuery дійшла до такого рівня, що ASP.NET MVC вона включена до набору стандарних бібліотек, і не потребує якогось додаткового підключення.

Рисунок 2.3 – Логотип ASP.NET

Через високу розподіленість функціоналу, проекти, написані з використання технології ASP.NET MVC досить легко тестувати. Навіть програмісти, які ніколи раніше не писали тестів, можуть , використовуючи Visual Studio, використовувати механізми автоматичного створення тестів.

ASP.NET пропонує багато вже готових рішень для вирішення таких завдань, як наприклад додавання ролей користувачів, аутентифікація, конвертація даних в різні формати, зміна мови.

Також платформа має відкритий код, який розповсюджується відповідно до відкритої ліцензії Microsoft. Розробники мають можливість завантажувати та змінювати початковий код, щоб отримати власну версію платформи. Це може бути корисно, якщо розробника не влаштовує один з компонентів платформи, і він хоче замінити його функціонал.

2.6 MS SQL

Microsoft SQL Server - це система управління реляційними базами даних, розроблена компанією Microsoft. Такі системи створюються з метою спрощення роботи з базами даних. Перша версія MS SQL вийшла ще в 1989 році, але компанія й до сьогодні продовжує покращувати свій продукт.

Реляційна база даних - це база даних, в якій набори даних організовані в виді таблиці. Кожний стовпчик зберігає певний тип даних, а рядок зберігає набір значень, який відповідає якійсь сутності або об’єкту. Рядки можуть мати унікальне значення-ідентифікатор, яке має назву первинний ключ. Також таблиці мають зовнішні ключі, за допомогою яких таблиці можуть встановлювати зв’язки. Є багато способів отримання доступу до даних, але для кожного з способів немає потреби реорганізовувати базу даних.

Переваги MS SQL:

- Зручний пошук. Є можливість проводити пошук по словам, по тексту, по індексам;
- Обробка транзакцій відбувається в інтерактивному режимі. Також існує динамічне блокування;
- Велика швидкість зчитування даних;

- Можливість співпрацювати з іншими програмами Microsoft, такими як Excel та Access;
- Масштабованість системи. Система автоматично підлаштовується під параметри комп’ютера, на якому відбувається робота з системою. В залежності від потужності комп’ютера, змінюється кількість оброблюваних запитів;
- Автоматизовано такі задачі як автоматичне керування пам'яттю та блокуваннями.

Недоліками MS SQL є:

- MS SQL в більшості випадків використовується для роботи з операційною системою Windows;
- Використовує лише один механізм зберігання даних;
- MS SQL ліцензований і вимагає придбання ліцензії для використання та запуску декількох баз даних.

ВИСНОВОК ДО РОЗДІЛУ 2

Другий розділ описує технології, які будуть використані при розробці веб-додатку для спрощення роботи працівників готелю. Було описано переваги та недоліки таких технологій як html, css, ASP.NET. Розібрано переваги та недоліки архітектурного шаблону MVC, а також аргументовано вибір системи управління базами даних MS SQL. Кожен з інструментів було досить детально досліджено, аргументована актуальність використання саме тієї технології.

РОЗДІЛ 3. СТВОРЕННЯ ВЕБ-ЗАСТОСУНКУ З АВТОМАТИЗАЦІЇ РОБОТИ ПРАЦІВНИКІВ ГОТЕЛЮ

3.1 Постановка задачі

Готовий веб-застосунок з автоматизації роботи працівників готелю мусить вміщати в собі інтуїтивно зрозумілий інтерфейс, готову базу даних, готову систему аутентифікації, можливості, розділені по ролях.

Для користувача мусить бути розроблена функціональність, яка дозволить йому зареєструватись на сайті, вибрати потрібний готель, вибрати потрібну кімнату, вибрати час приїзду. Також в користувача буде можливість переглядати вже створені замовлення, в разі необхідності відміняти їх, оплачувати замовлення. Всі дані мають зберігатися на сервері, та бути захищеними так, щоб інші користувачі не мали можливості отримати до них доступ.

Адміністратори та менеджери матимуть більше можливостей ніж звичайні користувачі. Крім функцій, доступних звичайним користувачам, менеджери зможуть переглядати інформацію про всіх користувачів, в них буде доступ до створених замовлень, а також можливість редагувати замовлення. Адміністратори, окрім функцій користувачів та менеджерів, матимуть можливість редагувати інформацію про готелі та кімнати. Також в них буде доступ до функціоналу зміни ролей. Тобто адміністратори зможуть надавати зареєстрованим акаунтам доступ до різних можливостей.

Всі дані про користувачів, їхні ролі, їхні замовлення зберігаються в базі даних на сервері, доступ до якої можна отримати лише через надсилання http-запитів на сервер.

Дані, отримані після аутентифікації та авторизації користувача, мусять відправлятися на пристрій користувача, кешуватися, та авторизовувати користувача кожного разу коли він буде заходити на сайт.

3.2 Підготовка до створення додатку

Для розробки проекту я обрав середовище розробки Visual Studio 2019. Ця програма є найбільш універсальною, так як дозволяє створювати додатки майже всіх типів. Програма розроблена командою з компанії Microsoft, тому має найбільшу підтримку для створення .NET застосунків.

Однією з переваг Visual Studio є вміст вбудованого Web-серверу. Так як для запуску додатків ASP.NET необхідно мати Web-сервер, середовище Visual Studio дозволяє легко вирішити цю проблему, без встановлення зайвих компонентів. Це також впливає на безпеку, так як інші комп'ютери не мають змогу отримати доступу до сервісу, бо запити приймаються тільки з локального комп'ютера.

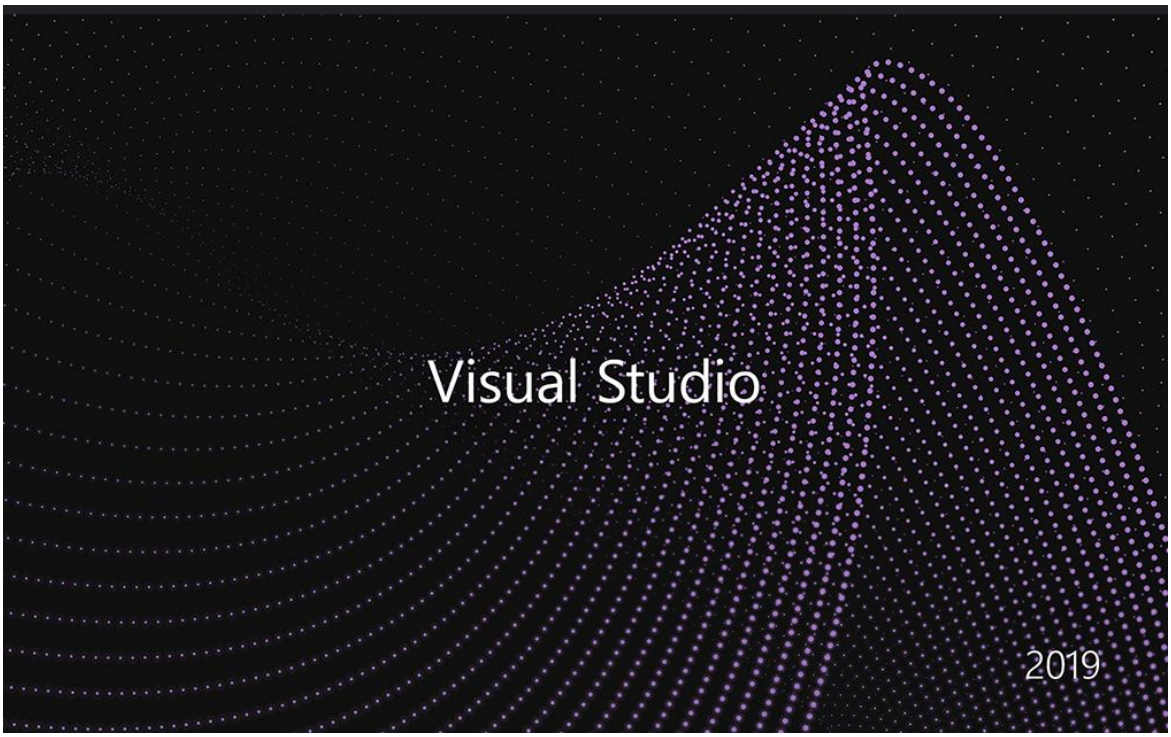

Рисунок 3.1 – Visual Studio 2019

Також перевагою Visual Studio є автоматичне форматування коду. По мірі написання коду програма сама додає відступи, вирівнює параграфи. Також є можливість запустити код в режимі налагодження. Так розробник зможе покроково перевіряти як змінюються значення змінних, чи зберігаються дані в базі, чи не з'являється помилка при введені різних даних.

Для початку нам потрібно створити проект. Visual Studio дозволяє обрати шаблон проекту ASP.NET MVC, та одразу довантажує потрібні бібліотеки.

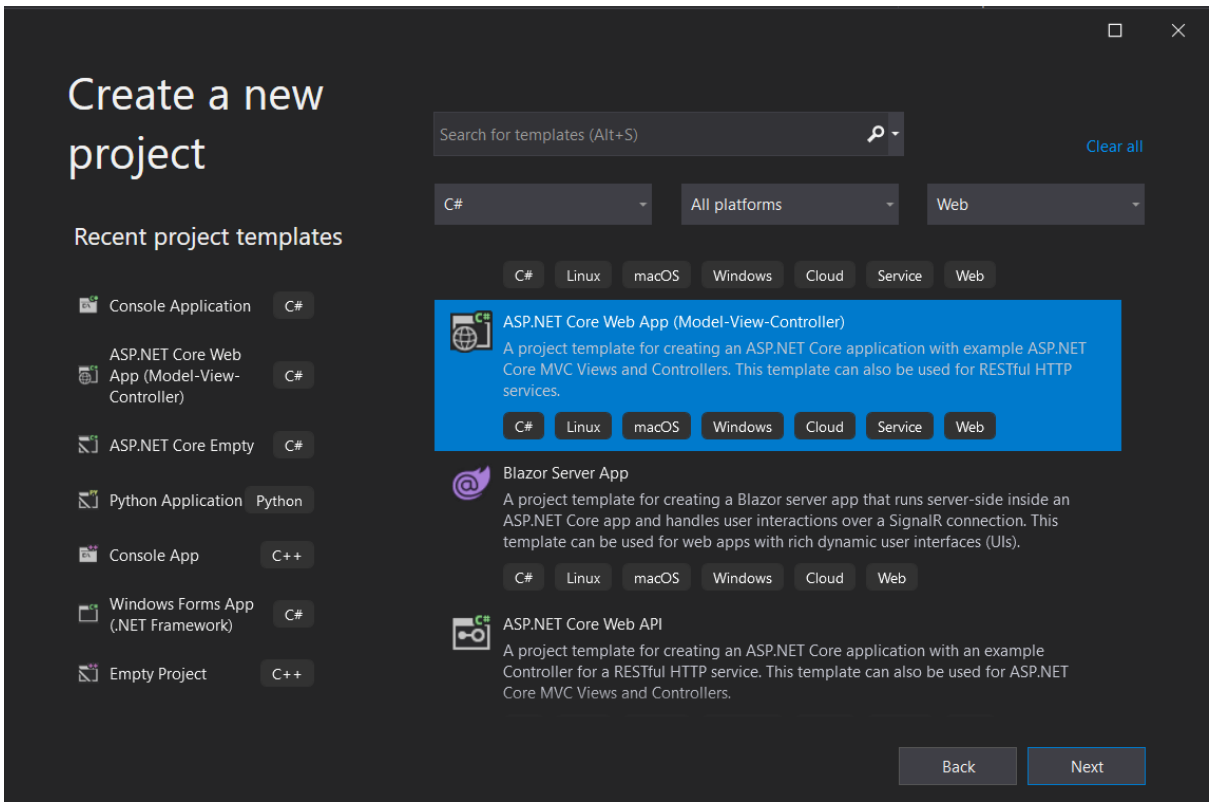

Рисунок 3.2 – Створення проекту MVC

Після обрання назви, та обрання директорії, де буде зберігатися наш проект, ми можемо його запустити. Нам відкриється шаблон майбутнього додатку, який ми будемо редагувати.

