МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА»
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

C# ТЕХНОЛОГІЇ ПРИКЛАДНОГО ПРОГРАМУВАННЯ. ПЛАТФОРМА DOTNET CORE

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт
з дисциплін
«Java та C# технології прикладного програмування»
та «Технології прикладного програмування»
для здобувачів вищої освіти
першого (бакалаврського) рівня вищої освіти
спеціальностей 121 – «Інженерія програмного забезпечення» та 123 – «Комп’ютерна інженерія»

Обговорено і рекомендовано
на засіданні кафедри ІТ та ПІ
протокол №1
від 22 січня 2024 р.

Чернігів НУ «Чернігівська політехніка» 2024

C# технології прикладного програмування. Платформа dotnet core. Методичні вказівки до виконання лабораторних робіт з дисциплін «Java та C# технології прикладного програмування» та «Технології прикладного програмування» для здобувачів вищої освіти першого (бакалаврського) рівня вищої освіти спеціальностей 121 – «Інженерія програмного забезпечення» та 123 – «Комп'ютерна інженерія» / Укл.: Задорожній А.О. – Чернігів: НУ «Чернігівська політехніка», 2024. – 124 с.

Укладачі: Задорожній А. О., к.т.н., доцент, доцент кафедри ІТ т ПІ

Відповідальний за випуск: Білоус Ірина Володимирівна, к.т.н., доцент, завідувач кафедри ІТ та ПІ

Рецензент: Базилевич В. М., к.е.н., доцент, завідувач кафедри ІКС

СПИСОК УМОВНИХ СКОРОЧЕНЬ

ORM	-	Object Relation Mapping
LINQ	-	Language Integrated Query
VS	-	Visual Studio
DAO	-	Data Access Object
HTML	-	Hyper Text Markup Language
XML	-	Extensible Markup Language
SQL	-	Structured Query Language
CRUD	-	Create, Read, Update and Delete
ASP	-	Active Server Pages
MVC	-	Model View Controller

ЗМІСТ

ВСТУП

Об’єктно-орієнтовні мови є ефективним засобом для створення складних застосунків. З часу появи перших об’єктна-орієнтованих мов виробився ряд підходів і технологій, які значно спрощують створення застосунків. Наприклад, однотипні дані прийнято зберігати в масивах чи колекціях даних, для роботи з базами даних прийнято використовувати драйвери баз даних, створені під певну мову програмування. Досвід, що накопичувався методо спроб і помилок переріс в технології, що значно спрощують розробку застосунків. Дані методичні вказівки створені, щоб допомогти отримати практичні навички по роботі з базовими технологіями платформи dotnet core, технологіями взаємодії з базами даних, web-технологіями і технологіями модульного тестування на базі платформи dotnet core.

В даних методичних вказівках розглядаються такі технології як ADO.NET – технологія роботи з базами даних, Fluent NHibernate – технологія об’єктно-реляційного відображення, ASP.NET core MVC – технологія створення web-застосунків, MS Unit Test Framework – технологія модульного тестування застосунків, а також розглядаються базові технології платформи dotnet core.

В деяких лабораторних роботах даних методичних вказівок описані шаблони проектування, які доцільно використовувати при вирішенні тих чи інших задач. Наприклад, в лабораторній роботі з об’єктна-реляційного відображення розглядається архітектурний шаблон Data Access Object, а також шаблон проектування Factory. Це повинно допомогти сформувати уявлення про використання шаблонів проектування при створенні застосунків з використанням об’єктно-орієнтованих мов програмування.

1 ЛАБОРАТОРНА РОБОТА № 1 КОЛЕКЦІЇ В DOTNET CORE

1.1 Мета роботи

Ознайомитися з особливостями створення додатків на базі платформи dotnet core на прикладі роботи з колекціями.

1.2 Теоретичні відомості

1.2.1 Загальні відомості про платформу .NET.

Платформа - це середовище виконання програм і все те, що визначає особливості розробки і виконання програмного коду – парадигми програмування, множина базових класів в т.д.

Microsoft.NET (.NET Framework) – програмна платформа, що є інтегрованим компонентом Windows, включає віртуальне середовище виконання (середовище CLR) і уніфікований набір бібліотек класів. В платформі .NET можна виділити наступні основні компоненти:

- Common Language Specification (CLS);
- Microsoft Intermediate Language (MSIL);
- Common Language Runtime (CLR);
- .NET Framework Class Library (.NET FCL).

CLS (Common Language Specification) – загальна специфікація мов програмування. Це набір конструкцій і обмежень, які є керівництвом для розробників бібліотек і компіляторів в середовищі .NET Framework. Бібліотеки, побудовані в відповідності до CLS, можуть бути використані з біль-якої мови програмування, що підтримує CLS. Мови, що відповідають CLS (до їх числа відносяться мови Visual Basic, Visual C++, Visual C#), можуть інтегруватися одна з одною. CLS – це основа міжмовної взаємодії в рамках платформи Microsoft.NET.

CLR (Common Language Runtime) – середовище виконання або віртуальна машина – основний компонент .NET Framework.

CLR забезпечує виконання збірки. Програмний код (модуль) збірки може бути я керованим (код, при виконанні якого CLR активує систему управління пам’яттю), так і некерованим. Під віртуальною машиною (середовищем часу виконання) розуміють абстракцію інкапсульованої (відокремленої) керованої операційної системи високого рівня, яка забезпечує виконання програмного коду і припускає вирішення наступних задач:

- управління кодом (завантаження і виконання);
- управління пам’яттю при розміщенні об’єктів;
- ізоляція пам’яті застосунків;
- перевірка безпечності коду;
- перетворення проміжної мови в машинний код;
- доступ до метаданих (розширена інформація про типи);

- обробка помилок, включаючи міжмовні помилки;
- взаємодія між керованим і некерованим кодом (в тому числі з COM - об’єктами);
- підтримка сервісів розробки (профілювання, відлагодження і т.д.).

Таким чином, CLR – це набір служб, необхідних для виконання збірки.

Сама CLR складається з двох головних компонентів:

- ядра (mscoree.dll);
- бібліотеки базових класів (mscorlib.dll).

Ядро середовища виконання реалізоване в вигляді бібліотеки mscoree.dll. При компонуванні збірки в неї вбудовується спеціальна інформація, яка при запуску застосунку (.exe) або при завантаженні бібліотеки (звернення до .dll із некерованого модуля - виклик функції LoadLibrary для завантаження керованої збірки) приводить до завантаження в ініціалізації CLR. Після завантаження CLR в адресний простір процесу, ядро середовище виконання виконує наступні дії:

- знаходить місцезнаходження збірки;
- завантажує збірку в пам'ять;
- виконує аналіз вмісту збірки (виявляє класи, структури, інтерфейси);
- виконує аналіз метаданих;
- забезпечує компіляцію коду на проміжній мові (IL) в платформонезалежні інструкції (асемблер ний код);
- виконує перевірки, пов’язані з забезпеченням безпеки;
- використовуючи основний потік застосунку, передає керування перетвореному в команди процесора фрагменту коду збірки.

.NET FCL (.NET Framework Class Library) – об’єктно-орієнтована бібліотека, що відповідає CLS специфікації і містить класи, інтерфейси і системи типів (типів-значень), що включаються в платформу .NET.

Ця бібліотека забезпечує доступ до функціональних можливостей системи і виступає в якості основи при розробці .NET застосунків, компонентів і елементів управління. Бібліотека .NET FCL є другим основним компонентом платформи .NET.

Бібліотеку .NET FCL можуть використовувати всі .NET-застосунки, незалежно від призначення, архітектури і використовуваної при розробці мови програмування. Бібліотека включає більше 4000 класів. Зокрема, вона містить:

1. Вбудовані (елементарні) типи, представлені у вигляді класів (на платформі .NET все побудовано на структурах і класах).
2. Класи для розробки графічного інтерфейсу користувача.
3. Класи для розробки web-застосунків і web-служб на основі технології ASP.NET (Web Forms).
4. Класи для розробки застосунків, що працюють з XML і internet-протоколами (FTP; HTTP; SMTP; SOAP).

5. Класи для розробки застосунків, що працюють з базами даних (ADO.NET).

.NET застосунок – застосунок, розроблений для виконання на платформі .NET і реалізований на мова програмування, що відповідають CLS.

MSIL (Microsoft Intermediate Language или IL – Intermedia Language) – проміжна мова платформи .NET. Вихідні тексти програм для .NET застосунків пишуться на мовах програмування, що відповідають специфікації CLS.

Для мов програмування, що відповідають специфікації CLS може бути побудований перетворювач в MSIL. Таким чином, програми на цих мовах можуть транслюватися в проміжний код на MSIL. Завдяки відповідності CLS, в результаті трансляції програмного коду, написаного на різних мовах, виходить сумісний IL код.

Фактично MSIL є асемблером віртуальної машини. Пр. перетворенні програмного коду в MSIL формується блок метаданих. Цей блок містить інформацію про дані, що використовуються в програмі.

Фактично блок метаданих – це набір таблиць, що містять інформацію про типи даних, визначених в модулі, і про типи даних, на які посилається даний модуль. Раніше така інформація зберігалась окремо. Наприклад, застосунок міг включати інформацію про інтерфейси, що описувались на Interface Definition Language (IDL). Тепер метадані є частино. Керованого модуля.

Зокрема, метадані використовуються для:

1. Збереження інформації про типи. При компіляції тепер нема необхідності в заголовочних і бібліотечних файлах. Всю необхідну інформацію компілятор читає безпосередньо з керованих модулів.
2. Верифікації коду в процесі виконання модуля.
3. Управління динамічною пам’яттю (вивільнення пам’яті) в процесі виконання модуля.
4. Динамічної підказки (IntelliSense) при розробці програми стандартними інструментальними засобами (MS Visual Studio.NET).

Мови, для яких реалізований переклад на MSIL достатньо широкий:

– Visual Basic;
– Visual C++;
– Visual C#;
– і більше 20 інших CLS - сумісних мов.

1.2.2 Загальні відомості про колекції C#

Колекція чи контейнер – це клас, що дозволяє зберігати і виконувати операції на множиною об’єктів. Колекції використовуються для збереження, отримання, маніпулювання даними і забезпечують агрегацію одних об’єктів з іншими.

В багатьох мовах програмування (Java, C, C++, Pascal) одним з вбудованих в мову програмування засобом збереження об’єктів є масиви. Однак масиви мають ряд недоліків, одним з яких є кінцевий розмір масиву, і як наслідок необхідність слідкувати за розміром масиву. Іншим недоліком є індексація, що обмежує можливості додання і видалення об’єктів. Стандартний набір колекцій C# слугує для позбавлення програміста від необхідності самостійно реалізовувати структури даних і надає йому додаткові можливості.

Головна перевага колекцій полягає в тому, що вони стандартизують обробку груп об’єктів в програмі.

Всі колекції розроблені на основі чітко визначених інтерфейсів. Деякі вбудовані реалізації таких інтерфейсів, в тому числі ArrayList, Hashtable, Stack і Queue, можуть застосовуватися в початковому вигляді і без будь-яких змін. Також є можливість реалізувати власну колекцію, хоча потреба в цьому виникає дуже рідко.

В середовищі .NET Framework підтримується п’ять типів колекцій: неузагальнені, спеціальні, з порозрядною адресацією, узагальнені і паралельні. Неузагальнені колекції реалізують ряд основних структур даних, включаючи динамічний масив, стек, чергу а також словники, в яких можна зберігати пар «ключ-значення». По відношенню неузагальнених колекцій важливо мати на увазі наступне: вони оперують даними типу object.

Таким чином, неузагальнені колекції можуть слугувати для зберігання даних будь-якого типу, причому в одній колекції допускається наявність різнотипних даних.

Очевидно, що такі колекції не типізовані, оскільки них зберігаються посилання на дані типу object. Класи і інтерфейси неузагальнених колекцій знаходяться в просторі імен *System.Collections.*

Спеціальні колекції оперують даними конкретного типу або ж роблять особливим чином. Наприклад, є спеціальні колекції для символьних рядків, а також спеціальні колекції, в яких використовується однонаправлений список. Спеціальні колекції об’являються в просторі імен *System.Collections.Specialized*.

В прикладному інтерфейсі Collections API визначена одна колекція з порозрядною організацією - це BitArray. Колекція типу BitArray підтримує порозрядні операції (операції над окремими двійковим розрядами), наприклад AND і XOR, а, відповідно, вона суттєво відрізняється своїми можливостями від інших типів колекцій. Колекція типу BitArray об’являється в просторі імен *System.Collections*.

Узагальнені колекції забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи зв’язані списки, стеки, черги і словники.

Такі колекції є типізованими в силу їх узагальненого характеру. Це значить, що в узагальнені колекції можуть зберігатися тільки такі елементи даних, які сумісні по типу з даними колекцій. Завдяки цьому виключається

випадкова розбіжність типів. Узагальнені колекції об’являються в просторі імен *System.Collections.Generic*.

Паралельні колекції підтримують багатопоточний доступ до колекції. Ці узагальнені колекції визначені в просторі імен *System.Collections.Concurrent*.

В просторі імен *System.Collections.ObjectModel* знаходиться також ряд класів, що підтримують створення користувачами власних узагальнених колекцій.

Основним для всіх колекцій є поняття енумератора, який підтримується в неузагальнених інтерфейсах IEnumerator і IEnumerable, а також в узагальнених інтерфейсах IEnumerator<T> і IEnumerable<T>.

Енумератор забезпечує стандартний спосіб почергового доступу до елементів колекції. Відповідно він перераховує вміст колекції.

В кожній колекції повинна бути реалізована узагальнена або неузагальнена форма інтерфейсу IEnumerable, тому елементи будь-якого класу колекції повинні бути доступні за допомогою методів, визначених в інтерфейсі IEnumerator чи IEnumerator<T>.

Це означає, що внісши мінімальні зміни в код циклічного звернення до колекції одного типу, його можна використовувати для аналогічного звернення до колекції іншого типу. Для почергового звернення до вмісту колекції в циклі foreach використовується енумератор.

З енумератором безпосередньо пов'язаний і інший засіб, що називається ітератором. Це засіб спрощує процес створення класів колекцій, наприклад, спеціальних, почергове звернення до яких організовується в циклі foreach.

І останнє зауваження: якщо у вас є деякий досвід програмування на C++, то вам, вірогідно, буде корисно знати, що класи колекцій по своїй суті подібні класам стандартної бібліотеки шаблонів (Standard Template Library — STL), визначеної в C++. Те, що в програмуванні на C++ називається контейнером, в програмуванні на C# називається колекцією. Це ж відноситься і до Java. Якщо ви знайомі з бібліотекою Collections Framework для Java, то навчитися користуватися колекціями C# не буде складно.

1.2.3 Узагальнені колекції C#

Узагальнені колекції забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи пов’язані списки, стеки, черги і словники.

Такі колекції є типізованими в силу їх узагальненого характеру.

Це значить, що в узагальненій колекції можуть зберігатися тільки такі елементи даних, які сумісні по типу з даною колекцією.

Завдяки цьому виключається випадкова невідповідність типів. Узагальнені колекції оголошуються в просторі імен *System.Collections.Generic*. Узагальнені колекції представлені в таблиці 1.1.

Таблиця 1.1 – Узагальнені колекції C#

List<T>	Представляє строго типізований список об’єктів, доступних по індексу. Підтримує методи для пошуку по списку, виконання сортування і інші операції зі списками
HashSet<T>	Представляє набір значень
LinkedList<T>	Представляє зв’язаний список
Dictionary<TKey, TValue>	Представляє колекцію ключів і значень
Queue<T>	Представляє колекцію об’єктів, що базується на принципі FIFO
SortedDictionary<TKey, TValue>	Представляє колекцію пар "ключ-значення", відсортованих по ключу
SortedList<TKey, TValue>	Представляє список, відсортований по ключу
SortedSet<T>	Представляє впорядковану колекцію об’єктів
Stack<T>	Представляє колекцію об’єктів, що базується на принципі LIFO

1.3 Порядок виконання лабораторної роботи

1. Завантажте з офіційного сайту Microsoft і встановіть найсвіжішу версію Dotnet Core SDK. На момент створення даних методичних вказівок найсвіжіша версія була Dotnet Core SDK 3.1, яку можна було завантажити за посиланням https://dotnet.microsoft.com/en-us/download/dotnet/3.1.

2. Завантажте з офіційного сайту і встановіть середовище розробки Visual Studio Code. На момент створення методичних вказівок середовище розробки Visual Studio Code можна було завантажити з сайту https://code.visualstudio.com/.

3. Відкрийте Visual Studio Code і додайте декілька плагінів, що допоможуть при розробці застосунків на мові C#. На рисунку 1.1 представлений список встановлених плагінів в середовищі розробки Visual Studio Code, що є далеко не остаточним. Плагін C# додасть можливість висвітлювати підказки під час написання коду, а плагін Auto-Using for C# допоможе автоматично додавати залежні пакети при написанні коду. Якщо ввести в поле пошуку рядок C#, то можна знайти багато додаткових плагінів, що значно спростять написання коду в середовищі Visual Studio Code.