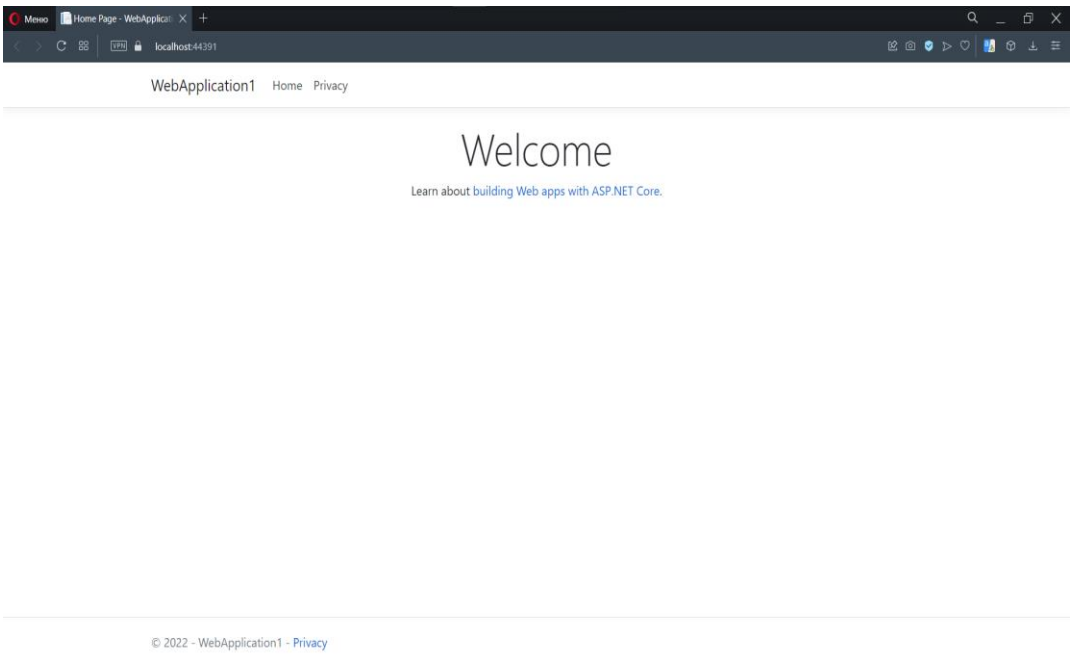

Рисунок 3.3 – Початкова сторінка проекту MVC

3.3 Обрання дизайну

Обрання підходящої палітри грає важливу роль в створенні сайту. Кольори мають бути обрані такі, щоб не створювали зайвого навантаження на очі. З багатьох палітр для свого проекту я обрав “кавову” палітру, так як вона підходить по стилю. В готелі для відвідувачів має бути створена комфортна атмосфера, так само комфортно користувачі мають почувати себе відвідуючи сайт готелю.

Рисунок 3.4 – Кольорова палітра додатку

3.4 Додавання авторизації

Наступне про що варто було подумати, це про авторизацію на сайті. Про ролі (адміністратор, менеджер, клієнт) людей, які будуть користуватись веб-сервісом. Microsoft має два варіанти рішення:

- Membership API
- ASP.NET Identity

Так як перший варіант є застарілим, я одразу обрав другий варіант – ASP.NET Identity.

ASP.NET Identity – це новий API-інтерфейс від компанії Microsoft для керування користувачами у додатках ASP.NET, покликаний замінити застарілий підхід на основі Membership API.

ASP.NET Identity можна використовувати з усіма фреймворками, такими як ASP.NET MVC, Web Forms, Web Pages. За замовчуванням ASP.NET Identity зберігатиме інформацію про користувачів у базі даних. У якому тепер також можна зберігати інформацію про різних постачальників сховища відповідно до вимог програми. Різними постачальниками даних можуть бути SharePoint, служби таблиць Azure.

Авторизація на основі ролі також відіграє важливу роль в ASP.NET Identity. ASP.NET Identity дозволяє легко створювати такі ролі, як «Адміністратор», «Клієнт”, та додати користувачів до цієї ролі, що також допомагає нам обмежувати доступ користувачам до всіх частин програми.

Щоб додати до нашого проекту ASP.NET Identity, нам потрібно відкрити менеджер пакетів nuget, знайти потрібний ресурс, та завантажити його до проекту.

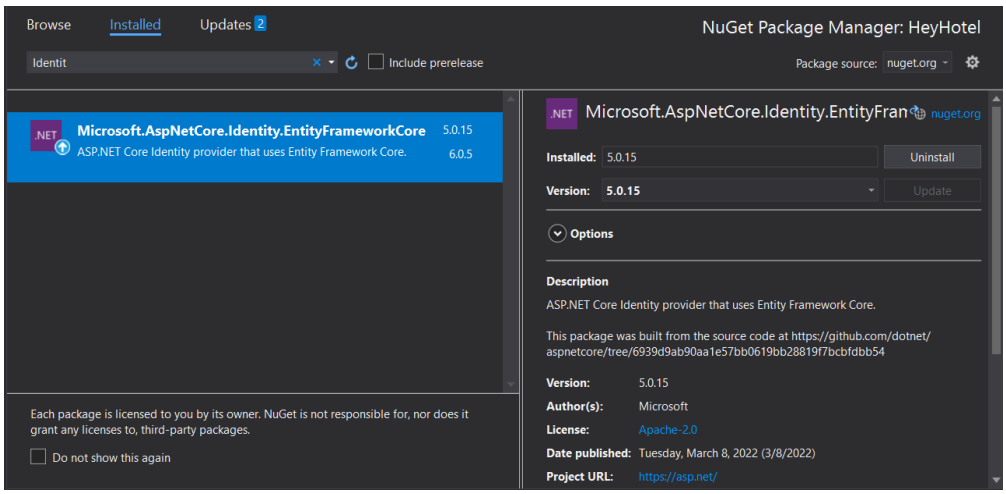

Рисунок 3.5 – Встановлений пакет ASP.NET Identity

Після встановлення пакету, нам треба дещо налаштувати проект. По-перше, нам потрібно створити клас, який буде відповідати за підключення до бази даних, а також клас, який буде представляти користувача.

ASP.NET Identity вже пропонує готовий клас User, але залишає можливість розробнику розширити його. В моєму випадку мені окрім існуючих властивостей, потрібно ще зберігати ім’я, прізвище, та рік народження користувача. Тому в папці Model я створив клас User так, як це показано на рисунку 3.6.

```
namespace HeyHotel.Models
{
    39 references | SerGobec, 75 days ago | 1 author, 3 changes
    public class User : IdentityUser
    {
        5 references | SerGobec, 84 days ago | 1 author, 1 change
        public string Name { get; set; }
        5 references | SerGobec, 84 days ago | 1 author, 1 change
        public string SName { get; set; }
        4 references | SerGobec, 84 days ago | 1 author, 1 change
        public int Year { get; set; }

        List<Order> Orders = new List<Order>();

        2 references | SerGobec, 84 days ago | 1 author, 1 change
        public User()
        {
            Orders = new List<Order>();
        }
    }
}
```

Рисунок 3.6 – Клас User

Для роботи з базою даних я створив клас UsersDbContext, який є нащадком стандартного класу IdentityDbContext. В конструкторі класу додана функція, яка, при першому зверненні, буде автоматично створювати базу даних.

```
namespace HeyHotel.Models
{
    8 references | SerGobec, 84 days ago | 1 author, 1 change
    public class UsersDbContext : IdentityDbContext<User>
    {
        0 references | SerGobec, 84 days ago | 1 author, 1 change
        public UsersDbContext(DbContextOptions<UsersDbContext> options) : base(options)
        {
            Database.EnsureCreated();
        }
    }
}
```

Рисунок 3.7 – Клас UsersDbContext

Далі в файлі appsettings.json потрібно вказати стрічку підключення до бази даних, а після цього в головному класі програми Startup, використовуючи dependency injection додати UsersDbContext до набору сервісів, які буде використовувати програма.

```
public void ConfigureServices(IServiceCollection services)
{
    string connection = Configuration.GetConnectionString("DefaultConnection");
    services.AddDbContext<HotelDbContext>(options =>
        options.UseSqlServer(connection));

    services.AddDbContext<UsersDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("UsersConnection")));

    services.AddIdentity<User, IdentityRole>().AddEntityFrameworkStores<UsersDbContext>();

    services.AddControllersWithViews();
}
```

Рисунок 3.8 – Налаштування сервісів в класі Startup

3.5 Створення бази даних

Окрім бази даних, яку використовуватиме ASP.NET Identity для збереження інформації про користувачів, необхідно створити базу даних, в яку

будуть збережені замовлення, інформація про готелі та номери, відгуки до замовлень. Для роботи з обома базами даних мені необхідно встановити Entity Framework через менеджер пакетів nuget.

Entity Framework дозволяє взаємодіяти з базами даних за допомогою використання класів, а не таблиць. Коли програміст працює з базами даних безпосередньо, він має піклуватися про те, щоб були створені всі зв'язки між таблицями, та в кожній таблиці були встановлені ключі.

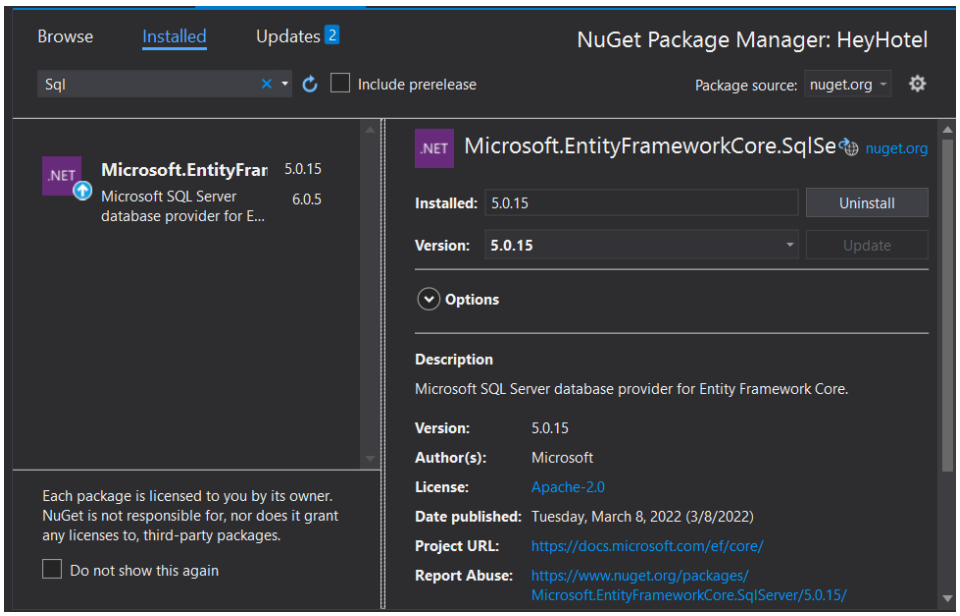

Рисунок 3.9 – Встановлений пакет EntityFramework

Я вирішив використати підхід Code First. Він полягає в тому, що розробник спочатку створює сущності(об'єкти) з таблиць, а потім Entity Framework автоматично генерує відповідну таблицю в базі даних. Отже в базі даних буде чотири таблиці:

- Hotels
- Rooms
- Orders
- Feedbacks

Створений клас Hotel має поля для зберігання інформації про ім'я готелю, його адресу, номер телефону, email. Окрім того, таблиця Hotels матиме зв'язок з таблицею Rooms як один до багатьох, що також треба прописати в коді. За

допомогою Entity Framework, ми можемо просто створити список з об’єктами класу Room, і таким чином встановити зв’язок. З викликом конструктора створюється пустий список, щоб запобігти ситуаціям, коли список буде рівний null, через що винакає багато неочікуваних помилок.

```
public class Hotel
{
    10 references | SerGobec, 95 days ago | 1 author, 1 change
    public int Id { get; set; }
    3 references | SerGobec, 17 days ago | 1 author, 1 change
    public string Name { get; set; }
    1 reference | SerGobec, 17 days ago | 1 author, 1 change
    public string City { get; set; }
    11 references | SerGobec, 95 days ago | 1 author, 1 change
    public string Location { get; set; }
    5 references | SerGobec, 95 days ago | 1 author, 1 change
    public string PhoneNumber { get; set; }
    5 references | SerGobec, 95 days ago | 1 author, 1 change
    public string Mail { get; set; }

    4 references | SerGobec, 95 days ago | 1 author, 1 change
    public List<Room> Rooms { get; set; }
    1 reference | SerGobec, 95 days ago | 1 author, 1 change
    public Hotel()
    {
        Rooms = new List<Room>();
    }
}
```

Рисунок 3.10.1 – Клас Hotel

Клас Room являє собою кімнату готелю. Властивості будуть зберігати номер кімнати, ціну за одну ніч, поверх, опис кімнати, та чи використовується кімната в даний момент. Окрім того, табриця Rooms матиме зв’язок з таблицею Orders як один до багатьох, тому нам так само потрібно створити список, але тепер з об’єктами класу Order.

```
public class Room
{
    20 references | SerGobec, 95 days ago | 1 author, 1 change
    public int Id { get; set; }
    9 references | SerGobec, 95 days ago | 1 author, 1 change
    public int HotelId { get; set; }
    10 references | SerGobec, 78 days ago | 1 author, 1 change
    public int RoomNumber { get; set; }
    15 references | SerGobec, 95 days ago | 1 author, 1 change
    public decimal Price { get; set;}
    9 references | SerGobec, 95 days ago | 1 author, 1 change
    public int NumberOfRooms { get; set; }
    9 references | SerGobec, 95 days ago | 1 author, 1 change
    public int Floor { get; set; }
    10 references | SerGobec, 95 days ago | 1 author, 1 change
    public bool IsUsing { get; set; }
    5 references | SerGobec, 78 days ago | 1 author, 1 change
    public string Description { get; set; }

    20 references | SerGobec, 95 days ago | 1 author, 1 change
    public Hotel Hotel { get; set; }

    public List<Order> Orders = new List<Order>();

    1 reference | SerGobec, 84 days ago | 1 author, 2 changes
    public Room()
    {
        this.Orders = new List<Order>();
    }
}
```

Рисунок 3.10.2 – Клас Room

Клас Order зберігає інформацію про замовлення. Властивості відповідають за дату замовлення, кількість ночей, суму замовлення, чи оплачено та чи закрито замовлення. Також таблиця Orders має зв’язок з таблицею Feedbacks як один до одного. Тобто кожному замовленню відповідає один відгук від користувача.

```
public class Order
{
    9 references | SerGobec, 84 days ago | 1 author, 1 change
    public int Id { get; set; }
    7 references | SerGobec, 79 days ago | 1 author, 2 changes
    public string UserId { get; set; }
    5 references | SerGobec, 84 days ago | 1 author, 1 change
    public int RoomId { get; set; }
    5 references | SerGobec, 84 days ago | 1 author, 1 change
    public DateTime Date { get; set; }
    3 references | SerGobec, 84 days ago | 1 author, 1 change
    public int NumOfNight { get; set; }
    5 references | SerGobec, 84 days ago | 1 author, 1 change
    public decimal Sum { get; set; }
    8 references | SerGobec, 76 days ago | 1 author, 1 change
    public bool IsPayed { get; set; }
    8 references | SerGobec, 76 days ago | 1 author, 1 change
    public bool IsClosed { get; set; }
    12 references | SerGobec, 84 days ago | 1 author, 1 change
    public Room Room { get; set; }
    1 reference | SerGobec, 12 days ago | 1 author, 1 change
    public Feedback Feedback{ get; set; }
}
```

Рисунок 3.10.3 – Клас Order

Клас Feedback являє собою відгук до замовлення. Користувач залишає відгук, після чого відгук показується іншим користувачам, при виборі тієї ж кімнати. Властивості класу зберігають оцінку від 1 до 10, та коментар.

```
public class Feedback
{
    0 references | SerGobec, 12 days ago | 1 author, 1 change
    public int Id { get; set; }
    2 references | SerGobec, 12 days ago | 1 author, 1 change
    public int OrderId { get; set; }
    3 references | SerGobec, 12 days ago | 1 author, 1 change
    public string FeedBack { get; set; }
    4 references | SerGobec, 12 days ago | 1 author, 1 change
    public byte Score { get; set; }
    0 references | SerGobec, 12 days ago | 1 author, 1 change
    public Order Order { get; set; }
}
```

Рисунок 3.10.4 – Клас Feedback

Останній необхідний клас, під назвою HotelDbContext, відповідатиме за зв’язок з базою даних. Він наслідується від стандартного класу DbContext, який відноситься до пакету Entity Framework, та вміщує в собі набори об’єктів, з яких

будуватиметься база даних, та має конструктор, який при першому виклику створить базу даних з правильними полями.

```
public class HotelDbContext : DbContext
{
    7 references | SerGobec, 95 days ago | 1 author, 1 change
    public DbSet<Hotel> Hotels { get; set; }
    11 references | SerGobec, 84 days ago | 1 author, 1 change
    public DbSet<Order> Orders { get; set; }
    18 references | SerGobec, 84 days ago | 1 author, 1 change
    public DbSet<Room> Rooms { get; set; }
    1 reference | SerGobec, 12 days ago | 1 author, 1 change
    public DbSet<Feedback> Feedbacks { get; set; }
    0 references | SerGobec, 95 days ago | 1 author, 1 change
    public HotelDbContext(DbContextOptions<HotelDbContext> options)
        : base(options)
    {
        Database.EnsureCreated();   // Створюємо базу даних при першому виклику
    }
}
```

Рисунок 3.10.5 – Клас HotelDbCpntext

Після запуску програми, Entity Framework створює бази даних, з якими тепер можна взаємодіяти.

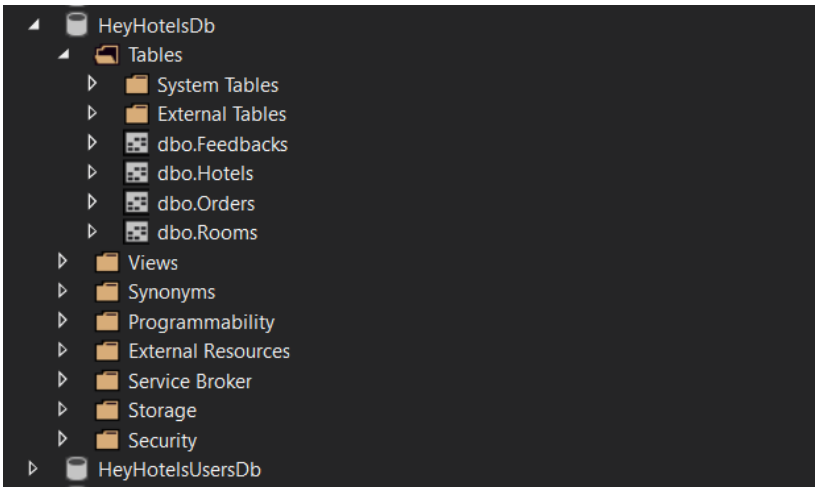

Рисунок 3.11 – Створені бази даних

3.6 Огляд готового додатку

Так як додаток відповідає архітектурному стилю MVC, ми маємо декілька контролерів, моделі, та сторінки Razor. Коли користувач натискає будь яку кнопку, відправляється http-запит на сервер, а в відповідь повертається інша

сторінка. Наприклад початкова сторінка, це відповідь контроллера Home, на запит без параметрів.

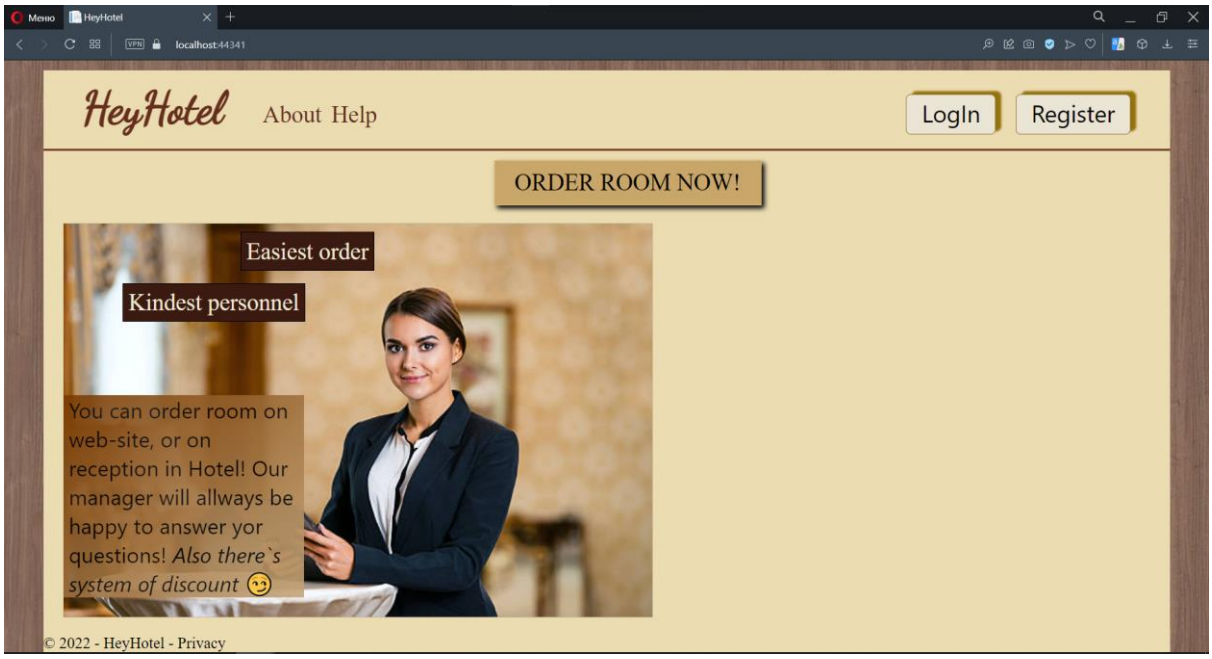

Рисунок 3.12 – Початкова сторінка HeyHotel

Коли користувач вперше заходить на сайт, він може переглядати доступні кімнати, але не може зробити замовлення. Створювати замовлення користувач може тільки після реєстрації. Також після авторизації користувачу стає доступним випадаюче меню, де він може обрати кнопку переглянути свій профіль, та переглянути замовлення.