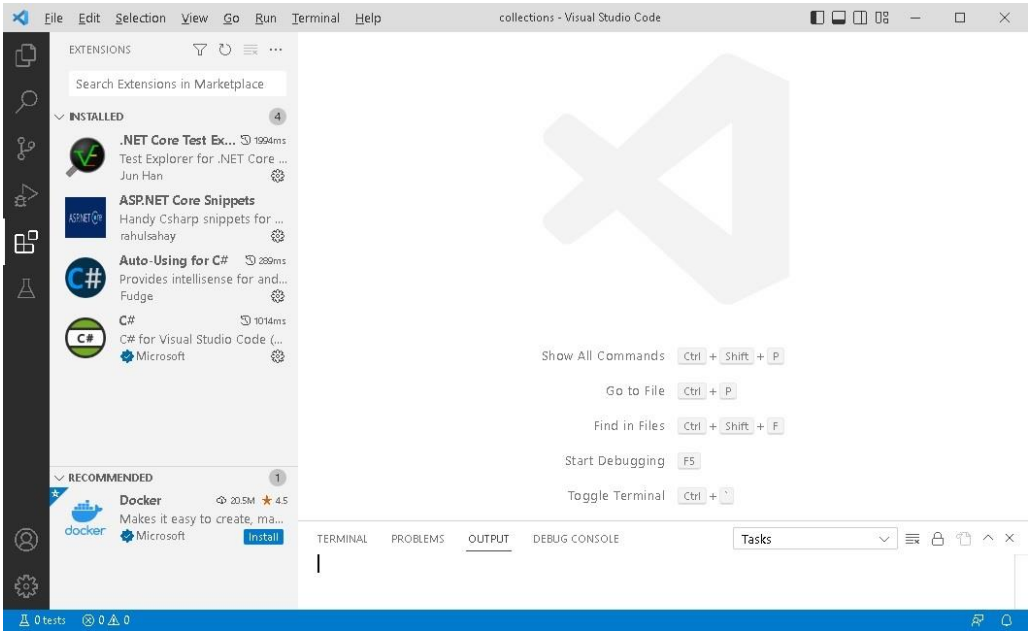

Рисунок 1.1 - Список встановлених плагінів в середовищі розробки Visual Studio Code

4. Створіть директорію, в якій будуть зберігатися проекти з циклу лабораторних робіт і всередині даної директорії створіть директорію для першої лабораторної роботи. В якості прикладу для зберігання проектів біла створена директорія csexamples а всередині директорії csexamples для зберігання проекту для першої лабораторної роботи була створена директорія collections.

5. Відкрийте середовище розробки Visual Studio Code і відкрийте директорію collections (File->Open Folder). Відкрийте термінал (Terminal -> New Terminal). Середовище розробки має виглядати як показано на рисунку 1.2.

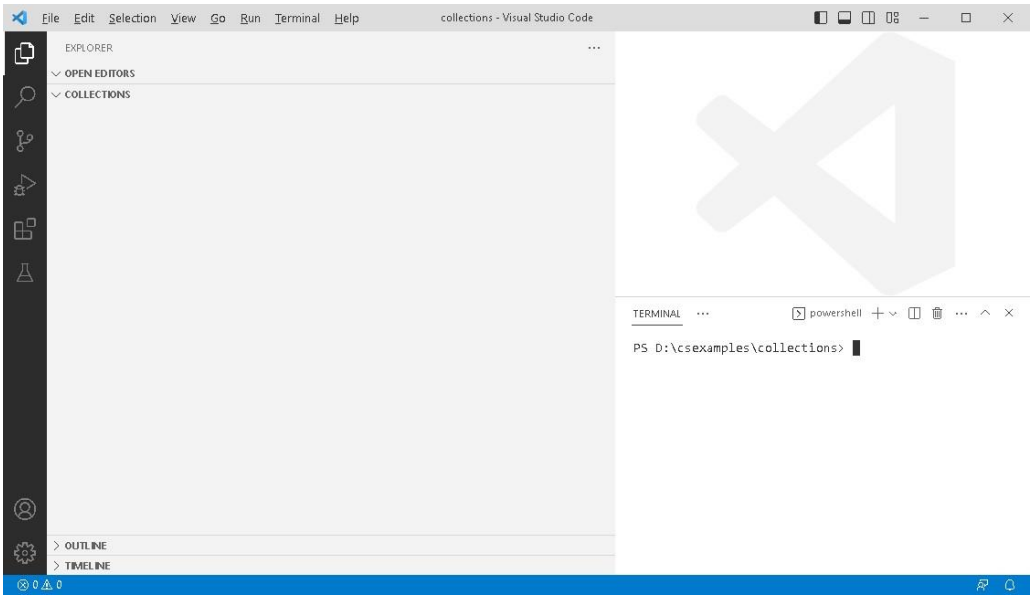

Рисунок 1.2 – Зовнішній вигляд середовища розробки після відкриття директорії проекту лабораторної роботи

6. В терміналі введіть команду dotnet new console --use-program-main. Це спричинить створення всіх необхідних файлів проекту. Якщо при виконанні команди буде отримана помилка, що команда dotnet не знайдена, необхідно додати до системної змінної Path директорію, де встановлена команда dotnet. Зазвичай це C:\Program Files\dotnet\. Середовище розробки має виглядати як показано на рисунку 1.3.

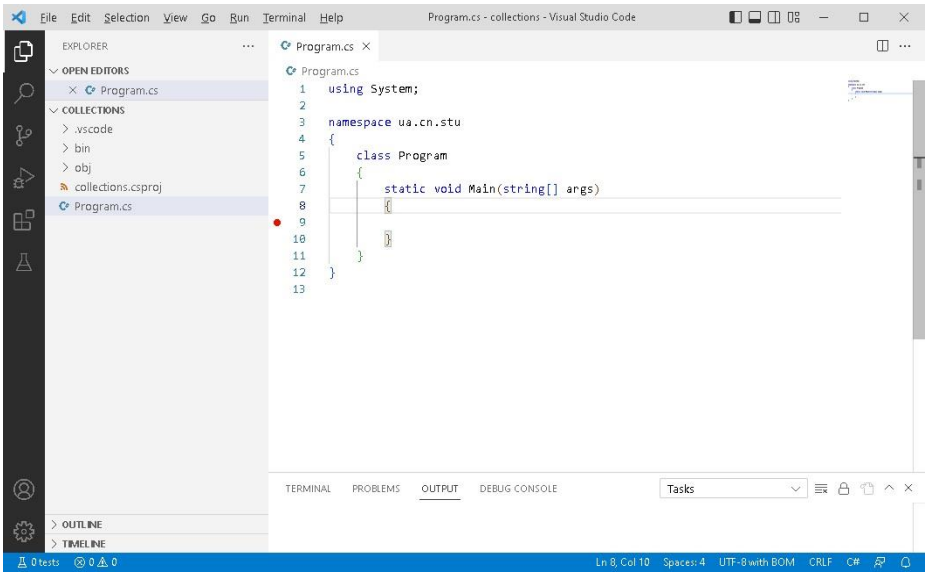

Рисунок 1.3 – Зовнішній вигляд середовища після створення нового проекту

7. Створіть новий клас Student.cs з кодом приведеним нижче:

```
namespace ua.cn.stu
{
    class Student
    {
        private string name;
        private string surname;
        private string  recordBookNumber;

        public Student(string name, string surname, string recordBookNumber)
        {
            this.name = name;
            this.surname = surname;
            this.recordBookNumber = recordBookNumber;
        }

        public string getName()
        {
            return this.name;
        }

        public string getSurname()
        {
            return this.surname;
        }

        public string getRecordBookNumber()
        {
            return this.recordBookNumber;
        }

        public void setName(string name)
        {
```

```
            this.name = name;
        }

        public void setSurname(string surname)
        {
            this.surname = surname;
        }

        public void setRecordBookNumber(string recordBookNumber)
        {
            this.recordBookNumber = recordBookNumber;
        }
    }
}
```

8. Змініть код класу Program, як показано нижче:

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

namespace ua.cn.stu
{
    class Program
    {

        private static IList<Student> studentList = new List<Student>();

        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Please enter command add, list or exit:");
                String command = Console.ReadLine();
                if (command == "add")
                {
                    Console.WriteLine("Please enter student name:");
                    String name = Console.ReadLine();
                    Console.WriteLine("Please enter student surname:");
                    String surname = Console.ReadLine();
                    Console.WriteLine("Please enter record book number:");
                    String recordBookNumber = Console.ReadLine();
                    Student student =
                        new Student(name, surname, recordBookNumber);
                    studentList.Add(student);
                    Console.WriteLine(
                        "Student was succesfully added to collection");
                }
                else if (command == "list")
                {
                    foreach(Student student in studentList)
                    {
                        Console.WriteLine("Student name -> "
                              + student.getName()
                            + ", Student surname -> " + student.getSurname()
                            + ", Student recordbook number -> "
                              + student.getRecordBookNumber());
                    }
                }
                else if (command == "exit")
                {
                    Console.WriteLine("Exiting the program.");
                    return;
                }
            }
```

```
            }
        }
}
```

9. Для того, щоб перевірити працездатність програми, необхідно її запустити. Visual Studio Code дозволяє запускати програму як у режимі трасування (Run->Start Debugging), так і у звичайному режимі (Run->Run Without Debugging). Консоль, вбудована в Visual Studio Code не дозволяє працювати з функціє Console.ReadLine(), тому перш ніж запускати програму на виконання необхідно налаштувати проект, щоб використовувати зовнішню консоль. Відкрийте в проекті файл lanch.json і змініть значення поля console з internalConsole на externalTerminal.

10. Запустіть програму, перевірте її працездатність. Запустіть програму в режимі трасування і проекспериментуйте з даним.

11. Виконайте завдання відповідно до варіанту.

1.4 Завдання до лабораторної роботи

Змінити застосунок, описаний в порядку виконання лабораторної роботи відповідності до варіанту завдання.

Застосунок повинен забезпечувати можливість додання, елементів в колекцію, видалення елементів із колекції, відображення елементів колекції.

Варіанти завдань, з яких необхідно обрати об’єкт предметної області і колекцію, приведені в таблиці 1.2. Для об’єкта предметної області необхідно запропонувати не менше ніж п’ять різнотипних полів, що містять інформацію. Номер варіанта необхідно вибрати по останній цифрі залікової книжки.

Таблиця 1.2 – Варіанти завдань

Остання цифра залікової книжки	Об’єкт предметної області	Колекція
0	Книжка	HashSet<T>
1	Громадянин	LinkedList<T>
2	Автомобіль	Dictionary<TKey, TValue>
3	Водій	Queue<T>
4	Комп’ютер	SortedDictionary<TKey, TValue>
5	Громадський транспорт	SortedList<TKey, TValue>
6	Вимірювальний прилад	SortedSet<T>
7	Будинок	Stack<T>
8	Мобільний телефон	HashSet<T>
9	Товар	LinkedList<T>

1.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;
- номер і назва лабораторної роботи;
- мета лабораторної роботи;

- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

1.6 Контрольні питання

1. Що таке платформа?
2. Із яких частин складається платформа .NET?
3. Для чого використовується *CLR?*
4. Для чого використовується MSIL?
5. Що таке колекція?
6. Які колекції ви знаєте?
7. Які узагальнені колекції ви знаєте?
8. Чим відрізняються узагальнені колекції від неузагальнених?
9. Чим відрізняється List від SortedList?
10. Чим відрізняється HashSet від SortedSet?

1.7 Завдання до захисту лабораторної роботи

1. Додати валідацію одного з полів об’єкта предметної області при доданні нового елементу.
2. Додати команду пошуку елементів колекції по одному з полів об’єкта предметної області.
3. Додати команду видалення елемента колекції по одному з полів об’єкта предметної області.
4. Додати команду редагування полів елемента колекції.
5. Додати команду збереження елементів колекції в JSON файл.
6. Додати команду відновлення елементів колекції з JSON файлу.
7. Додати команду збереження елементів колекції в XML файл.
8. Додати команду відновлення елементів колекції з XML файлу.
9. Додати пошуку елементів колекції по будь-якому полю з подальшим збереженням в текстовий файл.
10. Додати команду пошуку елементів колекції по будь-якому полю з використанням технології LINQ to Object.

2 ЛАБОРАТОРНА РОБОТА № 2
РОБОТА З ADO.NET В DOTNET CORE

2.1 Мета роботи

Ознайомитися з технологією ADO.NET і особливостями її використання в dotnet core. Отримати практичні навички по роботі з технологією ADO.NET.

2.2 Теоретичні відомості

2.2.1 Загальні відомості про ADO.NET

ADO.NET *(ActiveX Data Object .NET)* – набір класів, що використовується для доступу до джерел даних в платформі .NET.

ADO.NET підтримує асинхронний доступ до даних і серіалізацію даних, отриманих зі сховища, в форматі XML.

ADO.NET не пропонує єдиного набору типів зв'язку з усіма СУБД. В просторі імен System.Data.Common знаходяться базові класи і інтерфейси, а відповідні похідні класи для різних постачальників даних – в відповідних просторах імен System.Data.SqlClient, System.Data.Odbc і т.д.

І хоча імена цих реалізуючи типів різні, можна створити універсальний програмний код, завдяки використанню поліморфізму і спеціально обєкта класу DbProviderFactory, з допомогою якого витягуються об'єкти даних, специфічні для кожного постачальника.

Бібліотеки ADO.NET можуть використовуватися в рамках двох концептуально різних способів взаємодії: на зв'язаному рівні, для чого використовуються класи приєднаних об'єктів, або на незв'язаному рівні, де потрібні класи від'єднаних об'єктів (рисунок 2.1).

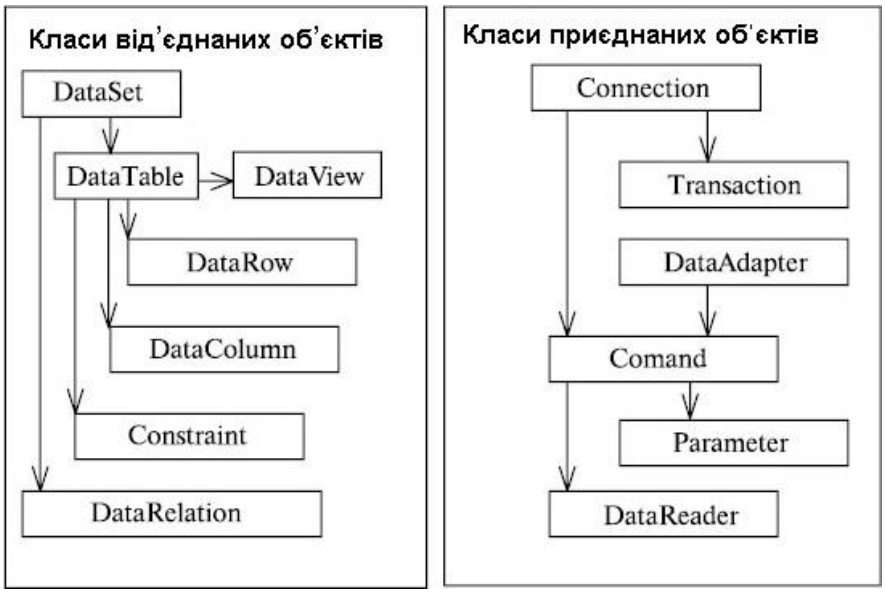

Рисунок 2.1 – Організація класів ADO.NET

При використанні зв’язаного рівня програмний код з’єднується безпосередньо з відповідним сховищем даних і від’єднується від нього, коли задачі взаємодії вирішені. Незв’язаний рівень, навпаки, дозволяє отримати набір об’єктів, що функціонують к клієнтські копії зовнішніх даних.

2.2.2 Зв’язаний рівень взаємодії ADO.NET

Робота зі зв’язаним рівнем дозволяє взаємодіяти з базою даних, використовуючи об’єкти з’єднання, об’єкт команди і об’єкт читання даних.

Першим кроком в роботі з постачальниками даних є створення з’єднання, для чого використовується об’єкт з’єднання похідний від *DbConnection.*

Припустимо, що в файлі конфігурації вказані рядки з’єднання і постачальника даних:

```
< configuration>
<appSettings>
<!-- Який постачальник? -->
<add key="provider" value="System.Data.SqlClient" />
<!-- Який рядок з’єднання? -->
<add key="cnStr" value=
"Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/> </appSettings>
</configuration>
```

Тоді відповідний програмний код для з’єднання може виглядати наступним чином:

```
// Отримання рядків з’єднання і постачальника даних
// з конфігураційного файлу
string dp = ConfigurationManager.AppSettings("provider");
string cnStr = ConfigurationManager.AppSettings("cnStr");
// Створення джерела постачальника
DbProviderFactory df = DbProviderFactories.GetFactory(dp);
// Створення об’єкта з’єднання
DbConnection cn = df.CreateConnection();
cn.CornneccionString = cnStr;
cn.Open();
```

Далі необхідно створити і налаштувати об’єкт команди, передавши йому об’єкт з’єднання у вигляді параметра конструктора або за допомогою властивості *Connection.* Вид відповідної команди вказується з допомогою властивості *CommandType,* яка може приймати значення *StoredProcedure*, *TableDirect,* а за замовчування приймає значення *Text.*

```
//Створення об’єкта команди
DbCommand cmd = df.CreateComnmand();
cmd.Connection = cn;
cmd.CommandText = "Select * From Authors";
```

Треба розуміти, в цей момент ще не виконується запит до бази даних безпосередньо, а тільки йде підготовка об’єкту команди для використання в майбутньому.

Після створення активного з’єднання і SQL-команди наступним кроком є виконання запиту до джерела даних. Тип *DbDataReader* (що реалізує

IDataReader) забезпечує найпростіший і найшвидший спосіб отримання даних від джерела даних.

Об'єкт читання даних створює однонаправлений і доступний тільки для читання потік даних, що повертає по одному запису за один раз. Тому повинно бути очевидно, що об'єкти читання даних використовуються для відправлення сховищу даних тільки SQL-операторів для вибирання даних.

Об'єкт читання даних є корисним тоді, коли необхідно дуже швидко переглянути великий об'єм даних і при цьому нема необхідності підкуватися про їх розміщенні в пам'яті. Наприклад, якщо йде запит на 20000 записів з таблиці для того щоб зберегти їх в текстовому файлі, то при зберіганні цієї інформації в *DataSet* виникає достатньо велике навантаження на пам'ять.

Більш вигідним рішенням є створення об'єкта читання даних, який буде оброблювати кожен запис настільки швидко, наскільки це можливо. Але при цьому треба враховувати те, що об'єкт читання даних (на відміну від об'єкта адаптера даних, який буде розглянутий пізніше) підтримує відкрите з'єднання з джерелом даних, поки явно не закриється сеанс зв'язку.

Об'єкт читання даних отримується з об'єкта команд з допомогою виклику *ExecuteReader*(). При виклику цього методу об'єкту читання даних можна додатково дати інструкцію автоматично закривати відповідний об'єкт з'єднання, передавши в метод *ExecuteReader*() в якості параметра значення *CommandBehavior.CloseConnection.*

Після підготовки можна обробити кожен запис, використовуючи метод Read() об'єкта читання даних.