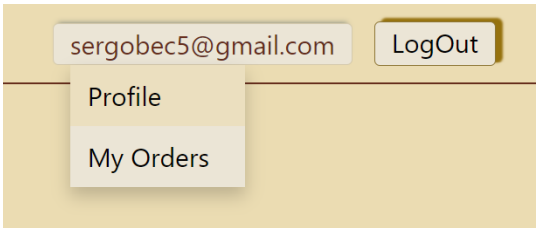

Рисунок 3.13 – Випадаюче меню

Якщо користувач раніше не робив замовлень, то список буде пустим. Щоб створити замовлення, користувачу, на головній сторінці, треба натиснути кнопку “Order room now”, після чого він перейде до вибору готелю.

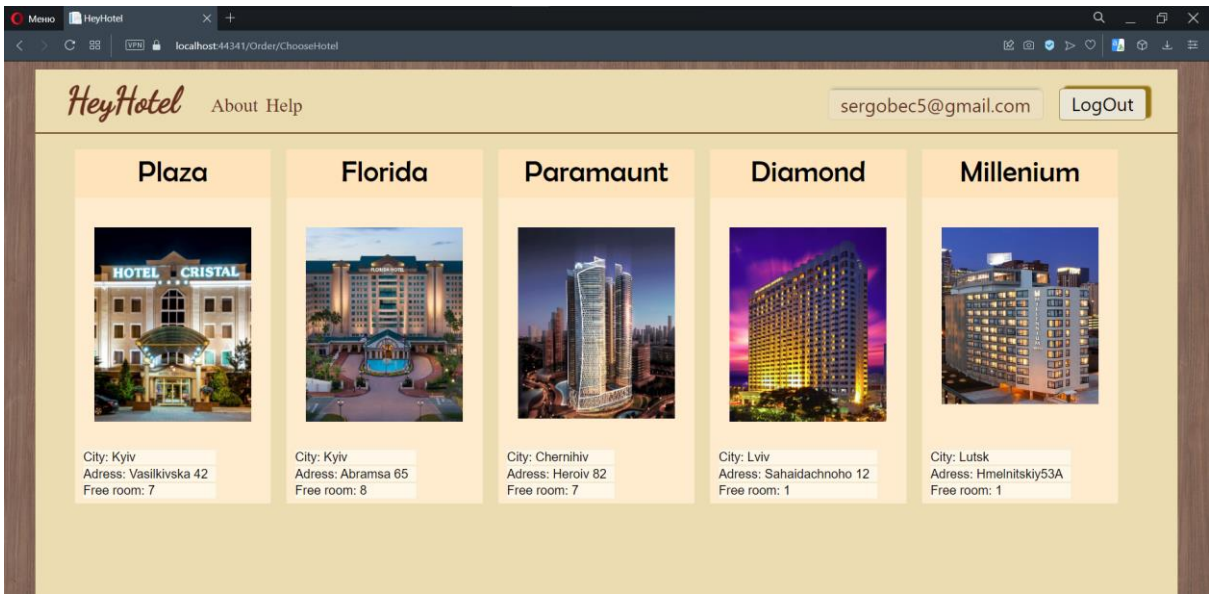

Рисунок 3.14 – Обрання готелю

Для кожного готелю пишеться його адреса, місто, та кількість вільних номерів. Після обрання готелю, користувачу пропонується обрати підходящу кімнату.

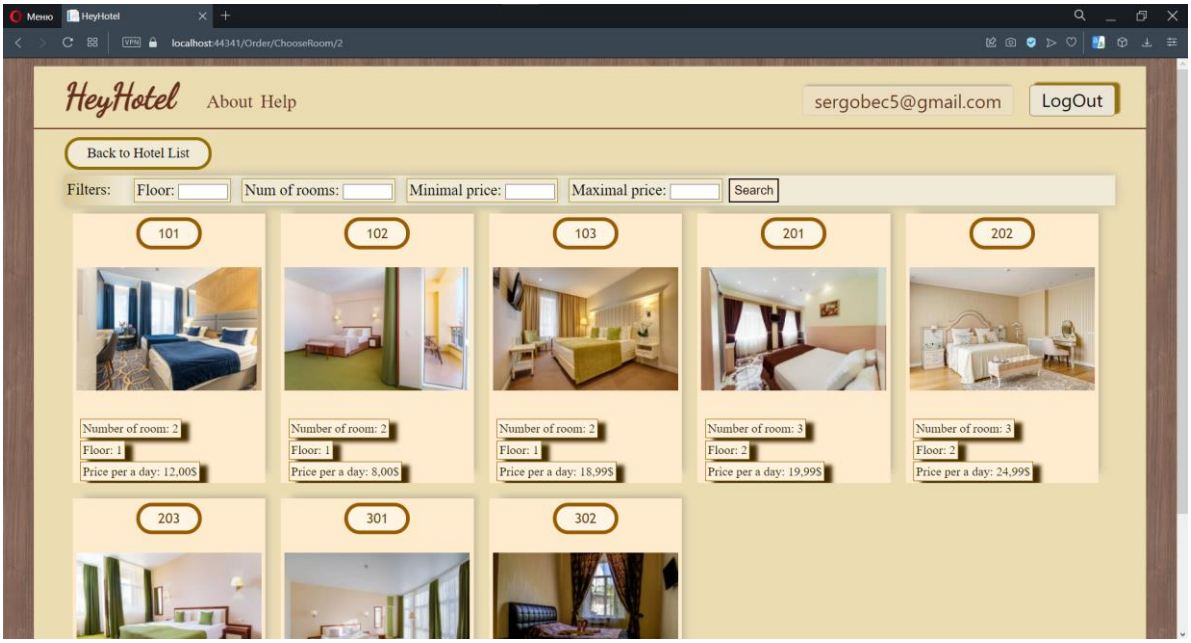

Рисунок 3.15 – Обрання кімнати

При наведенні на картку кімнати, з’являється інформація про кімнату, також користувач за необхідності може скористатися фільтрами, які допоможуть йому швидше знайти підходящу кімнату.

Рисунок 3.16 – Примінення фільтрів для пошуку кімнат

Коли користувач обирає кімнату, йому показується 3 секції, а саме:

- Інформація про кімнату та готель;
- Вікно оформлення замовлення;
- Відгуки користувачів про кімнату.

Після введення бажаного часу прибуття та кількості ночей, користувач натискає кнопку “Create order”, підтверджує замовлення, і тепер замовлення буде відображатись в списку замовлень.

Рисунок 3.17 – Форма створення замовлення

Рисунок 3.18 – Секція з відгуками користувачів

Коли користувач переходить на сторінку з його замовленнями, він має можливість оплатити замовлення(якщо воно активне та не оплачене), та відмовитись від замовлення. Для цього потрібно натиснути на кнопки “Pay Online” та “Cancel the order” відповідно.

Рисунок 3.19.1 – Вікно з замовленнями користувача.

Після того, як користувач оплатить замовлення, та вже виселиться з кімнати, його замовлення стане не акривним, і йому буде доступна функція “Leave feedback”, де користувач може залишити відгук про кімнату.

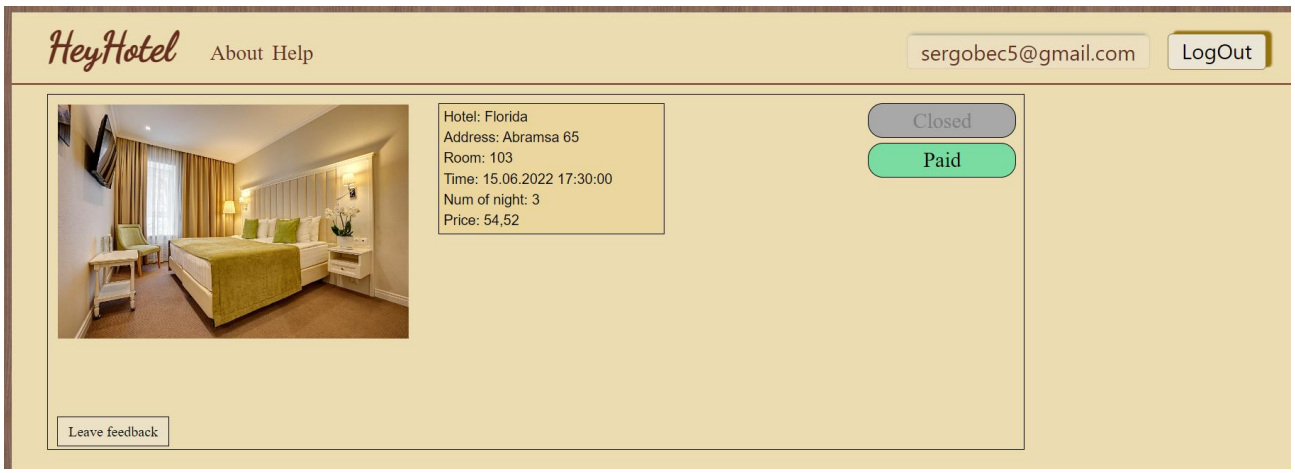

Рисунок 3.19.2 – Оплачене та неактивне замовлення

Але якщо користувач не оплати замовлення, та, по деяким причинам, відмовився від нього, то йому не буде доступна функція “Leave feedback”.

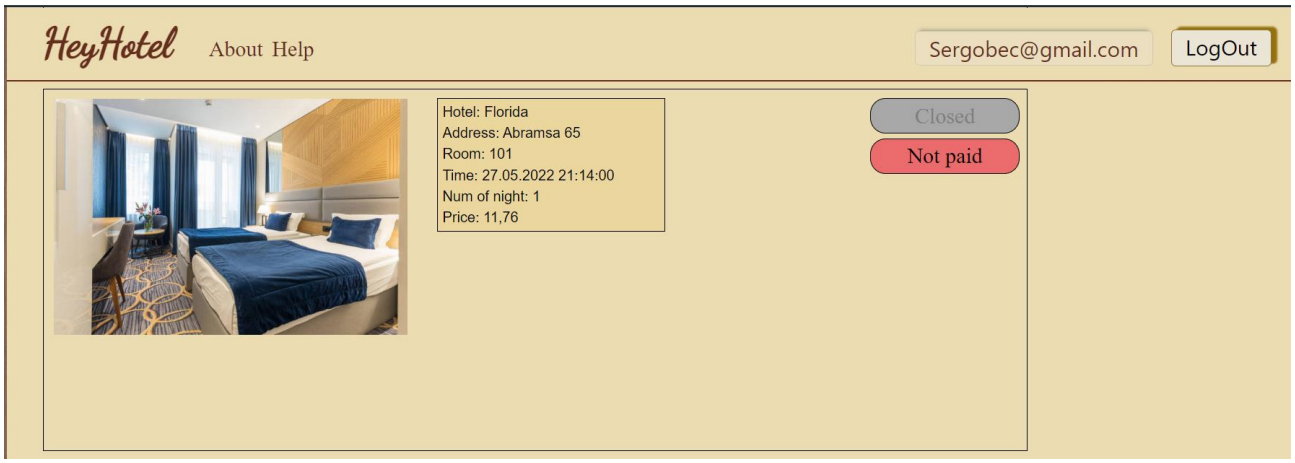

Рисунок 3.19.3 – Неоплачене закрите замовлення

Окрім звичайних користувачів, сервісом можуть користуватися адміністратори та менеджери. Їх акаунти матимуть спеціальні можливості та функції, недоступні для звичайного користувача. Наприклад менеджер може переглянути список всіх замовлень. Ця функція також доступна і для адміністратора. За необхідності можна використати пошук по замовленням, а також використати фільтри.

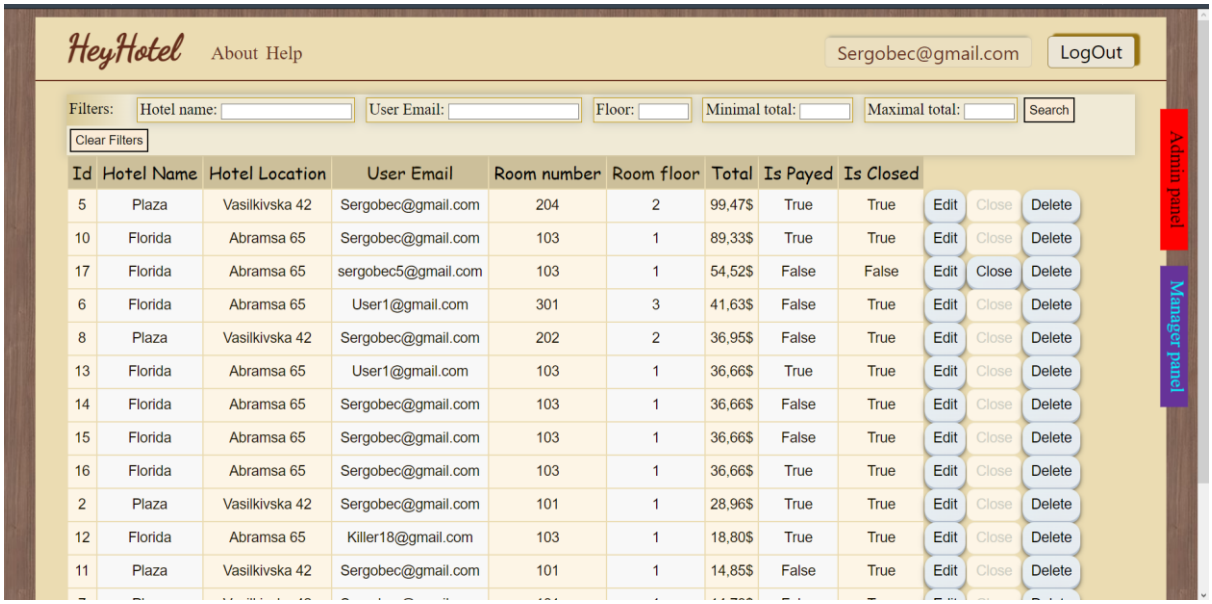

Рисунок 3.20 – Замовлення, відсортовані за вартістю замовлення від більшого до меншого.

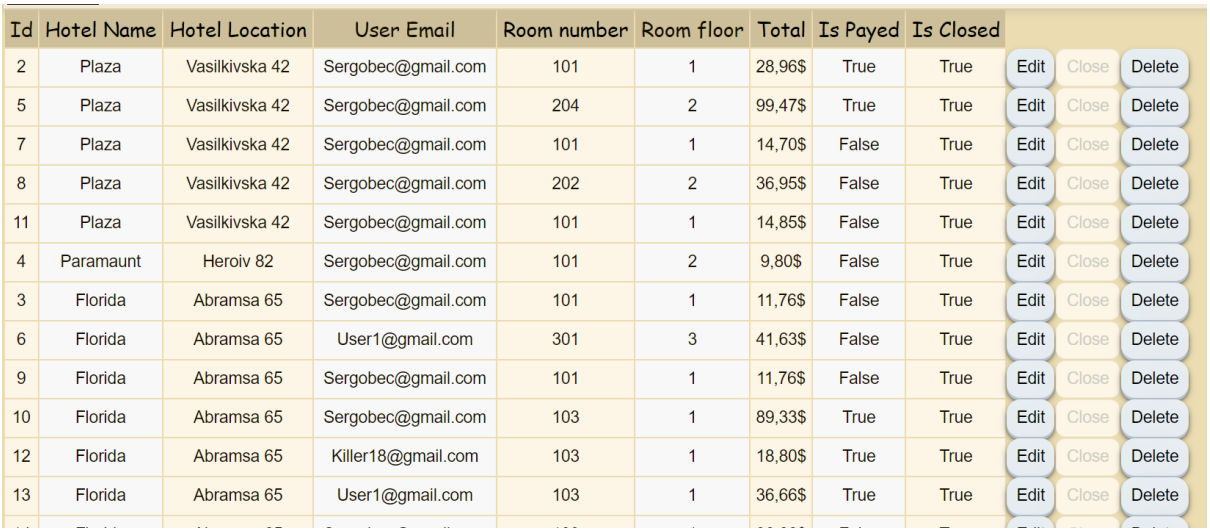

Рисунок 3.21 – Замовлення, відсортовані по назві готелю.

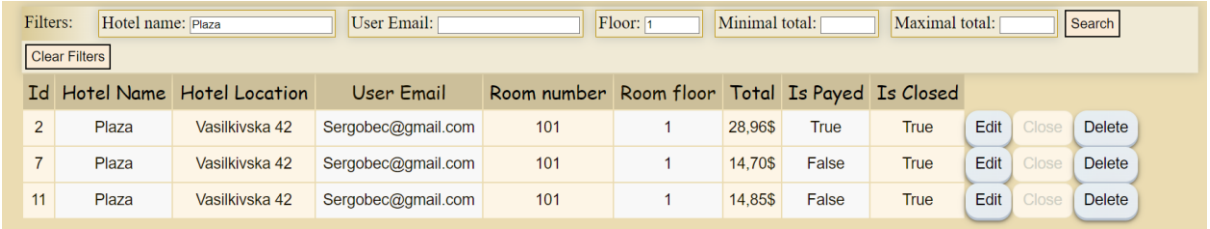

Рисунок 3.22 – Використання фільтрів при пошуку замовлень.

Також менеджер має можливості переглядати список готелів та кімнат. Він може редагувати інформацію про готелі, але не видалити готель з бази даних.

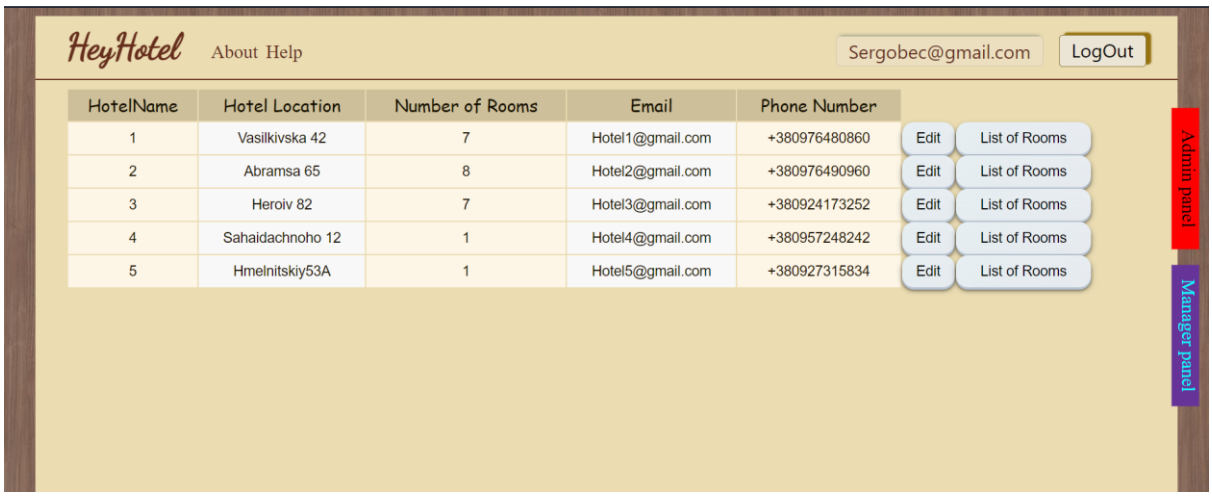

Рисунок 3.23 – Перегляд списку готелів

Адміністратор може все, що може менеджер, а також:

- Додавати кімнати до бази даних;
- Додавати готелі до бази даних;
- Переглядати список всіх користувачів;
- Змінювати ролі користувачів;
- Змінювати персональну інформацію користувачів;
- Видаляти користувачів з бази даних.

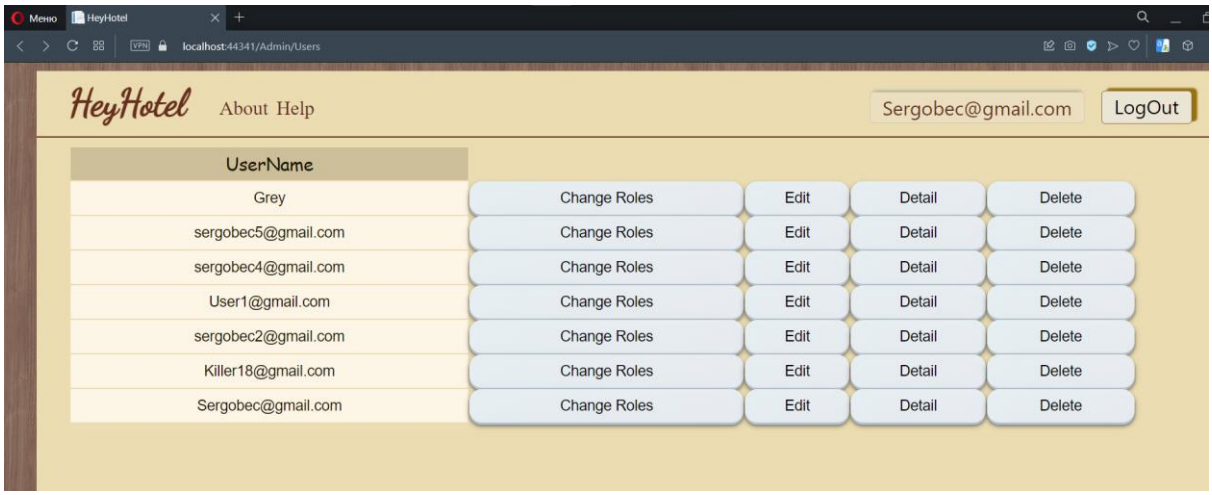

Рисунок 3.24 – Перегляд списку користувачів.

Рисунок 3.25 – Детальна інформація про користувача

Для зручності, кнопки для переходу на функціонал Адміністратора та менеджера було прикріплено до правого краю сторінки. Звичайним користувачам вони відображатися не будуть, і навіть при введені правильного посилання в пошуковий рядок, користувачам, без відповідних прав, буде відмовлено в доступі.

Рисунок 3.25 – Кнопки для переходу до функцій адміністратора та менеджера

До проекту також входить консольний додаток, роль якого полягає в тому, щоб закривати замовлення які не є активними. Менеджер також може закривати замовлення вручну, натиснувши кнопку “Close”, в списку з замовленнями.

ВИСНОВОК ДО РОЗДІЛУ 3

В цьому розділі було описано процес проектування та створення бази даних, а також процес створення самого застосунку. Проект зпроектований з використанням архітектурного патерну MVC, а реалізовано з використанням технології ASP.NET.

Реалізовано розділення по ролям. А також система аутентифікації та авторизації. Робота з базою даних відбувається за допомогою фреймворку Entity. Інтерфейс користувача дуже зручний, та не створює навантаження на очі.