```
// Виведення даних з допомогою обєкта читання даних
DbDataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
while (dr.Read())
Console.WriteLine("{0}, {1}", dr[au.lname"], dr[au.fname");
// Оскільки було передано значення CommandBehavior.CloseConnection,
// то нема необхідності явно викликати метод Close()
dr.Close();
```

Якщо необхідно застосувати команду, в результаті якої повинна відбутися модифікація таблиці, то необхідно викликати метод *ExecuteNonQuery*.Цей метод виконує вставку, оновлення і видалення в відповідності до формату відповідної команди.

Також об'єкт команди ADO.NET підтримує колекцію дискретних типів параметрів. За замовчування ця колекція пуста, але можна додати в неї будь-яку кількість об'єктів параметрів, які замінять собою параметри в SQL-запиті. Щоб асоціювати параметри в SQL-запиті з елементом колекції параметрів даного обєкта команди, необхідно додати до текстового SQL-параметру префікс @ (це працює, як мінімум, при використанні Microsoft SQL Server, але таке позначення підтримують не всі СУБД).

```
string sql = string.Format("Insert Into Inventory" + "(CarlD, Make, Color,
PetName, Values" + "(@CarID, @Make, @Color, @PetName)");
// Заповнення колекції параметрів
SqlParameter param = new SqlParameter();
param.ParameterName = "@CarID";
```

```
param.Value = newCarID;
param.SqlDbType = SqlDbType.Int;
cmd.Parameters.Add(param);
```

2.2.3 Незв’язаний рівень взаємодії ADO.NET

При роботі з незв’язним рівнем ADO.NET все ще необхідно використовувати об’єкт з’єднання і об’єкт команди. Крім того, необхідно використовувати спеціальний об’єкт, що називається адаптером даних (який розширює абстрактний клас *DbDataAdapter*) щоб отримувати і оновлювати дані.

Об’єкт адаптера даних оброблює з’єднання з базою даних автоматично. З метою розширення можливостей масштабованості адаптери даних зберігають з’єднання відкритим мінімально можливу кількість часу. Як тільки визиваючи сторона отримує об’єкт *DataSet* з’єднання з СУБД закривається і викликаючи сторона залишається зі своєю локальною копією віддалених даних.

DataSet дозволяє клієнту імітувати постійно відкрите з єднання, в той час як реальні операції виконуються з наборами даних, що знаходяться в пам’яті.

На відміну від зв’язаного рівня, дані, отримані з допомогою адаптера даних, не оброблюються з допомогою об’єкта читання даних. Замість цього для переміщення даних між стороною, що викликає, і джерелом даних об’єкт адаптера даних (*DataAdapter*) використовує об’єкт *DataSet*. Тип *DataSet* – це контейнер, що використовується для будь-якої кількості об’єктів *DataTable*, кожний з яких містить колекцію об’єктів *DataRow* і *DataColumn*.

В рамках від’єднаного рівня ADO.NET об’єкт *DataSet* є важливим елементом технології від’єднаного рівня доступу. Об’єкт-представник *DataSet* містить в собі множину таблиць.

Для успішного вирішення задачі відображення структури бази даних в *DataSet* є все необхідне. Його функціональні можливості дозволяють завантажити в локальне сховище на стороні застосунку дані з будь-якого доступного для ADO.NET джерела: SQL Server, Microsoft Access, XML-файл.

Серед даних-члені цього класу є набір *Tables*. Об’єкт *DataSet* може містити таблиці, кількість яких обмежується лише можливостями набору *Tables*.

Для кожної таблиці - елементу набору *Tables* може бути (і повинна бути) визначена структура таблиці. У випадку, коли застосунок взаємодіє з реальною базою даних, структура таблиць в *DataSet* повинна відповідати структурі таблиць в базі даних. *DataSet* – це об’єкт ADO.NET, що знаходиться в пам’яті і використовується в застосунку для представлення даних. Він визначає узгоджену реляційну модель бази даних, яка не залежить від джерела даних і даних, що в ньому містяться. Ступінь повноти моделі визначається завданнями, які вирішує застосунок.

Об’єкт *DataSet* може представляти абсолютно точну модель бази даних, і в такому випадку ця модель повинна буде включати повний набір

структурних елементів бази даних, включаючи таблиці, що містять дані, з урахуванням встановлених обмежень і відношень між таблицями.

Інформацію, що міститься в об'єкті можна міняти незалежно від джерела даних. Відповідні значення формуються безпосередньо в програмі і додаються в таблицю.

При роботі з базою даних дані можуть збиратися з різних таблиць, локальне відображення яких забезпечується різноманітними об'єктами – представниками класів *DataSet*. В класі *DataSet* визначено чимало перевантажених методів *Merge* , які позволяють об'єднувати вміст декількох об'єктів *DataSet*.

Будь-який об'єкт-представник класу *DataSet* дозволяє організувати читання і запис вмісту (теоретично – інформації з бази) в файл чи область пам'яті. При цьому можна читати і зберігати:

– тільки вміст об'єкту (відповідно дані за бази);
– тільки структуру об'єкта-представника класу *DataSet*;
– повний образ *DataSet* (вміст і структуру).

Таким чином, *DataSet* є основою для побудови різних варіантів від'єднаних об'єктів – сховищ інформації.

Клас *DataSet* – клас і не абстрактний і не інтерфейс. Це значить, що існує велика кількість варіантів побудови від'єднаних сховищ. На основі базового класу *DataSet* можна визначати похідні класи певної конфігурації, яка відповідає структурі бази даних.

Можна також створити об'єкт-представник класу *DataSet* оригінальної конфігурації і додати безпосередньо до цього об'єкту всі необхідні складові у вигляді таблиць (об'єктів-представників класу Table) відповідної структури і множини відношень Relation.

Об'єкт-представник класу *DataSet* і сам по собі буз супутнього оточення має певну цінність. Дані, що представляються в застосунках у вигляді таблиць, не обов'язково повинні мати зовнішнє джерело даних у вигляді реальної бази даних. Ніщо не заважає забезпечити в застосунку читання звичайного файлу або навіть просто накопичити необхідні дані шляхом інтерактивної взаємодії з користувачем, використовуючи при цьому звичайний діалог.

З рештою, база даних – це один із можливих способів організації даних. Не випадково *DataSet* представляє від'єднанні дані.

База даних характеризується множиною таблиць і множиною відношень між таблицями.

Оголошення класу *DataSet* включає:

– набір (можливо, пустий) оголошень класів *DataTable* (фактично це опис структури таблиць, що входять в даний клас *DataSet*);
– набір оголошень класів *DataRelations*, який забезпечує встановлення зв'язків між різними таблицями в рамках даного *DataSet*.

Структур *DataSet* може цілком повторювати структуру БД і містити повний перелік таблиць і відношень, а може бути частковою копією БД і

містити, відповідно, лише підмножину таблиць і підмножину відношень. Все визначається задачею, що вирішується з допомогою даного об’єкту *DataSet*.

Об’єкт-представник даного класу *DataSet* забезпечує відображення в пам’яті комп’ютера фрагмента БД. Цей об’єкт є локальним відображенням (фрагмента) БД.

В свою чергу, застосунку, що забезпечує взаємодію з базою даних, об’єкт *DataSet* функціонує виключно за рахунок об’єкту *DataAdapter*, який обслуговує *DataSet*.

При цьому *DataAdapter* є центральним компонентом архітектури від’єднаного рівня ADO.NET.

Об’єкт *DataAdapter* – складова частина провайдера даних. Це приєднаний компонент об’єктної моделі ADO.NET. Він забезпечує зв'язок між джерелом даних і об’єктом DataSet і використовується для двох цілей – заповнення об’єкта DataSet даними і модифікації джерела даних.

Функціональні можливості DataAdapter реалізуються за рахунок метода Fill(), який змінює дані в DataSet. При виконанні методу Fill() об’єкт DataAdapter заповнює DataTable або DataSet даними, отриманими з БД.

Після обробки даних, завантажених в пам'ять, з допомогою методу Update() можна записати модифіковані записи в БД. Метод Update() дозволяє міняти дані в джерелі даних з метою досягнення зворотної відповідності даних в джерелі даних по відношенню до даних в DataSet.

Фактично, *DataAdapter* управляє обміном даних і оновленням вмісту джерела даних.

DataAdapter має набір команд для підключення до БД і модифікації даних. Достойні особливої уваги чотири властивості цього класу, що фактично представляють команди БД. Через ці команди об’єкт *DataAdapter* і впливає на *DataSet* і БД:

1. *SelectCommand* – містить текст (рядок sql) чи об’єкт команди, що здійснює отримання даних з БД. При виклику метода Fill() ця команда виконується і заповнює об’єкт DataTable чи об’єкт DataSet.
2. *InsertCommand* – містить текст (рядок sql) чи об’єкт команди, що здійснює вставку записів в таблицю.
3. *DeleteCommand* – містить текст (рядок sql) чи об’єкт команди, що здійснює видалення запису з таблиці.
4. *UpdateCommand* – містить текст (рядок sql) чи об’єкт команди, що здійснює оновлення значень в БД.

2.3 Порядок виконання лабораторної роботи

В даному порядку виконання лабораторної роботи буде розроблений застосунок, який буде працювати з даними з бази даних (відображати, додавати, видаляти, редагувати дані). Доступ до бази даних буде здійснюватись з застосуванням технології ADO.NET. В якості СУБД була обрана PostgreSQL. Предметна область буде складатися з одного об’єкта - «Студент». В подальших лабораторних роботах до предметної області буде доданий ще один об’єкт.

1. Застосунок в даній лабораторній роботі (як і у всіх наступних лабораторних роботах) буде взаємодіяти з системою управління базами даних (СУБД) postgresql, тому перед початком виконання лабораторної роботи необхідно встановити цю СУБД. Її можна завантажити з офіційного сайту postgresql https://www.postgresql.org/download/. В даній лабораторній роботі була використана версія postgresql 12.13. Замість СУБД postgresql можна використовувати будь-яку іншу СУБД. Для полегшення взаємодії з СУБД необхідно встановити графічний інтерфейс, наприклад DBeaber, який можна завантажити з офіційного сайту даного графічного інтерфейсу https://dbeaver.io/download/. Можна обійтися і без графічного інтерфейсу, але треба бути готовим виконувати запити в консольному режимі.

2. Створіть в директорії з проектами (в якості прикладу у всіх лабораторних роботах використовується директорія D:\csexamples, але директорія з проектами може знаходитись в будь-якому зручному місці) директорію для нового проекту з назвою ado (або з будь-якою іншою назвою). Відкрийте Visual Studio Code і змініть директорію проекту на D:\csexamples\ado (File->Open Folder). Відкрийте термінал в Visual Studio Code, якщо він не з'явився автоматично (Terminal->New Terminal).

3. Зараз необхідно створити нову базу даних в postgres, для того, щоб до неї можна було підключитися з DBeaber. В терміналі введіть команду:

```
psql --username=postgres --password=Welcome1
```

Ім'я користувача і пароль для postgresql залежить від налаштувань СУБД і може бути іншим.

В командному рядку postgres введіть команду:

```
create database university;
```

4. Відкрийте DBeaber і створіть нове підключення PostgreSQL. Параметри підключення введіть, як показано на рисунку 2.2.

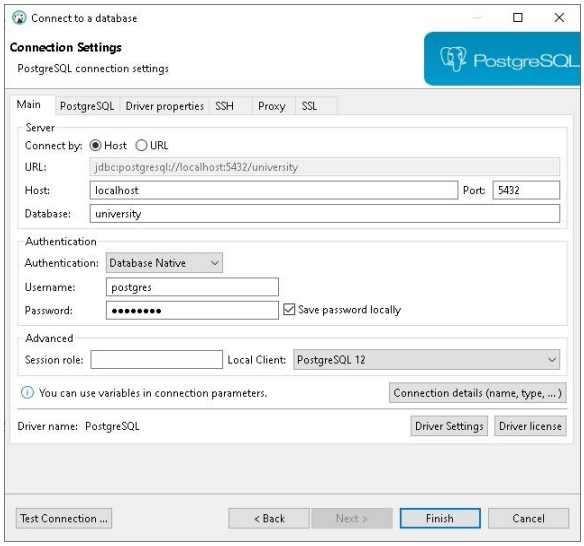

Рисунок 2.2 – Параметри підключення до бази PostgreSQL в DBeaber

5. В DBeaber відкрийте вкладку SQL для бази даних university та виконайте запит за створення таблички бази даних:

```
--Таблиця студентів
CREATE TABLE Students(
      id bigserial NOT NULL,
      firstname varchar(20),
      lastname varchar(20),
      sex varchar(1)    NOT NULL CHECK (sex = 'M' or sex = 'F'),
      age integer,
      PRIMARY KEY (id)
);
```

Результат виконання запиту на створення таблиці в базі даних показаний на рисунку 2.3.

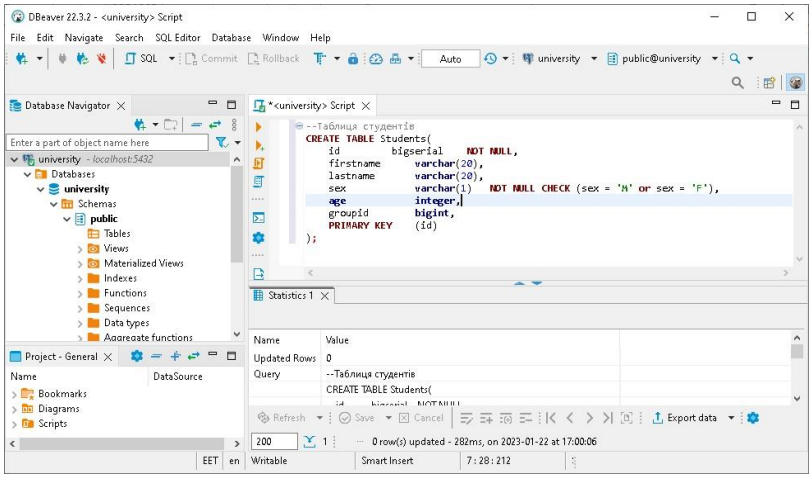

Рисунок 2.3 – Результат виконання запиту на створення таблиці в базі даних в DBeaber

6. В терміналі Visual Studio Code введіть команду exit для того, щоб вийти з postgres. Введіть команду для створення нового WPF (Windows Presentation Foundation) застосунку в директорії D:\csexamples\ado:

```
dotnet new wpf
```

7. В даній лабораторній роботі нам необхідно використовувати додаткову бібліотеку-драйвер для взаємодії з СУБД, яку необхідно додати до проекту шляхом виконання команди:

```
dotnet add package npgsql
```

8. Додайте клас доменного об’єкту Student до проекту:

```
namespace ado
{
    public class Student
    {
        public long Id {get; set;}
        public string FirstName {get;set;}
        public string SecondName {get; set;}
        public string Sex {get; set;}
        public int Age {get; set;}
    }
}
```

9. Для полегшення роботи з базою даних в даній лабораторній роботі буде використаний шаблон проектування «Репозитарій». Шаблону «Репозитарій» зазвичай складається з інтерфейсу, в якому описані всі необхідні методи для роботи з джерелом даних, і з реалізації цього інтерфейсу. Шаблон «Репозитарій» може використовуватися не тільки для взаємодії з базою даних але і для взаємодії з іншими джерелами даних. Додайте до проекту інтерфейс шаблону «Репозитарій»:

```
using System.Collections.Generic;

using System.Collections.Generic;


namespace ado
{
    interface IStudentRepository
    {
        public void Save(Student student);
        public void Update(Student student);
        public IList<Student> GetAll();
        public Student FindById(long id);
        public void Delete(long id);
        public void Destroy();
    }
}
```

10. Додайте до проекту клас-реалізацію шаблону репозитарій:

```
using System.Data;
using System;
using System.Collections.Generic;
using System.Data.Common;
using Npgsql;

namespace ado
{
 class StudentRepository : IStudentRepository
 {
  private StudentRepository() { }

  private NpgsqlConnection connection;

  private static IStudentRepository instance;

  public static IStudentRepository getInstance()
  {
   if (null == instance)
   {
    instance = new StudentRepository();
   }
   return instance;
  }
```

```
private NpgsqlConnection getConnection()
{
 if (null == connection)
 {
  String connectionStr = "Server=127.0.0.1;Port=5432;
   User Id=postgres;Password=Welcome1;Database=university;";
  connection = new NpgsqlConnection(connectionStr);
  connection.Open();
 }
 return connection;
}

public Student FindById(long id)
{
 DbCommand command = NpgsqlFactory.Instance.CreateCommand();
 command.Connection = getConnection();
 command.CommandText = "SELECT * FROM Students WHERE id = @id";
 addParameterToCommand(command, "@id", DbType.Int64, id);
 DbDataReader row = command.ExecuteReader();
 Student student = null;
 while (row.Read())
 {
  long studentId = (long)row["id"];
  String firstName = (String)row["firstname"];
  String lastName = (String)row["lastname"];
  String sex = (String)row["sex"];
  int age = (int)row["age"];
  student = new Student();
  student.FirstName = firstName;
  student.LastName = lastName;
  student.Sex = sex;
  student.Age = age;
 }
 row.Close();
 return student;
}

public IList<Student> GetAll()
{
 IList<Student> studentList = new List<Student>();
 DbCommand command = NpgsqlFactory.Instance.CreateCommand();
 command.Connection = getConnection();
 command.CommandText = "SELECT * FROM Students";
 DbDataReader row = command.ExecuteReader();
 while (row.Read())
 {
  long id = (long)row["id"];
  String firstName = (String)row["firstname"];
  String lastName = (String)row["lastname"];
  String sex = (String)row["sex"];
  int age = (int)row["age"];
  Student student = new Student();
  student.Id = id;
  student.FirstName = firstName;
  student.LastName = lastName;
  student.Sex = sex;
  student.Age = age;
  studentList.Add(student);
 }
 row.Close();
 return studentList;
}
```

```
  public void Save(Student student)
  {
   DbCommand command = NpgsqlFactory.Instance.CreateCommand();
   command.Connection = getConnection();
   command.CommandText = "INSERT INTO Students(firstname, lastname, sex, age)
    VALUES(@firstname, @lastname, @sex, @age)";
   addParameterToCommand(command, "@firstname", DbType.String,
    student.FirstName);
   addParameterToCommand(command, "@lastname", DbType.String,
    student.LastName);
   addParameterToCommand(command, "@sex", DbType.String,
    student.Sex);
   addParameterToCommand(command, "@age", DbType.Int32,
    student.Age);
   command.ExecuteNonQuery();
  }

  public void Update(Student student)
  {
   DbCommand command = NpgsqlFactory.Instance.CreateCommand();
   command.Connection = getConnection();
   command.CommandText = "UPDATE Students
    SET firstname=@firstname, lastname=@lastname,
    sex=@sex, age=@age WHERE id=@id";
   addParameterToCommand(command, "@id", DbType.Int64,
    student.Id);
   addParameterToCommand(command, "@firstname", DbType.String,
    student.FirstName);
   addParameterToCommand(command, "@lastname", DbType.String,
    student.LastName);
   addParameterToCommand(command, "@sex", DbType.String,
    student.Sex);
   addParameterToCommand(command, "@age", DbType.Int32,
    student.Age);
   command.ExecuteNonQuery();
  }

   public void Delete(long id)
   {
    DbCommand command = NpgsqlFactory.Instance.CreateCommand();
    command.Connection = getConnection();
    command.CommandText = "DELETE FROM Students WHERE id=@id";
    addParameterToCommand(command, "@id", DbType.Int64, id);
    command.ExecuteNonQuery();
   }

  private void addParameterToCommand(DbCommand command,
   string parameterName, DbType parameterType,
   object parameterValue)
  {
   NpgsqlParameter parameter = new NpgsqlParameter();
   parameter.ParameterName = parameterName;
   parameter.DbType = parameterType;
   parameter.Value = parameterValue;
   command.Parameters.Add(parameter);
  }

  public void Destroy()
  {
   getConnection().Close();
  }
 }
}
```

11. В даній лабораторній роботі для побудови графічного інтерфейсу буде використовуватись фреймворк WPF. Найпростіший застосунок WPF складається з розмітки на мові XAML і класу-обробника подій на мові C#. За замовчуванням файл розмітки має назву MainWindow.xaml, а клас обробник має назву MainWindow.xaml.cs. Додайте розмітку в файл MainWindow.xaml.

```
<Window x:Class="ado.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 xmlns:local="clr-namespace:ado"
 mc:Ignorable="d"
 Title="MainWindow" Height="450" Width="800">
 <DockPanel LastChildFill="True">
  <Border Width="350" Background="PaleGreen" BorderBrush="Black"
   BorderThickness="1" DockPanel.Dock="Right">
   <StackPanel Grid.Column="2" Grid.Row="1"  Orientation="Vertical">
    <StackPanel.Resources>
     <Style TargetType="{x:Type TextBox}">
      <Setter Property="Margin" Value="10,10,10,10"/>
     </Style>
     <Style TargetType="{x:Type Button}">
      <Setter Property="Margin" Value="10,10,10,10"/>
     </Style>
    </StackPanel.Resources>
    <Label Content="Id" Height="28" HorizontalAlignment="Left"
     Name="Label1" VerticalAlignment="Top"/>
    <TextBox MinWidth="300" HorizontalAlignment="Center"
     VerticalAlignment="Center" Name="InputTextBox1" IsEnabled="False"/>
    <Label Content="First Name" Height="28" HorizontalAlignment="Left"
     Name="Label2" VerticalAlignment="Top" />
    <TextBox MinWidth="300" HorizontalAlignment="Center"
     VerticalAlignment="Center" Name="InputTextBox2"/>
    <Label Content="Last Name" Height="28" HorizontalAlignment="Left"
     Name="Label3" VerticalAlignment="Top" />
    <TextBox MinWidth="300" HorizontalAlignment="Center"
     VerticalAlignment="Center" Name="InputTextBox3"/>
    <Label Content="Sex" Height="28" HorizontalAlignment="Left" Name="Label4"
     VerticalAlignment="Top" />
    <ComboBox MinWidth="300" HorizontalAlignment="Center"
     VerticalAlignment="Center" Name="ComboBox1">
     <ComboBoxItem>M</ComboBoxItem>
     <ComboBoxItem>F</ComboBoxItem>
    </ComboBox>
    <Label Content="Age" Height="28" HorizontalAlignment="Left" Name="Label5"
     VerticalAlignment="Top" />
    <TextBox MinWidth="300" HorizontalAlignment="Center"
     VerticalAlignment="Center" Name="InputTextBox4"/>
    <Button Content="Add student" Name="AddButton" Click="ButtonAdd_Click"/>
    <Button Content="Edit student" Name="EditButton" Click="ButtonEdit_Click"
     IsEnabled="False"/>
    <Button Content="Cancel Edit" Name="CancelButton"
     Click="ButtonCancel_Click" IsEnabled="False"/>
   </StackPanel>
  </Border>
  <Border Background="White" BorderBrush="Black" BorderThickness="1">
   <DataGrid Name="StudentGrid"  AutoGenerateColumns="False"
    IsReadOnly="True" Width="Auto">
    <DataGrid.Resources>
     <Style TargetType="DataGridCell">
      <Style.Triggers>
```

```
      <Trigger Property="IsSelected" Value="True">
       <Setter Property="Background" Value="Green" />
      </Trigger>
     </Style.Triggers>
    </Style>
   </DataGrid.Resources>
   <DataGrid.ContextMenu>
    <ContextMenu>
     <MenuItem Header="Edit" Click="MenuItemEdit_Click"/>
     <MenuItem Header="Delete" Click="MenuItemDelete_Click"/>
    </ContextMenu>
   </DataGrid.ContextMenu>
   <DataGrid.Columns>
    <DataGridTextColumn Header="Id" Binding="{Binding Path=Id}" Width="50"/>
    <DataGridTextColumn Header="First Name"
     Binding="{Binding Path=FirstName}" Width="100"/>
    <DataGridTextColumn Header="Last Name"
     Binding="{Binding Path=LastName}" Width="100"/>
    <DataGridTextColumn Header="Sex"
     Binding="{Binding Path=Sex}" Width="100"/>
    <DataGridTextColumn Header="Age"
     Binding="{Binding Path=Age}" Width="100"/>
   </DataGrid.Columns>
  </DataGrid>
 </Border>
</DockPanel>
</Window>
```

12. Додайте обробники подій головного вікна в файл MainWindow.xaml.cs.

```
using System.ComponentModel;
using System;
using System.Collections.Generic;
using System.Windows;

namespace ado
{
 public partial class MainWindow : Window
 {
  public MainWindow()
  {
   InitializeComponent();
   IList<Student> studentList = StudentRepository.getInstance().GetAll();
   StudentGrid.ItemsSource = studentList;
   Application.Current.MainWindow.Closing +=
    new CancelEventHandler(MainWindow_Closing);
  }

  private void MenuItemEdit_Click(object sender, EventArgs e)
  {
   Student student = (Student)StudentGrid.SelectedItem;
   InputTextBox1.Text = student.Id.ToString();
   InputTextBox2.Text = student.FirstName;
   InputTextBox3.Text = student.LastName;
   ComboBox1.Text = student.Sex;
   InputTextBox4.Text = student.Age.ToString();
   AddButton.IsEnabled = false;
   EditButton.IsEnabled = true;
   CancelButton.IsEnabled = true;
  }

  private void MenuItemDelete_Click(object sender, EventArgs e)
```

```
{
 string messageBoxText = "Do you want to delete student?";
 string caption = "Delete student";
 MessageBoxButton button = MessageBoxButton.OKCancel;
 MessageBoxImage icon = MessageBoxImage.Question;
 MessageBoxResult result = MessageBox.Show(messageBoxText, caption,
  button, icon, MessageBoxResult.Yes);
 if (result.Equals(MessageBoxResult.OK))
 {
  Student student = (Student)StudentGrid.SelectedItem;
  StudentRepository.getInstance().Delete(student.Id);
  IList<Student> studentList = StudentRepository.getInstance().GetAll();
  StudentGrid.ItemsSource = studentList;
 }
}

private void ButtonAdd_Click(object sender, EventArgs e)
{
 Student student = new Student();
 student.FirstName = InputTextBox2.Text;
 student.LastName = InputTextBox3.Text;
 student.Sex = ComboBox1.Text;
 student.Age = Int32.Parse(InputTextBox4.Text);
 StudentRepository.getInstance().Save(student);
 IList<Student> studentList = StudentRepository.getInstance().GetAll();
 StudentGrid.ItemsSource = studentList;
}

private void ButtonEdit_Click(object sender, EventArgs e)
{
 Student student = new Student();
 student.Id = Int64.Parse(InputTextBox1.Text);
 student.FirstName = InputTextBox2.Text;
 student.LastName = InputTextBox3.Text;
 student.Sex = ComboBox1.Text;
 student.Age = Int32.Parse(InputTextBox4.Text);
 StudentRepository.getInstance().Update(student);
 IList<Student> studentList = StudentRepository.getInstance().GetAll();
 StudentGrid.ItemsSource = studentList;
 InputTextBox1.Text = "";
 InputTextBox2.Text = "";
 InputTextBox3.Text = "";
 ComboBox1.Text = "";
 InputTextBox4.Text = "";
 EditButton.IsEnabled = false;
 CancelButton.IsEnabled = false;
 AddButton.IsEnabled = true;
}

private void ButtonCancel_Click(object sender, EventArgs e)
{
 InputTextBox1.Text = "";
 InputTextBox2.Text = "";
 InputTextBox3.Text = "";
 ComboBox1.Text = "";
 InputTextBox4.Text = "";
 EditButton.IsEnabled = false;
 CancelButton.IsEnabled = false;
 AddButton.IsEnabled = true;
}

void MainWindow_Closing(object sender, CancelEventArgs e)
{
 StudentRepository.getInstance().Destroy();
```