Список використаних джерел

1. yiiframework.com.ua [Електронний ресурс]. – Режим доступу: https://yiiframework.com.ua/uk/doc/guide/topics.webservice/
2. webcase.com.ua [Електронний ресурс]. – Режим доступу: https://webcase.com.ua/blog/cho-takoe-web-prilozhenie-vse-vidy/
3. dotnet.microsoft.com [Електронний ресурс]. – Режим доступу: https://dotnet.microsoft.com/en-us/learn/aspnet/hello-world-tutorial/run
4. professorweb.ru [Електронний ресурс]. – Режим доступу: https://professorweb.ru/my/ASP_NET/mvc/level3/3_2.php
5. web.spt42.ru [Електронний ресурс]. – Режим доступу: http://web.spt42.ru/index.php/chto-takoe-html#:~:text=Что%20такое%20HTML,языке%20HTML%20(или%20XHTML).
6. htmlbook.ru [Електронний ресурс]. – Режим доступу: http://htmlbook.ru/css/hover
7. w3schools.com [Електронний ресурс]. – Режим доступу: https://www.w3schools.com/css/css_border.asp
8. metanit.com [Електронний ресурс]. – Режим доступу: https://metanit.com/sharp/tutorial/12.2.php
9. metanit.com [Електронний ресурс]. – Режим доступу: https://metanit.com/sharp/aspnet5/10.4.php
10. uk.wikipedia.org [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Entity_Framework
11. docs.microsoft.com [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/ef/core/modeling/entity-types?tabs=data-annotations
12. habr.com [Електронний ресурс]. – Режим доступу: https://habr.com/ru/company/otus/blog/500012/
13. docs.microsoft.com [Електронний ресурс]. – Режим доступу: https://docs.microsoft.com/en-us/ef/core/modeling/table-splitting

14.jakeydocs.readthedocs.io [Електронний ресурс]. – Режим доступу: https://jakeydocs.readthedocs.io/en/latest/security/authentication/identity.html

15.professorweb.ru [Електронний ресурс]. – Режим доступу: https://professorweb.ru/my/ASP_NET/identity/level1/

ДОДАТКИ

ДОДАТОК А

Код програмної реалізації:

Startup.cs

```
using HeyHotel.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
```

```
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            string connection = Configuration.GetConnectionString("DefaultConnection");
            services.AddDbContext<HotelDbContext>(options =>
                options.UseSqlServer(connection));

            services.AddDbContext<UsersDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("UsersConnection")));

            services.AddIdentity<User, IdentityRole>().AddEntityFrameworkStores<UsersDbContext>();

            services.AddControllersWithViews();
        }
```

```
        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
```

```
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
  }
}
```

Program.cs

```
using HeyHotel.Models;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel
```

```
{
    public class Program
    {
        public static async Task Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            using(var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var userManager =
services.GetRequiredService<UserManager<User>>();
                    var rolesManager =
services.GetRequiredService<RoleManager<IdentityRole>>();
                    await DataInitializer.RoleInitialize(userManager, rolesManager);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred while seeding the database.");
                }
            }
```

```
            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

_ParentPage.schtml

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>HeyHotel</title>
    <link rel="stylesheet" href="~/css/mainpage/mainpage.css" />
    <link rel="stylesheet" href="~/css/mainpage/HomePage.css" />
    <link rel="stylesheet" href="~/css/mainpage/OrdersStyle.css" />
    <link rel="stylesheet" href="~/css/mainpage/GeneralStyle.css" />
```

```
    <script src="@Url.Content("~/lib/jquery/dist/jquery.min.js")"></script>
    <script src="@Url.Content("~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js")"></script>
</head>
<body>
    <header>
        <div class="left-part-nav">
            <ul>
                <li ><a id="MainButton" asp-action="Index" asp-
controller="Home">HeyHotel</a></li>
                <li><a asp-action="About" asp-controller="Home">About</a></li>
                <li><a asp-action="Help" asp-controller="Home">Help</a></li>
            </ul>
        </div>
        <div class="right-part-nav">
            @if (User.Identity.IsAuthenticated)
            {

            <div style="font-family:system-ui"
id="UserNameDiv">@User.Identity.Name


                <div class="dropdown-content">
                    <a href="#">Profile</a>
                    <a asp-controller="Order" asp-action="OrdersOfUser">My Orders</a>
```

```
            </div>

        </div>

            <a asp-action="Logout" asp-controller="Account"
class="HeaderLink"><div class="HeaderLinkDiv">LogOut</div></a>
        }
        else
        {
            <a asp-action="Login" asp-controller="Account"
class="HeaderLink"><div class="HeaderLinkDiv">LogIn</div></a>
            <a asp-action="Register" asp-controller="Account"
class="HeaderLink"><div class="HeaderLinkDiv">Register</div></a>
        }
    </div>
</header>

<div class="conteiner-for-main">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>

</div>
```

```
    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2022 - HeyHotel - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>

    @if (User.IsInRole("admin"))
    {
        <a asp-controller="Admin" asp-action="Index"><div class="AdminLink">Admin panel</div></a>
    }

    @if (User.IsInRole("admin") || User.IsInRole("manager"))
    {
        <a asp-controller="Manager" asp-action="Index"><div class="ManagerLink">Manager panel</div></a>
    }

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
    <script> </script>
```

```
</body>
</html>
```

AccountController.cs

```
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using HeyHotel.Models;
using HeyHotel.ViewModels;
using Microsoft.AspNetCore.Identity;

namespace HeyHotel.Controllers
{
    public class AccountController : Controller
    {
        private readonly UserManager<User> _userManager;
        private readonly SignInManager<User> _signInManager;
        public AccountController(UserManager<User> userManager,
SignInManager<User> signInManager)
        {
```

```
        this._userManager = userManager;
        this._signInManager = signInManager;
    }

    [HttpGet]
    public IActionResult Register()
    {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult> Register(RegisterViewModel model)
    {
        if (ModelState.IsValid)
        {
            User user = new User { Email = model.Email, UserName = model.Email,
Year = model.Year };
            var result = await _userManager.CreateAsync(user, model.Password);
            if (result.Succeeded)
            {
                await _userManager.AddToRoleAsync(user, "user");
                await this._signInManager.SignInAsync(user, false);
                return RedirectToAction("Index", "Home");
            } else
```

```
            {
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError("", error.Description);
                }
            }
        }
        return View(model);
    }

    [HttpGet]
    public IActionResult Login(string returnUrl = null)
    {
        return View(new LoginViewModel { ReturnUrl = returnUrl });
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginViewModel model)
    {
        if (ModelState.IsValid)
        {
            var result = await _signInManager.PasswordSignInAsync(model.Email,
model.Password, model.RememberMe, false);
```

```
            if (result.Succeeded)
            {
                // проверяем, принадлежит ли URL приложению
                if (!string.IsNullOrEmpty(model.ReturnUrl) &&
Url.IsLocalUrl(model.ReturnUrl))
                {
                    return Redirect(model.ReturnUrl);
                }
                else
                {
                    return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("", "Неправильный логин и (или)
пароль");
            }
        }
        return View(model);
    }

    [HttpGet]
    // [ValidateAntiForgeryToken]
    public async Task<IActionResult> Logout()
```

```
        {
            await _signInManager.SignOutAsync();
            return RedirectToAction("Index", "Home");
        }
    }
}
```

AdminController.cs

```
using HeyHotel.Models;
using HeyHotel.ViewModels;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Controllers
{
    [Authorize(Roles = "admin")]
    public class AdminController : Controller
```

```
{
    private readonly UserManager<User> _userManager;
    private readonly RoleManager<IdentityRole> _roleManager;
    private readonly SignInManager<User> _signInManager;
    private readonly HotelDbContext _dbContext;

    public AdminController(UserManager<User> userManager,
RoleManager<IdentityRole> roleManager, SignInManager<User> signInManager,
HotelDbContext dbContext)
    {
        this._userManager = userManager;
        this._roleManager = roleManager;
        this._signInManager = signInManager;
        this._dbContext = dbContext;
    }
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Users()
    {
        var users = _userManager.Users.ToList();
        return View(users);
    }
```

```
public async Task<IActionResult> UserDetail(string? id)
{
    User user = await _userManager.FindByIdAsync(id);
    if (user != null)
    {
        return View(user);
    }
    else return NotFound();
}

[HttpGet]
public async Task<IActionResult> ChangeRoles(string id)
{
    User user = await _userManager.FindByIdAsync(id);
    if (user != null)
    {
        ChangeRolesViewModel model = new ChangeRolesViewModel();
        model.userId = user.Id;
        model.UserName = user.UserName;
        model.AllRoles = _roleManager.Roles.ToList();
        model.UserRoles = await _userManager.GetRolesAsync(user);
        return View(model);
    }
```

```
        return NotFound();
    }

    [HttpPost]
    public async Task<IActionResult> ChangeRoles(string UserId, List<string>
chosenRoles)
    {
        User user = await _userManager.FindByIdAsync(UserId);
        if (user != null)
        {

            var userRoles = await _userManager.GetRolesAsync(user);
            var addedRoles = chosenRoles.Except(userRoles);
            var removedRoles = userRoles.Except(chosenRoles);

            await _userManager.AddToRolesAsync(user, addedRoles);
            await _userManager.RemoveFromRolesAsync(user, removedRoles);
            return RedirectToAction("Index", "Admin");
        }
        return NotFound();
    }

    [HttpGet]
    public async Task<IActionResult> EditUser(string id)
```

```
{
    User user = await _userManager.FindByIdAsync(id);
    if (user != null)
    {
        EditUserViewModel model = new EditUserViewModel
        {
            Id = user.Id,
            Name = user.Name,
            Sname = user.SName,
            userName = user.UserName,
            mail = user.Email,
            PhoneNumber = user.PhoneNumber,
            Year = user.Year
        };
        return View(model);
    }
    return NotFound();
}

[HttpPost]
public async Task<IActionResult> EditUser(EditUserViewModel model)
{
    User user = await _userManager.FindByIdAsync(model.Id);
    if (user != null)
```

```
    {
        user.Name = model.Name;
        user.SName = model.Sname;
        user.UserName = model.userName;
        user.Email = model.mail;
        user.PhoneNumber = model.PhoneNumber;
        user.Year = model.Year;

        var result = await _userManager.UpdateAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }
    }
    return View(model);
}
```

```
[HttpGet]
public async Task<IActionResult> DeleteUser(string id)
{
   User user = await _userManager.FindByIdAsync(id);
   if(user != null)
   {
         DeleteViewModel model = new DeleteViewModel()
         {
            Id = user.Id,
            Name = user.Name,
            SName = user.SName,
            Email = user.Email,
            UserName = user.UserName
         };
         return View(model);
   }
   return NotFound();

}

[HttpPost]
public async Task<IActionResult> DeleteUserPost(DeleteViewModel model)
{
```

```
    User user = await _userManager.FindByIdAsync(model.Id);
    if(user != null)
    {
        var result = await _userManager.DeleteAsync(user);
        if (result.Succeeded)
        {
            return RedirectToAction("Users", "Admin");
        }
        else
        {
            return NotFound();
        }
    }
    return NotFound();
}

[HttpGet]
public IActionResult CreateHotel()
{
    return View();
}

[HttpPost]
public async Task<IActionResult> CreateHotel(CreateHotelViewModel model)
```

```
{
    if (ModelState.IsValid)
    {
        Hotel hotel = new Hotel()
        {
            Location = model.Location,
            PhoneNumber = model.PhoneNumber,
            Mail = model.Mail
        };
        _dbContext.Add(hotel);
        int a = await _dbContext.SaveChangesAsync();
        return RedirectToAction("index", "admin");
    }
    return View(model);
}

[HttpGet]
public IActionResult CreateRoom()
{
    CreateRoomViewModel model = new CreateRoomViewModel();
    foreach(var hotel in _dbContext.Hotels)
    {
        model.hotelsLocations.Add(new SelectListItem
        {
```

```
                Text = hotel.Location,
                Value = hotel.Id + ""
            });
        }
        return View(model);
    }

    [HttpPost]
    public async Task<IActionResult> CreateRoom(CreateRoomViewModel
model)
    {
        if (ModelState.IsValid)
        {
            Room room = new Room()
            {
                Description = model.Description,
                Floor = model.Floor,
                HotelId = model.HotelId,
                IsUsing = false,
                NumberOfRooms = model.NumberOfRooms,
                Price = model.Price,
                RoomNumber = model.RoomNumber
            };
            if(_dbContext.Rooms.Where(el => el.HotelId == room.HotelId
```

```
                        && el.RoomNumber == room.RoomNumber)
                        .FirstOrDefault() == null)
            {
                _dbContext.Add(room);
                await _dbContext.SaveChangesAsync();
                return RedirectToAction("index", "admin");
            }
            return Content("Room already exist");
        }
        return View(model);
    }
  }
}
```