```
    }
  }
}
```

13. Запустіть застосунок (Run -> Start Without Debugging), від лагодьте помилки компіляції. Після запуску застосунку він має виглядати як на рисунку 2.4.

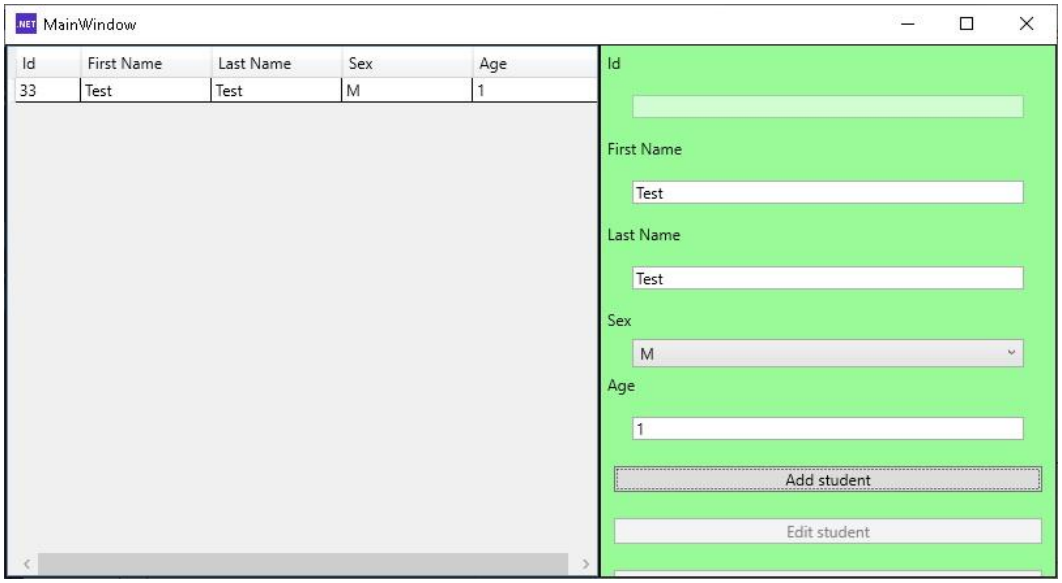

Рисунок 2.4 – Вигляд застосунку після запуску

2.4 Завдання до лабораторної роботи

Створити застосунок, описаний в ході виконання лабораторної роботи.

Замінити об’єкт предметної «Студент» на об’єкт предметної області відповідно до варіанту з таблиці 2.1 та внести необхідні зміни в код застосунку. Номер варіанта визначається за останньою цифрою залікової книжки.

Таблиця 2.1 – Варіанти завдань

Номер варіанта	Предметна область	Об’єкт предметної області
0	Аеропорт	Пасажир
1	Піцерія	Замовлення
2	Крамниця	Товар
3	Супермаркет	Постачальник
4	Відділ поліції	Порушник
5	Лікарня	Пацієнт
6	Кінотеатр	Сеанс
7	Корабель	Пасажир
8	Підприємство	Співробітник
9	Бібліотека	Книга

2.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;

- номер і назва лабораторної роботи;
- мета лабораторної роботи;
- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

2.6 Контрольні питання

1. Які особливості моделі доступу до даних в ADO.NET?
2. Які відмінності в застосуванні моделей доступу до даних на зв’язаному і незв’язаному рівнях?
3. Які основні об’єкти використовуються для доступу до даних на зв’язаному рівні ADO.NET?
4. Які основні об’єкти використовуються для доступу до даних на незв’язаному рівні ADO.NET?
5. Які три об’єкти необхідно створити в зв’язаному рівні ADO.NET, для того, щоб прочитати дані з таблиці?
6. Яким чином можна налаштувати об’єкт читання даних, щоб він автоматично закривав з’єднання за базою даних після завершення читання?
7. Чим запит на отримання даних відрізняється від запиту на модифікацію даних в зв’язаному рівні ADO.NET?
8. Як передати параметр в запит в ADO.NET?
9. Для чого використовується адаптер даних в незв’язаному рівні ADO.NET?
10. Які властивості адаптера даних відповідають за виконання запитів до джерела даних?

2.7 Завдання до захисту лабораторної роботи

1. Додати валідацію для одного з полів об’єкта предметної області.
2. Зберегти табличку бази даних в XML файл.
3. Завантажити дані з XML файлу в табличку бази даних з лабораторної роботи.
4. Зберегти табличку бази даних в JSON файл.
5. Завантажити дані з JSON файлу в табличку бази.
6. Зберегти табличку бази даних в CSV файл.
7. Завантажити дані з CSV файлу в табличку бази.
8. Додати функцію пошуку в таблиці по заданому полю.
9. Додати функцію пошуку в таблиці по всім полям.
10. Додати функцію збереження даних в JSON файл в зашифрованому вигляді.

3 ЛАБОРАТОРНА РОБОТА № 3
РОБОТА З ORM В DOTNET CORE

3.1 Мета роботи

Ознайомитися з технологією Fluent Nhibernate і особливостями її використання в dotnet core. Отримати практичні навички по роботі з технологією Fluent Nhibernate.

3.2 Теоретичні відомості

3.2.1 Технологія Fluent Hibernate

Для суміщення в одній системі переваг об'єктно-орієнтовного програмування і реляційних баз даних необхідно використовувати ORM (object relation mapping) – об'єктно-реляційне відображення. Об'єктно-реляційне відображення забезпечує можливість роботи з записами таблиці реляційної бази даних як з об'єктами.

Суть об'єктно-реляційного відображення в тому, щоб відділити зберігання даних від роботи з ними, або іншими словами – відділити бізнес-логіку і логіку зберігання даних. При розробці системи відображення до неї висуваються вимоги наступного плану – необхідно розробити систему таким чином, щоб при описі бізнес компонентів мінімум часу витрачалося на реалізацію необхідного для цієї системи відображення.

Важливою перевагою відображення є можливість розглянути окремі таблиці БД як колекції об'єктів відповідних класів.

В технології .NET використовується два підходи для реалізації відображення – з використанням атрибутів чи з використанням XML-файлів для зовнішнього співставлення.

Суть відображення на основі атрибутів базується на механізмі Reflection і полягає у наступному:

1. Визначаються атрибути, що описують, які таблиці зберігають дані відповідних класів і атрибути, що спів ставляють поля цих класів полям відповідних таблиць. При цьому помічаються ключові поля.
2. При описі наступної сутності, що відображається, над класом розміщається атрибут, що вказує, в якій таблиці зберігаються сутності даного типу, а над полями вказуються атрибути, що ставлять їм у відповідність поля таблиці в базі даних.
3. Створюється мапер (виконавець відображення), робота якого побудована на цих атрибутах. Інформацію про відображення поля цей мапер отримує виключно використовуючи механізм Reflection. Значення полів також обираються і встановлюються через механізм Reflection.

З допомогою XML-файлів метадані для співставлення можна зберігати за межами коду застосунку. Такий файл використовується для співставлення між моделлю даних БД і об'єктною моделлю. Файл зовнішнього співставлення має наступні переваги:

1. Код співставлення можна зберігати за межами коду застосунку. Цей підхід зменшує перевантаженість коду застосунку.

2. Файл зовнішнього співставлення можна вважати подібним файлу конфігурації. Наприклад, можна змінити поведінку застосунку після створення двійкових файлів, просто змінивши файл зовнішнього співставлення.

Популярними ORM для платформи .NET є LINQ to Entities, NHibernate, Fluent NHibernate. В даній лабораторній роботі використовується ORM Fluent NHibernate.

Використання Fluent NHibernate дозволяє не задумуватись про структуру бази даних, а працювати з базою як з колекцією пов'язаних один з одним об'єктів. Така можливість забезпечується завдяки механізму відображення об'єкта на таблицю бази даних шляхом правил, що в NHibernate задаються map-класами. В map-класі вказуються правила, з допомогою яких поля об'єктів будуть відображатися на колонки бази даних, а також правила для зв'язку об'єктів один з одним. На етапі запуску застосунку map-класи перетворюються в набори запитів, що дозволяють додавати, видаляти і модифікувати дані в таблицях бази даних, а також виконувати вибірку даних. Окрім map-файлів в форматі XML в більш пізніх версіях NHibernate правила відображення можна задавати з допомогою атрибутів, а в Fluent NHibernate з допомогою класів.

3.2.2 Зв'язки між об'єктами в Fluent NHibernate

Fluent NHibernate дозволяє задавати зв'язки один до одного, один до багатьох, багато до багатьох з використанням спеціальних конструкцій HasOne, HasMany, HasManyToMany.

Розглянемо приклад створення цих трьох типів зв'язків на прикладах.

3.2.3 Зв'язок один до одного

Для зв'язку один до одного візьмемо об'єкти «Студент» і «Залікова книжка» - один студент може мати лише одну залікову книжку. Для початку створимо класи студента і залікової книжки:

```
namespace Fluent.Domain
{
    //Домен студента
    public class Student
    {
        public virtual long Id { get; set; }

        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

        //Посилання на залікову книжку
        public virtual RecordBook RecordBook { get; set; }
```

```
    }
}

namespace Fluent.Domain
{
    //Домен залікової книжки
    public class RecordBook
    {
        public virtual long Id { get; set; }

        public virtual string Number { get; set; }

        //Посилання на студента
        public virtual Student Student { get; set; }
    }
}
```

Тепер необхідно створити класи відображення (map-класси):

```
namespace Fluent.Mappings
{
    //Клас відображення залікової книжки
    public class RecordBookMap : ClassMap<RecordBook>
    {
        public RecordBookMap()
        {
            //Назва таблиці
            Table("RecordBooks");
            //Відображення ідентифікатора
            Id(x => x.Id).GeneratedBy.Native();
            //Відображення звичайного поля
            Map(x => x.Number);
            //Посилання на студента
            References(x => x.Student).Column("StudentId").Cascade.All();
        }
    }
}

namespace Fluent.Mappings
{
    //Клас відображення студента
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
            // Зв'язок один до одного
            HasOne(x => x.RecordBook).ForeignKey("StudentId").Cascade.All();
        }
    }
}
```

Як видно із прикладу, кожний клас відображення містить конструкцію для зазначення назви таблиці в базі даних (Table), конструкцію для відображення ключового поля на таблицю бази даних (Id), конструкції для

відображення полів на таблицю бази даних (Map), конструкції для зв’язування об’єктів.

В даному прикладі для того, щоб зв’язати об’єкти «Студент» і «Залікова книжка» зв’язком один до одного, в класі «Студент» необхідне посилання на об’єкт «Залікова книжка»:

```
public virtual RecordBook RecordBook { get; set; }
```

В класі «Залікова книжка» для того, щоб з об’єкта залікової книжки можна було отримати доступ до студента, необхідне посилання на об’єкт «Студент»:

```
public virtual Student Student { get; set; }
```

В класах відображення також повинні бути відповідні поля. Зі сторони студента:

```
HasOne(x => x.RecordBook).ForeignKey("StudentId").Cascade.All();
```

Зі сторони залікової книжки повинне бути поле:

```
References(x => x.Student).Column("StudentId").Cascade.All();
```

У відповідності до класів відображення в базі даних автоматично створюються таблиці, що представлені на рис 3.1 і 3.2.

id [PK] bigserial	number character var	studentid bigint

Рисунок 3.1 – Таблиця залікової книжки

id [PK] bigserial	firstname character var	lastname character var	sex character(1)	year integer

Рисунок 3.2 – Таблиця студента

Розглянемо тепер зв'язок один до багатьох.

3.2.4 Зв'язок один до багатьох

Для зв’язку один до багатьох візьмемо об’єкти «Група» і «Студент». В одній групі може бути багато студенті. Один студент може бути членом тільки однієї групи.

Створимо класи доменів для групи і студента.

```
namespace Fluent.Domain
{
    //Домен групи
    public class Group
    {
        private IList<Student> studentList = new List<Student>();

        public virtual long Id { get; set; }

        public virtual string GroupName { get; set; }
```

```
        public virtual string CuratorName { get; set; }

        public virtual string HeadmanName { get; set; }

        public virtual IList<Student> StudentList
        {
            get { return studentList; }
            set { studentList = value; }
        }

    }
}

namespace Fluent.Domain
{
    //Домен студента
    public class Student
    {
        public virtual long Id { get; set; }

        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

        public virtual Group Group { get; set; }

    }
}
```

Зверніть увагу на те, що зі сторони групи міститься посилання на колекцію для зберігання студентів.

Створимо класи відображення для груп і студентів:

```
using System;
using System.Collections.Generic;
using FluentNHibernate.Mapping;
using Fluent.Domain;

namespace Fluent.Mappings
{
    //Клас відобрадення групи
    public class GroupMap:ClassMap<Group>
    {
        public GroupMap()
        {
            Table("Groups");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.GroupName);
            Map(x => x.CuratorName);
            Map(x => x.HeadmanName);
            //Зв'язок один до багатьох
            HasMany(x => x.StudentList)
                .KeyColumns.Add("GroupId")
                .Inverse()
                .Cascade.All();
        }

    }
}
```

```
namespace Fluent.Mappings
{
    //Клас відображення групи
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
            //Посилання на групу
            References(x => x.Group).Column("GroupId").Cascade.All();
        }
    }
}
```

На відміну від зв’язку один до одного, зі сторони один в класі відображення знаходиться конструкція:

```
HasMany(x => x.StudentList)
    .KeyColumns.Add("GroupId")
    .Inverse()
    .Cascade.All();
```

А зі сторони багатьох, як і при зв'язку один до одного знаходится конструкція:

```
References(x => x.Group).Column("GroupId").Cascade.All();
```

У відповідності з класами відображення в базі даних автоматично створюються таблиці, що зображені на рисунках 3.3 і 3.4.

id [PK] bigserial	groupname character var	curatorname character var	headmannam character var

Рисунок 3.3 – Таблиця групи

id [PK] bigserial	firstname character var	lastname character var	sex character(1)	year integer	groupid bigint

Рисунок 3.4 – Таблиця студента

Розглянемо тепер зв'язок багато до багатьох.

3.2.5 Зв'язок багато до багатьох

Для зв’язку багато до багатьох оберемо об’єкти «Викладач» і «Предмет». Один викладач може вести багато предметів і один предмет може вести декілька викладачів.

Створимо класи домену для викладачів і для предметів:

```
namespace Fluent.Domain
{
    //Клас домена предмета
```

```
    public class Subject
    {
        private IList<Teacher> teacherList = new List<Teacher>();

        public virtual long Id { get; set; }

        public virtual string SubjectName { get; set; }

        public virtual int HoursNumber { get; set; }

        public virtual IList<Teacher> TeacherList
        {
            get { return teacherList; }
            set { teacherList = value; }
        }
    }
}

namespace Fluent.Domain
{
    //Клас домена викладача
    public class Teacher
    {
        private IList<Subject> subjectList = new List<Subject>();

        public virtual long Id { get; set; }

        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual IList<Subject> SubjectList
        {
            get { return subjectList; }
            set { subjectList = value; }
        }
    }
}
```

Зверніть увагу, що клас предмета містить колекцію для зберігання викладачів, а клас викладача містить колекцію для зберігання предметів.

Створимо класи відображення для предмета і викладача:

```
namespace Fluent.Mappings
{
    //Клас відображення предмета
    public class SubjectMap:ClassMap<Subject>
    {
        public SubjectMap()
        {
            Table("Subjects");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.SubjectName);
            Map(x => x.HoursNumber);
            // Зв'язок багато до багатьох
            HasManyToMany(x => x.TeacherList)
                .Table("TeacherSubject")
                .ParentKeyColumn("SubjectId")
                .ChildKeyColumn("TeacherId");
        }
    }
}
```

```
namespace Fluent.Mappings
{
    //Клас відображення викладача
    public class TeacherMap:ClassMap<Teacher>
    {
        public TeacherMap()
        {
            Table("Teachers");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Зв'язок багато до багатьох
            HasManyToMany(x => x.SubjectList)
                .Table("TeacherSubject")
                .ParentKeyColumn("TeacherId")
                .ChildKeyColumn("SubjectId");
        }
    }
}
```

Зверніть увагу, що в класі відображення з обох сторін вказується конструкція HasManyToMany.

Оскільки зв'язок багато до багатьох в реляційних базах даних здійснюється через окрему таблицю, то в базі даних буде автоматично створюватися три таблиці, що представлені на рисунках 3.5, 3.6, 3.7.

id [PK] bigserial	subjectname character var	hoursnumber integer

Рисунок 3.5 – Таблиця предмету

id [PK] bigserial	firstname character var	lastname character var

Рисунок 3.6 – Таблиця викладача

subjectid bigint	teacherid bigint

Рисунок 3.7 – Проміжна таблиця викладача і предмету

Більш детально зі зв’язками між об’єктами, а також з параметрами зв’язків можна ознайомитись в книгах NHibernate in Action і NHibernate Cookbook.

3.2.6 Способи отримання об’єктів в Fluent NHibernate

В Fluent NHibernate існує 4 способи отримання об’єктів:

- Native SQL;
- запити по критерію;
- запити по прикладу;
- мова запитів HQL (Hibernate Query Language).

3.3 Порядок виконання лабораторної роботи

Застосунок в даній лабораторній роботі буде взаємодіяти з реляційною базою даних з використанням технології Fluent Nhibernate. Взаємодія буде відбуватися з двома табличками бази даних «Група» і «Студент», що зв'язані між собою зв'язком один до багатьох. На відміну від попередньої лабораторної роботи в якій був використаний шаблон проектування репозитарій, в даній лабораторній роботі буде використаний шаблон проектування DAO (Data Access Object), що краще підходить при використанні технологій ORM. Таблички в базі даних і зв'язки між ними будуть створені автоматично під час запуску застосунку з використанням класів відображень доменних об'єктів.

1. Створіть в директорії з проектами (в якості прикладу у всіх лабораторних роботах використовується директорія D:\csexamples, але директорія з проектами може знаходитись в будь-якому зручному місці) директорію для нового проекту з назвою orm (або з будь-якою іншою назвою). Відкрийте Visual Studio Code і змініть директорію проекту на D:\csexamples\orm (File->Open Folder). Відкрийте термінал в Visual Studio Code, якщо він не з'явився автоматично (Terminal->New Terminal).

2. Зараз необхідно створити нову базу даних в postgres з назвою universityorm. В терміналі введіть команду:

```
psql --username=postgres --password=Welcome1
```

Ім'я користувача і пароль для postgresql залежить від налаштувань СУБД і може бути іншим.

В командному рядку postgres введіть команду:

```
create database universityorm;
```

3. В терміналі Visual Studio Code введіть команду exit для того, щоб вийти з postgres. Введіть команду для створення нового WPF (Windows Presentation Foundation) застосунку в директорії D:\csexamples\orm:

```
dotnet new wpf
```

4. В даній лабораторній роботі будуть використовуватися додаткові бібліотеки для роботи з технологією Fluent Nhibernate, які необхідно додати до проекту шляхом виконання наступних команд:

```
dotnet add package fluentnhibernate
dotnet add package npgsql
```

5. В даній лабораторній роботі буде створено досить багато класів, тому в директорії проекту (D:\csexamples\orm) необхідно створити декілька внутрішніх директорій для кращого структурування проекту. Для класів доменів створіть директорію domain, для класів DAO створіть директорію dao.

6. Додайте класи доменних об'єктів EntityBase, Group і Student в директорію domain:

```
//Базовий клас
```

```
namespace orm
{
    public abstract class  EntityBase
    {
        public virtual long Id { get; set; }
    }
}

using System.Collections.Generic;

//Доменний клас групи
namespace orm
{
    public class Group : EntityBase
    {
        private IList<Student> studentList = new List<Student>();

        public virtual string GroupName { get; set; }

        public virtual string CuratorName { get; set; }

        public virtual string HeadmanName { get; set; }

        public virtual IList<Student> StudentList
        {
            get { return studentList; }
            set { studentList = value; }
        }
    }
}

//Доменний клас студента
namespace orm
{
    public class Student : EntityBase
    {
        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

        public virtual Group Group { get; set; }
    }
}
```

7. Додайте класи відображень для доменних об’єктів групи і студента в директорію domain:

```
//Клас відображення групи
using FluentNHibernate.Mapping;

namespace orm
{
    public class GroupMap : ClassMap<Group>
    {
        public GroupMap()
        {
            Table("Groups");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.GroupName);
```

```
            Map(x => x.CuratorName);
            Map(x => x.HeadmanName);
            HasMany(x => x.StudentList)
                .Cascade.All()
                .KeyColumn("GroupId");
        }
    }
}

//Клас відображення студента
using FluentNHibernate.Mapping;

namespace orm
{
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
            References(x => x.Group, "GroupId");
        }
    }
}
```

8. Далі необхідно створити набір класів для реалізації шаблона проектування DAO. Додайте інтерфейс GenericDAO з базовим набором методів для роботи з доменними об’єктами в директорію dao:

```
using System.Collections.Generic;

namespace orm
{
    public interface IGenericDAO<T>
    {
        void SaveOrUpdate(T item);

        T GetById(long id);

        List<T> GetAll();

        void Delete(T item);
    }
}
```

9. Додайте реалізацію інтерфейсу GenericDAO в директорію dao:

```
using System.Collections.Generic;
using NHibernate;

namespace orm
{
    public class GenericDAO<T> : IGenericDAO<T>
    {
        protected ISession session;

        public GenericDAO() { }
```

```
        public GenericDAO(ISession session)
        {
            this.session = session;
        }

        public void SaveOrUpdate(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.SaveOrUpdate(item);
            transaction.Commit();
        }

        public T GetById(long id)
        {
            return session.Get<T>(id);
        }

        public List<T> GetAll()
        {
            return new List<T>(session.CreateCriteria(typeof(T)).List<T>());
        }

        public void Delete(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.Delete(item);
            transaction.Commit();
        }
    }
}
```

10. Додайте інтерфейси DAO для доменних об’єктів групи IGroupDAO і студента IStudentDAO в директорію dao. Вони потрібні у випадку, якщо для роботи з доменними об’єктами необхідно реалізувати додаткові методи, яких нема в IGenericDAO:

```
//Інтерфейс DAO для доменного об’єкта групи
namespace orm
{
    public interface IGroupDAO:IGenericDAO<Group>
    {

    }
}

//Інтерфейс DAO для доменного об’єкта студента
using System.Collections.Generic;

namespace orm
{
    public interface IStudentDAO:IGenericDAO<Student>
    {
        IList<Student> getStudentsByGroup(long groupId);

    }
}
```

11. Додайте до директорії dao реалізації інтерфейсів IGroupDAO і IStudentDAO:

```
//Реалізація інтерфейсу IGroupDAO
```

```
using NHibernate;

namespace orm
{
    public class GroupDAO : GenericDAO<Group>, IGroupDAO
    {
        public GroupDAO(ISession session) : base(session) { }

    }
}

//Реалізація інтерфейсу IStudentDAO
using System.Collections.Generic;
using NHibernate;

namespace orm
{
    public class StudentDAO : GenericDAO<Student>, IStudentDAO
    {

        public StudentDAO(ISession session) : base(session) { }

        public IList<Student> getStudentsByGroup(long groupId)
        {
            var list = session.CreateSQLQuery(
               "SELECT Students.* FROM Students JOIN Groups" +
               " ON Students.GroupId = Groups.Id" +
               " WHERE Groups.Id=" + groupId)
               .AddEntity("Student", typeof(Student))
               .List<Student>();
            return list;
        }
    }
}
```

12. Додатково до шаблону DAO в даній лабораторній роботі використовується шаблон фабрика. Додайте до директорії dao абстрактну фабрику DAOFactory:

```
namespace orm
{
    abstract public class DAOFactory
    {
        public abstract IStudentDAO getStudentDAO();

        public abstract IGroupDAO getGroupDAO();

        public abstract void destroy();

    }
}
```

13. Додайте до директорії dao реалізацію фабрики DAOFactory для фреймворку Fluent Nhibernate:

```
using System;
using System.Reflection;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;
```

```
namespace orm
{
    public class NHibernateDAOFactory : DAOFactory
    {
        private static ISessionFactory factory;

        private static DAOFactory instance;

        private ISession session;

        private IGroupDAO groupDAO;
        private IStudentDAO studentDAO;

        public static DAOFactory getInstance()
        {
            if (null == instance)
            {
                ISession session = openSession("127.0.0.1",
                    Convert.ToInt32("5432"), "universityorm",
                    "postgres", "Welcome1");
                instance = new NHibernateDAOFactory(session);
            }
            return instance;
        }

        public NHibernateDAOFactory(ISession session)
        {
            this.session = session;
        }

        public override IGroupDAO getGroupDAO()
        {
            if (null == groupDAO)
            {
                groupDAO = new GroupDAO(session);
            }
            return groupDAO;
        }
        public override IStudentDAO getStudentDAO()
        {
            if (null == studentDAO)
            {
                studentDAO = new StudentDAO(session);
            }
            return studentDAO;
        }

        public override void destroy()
        {
            session.Close();
        }

        private static ISession openSession(String host, int port,
            String database, String user, String password)
        {
            ISession session = null;
            Assembly mappingsAssemly = Assembly.GetExecutingAssembly();
            if (null == factory)
            {
                factory = Fluently.Configure()
                   .Database(PostgreSQLConfiguration
                   .PostgreSQL82.ConnectionString(c => c
                   .Host(host)
```

```
                    .Port(port)
                    .Database(database)
                    .Username(user)
                    .Password(password)))
                    .Mappings(m => m.FluentMappings
                    .AddFromAssembly(mappingsAssemly))
                    .ExposeConfiguration(BuildSchema)
                    .BuildSessionFactory();
            }
            session = factory.OpenSession();
            return session;
        }

        //Метод, що дозволяє автоматично згенерувати схему
        //бази даних
        private static void BuildSchema(Configuration config)
        {
            new SchemaExport(config).Create(true, true);
        }
    }
}
```

14. Далі необхідно створити інтерфейс користувача, який, як і в попередній лабораторній роботі, побудований на технології WPF, але, на відміну від попередньої лабораторної роботи, буде складатися не з одного вікна а з трьох. Додатково до файлу розмітки XAML MainWindow.xaml і файлу з класом MainWindow.xaml.cs необхідно буде створити файли GroupWindow.xaml, GroupWindow.xaml.cs і StudentWindow.xaml, StudentWindow.xaml.cs. Додайте до кореня проекту фай з розміткою для вікна додання і редагування групи GroupWindow.xaml:

```
<Window x:Class="orm.GroupWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 xmlns:local="clr-namespace:orm"
 mc:Ignorable="d"
 Title="Group" Height="400" Width="400">
 <Grid>
  <StackPanel Grid.Column="2" Grid.Row="1"  Orientation="Vertical">
   <StackPanel.Resources>
    <Style TargetType="{x:Type Label}">
     <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
     <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
    <Style TargetType="{x:Type Button}">
     <Setter Property="Margin" Value="5,10,10,5"/>
    </Style>
   </StackPanel.Resources>
   <Label Content="Group Name" Height="28" HorizontalAlignment="Left"
    Name="Label1" VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox1"/>
   <Label Content="Curator Name" Height="28" HorizontalAlignment="Left"
    Name="Label2" VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox2"/>
```

```
   <Label Content="HeadmanName" Height="28" HorizontalAlignment="Left"
    Name="Label3" VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox3"/>
   <Button Content="Add group" Name="AddButton" Click="ButtonAdd_Click"
    IsEnabled="False"/>
   <Button Content="Edit group" Name="EditButton" Click="ButtonEdit_Click"
    IsEnabled="False"/>
   <Button Content="Cancel" Name="CancelButton" Click="ButtonCancel_Click"
    IsEnabled="False"/>
  </StackPanel>
 </Grid>
</Window>
```

15. Додайте до кореня проекту фай з класом для обробки подій вікна додання і редагування групи GroupWindow.xaml.cs:

```
using System;
using System.ComponentModel;
using System.Windows;
namespace orm
{
    public partial class GroupWindow : Window
    {

        private MainWindow parentWindow;

        public Group Group { get; set; }

        public GroupWindow(MainWindow parentWindow)
        {
            this.parentWindow = parentWindow;
            this.Closing += new CancelEventHandler(Window_Closing);
            InitializeComponent();
        }

        public MainWindow getParentWindow()
        {
            return parentWindow;
        }

        //Обробник натиснення на кнопку додання групи
        private void ButtonAdd_Click(object sender, EventArgs e)
        {
            Group group = new Group();
            group.GroupName = InputTextBox1.Text;
            group.CuratorName = InputTextBox2.Text;
            group.HeadmanName = InputTextBox3.Text;
            getParentWindow().getDaoFactory()
                .getGroupDAO().SaveOrUpdate(group);
            getParentWindow().GroupGrid.ItemsSource = getParentWindow()
                .getDaoFactory().getGroupDAO().GetAll();
            closeWindow();
        }

        //Обробник натиснення на кнопку редагування групи
        private void ButtonEdit_Click(object sender, EventArgs e)
        {
            Group group = getParentWindow()
                .getDaoFactory().getGroupDAO().GetById(Group.Id);
            group.GroupName = InputTextBox1.Text;
            group.CuratorName = InputTextBox2.Text;
            group.HeadmanName = InputTextBox3.Text;
```

```
            getParentWindow().getDaoFactory().getGroupDAO()
                .SaveOrUpdate(group);
            getParentWindow().GroupGrid.ItemsSource = getParentWindow()
                .getDaoFactory().getGroupDAO().GetAll();
            closeWindow();
        }

        //Обробник натиснення на кнопку Cancel
        private void ButtonCancel_Click(object sender, EventArgs e)
        {
            closeWindow();
        }

        //Обробник закриття вікна групи
        private void Window_Closing(object sender,
            System.ComponentModel.CancelEventArgs e)
        {
            e.Cancel = true;
            closeWindow();
        }

        private void closeWindow()
        {
            this.AddButton.IsEnabled = false;
            this.EditButton.IsEnabled = false;
            this.CancelButton.IsEnabled = false;
            this.InputTextBox1.Text = "";
            this.InputTextBox2.Text = "";
            this.InputTextBox3.Text = "";
            this.Hide();
        }
    }
}
```

16. Додайте до кореня проекту фай з розміткою для вікна додання і редагування студента StudentWindow.xaml:

```
<Window x:Class="orm.StudentWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:orm"
  mc:Ignorable="d"
  Title="Student" Height="450" Width="400">
 <Grid>
  <StackPanel Grid.Column="2" Grid.Row="1"  Orientation="Vertical">
   <StackPanel.Resources>
    <Style TargetType="{x:Type Label}">
     <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
     <Setter Property="Margin" Value="5,5,5,5"/>
    </Style>
    <Style TargetType="{x:Type Button}">
     <Setter Property="Margin" Value="5,10,10,5"/>
    </Style>
   </StackPanel.Resources>
   <Label Content="First Name" Height="28" HorizontalAlignment="Left"
    Name="Label1" VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox1"/>
   <Label Content="Last Name" Height="28" HorizontalAlignment="Left"
```

```
    Name="Label2" VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox2"/>
   <Label Content="Sex" Height="28" HorizontalAlignment="Left" Name="Label3"
    VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox3"/>
   <Label Content="Age" Height="28" HorizontalAlignment="Left" Name="Label4"
    VerticalAlignment="Top" />
   <TextBox MinWidth="300" HorizontalAlignment="Left"
    VerticalAlignment="Center" Name="InputTextBox4"/>
   <Button Content="Add student" Name="AddButton" Click="ButtonAdd_Click"
    IsEnabled="False"/>
   <Button Content="Edit student" Name="EditButton" Click="ButtonEdit_Click"
    IsEnabled="False"/>
   <Button Content="Cancel" Name="CancelButton" Click="ButtonCancel_Click"
    IsEnabled="False"/>
  </StackPanel>
 </Grid>
</Window>
```

17. Додайте до кореня проекту фай з класом для обробки подій вікна додання і редагування студента StudentWindow.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
namespace orm
{
    public partial class StudentWindow : Window
    {
        private MainWindow parentWindow;

        public MainWindow getParentWindow()
        {
            return parentWindow;
        }

        public Group Group { get; set; }

        public Student Student { get; set; }

        public StudentWindow(MainWindow parentWindow)
        {
            this.parentWindow = parentWindow;
            this.Closing += new CancelEventHandler(Window_Closing);
            InitializeComponent();
        }

        //Обробник натиснення на кнопку додання студента
        private void ButtonAdd_Click(object sender, EventArgs e)
        {
            Student student = new Student();
            student.FirstName = InputTextBox1.Text;
            student.LastName = InputTextBox2.Text;
            student.Sex = InputTextBox3.Text.ToCharArray()[0];
            student.Year = Int32.Parse(InputTextBox4.Text);
            Group group = getParentWindow()
                .getDaoFactory().getGroupDAO().GetById(Group.Id);
            student.Group = group;
            group.StudentList.Add(student);
            getParentWindow().getDaoFactory()
```

```
                .getStudentDAO().SaveOrUpdate(student);
            IList<Student> studentList = getParentWindow()
                .getDaoFactory().getStudentDAO()
                .getStudentsByGroup(group.Id);
            getParentWindow().StudentGrid.ItemsSource = studentList;
            closeWindow();
        }

        //Обробник натиснення на кнопку редагування студента
        private void ButtonEdit_Click(object sender, EventArgs e)
        {
            Student student = getParentWindow()
                .getDaoFactory().getStudentDAO().GetById(Student.Id);
            student.FirstName = InputTextBox1.Text;
            student.LastName = InputTextBox2.Text;
            student.Sex = InputTextBox3.Text.ToCharArray()[0];
            student.Year = Int32.Parse(InputTextBox4.Text);
            getParentWindow().getDaoFactory().getStudentDAO()
                .SaveOrUpdate(student);
            getParentWindow().StudentGrid.ItemsSource = getParentWindow()
                .getDaoFactory().getStudentDAO()
                .getStudentsByGroup(Group.Id);
            closeWindow();
        }

        //Обробник натиснення на кнопку Cancel
        private void ButtonCancel_Click(object sender, EventArgs e)
        {
            closeWindow();
        }

        //Обробник закриття вікна студента
        private void Window_Closing(object sender,
            System.ComponentModel.CancelEventArgs e)
        {
            e.Cancel = true;
            closeWindow();
        }

        private void closeWindow()
        {
            this.AddButton.IsEnabled = false;
            this.EditButton.IsEnabled = false;
            this.CancelButton.IsEnabled = false;
            this.InputTextBox1.Text = "";
            this.InputTextBox2.Text = "";
            this.InputTextBox3.Text = "";
            this.InputTextBox4.Text = "";
            this.Hide();
        }
    }
}
```

18. Змініть файл розмітки головного вікна MainWindow.xaml як показано нижче:

```
<Window x:Class="orm.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:orm"
  mc:Ignorable="d"
```

```
 Title="University" Height="450" Width="800">
<Grid>
 <Grid.ColumnDefinitions>
  <ColumnDefinition />
  <ColumnDefinition />
 </Grid.ColumnDefinitions>
 <Grid Grid.Column="0">
  <Grid.RowDefinitions>
   <RowDefinition Height="auto"></RowDefinition>
   <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Label Grid.Row="0" Content="Group list" Height="28"
   HorizontalAlignment="Left" Name="Label1" VerticalAlignment="Top"/>
  <DataGrid Grid.Row="1" Name="GroupGrid" AutoGenerateColumns="False"
   IsReadOnly="True" Width="Auto">
   <DataGrid.Resources>
    <Style TargetType="DataGridCell">
     <Style.Triggers>
      <Trigger Property="IsSelected" Value="True">
       <Setter Property="Background" Value="Green" />
      </Trigger>
     </Style.Triggers>
     <EventSetter Event="MouseLeftButtonUp" Handler="DataGrid_Click"/>
    </Style>
   </DataGrid.Resources>
   <DataGrid.ContextMenu>
    <ContextMenu>
     <MenuItem Header="Add group" Click="MenuItemAddGroup_Click"/>
     <MenuItem Header="Edit group" Click="MenuItemEditGroup_Click"/>
     <MenuItem Header="Delete group" Click="MenuItemDeleteGroup_Click"/>
    </ContextMenu>
   </DataGrid.ContextMenu>
   <DataGrid.Columns>
    <DataGridTextColumn Header="Id" Binding="{Binding Path=Id}" Width="50"/>
    <DataGridTextColumn Header="Group Name" Binding="{Binding
     Path=GroupName}" Width="100"/>
    <DataGridTextColumn Header="Curator Name" Binding="{Binding
     Path=CuratorName}" Width="100"/>
    <DataGridTextColumn Header="Headman Name" Binding="{Binding
     Path=HeadmanName}" Width="100"/>
   </DataGrid.Columns>
  </DataGrid>
 </Grid>
 <GridSplitter Grid.Column="0" Width="2" ShowsPreview="True"/>
 <Grid Grid.Column="1">
  <Grid.RowDefinitions>
   <RowDefinition Height="auto"></RowDefinition>
   <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Label Grid.Row="0" Content="Student list" Height="28"
   HorizontalAlignment="Left" Name="Label2" VerticalAlignment="Top"/>
  <DataGrid Grid.Row="1" Name="StudentGrid"  AutoGenerateColumns="False"
   IsReadOnly="True" Width="Auto">
   <DataGrid.Resources>
    <Style TargetType="DataGridCell">
     <Style.Triggers>
      <Trigger Property="IsSelected" Value="True">
       <Setter Property="Background" Value="Green" />
      </Trigger>
     </Style.Triggers>
    </Style>
   </DataGrid.Resources>
   <DataGrid.ContextMenu>
    <ContextMenu>
```

```
      <MenuItem Header="Add student" Click="MenuItemAddStudent_Click"/>
      <MenuItem Header="Edit student" Click="MenuItemEditStudent_Click"/>
      <MenuItem Header="Delete student" Click="MenuItemDeleteStudent_Click"/>
     </ContextMenu>
    </DataGrid.ContextMenu>
    <DataGrid.Columns>
     <DataGridTextColumn Header="Id" Binding="{Binding Path=Id}" Width="50"/>
     <DataGridTextColumn Header="First Name" Binding="{Binding
      Path=FirstName}" Width="100"/>
     <DataGridTextColumn Header="Last Name" Binding="{Binding Path=LastName}"
      Width="100"/>
     <DataGridTextColumn Header="Sex" Binding="{Binding Path=Sex}"
      Width="100"/>
     <DataGridTextColumn Header="Year" Binding="{Binding Path=Year}"
      Width="100"/>
    </DataGrid.Columns>
   </DataGrid>
  </Grid>
 </Grid>
</Window>
```