HomeController.cs

```
using HeyHotel.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
```

```
using System.Threading.Tasks;

namespace HeyHotel.Controllers
{
    public class HomeController : Controller
    {
        private readonly ILogger<HomeController> _logger;
        private HotelDbContext _context;
        private readonly UsersDbContext _usersContext;

        public HomeController(ILogger<HomeController> logger, HotelDbContext
context, UsersDbContext usersContext)
        {
            _logger = logger;
            this._context = context;
            this._usersContext = usersContext;
        }

        public IActionResult Index()
        {
            return View();
        }

        public IActionResult Privacy()
```

```
        {
            return View();
        }

        [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
        }
    }
}
```

ManagerController.cs

```
using HeyHotel.Models;
using HeyHotel.ViewModels;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;

namespace HeyHotel.Controllers
{
    [Authorize(Roles = "manager,admin")]
    public class ManagerController : Controller
    {
        HotelDbContext _dbContext;
        UsersDbContext _usersDbContext;
        public ManagerController(HotelDbContext dbContext, UsersDbContext
usersDbContext)
        {
            this._dbContext = dbContext;
            this._usersDbContext = usersDbContext;
        }
        public IActionResult Index()
        {
            return View();
        }

        public IActionResult HotelList()
        {
            List<Hotel> hotels = _dbContext.Hotels.Include(x => x.Rooms).ToList();
            return View(hotels);
```

```
}

[HttpGet]
public IActionResult EditHotel(int id)
{
    Hotel hotel = _dbContext.Hotels.Where(el => el.Id == id).FirstOrDefault();
    if (hotel != null)
    {
        return View(new EditHotelViewModel()
        {
            Id = hotel.Id,
            Location = hotel.Location,
            Mail = hotel.Mail,
            PhoneNumber = hotel.PhoneNumber
        });
    }
    return NotFound();
}

[HttpPost]
public async Task<IActionResult> EditHotel(EditHotelViewModel model)
{
    if (ModelState.IsValid)
    {
```

```
            Hotel hotel = _dbContext.Hotels.Where(el => el.Id ==
model.Id).FirstOrDefault();
            if(hotel != null)
            {
                hotel.Location = model.Location;
                hotel.Mail = model.Mail;
                hotel.PhoneNumber = model.PhoneNumber;
                _dbContext.Hotels.Update(hotel);
                await _dbContext.SaveChangesAsync();
                return RedirectToAction("HotelList", "Manager");
            }
        }
        return View(model);
    }



    public IActionResult RoomsList(int? id)
    {
        if(id != null)
        {
            List<Room> rooms = _dbContext.Rooms.Where(room => room.HotelId
== id).Include(x => x.Hotel).ToList();
            return View("RoomsList", rooms);
        } return NotFound();
    }
```

```
[HttpGet]
public IActionResult EditRoom(int? id)
{
    if(id != null)
    {
        Room room = _dbContext.Rooms.Where(room => room.Id ==
id).FirstOrDefault();
        if(room != null)
        {
            return View(new EditRoomViewModel()
            {
                Id = room.Id,
                Description = room.Description,
                Floor = room.Floor,
                NumberOfRooms = room.NumberOfRooms,
                Price = room.Price,
                RoomNumber = room.RoomNumber
            });
        }
    }
    return NotFound();
}

[HttpPost]
```

```
public async Task<IActionResult> EditRoomPost(EditRoomViewModel model)
{
    if (ModelState.IsValid)
    {
        Room room = _dbContext.Rooms.Where(room => room.Id ==
model.Id).FirstOrDefault();
        if(room != null)
        {
            room.NumberOfRooms = model.NumberOfRooms;
            room.Price = model.Price;
            room.RoomNumber = model.RoomNumber;
            room.Floor = model.Floor;
            room.Description = model.Description;
            _dbContext.Update(room);
            await _dbContext.SaveChangesAsync();
            //return RedirectToAction("HotelList", "Manager");
            return RoomsList(room.HotelId);
        }
        return NotFound();
    }
    return View(model);
}

[HttpGet]
```

```
public IActionResult RoomDetail(int? id)
{
    Room room = _dbContext.Rooms.Where(room => room.Id ==
id).Include(room => room.Hotel).FirstOrDefault();
    if(room != null)
    {
        return View(room);
    }
    return NotFound();
}

[Authorize(Roles = "admin")]
[HttpGet]
public async Task<IActionResult> DeleteRoom(int? id)
{
    Room room = _dbContext.Rooms.Where(el => el.Id == id).FirstOrDefault();
    if(room != null)
    {
        _dbContext.Rooms.Remove(room);
        await _dbContext.SaveChangesAsync();
        return RoomsList(room.HotelId);
    }
    return NotFound();
}
```

```
[HttpGet]
public IActionResult OrdersList(string? sortOrder, string? HotelName ,int? Floor, string? Email, int? MinTotal, int? MaxTotal)
{
    OrdersListViewModel model = new OrdersListViewModel();

    ViewBag.HotelNameSortParm = String.IsNullOrEmpty(sortOrder) ? "Name_desc" : "";
    ViewBag.HotelLocationSortParm = sortOrder == "Location" ? "Location_desc" : "Location";
    ViewBag.RoomNumberSortParm = sortOrder == "Rnumb" ? "Rnumb_desc" : "Rnumb";
    ViewBag.TotalSortParm = sortOrder == "Total" ? "Total_desc" : "Total";
    ViewBag.IsPayedSortParm = sortOrder == "Payed" ? "Payed_desc" : "Payed";
    ViewBag.IsClosedSortParm = sortOrder == "Closed" ? "Closed_desc" : "Closed";
    ViewBag.LastValueSortOrder = sortOrder;
    ViewBag.LVFloor = Floor;
    ViewBag.LVFEmail = Email;
    ViewBag.LVMinTotal = MinTotal;
    ViewBag.LVMaxTotal = MaxTotal;
    ViewBag.LVHotelName = HotelName;
```

```
        model.Orders = _dbContext.Orders.Include(el => el.Room).Include(el =>
el.Room.Hotel).ToList();
        model.UsersInfo = new List<ShortUserInfo>();

        if(Floor != null)
        {
            model.Orders = model.Orders.Where(el => el.Room.Floor ==
Floor).ToList();
        }
        if (MinTotal != null)
        {
            model.Orders = model.Orders.Where(el => el.Sum >= MinTotal).ToList();
        }
        if (MaxTotal != null)
        {
            model.Orders = model.Orders.Where(el => el.Sum <= MaxTotal).ToList();
        }
        if(HotelName != null)
        {
            model.Orders = model.Orders.Where(el =>
el.Room.Hotel.Name.Contains(HotelName)).ToList();
        }

        switch (sortOrder)
        {
```

```
            case "Name_desc":
                model.Orders = model.Orders.OrderByDescending(el =>
el.Room.Hotel.Name).ToList();
                break;
            case "Location":
                model.Orders = model.Orders.OrderBy(el =>
el.Room.Hotel.Location).ToList();
                break;
            case "Location_desc":
                model.Orders = model.Orders.OrderByDescending(el =>
el.Room.Hotel.Location).ToList();
                break;
            case "Rnumb":
                model.Orders = model.Orders.OrderBy(el =>
el.Room.NumberOfRooms).ToList();
                break;
            case "Rnumb_desc":
                model.Orders = model.Orders.OrderByDescending(el =>
el.Room.NumberOfRooms).ToList();
                break;
            case "Total":
                model.Orders = model.Orders.OrderBy(el => el.Sum).ToList();
                break;
            case "Total_desc":
```

```
                model.Orders = model.Orders.OrderByDescending(el =>
el.Sum).ToList();
                break;
            case "Payed":
                model.Orders = model.Orders.OrderBy(el => el.IsPayed).ToList();
                break;
            case "Payed_desc":
                model.Orders = model.Orders.OrderByDescending(el =>
el.IsPayed).ToList();
                break;
            case "Closed":
                model.Orders = model.Orders.OrderBy(el => el.IsClosed).ToList();
                break;
            case "Closed_desc":
                model.Orders = model.Orders.OrderByDescending(el =>
el.IsClosed).ToList();
                break;
            default:
                model.Orders = model.Orders.OrderBy(el =>
el.Room.Hotel.Name).ToList();
                break;
        }

        foreach (Order order in model.Orders)
        {
```

```
            User user = _usersDbContext.Users.Where(el => el.Id ==
order.UserId).FirstOrDefault();
            if(user != null)
            {
                model.UsersInfo.Add(new ShortUserInfo()
                {
                    Id = user.Id,
                    SName = user.SName,
                    Email = user.Email,
                    Name = user.Name
                });
            }
        }

        if(Email != null)
        {
            model.Orders = model.Orders.Where(el =>
            {
                ShortUserInfo user = model.FindUserById(el.UserId);
                if (user != null && user.Email.Contains(Email)) return true;
                return false;
            }).ToList();
        }
        return View(model);
```

```
        }
    }
}
```

OrderController.cs

```
using HeyHotel.Models;
using HeyHotel.ViewModels;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Controllers
{
    public class OrderController : Controller
    {
        HotelDbContext _dbContext;
        UserManager<User> _userManager;
```

```
public OrderController(HotelDbContext dbContext, UserManager<User>
userManager)
{
    _dbContext = dbContext;
    _userManager = userManager;
}
public IActionResult Index()
{
    return View();
}

public IActionResult ChooseHotel()
{
    return View(_dbContext.Hotels.Include(el => el.Rooms).ToList());
}

public IActionResult ChooseRoom(int? id, int? Floor, int? NumOfRooms, int?
MinPrice, int? MaxPrice)
{
    ViewBag.HotelId = id;
    if(_dbContext.Hotels.Where(el => el.Id == id) != null)
    {
        if(_dbContext.Rooms.Where(el => el.HotelId == id &&
!el.IsUsing).Count() == 0)
        {
```

```
            return NotFound();
        }
        List<Room> rooms = _dbContext.Rooms.Where(el => el.HotelId == id
&& !el.IsUsing).Include(el => el.Hotel).ToList();
        if(Floor != null)
        {
            rooms = rooms.Where(el => el.Floor == Floor).ToList();
        }
        if(NumOfRooms != null)
        {
            rooms = rooms.Where(el => el.NumberOfRooms ==
NumOfRooms).ToList();
        }
        if(MinPrice != null && MaxPrice != null && MinPrice <= MaxPrice)
        {
            rooms = rooms.Where(el => ((int)el.Price) >= MinPrice &&
((int)el.Price) <= MaxPrice).ToList();
        }
        if (MinPrice != null && MaxPrice != null && MinPrice <= MaxPrice)
        {
            rooms = rooms.Where(el => ((int)el.Price) >= MinPrice &&
((int)el.Price) <= MaxPrice).ToList();
        }
        return View(rooms);
    }
```

```
            return NotFound();
        }

        [Authorize]
        public async Task<IActionResult> CreateOrder(int? id)
        {
            if (_dbContext.Rooms.Where(el => el.Id == id).FirstOrDefault() != null)
            {
                CreateOrderViewModel model = new CreateOrderViewModel();
                Room room = _dbContext.Rooms.Where(el => el.Id == id).Include(el =>
el.Hotel).FirstOrDefault();
                List<Order> orders = _dbContext.Orders.Where(el => el.RoomId ==
room.Id).ToList();
                List<FeedbackUserPair> feedbacks = new List<FeedbackUserPair>();
                foreach(Order order in orders)
                {
                    Feedback fb = _dbContext.Feedbacks.Where(el => el.OrderId ==
order.Id).FirstOrDefault();
                    User user1 = await _userManager.FindByIdAsync(order.UserId);
                    if (fb != null && user1 != null)
                    {
                        feedbacks.Add(new FeedbackUserPair
                        {
                            Feedback = fb,
                            User = user1
```

```
                });
            }
        }
        ViewBag.Feedbacks = feedbacks;
        User user = await
_userManager.FindByNameAsync(HttpContext.User.Identity.Name);
        model.UserId = user.Id;
        model.Room = room;
        model.RoomId = room.Id;
        model.PersonalDiscount = PersonalDiscount(user.Id);
        return View(model);
      }
      return NotFound();
    }

    [Authorize]
    [HttpGet]
    public IActionResult ConfirmOrder(CreateOrderViewModel model)
    {
      if (ModelState.IsValid)
      {
        ConfirmOrderViewModel ConfirmModel = new
ConfirmOrderViewModel();
        ConfirmModel.NumOfNight = model.NumOfNight;
```

```
            ConfirmModel.Sum = CalcRoomPrice(model.RoomId, model.UserId,
model.NumOfNight);
            ConfirmModel.Room = _dbContext.Rooms.Where(el => el.Id ==
model.RoomId).FirstOrDefault();
            ConfirmModel.Date = model.Date.Add(model.Time);
            ConfirmModel.RoomId = model.RoomId;
            ConfirmModel.UserId = model.UserId;
            return View(ConfirmModel);
        }
        return NotFound();
    }

    [HttpPost]
    public async Task<IActionResult> ConfirmOrder(ConfirmOrderViewModel
model)
    {
        if (ModelState.IsValid)
        {
            Order order = new Order()
            {
                Sum = model.Sum,
                Date = model.Date,
                IsClosed = false,
                IsPayed = false,
                RoomId = model.RoomId,
```

```
                NumOfNight = model.NumOfNight,
                UserId = model.UserId
            };
            Room room = _dbContext.Rooms.Where(el => el.Id ==
model.RoomId).FirstOrDefault();
            if(room != null && !room.IsUsing)
            {
                room.IsUsing = true;

                _dbContext.Add(order);
                _dbContext.Update(room);

                await _dbContext.SaveChangesAsync();
                return RedirectToAction("Index", "Home");
            }
            return Content("Room alred ordered, or ");
        }
        return View(model);
    }

    [Authorize(Roles = "manager,admin")]
    public async Task<IActionResult> CloseOrderForManager(int? OrderId)
    {
        Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
```

```
        Room room = _dbContext.Rooms.Where(el => el.Id ==
order.RoomId).FirstOrDefault();
        if (order != null && room != null)
        {
            order.IsClosed = true;
            room.IsUsing = false;
            _dbContext.Update(order);
            _dbContext.Update(room);
            await _dbContext.SaveChangesAsync();
            return RedirectToAction("OrdersList", "Manager");
        }
        return Content("Order or room not found");
    }

    public async Task<IActionResult> CloseOrderForUser(int? OrderId)
    {
        Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
        User user = await _userManager.FindByNameAsync(User.Identity.Name);
        Room room = _dbContext.Rooms.Where(el => el.Id ==
order.RoomId).FirstOrDefault();
        if (order != null && room != null && user != null && user.Id ==
order.UserId)
        {
            order.IsClosed = true;
```

```
            room.IsUsing = false;
            _dbContext.Update(order);
            _dbContext.Update(room);
            await _dbContext.SaveChangesAsync();
            return RedirectToAction("OrdersOfUser", "Order");
        }
        return Content("Order or room not found");
    }

    public async Task<IActionResult> DeleteOrder(int? OrderId)
    {
        Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
        Room room = _dbContext.Rooms.Where(el => el.Id ==
order.RoomId).FirstOrDefault();
        if(order != null && room != null)
        {
            if(room.IsUsing && !order.IsClosed)
            {
                room.IsUsing = false;
                _dbContext.Update(room);
            }
            _dbContext.Remove(order);
            await _dbContext.SaveChangesAsync();
            return RedirectToAction("OrdersList", "Manager");
```

```
            }
            return Content("Order or room not found");
        }

        [HttpGet]
        public IActionResult EditOrder(int? OrderId)
        {
            Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
            if(order != null)
            {
                EditOrderViewModel model = new EditOrderViewModel();
                model.Id = order.Id;
                model.IsPayed = order.IsPayed;
                model.NumOfNight = order.NumOfNight;
                model.Date = order.Date;
                model.Time = order.Date.TimeOfDay;
                return View(model);
            }
            return NotFound();
        }

        [HttpPost]
        public async Task<IActionResult> EditOrder(EditOrderViewModel model)
```

```
        {
            if (ModelState.IsValid)
            {
                Order order = _dbContext.Orders.Where(el => el.Id ==
model.Id).FirstOrDefault();
                if (order != null)
                {
                    order.IsPayed = model.IsPayed;
                    order.NumOfNight = model.NumOfNight;
                    order.Date = model.Date.Add(model.Time);
                    _dbContext.Update(order);
                    await _dbContext.SaveChangesAsync();
                    return RedirectToAction("OrdersList", "Manager");
                }
            }
            return View(model);
        }


        public decimal PersonalDiscount(string userId)
        {
            int closedOrders = _dbContext.Orders.Where(el => el.UserId == userId &&
el.IsPayed && el.IsClosed).Count();
            decimal discount = 1;
            if (closedOrders > 0 && closedOrders < 3) discount = 2;
```

```
        if (closedOrders > 3 && closedOrders < 6) discount = 5;
        if (closedOrders > 6 && closedOrders < 10) discount = 7;
        if (closedOrders > 10) discount = 10;
        return discount;
    }

    public decimal CalcRoomPrice(int? roomId, string userId, int numOfDays)
    {
        decimal discount = PersonalDiscount(userId);

        Room room = _dbContext.Rooms.Where(el => el.Id ==
roomId).FirstOrDefault();
        if(room != null)
        {
            return (room.Price + room.Price * (numOfDays - 1) * 0.95M) * ( 1 -
discount/100);
        }
        return -1;
    }

    [HttpGet]
    public IActionResult CalcRoomPriceGet(int RoomId, string userId, int
numOfDays)
    {
        //, string userId, int numOfDays
```

```
        //return Content(RoomId.ToString() + "_" + userId + "_" +
numOfDays.ToString());
        decimal discount = PersonalDiscount(userId);

        Room room = _dbContext.Rooms.Where(el => el.Id ==
RoomId).FirstOrDefault();
        if (room != null)
        {
            decimal result = (room.Price + room.Price * (numOfDays - 1) * 0.95M) *
(1 - discount / 100);
            return Content(result.ToString("0.00"));
        }
        return Content("Can`t count...");
    }

    [Authorize]
    [HttpGet]

    public async Task<IActionResult> OrdersOfUser()
    {
        User user = await _userManager.FindByNameAsync(User.Identity.Name);
        if(user != null)
        {
            List<Order> orders = _dbContext.Orders.Where(el => el.UserId ==
user.Id).Include(el => el.Feedback).Include(el => el.Room).Include(el =>
```

```
el.Room.Hotel).OrderBy(el => el.IsClosed).ThenBy(el => !el.IsPayed).ThenBy(el =>
el.Date).ToList();
            return View(orders);
        }
        return NotFound();
    }

    [HttpGet]
    public async Task<IActionResult> PayForOrder(int OrderId)
    {
        Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
        if(order != null)
        {
            order.IsPayed = true;
            _dbContext.Update(order);
            await _dbContext.SaveChangesAsync();
            return RedirectToAction("OrdersOfUser");
        }
        return NotFound();
    }

    [HttpGet]
    public async Task<IActionResult> LeaveFeedback(int OrderId)
    {
```

```
            Order order = _dbContext.Orders.Where(el => el.Id ==
OrderId).FirstOrDefault();
            User user = await _userManager.FindByNameAsync(User.Identity.Name);
            if(user.UserName != User.Identity.Name)
            {
                return StatusCode(405);
            }
            if(order != null)
            {
                return View(order);
            }

            return NotFound();
        }

        [HttpPost]
        public async Task<IActionResult> LeaveFeedback(int OrderId, byte Score,
string Feedback)
        {
            Feedback fb = new Feedback
            {
                Score = Score,
                FeedBack = Feedback,
                OrderId = OrderId
            };
```

```
            _dbContext.Add(fb);
            await _dbContext.SaveChangesAsync();
            return RedirectToAction("OrdersOfUser");
        }
    }
}
```

Feedback.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Models
{
    public class Feedback
    {
        public int Id { get; set; }
        public int OrderId { get; set; }
        public string FeedBack { get; set; }
        public byte Score { get; set; }
        public Order Order { get; set; }
```

```
    }
}
```

Hotel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Models
{
    public class Hotel
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string City { get; set; }
        public string Location { get; set; }
        public string PhoneNumber { get; set; }
        public string Mail { get; set; }

        public List<Room> Rooms { get; set; }
        public Hotel()
```

```
        {
            Rooms = new List<Room>();
        }
    }
}
```

HotelDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Models
{
    public class HotelDbContext : DbContext
    {
        public DbSet<Hotel> Hotels { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<Room> Rooms { get; set; }
        public DbSet<Feedback> Feedbacks { get; set; }
        public HotelDbContext(DbContextOptions<HotelDbContext> options)
```

```
        : base(options)
    {
        Database.EnsureCreated();   // Створюємо базу даних при першому виклику
    }
  }
}
```

Order.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Models
{
  public class Order
  {
    public int Id { get; set; }
    public string UserId { get; set; }
    public int RoomId { get; set; }
```

```
        public DateTime Date { get; set; }
        public int NumOfNight { get; set; }
        public decimal Sum { get; set; }
        public bool IsPayed { get; set; }
        public bool IsClosed { get; set; }
        public Room Room { get; set; }
        public Feedback Feedback{ get; set; }
    }
}
```

Room.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace HeyHotel.Models
{
    public class Room
    {
        public int Id { get; set; }
        public int HotelId { get; set; }
```

```
        public int RoomNumber { get; set; }
        public decimal Price { get; set;}
        public int NumberOfRooms { get; set; }
        public int Floor { get; set; }
        public bool IsUsing { get; set; }
        public string Description { get; set; }

        public Hotel Hotel { get; set; }

        public List<Order> Orders = new List<Order>();

        public Room()
        {
            this.Orders = new List<Order>();
        }
    }
}
```

ShortUserInfo.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Threading.Tasks;

namespace HeyHotel.Models
{
    public class ShortUserInfo
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public string SName { get; set; }
        public string Email { get; set; }
    }
}
```

User.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;

namespace HeyHotel.Models
{
```

```
    public class User : IdentityUser
    {
        public string Name { get; set; }
        public string SName { get; set; }
        public int Year { get; set; }

        List<Order> Orders = new List<Order>();

        public User()
        {
            Orders = new List<Order>();
        }
    }
}
```

UserDbContext.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
```

```
namespace HeyHotel.Models
{
    public class UsersDbContext : IdentityDbContext<User>
    {
        public UsersDbContext(DbContextOptions<UsersDbContext> options) :
base(options)
        {
            Database.EnsureCreated();
        }
    }
}
```