19. Змініть файл з класом обробником подій головного вікна MainWindow.xaml.cs як показано нижче:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;

namespace orm
{
    public partial class MainWindow : Window
    {
        private DAOFactory daoFactory;

        public DAOFactory getDaoFactory()
        {
            if(null == daoFactory)
            {
                daoFactory = NHibernateDAOFactory.getInstance();
            }
            return daoFactory;
        }
        private GroupWindow groupWindow;

        private StudentWindow studentWindow;

        public GroupWindow getGroupWindow()
        {
            if (null == groupWindow)
            {
                groupWindow = new GroupWindow(this);
            }
            return groupWindow;
        }

        public StudentWindow getStudentWindow()
        {
            if (null == studentWindow)
            {
                studentWindow = new StudentWindow(this);
            }
            return studentWindow;
```

```
}

public MainWindow()
{
    this.Closing += new CancelEventHandler(MainWindow_Closing);
    InitializeComponent();
    GroupGrid.ItemsSource = getDaoFactory().getGroupDAO().GetAll();
}
//Обробник натиснення на контекстне меню додання групи
private void MenuItemAddGroup_Click(object sender, EventArgs e)
{
    GroupWindow groupWindow = getGroupWindow();
    groupWindow.AddButton.IsEnabled = true;
    groupWindow.CancelButton.IsEnabled = true;
    groupWindow.ShowDialog();
}
//Обробник натиснення на контекстне меню редагування групи
private void MenuItemEditGroup_Click(object sender, EventArgs e)
{
    Group group = (Group)GroupGrid.SelectedItem;
    if (null == group)
    {
        MessageBox.Show("Please select group", "Nothing to edit",
            MessageBoxButton.OK, MessageBoxImage.Information,
            MessageBoxResult.No);
    }
    else
    {
        GroupWindow groupWindow = getGroupWindow();
        groupWindow.EditButton.IsEnabled = true;
        groupWindow.CancelButton.IsEnabled = true;
        groupWindow.Group = group;
        groupWindow.InputTextBox1.Text = group.GroupName;
        groupWindow.InputTextBox2.Text = group.CuratorName;
        groupWindow.InputTextBox3.Text = group.HeadmanName;
        groupWindow.ShowDialog();
    }
}
//Обробник натиснення на контекстне меню видалення групи
private void MenuItemDeleteGroup_Click(object sender, EventArgs e)
{
    Group group = (Group)GroupGrid.SelectedItem;
    if (null == group)
    {
        MessageBox.Show("Please select group", "Nothing to delete",
            MessageBoxButton.OK, MessageBoxImage.Information,
            MessageBoxResult.No);
    }
    else
    {
        getDaoFactory().getGroupDAO().Delete(group);
        GroupGrid.ItemsSource =
            getDaoFactory().getGroupDAO().GetAll();
    }
}
//Обробник натиснення на контекстне меню додання студента
private void MenuItemAddStudent_Click(object sender, EventArgs e)
{
    Group group = (Group)GroupGrid.SelectedItem;
    if (null == group)
    {
        MessageBox.Show("Please select group",
            "Student must be added to group",
            MessageBoxButton.OK, MessageBoxImage.Information,
```

```
            MessageBoxResult.No);
    }
    else
    {
        StudentWindow studentWindow = getStudentWindow();
        studentWindow.AddButton.IsEnabled = true;
        studentWindow.CancelButton.IsEnabled = true;
        studentWindow.Group = group;
        studentWindow.ShowDialog();
    }
}

//Обробник натиснення на контекстне меню редагування студента
private void MenuItemEditStudent_Click(object sender, EventArgs e)
{
    Group group = (Group)GroupGrid.SelectedItem;
    Student student = (Student)StudentGrid.SelectedItem;
    if (null == group)
    {
        MessageBox.Show("Please select student", "Nothing to edit",
            MessageBoxButton.OK, MessageBoxImage.Information,
            MessageBoxResult.No);
    }
    else
    {
        StudentWindow studentWindow = getStudentWindow();
        studentWindow.EditButton.IsEnabled = true;
        studentWindow.CancelButton.IsEnabled = true;
        studentWindow.Group = group;
        studentWindow.Student = student;
        studentWindow.InputTextBox1.Text = student.FirstName;
        studentWindow.InputTextBox2.Text = student.LastName;
        studentWindow.InputTextBox3.Text = student.Sex.ToString();
        studentWindow.InputTextBox4.Text = student.Year.ToString();
        studentWindow.ShowDialog();
    }
}

//Обробник натиснення на контекстне меню видалення студента
private void MenuItemDeleteStudent_Click(object sender, EventArgs e)
{
    Student student = (Student)StudentGrid.SelectedItem;
    if (null == student)
    {
        MessageBox.Show("Please select student", "Nothing to delete",
            MessageBoxButton.OK, MessageBoxImage.Information,
            MessageBoxResult.No);
    }
    else
    {
        student = getDaoFactory().getStudentDAO()
            .GetById(student.Id);
        student.Group.StudentList.Remove(student);
        getDaoFactory().getStudentDAO().Delete(student);
        Group group = (Group)GroupGrid.SelectedItem;
        IList<Student> studentList = getDaoFactory()
            .getStudentDAO().getStudentsByGroup(group.Id);
        StudentGrid.ItemsSource = studentList;
    }
}

//Обробник натиснення на рядок в таблиці групи
private void DataGrid_Click(object sender, RoutedEventArgs e)
{
```

```
            Group group = (Group)GroupGrid.SelectedItem;
            if (null != group)
            {
                group = getDaoFactory().getGroupDAO().GetById(group.Id);
                StudentGrid.ItemsSource = group.StudentList;
            }
        }

        //Обробник закриття головного вікна
        void MainWindow_Closing(object sender, CancelEventArgs e)
        {
            getDaoFactory().destroy();
        }
    }
}
```

20. Додайте в файл розмітки застосунку App.xaml в елемент Application атрибут ShutdownMode="OnMainWindowClose", як показано нижче. Це необхідно для того, щоб зупиняти застосунок при закритті головного вікна.

```
<Application x:Class="orm.App"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:local="clr-namespace:orm"
 StartupUri="MainWindow.xaml"
 ShutdownMode="OnMainWindowClose">
 <Application.Resources>

 </Application.Resources>
</Application>
```

21. Відправте застосунок на виконання. При наявності помилок їх необхідно відлагодити. Після запуску застосунку він повинен виглядати, як показано на рисунку 3.8. Додайте декілька груп і студентів.

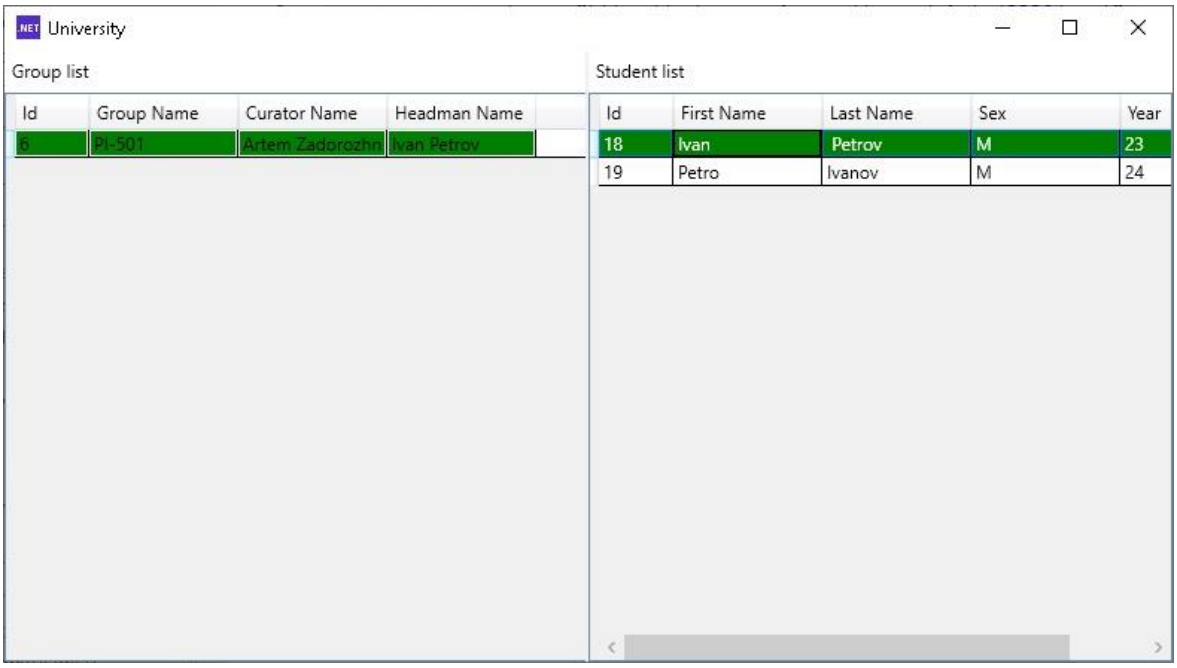

Рисунок 3.8 – Зовнішній вигляд застосунку після запуску

22. Зупиніть застосунок, відкрийте клас NHibernateDAOFactory та змініть метод BuildSchema, як показано нижче. Це необхідно для того, що не

запускати генерацію структури бази даних при кожному запуску програми. Якщо цю зміну не внести, то всі дані, введені під час виконання програми, будуть видалятися кожного разу під час наступного запуску програми.

```
private static void BuildSchema(Configuration config)
{
    new SchemaExport(config).Create(true, false);
}
```

23. Відправте застосунок на виконання ще раз і переконайтеся, що дані не зникли.

3.4 Завдання до лабораторної роботи

Реалізувати застосунок, описаний в ході виконання лабораторної роботи, що може працювати з двома таблицями СУБД. На відміну від застосунку, описаного в лабораторній роботі, предметну область необхідно вибрати з таблиці 3.1. Номер варіанту необхідно вибрати відповідно до останньої цифри номеру залікової книжки.

Таблиця 3.1 – Варіанти завдань

Номер варіанта	Предметна область	Об’єкти предметної області
0	Аеропорт	Рейс, Пасажир
1	Піцерія	Офіціант, Замовлення
2	Крамниця	Замовлення, Товар
3	Супермаркет	Постачальник, Товар
4	Відділ поліції	Поліцейський, Порушник
5	Лікарня	Лікар, Пацієнт
6	Кінотеатр	Сеанс, Відвідувач
7	Корабель	Каюта, Пасажир
8	Підприємство	Відділ, Співробітник
9	Бібліотека	Автор, Книга

3.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;
- номер і назва лабораторної роботи;
- мета лабораторної роботи;
- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

3.6 Контрольні питання

1. Що таке ORM?
2. Чим відрізняється NHibernate від Fluent NHibernate?
3. Як в Fluent NHibernate відобразити поле класу на базу даних.
4. Як задати зв’язки між об’єктами в Fluent NHibernate?

5. Зв'язок HasOne. Призначення і синтаксис.
6. Зв'язок HasMany. Призначення і синтаксис.
7. Зв'язок HasManyToMany. Призначення і синтаксис.
8. Які способи отримання даних є в NHibernate?
9. Чим запит Native SQL відрізняється від запиту по критерію?
10. Чим запит по прикладу відрізняється від запиту HQL?

3.7 Завдання до захисту лабораторної роботи

1. Написати метод на мові C#, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів Native SQL.
2. Написати метод на мові C#, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів по критерію.
3. Написати метод на мові C#, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів за прикладом.
4. Написати метод на мові C#, що повертає TOP 10 об’єктів зі сторони багато (відповідно до варіанту завдання), з використанням запитів HQL.
5. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати мову запитів HQL.
6. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати запити по критерію.
7. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати запити за прикладом.
8. Реалізувати пошук доменних об’єктів (відповідно до варіанту лабораторної роботи) по будь якому полю чи частині поля. Для пошуку використати запити Native SQL.
9. Використати стратегію спадкування Single Table Strategy для класів-доменів, що спадкуються від класу BaseEntity.
10. Використати стратегію спадкування Table Per Class Strategy для класів-доменів, що спадкуються від класу BaseEntity.

4 ЛАБОРАТОРНА РОБОТА № 4 СТВОРЕННЯ ВЕБ-ЗАСТОСУНКІВ НА БАЗІ ПЛАТФОРМИ DOTNET CORE

4.1 Мета роботи

Ознайомитися з технологією ASP.NET Core MVC. Отримати практичні навички по роботі з технологією ASP.NET Core MVC.

4.2 Теоретичні відомості

4.2.1 Відображення в ASP.NET Core MVC

В шаблоні MVC (Model-View-Controller) відображення відповідає за вивід даних застосунку і взаємодію з користувачем. Відображення – це html-шаблон з вбудованою Razor розміткою. Razor розмітка – це код, що взаємодіє з розміткою HTML для створення веб-сторінки, яка відправляється клієнту. В ASP.NET Core MVC відображення є .cshtml файлами, що використовують мову програмування C# в Razor розмітці. Зазвичай, файли відображень об’єднуються в директорії з іменами, що відповідають окремим контролерам застосунку. Директорії зберігаються в директорії Views, що зазвичай знаходиться в корені застосунку.

Відображення допомагають реалізувати розподіл задач в застосунку MVC, ізолюючи розмітку інтерфейсу користувача від інших частин застосунку. Дотримання принца SoC (Separation of Concerns) робить застосунок модульним, що надає декілька переваг:

1. Застосунок простіше обслуговувати, так як воно краще організоване. Відображення, як правило, групуються по функціональним можливостям застосунку. Це спрощує пошук зв’язаних відображень пр. роботі на певною функцією.

2. Частини застосунку слабко пов’язані одна з одною. Є можливість розроблювати і оновлювати відображення застосунку окремо від компонентів бізнес-логіки і доступу до даних. При оновленні відображень застосунку не обов’язково оновлювати інші його частини.

3. Частини інтерфейсу користувача простіше тестувати, бо відображення є окремими модулями.

4. Завдяки більш упорядкованій структурі є меншою вірогідність того, що розділи інтерфейсу користувача будуть випадково повторюватися.

4.2.2 Елементи відображення в ASP.NET Core MVC

Відображення, що відносяться до контролера, створюються в директорії Views/[ControllerName]. Відображення, що використовуються декількома контролерами, розміщують в директорії Views/Shared. Щоб створити відображення, необхідно додати новий файл і привласнити йому те ж ім’я, що і пов’язана з ним дія в котроллері з розширенням файлу .cshtml.

Файл відображення містить в собі розмітку Razor. Razor – це синтаксис розмітки для включення коду на основі .NET в веб-сторінки. Синтаксиси Razor складається з розмітки Razor, C# і HTML. Синтаксис Razor аналогічний механізмам створення шаблонів для різних платформ одно сторінкових застосунків JavaScript, таких як, Angular, React, VueJs и Svelte.

Мова за замовчуванням Razor це HTML. Відображення HTML з Razor розмітки нічим не відрізняється від відображення HTML з HTML-файлу. Розмітка HTML в .cshtml Razor файлах відображається сервером без змін.

Razor підтримує C# і використовує символ @ для переходу з HTML на C#. Razor вираховує вирази C# і відображає їх в вигляді даних HTML.

До елементів Razor можна віднести Razor вирази, Razor блоки коду, керуючі структури і директиви. Razor вирази можуть бути як явними так і неявними. Неявні Razor вирази починаються з символу @ за яким слідує код на мові C#. Приклад неявних виразів приведений нижче:

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

Неявні вирази не повинні містити пробілів. Виключення м ключове слово C# await. Якщо оператор C# має чітке закінчення, пробіли вставляти можна. Неявні вирази не можуть містити універсальні шаблони C#, так як символи в кутових дужках (<>) інтерпретуються як тег HTML.

Явні Razor вирази містять символ @ за яким слідують круглі дужки:

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Явні вирази можна використовувати для виводу даних з універсальних методів в .cshtml файлах. В наступному прикладі кутові дужки (<>) не будуть інтерпретуватися як тег HTML:

```
<p>@(GenericMethod<int>())</p>
```

Razor блоки коду починаються символом @ і обгорнуті в фігурні дужки. На відміну від виразів, код C# всередині блоків коду не обробляється. Блоки коду і вирази в відображеннях використовують загальну область видимості і, таким чином, змінні, задані в блоці коду, можна використовувати в виразах:

```
@{
    var quote = "Hello world";
}

<p>@quote</p>
```

В блоках коду можна оголошувати локальні функції з допомогою розмітки для використання в якості метолів створення шаблонів:

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
```

```
    RenderName("Ivan Franko");
    RenderName("Taras Shevchenko");
}
```

HTML код, що буде відображений в даному випадку приведений нижче:

```
<p>Name: <strong>Ivan Franko</strong></p>
<p>Name: <strong>Taras Shevchenko</strong></p>
```

Блоки коду можуть бути з явними і неявними переходами. Мовою за замовчуванням в блоці коду є C#, але Razor сторінка може повернутися в HTML. В такому випадку відбудеться неявний перехід:

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Щоб визначити частину коду, яка повинна відображати HTML, необхідно обгорнути значення для виводу тегом Razor <text>. В такому випадку відбудеться явний перехід:

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Для відображення частини рядка, що залишилась, в вигляді HTML, всередині блока коду необхідно використовувати синтаксиси @: для здійснення явного переносу рядка. Без додання на початок рядка @: виникне помилка середовища виконання.

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

До управляючих структур відносяться умови @if, else if, else, і @switch, цикли @for, @foreach, @while, і @do while, обробники виключень @try, catch, finally. Приклади управляючих структур наведені нижче:

```
//Приклад умови if
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
```

```
}

//Приклад умови switch
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}

//Приклад циклу for
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}

//Приклад циклу foreach
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}

//Приклад циклу while
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}

//Приклад циклу do while
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);

//Приклад обробника виключень
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
```

```
    <p>The finally statement.</p>
}
```

Директиви Razor представлені неявними виразами з зарезервованими ключовими словами після символу @. Зазвичай директиви змінюють спосіб аналізу відображення або відкривають доступ до додаткових функцій. Далі розглянемо декілька директив Razor.

Директива @attribute додає даний атрибут до класу створеної сторінки або відображення.

```
@attribute [Authorize]
```

Директива @code дозволяє додавати елементи C# (поля, властивості і методи) в компоненті. Є псевдонімом директиви @functions.

```
@code {
    // C# members (fields, properties, and methods)
}
```

Директива @functions дозволяє додавати елементи C# (поля, властивості і методи) в компоненті.

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

Директива @model визначає тип моделі, що передається в відображення або сторінку.

```
@model LoginViewModel
```

Директива @section використовується в поєднанні з макетами MVC, щоб відображення або сторінка могли відображати вміст в різних частинах сторінки. Дана директива більш детально буде розглянута в розділі про макети.

Директива @using додає директиву C# using в створене відображення.

```
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

Директива @addTagHelper робить допоміжні функції тегів доступними в відображенні.

Директива @removeTagHelper видаляє із відображення допоміжні функції тегів, що були додані раніше.

Директива @tagHelperPrefix задає префікс тегу, який активує підтримку допоміжної функції тегу і її використання в явному вигляді.

4.2.3 Визначення відображення в контролерах

Відображення зазвичай повертаються з дії у вигляді типу ViewResult або ActionResult. Метод дії може створювати і повертати об’єкт ViewResult напряму, однак зазвичай так не робиться. Так як більшість контролерів спадкуються від класу Controller, то для повернення відображення просто використовується допоміжний метод View:

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

Допоміжний метод View має декілька перевантажень. В метод View можна явно передати ім’я відображення, модель, ім’я відображення і модель:

```
//Явна передача імені відображення
return View("Orders");

//Явна передача моделі
return View(Orders);

// Явна передача імені відображення і моделі
return View("Orders", Orders);
```

Коли дія контролера повертає відображення, відбувається процес, який називається виявлення відображення. Він слугує для визначення файлу відображення, що використовується, з використанням імені відображення. Метод View за замовчуванням повертає відображення з тим же ім’ям, що і метод дії, з якого він був викликаний. По-перше, середовище виконання шукає відображення в директорії Views/[ControllerName]. Якщо в директорії нема відповідного відображення, середовище виконує пошук в директорії Shared. Не має значення, чи повертається відображення неявно з допомогою метода View, чи ім’я відображення явно передається в метод View з допомогою return View("<ViewName>"); в обох випадках пошук файлу відображення відбувається в наступному порядку:

1. Views/\[ControllerName]/\[ViewName].cshtml
2. Views/Shared/\[ViewName].cshtml

Замість імені файлу можна передавати шлях до файлу відображення. Якщо використовується абсолютний шлях, що починається з кореневого каталогу застосунку, необхідно вказати розширення файлу:

```
return View("Views/Home/About.cshtml");
```

4.2.4 Передача даних в відображення

Існує декілька підходів до передачі даних в відображення:

– строго типізовані дані – viewmodel;
– слабко типізовані дані (ViewData, ViewBag);

Найбільш надійний підхід – визначення типу моделі в відображенні. Така модель називається viewmodel. Використання viewmodel для передачі даних в відображення дозоляє застосовувати строгу перевірку типів. Вказати модель можна з допомогою директиви @model, а використовувати моделі можна з допомогою виразу @Model:

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

Щоб надати модель для відображення, контролер передає її в якості параметра:

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";

    var viewModel = new Address()
    {
        Name = "Google",
        Street = "1600 Amphitheatre Pkwy",
        City = "Mountain View",
        State = "CA",
        PostalCode = "94043"
    };

    return View(viewModel);
}
```

По відношенню до типів моделей, які можна надавати для відображення, обмежень не існує. Рекомендовано використовувати POCO (Planed Old CLR Object) класи. Класи моделі відображення можуть зберігатися в директорії Models або ViewModels, що знаходяться в корені застосунку, але не забороняється використовувати в якості класів-моделей класи бізнес-моделі.

Окрім строго типізованих даних, відображення мають доступ до слабко типізованої колекції даних. При такому способі передачі даних в відображення передбачається, що тип даних, що використовується, не оголошується явними чином. Колекцію слабко типізованих даних можна використовувати для передачі невеликих об’ємів даних. Працювати з не типізованою колекцією даних можна через властивості ViewData або ViewBag. Властивість ViewData є словником слабко типізованих об’єктів. Властивість ViewBag є обгорткою для властивості ViewData, яка надає динамічні властивості для базової колекції ViewData.

ViewData – це об’єкт ViewDataDictionary доступ до елементів якого здійснюється з допомогою string ключів. Рядкові дані можна і використовувати напряму без приведення типів, але при отриманні значень інших типів їх необхідно приводити до відповідних типів. Нижче показаний

приклад, в який з допомогою ViewData в дії контролера задаються значення для вітання і адреси:

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Ivan",
        Street = "1600 Amphitheatre Pkwy",
        City = "Mountain View",
        State = "CA",
        PostalCode = "94043"
    };

    return View();
}
```

Робота з даними в відображенні буде виглядати наступним чином:

```
@{
    //Об’єкт необхідно примусово привести до типу
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

ViewBag – це об’єкт DynamicViewData, що надає динамічний доступ до об’єктів, що зберігаються в ViewData. Приведений нижче приклад показує, яким чином можна працювати з об’єктом ViewBag:

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Ivan",
        Street = "1600 Amphitheatre Pkwy",
        City = "Mountain View",
        State = "CA",
        PostalCode = "94043"
    };

    return View();
}
```

Робота з даними в відображенні буде виглядати наступним чином:

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State @ViewBag.Address.PostalCode
</address>
```

Відмінності між ViewData і ViewBag представлені нижче:

1. ViewData:

– є похідним від ViewDataDictionary, томі він має властивості словника, котрі можуть бути корисні;
– ключі в словнику є рядками, тому пробіли є допустимими (наприклад ViewData["Some Key With Whitespace"]);
– об’єкт будь-якого типу крім string повинен бути примусово приведений до типу в відображенні;

2. ViewBag:

– є похідним від DynamicViewData, тому він дозволяє створювати динамічні властивості з використанням нотації крапок (@ViewBag.SomeKey = <value or object>);
– простіше перевіряти значення NULL (@ViewBag.Person?.Name).

4.2.5 Макети в ASP.NET Core MVC

На сторінках відображень зазвичай є загальні візуальні і програмні елементи. Більшість веб-застосунків мають загальний макет, що забезпечує узгоджений інтерфейс користувача при переході між сторінками. Макет зазвичай включає в себе загальні елементи інтерфейсу користувача, такі як верхній і нижній колонтитули, а також елементи навігації і меню. Структури HTML такі як скрипти і таблиці стилів також використовуються декількома сторінками в застосунку одночасно. Всі ці загальні елементи можуть бути визначені в файлі макету, на який потім може посилатися будь-ке відображення, що використовується в застосунку. Макет визначає шаблон верхнього рівня для відображень в застосунку. Для застосунку макет не є обов’язковим. В застосунку може бути визначено декілька макетів для різних відображень. Нижче приведений приклад макету, використаного в лабораторній роботі. Місце, в яке буде вставлена розмітка відображення, яке використовує макет, позначене викликом методу @RenderBody().

```
<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="utf-8" />
 <title>@ViewData["Title"]</title>
 <link rel="stylesheet" href="~/css/university.css" />
</head>

<body>
 <div class="main">

  @RenderBody()
 </div>
</body>

</html>
```

Для того щоб відображення почало використовувати макет необхідно задати значення змінної Layout:

```
@{
    Layout = "Layout";
}
```

Макет може при необхідності посилатися на один або декілька розділів, викликаючи метод RenderSection. Розділи - це засіб впорядкування розміщення певних елементів на сторінці. При кожному виклику RenderSection можна вказати чи є розділ обов'язковим чи ні.

Нижче приведений код, який показує, яким чином вивести секцію в макеті:

```
@RenderSection("Scripts", required: false)
```

Код, який показує яким чином в відображення, що використовує шаблон, позначити частину коду, як секцію, показаний нижче:

```
@section Scripts {
     <script type="text/javascript" src="~/scripts/main.js"></script>
}
```

4.3 Порядок виконання лабораторної роботи

Застосунок в даній лабораторній роботі буде взаємодіяти з реляційною базою даних з використанням технології Fluent Nhibernate. Взаємодія буде відбуватися з однією табличкою бази даних «Студент». На відміну від попередньої лабораторної роботи, в якій для прошарку відображення була використана технологія WPF, в даній лабораторній роботі була використана технологія ASP.NET Core MVC.

1. Створіть в директорії з проектами (в якості прикладу у всіх лабораторних роботах використовується директорія D:\csexamples, але директорія з проектами може знаходитись в будь-якому зручному місці) директорію для нового проекту з назвою webapp (або з будь-якою іншою назвою). Відкрийте Visual Studio Code і змініть директорію проекту на D:\csexamples\webapp (File->Open Folder). Відкрийте термінал в Visual Studio Code, якщо він не з'явився автоматично (Terminal->New Terminal).

2. Зараз необхідно створити нову базу даних в postgres з назвою universityweb. В терміналі введіть команду:

```
psql --username=postgres --password=Welcome1
```

Ім'я користувача і пароль для postgresql залежить від налаштувань СУБД і може бути іншим.

В командному рядку postgres введіть команду:

```
create database universityweb;
```

3. В терміналі Visual Studio Code введіть команду exit для того, щоб вийти з postgres. Введіть команду для створення нового порожнього web-застосунку в директорії D:\csexamples\webapp:

```
dotnet new web
```

4. В даній лабораторній роботі будуть використовуватися додаткові бібліотеки для роботи з технологією Fluent Nhibernate, які необхідно додати до проекту шляхом виконання наступних команд:

```
dotnet add package fluentnhibernate
dotnet add package npgsql
```

5. Створіть в корені проекту директорію Domain. Створіть в директорії Domain класи EntityBase.cs, Student.cs та StudentMap.cs:

```
//Базовий доменний клас
namespace webapp.Domain
{
    public abstract class  EntityBase
    {
        public virtual long Id { get; set; }
    }
}

//Доменний клас студента
namespace webapp.Domain
{
    public class Student : EntityBase
    {
        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

    }
}

//Клас відображення студента
using FluentNHibernate.Mapping;

namespace webapp.Domain
{
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
        }
    }
}
```

6. Створіть в корені проекту директорію Dao. Додайте до директорії Dao інтерфейси і класи шаблону DAO IGenericDAO.cs, IGenericDAO.cs, IStudentDAO.cs, StudentDAO.cs, DAOFactory.cs, NHibernateDAOFactory.cs:

```
//Інтерфейс IGenericDAO
namespace webapp.Dao
{
```

```
    public interface IGenericDAO<T>
    {
        void SaveOrUpdate(T item);

        T GetById(long id);

        List<T> GetAll();

        void Delete(T item);
    }
}

//Клас GenericDAO
using NHibernate;
using ISession = NHibernate.ISession;

namespace webapp.Dao
{
    public class GenericDAO<T> : IGenericDAO<T>
    {
        protected ISession session;

        public GenericDAO() { }

        public GenericDAO(ISession session)
        {
            this.session = session;
        }

        public void SaveOrUpdate(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.SaveOrUpdate(item);
            transaction.Commit();
        }

        public T GetById(long id)
        {
            return session.Get<T>(id);
        }

        public List<T> GetAll()
        {
            return new List<T>(session.CreateCriteria(typeof(T)).List<T>());
        }

        public void Delete(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.Delete(item);
            transaction.Commit();
        }
    }
}

//Інтерфейс IStudentDAO
using webapp.Domain;

namespace webapp.Dao
{
    public interface IStudentDAO:IGenericDAO<Student>
    {

    }
```

```
}

//Клас StudentDAO
using ISession = NHibernate.ISession;
using webapp.Domain;

namespace webapp.Dao
{
    public class StudentDAO : GenericDAO<Student>, IStudentDAO
    {

        public StudentDAO(ISession session) : base(session) { }

    }
}

//Клас DAOFactory
namespace webapp.Dao
{
    abstract public class DAOFactory
    {
        public abstract IStudentDAO getStudentDAO();

        public abstract void destroy();

    }
}

//Клас NHibernateDAOFactory
using System.Reflection;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;
using ISession = NHibernate.ISession;

namespace webapp.Dao
{
    public class NHibernateDAOFactory : DAOFactory
    {
        private static ISessionFactory factory;

        private static DAOFactory instance;

        private ISession session;

        private IStudentDAO studentDAO;

        public static DAOFactory getInstance()
        {
            if (null == instance)
            {
                ISession session = openSession("127.0.0.1",
                    Convert.ToInt32("5432"), "universityweb",
                    "postgres", "Welcome1");
                instance = new NHibernateDAOFactory(session);
            }
            return instance;
        }

        public NHibernateDAOFactory(ISession session)
        {
            this.session = session;
```

```
        }

        public override IStudentDAO getStudentDAO()
        {
            if (null == studentDAO)
            {
                studentDAO = new StudentDAO(session);
            }
            return studentDAO;
        }

        public override void destroy()
        {
            session.Close();
        }

        private static ISession openSession(String host, int port,
            String database, String user, String password)
        {
            ISession session = null;
            Assembly mappingsAssemly = Assembly.GetExecutingAssembly();
            if (null == factory)
            {
                factory = Fluently.Configure()
                   .Database(PostgreSQLConfiguration
                   .PostgreSQL82.ConnectionString(c => c
                   .Host(host)
                   .Port(port)
                   .Database(database)
                   .Username(user)
                   .Password(password)))
                   .Mappings(m => m.FluentMappings
                   .AddFromAssembly(mappingsAssemly))
                   .ExposeConfiguration(BuildSchema)
                   .BuildSessionFactory();
            }
            session = factory.OpenSession();
            return session;
        }

        private static void BuildSchema(Configuration config)
        {
            new SchemaExport(config).Create(true, true);
        }
    }
}
```

7. Оскільки в даній лабораторній роботі в якості прошарку відображення використовується технологія ASP.NET Core MVC, застосунок має містити класи контролерів (Controllers), класи моделей (Models) і файли відображень (Views). Кожен з елементів моделі MVC має за замовчуванням міститися в директорії з заданим іменем. Контролери мають міститися в директорії Controllers, а файли відображень CSHTML мають за замовчуванням знаходитися в директорії Views. Класи моделей можуть міститися (але це не є обов'язковою умовою) в директорії Models. Назви директорій Controllers і Views, а також їх місцезнаходження можна змінити шляхом додаткових налаштувань.

Створіть в корені проекту директорію Controllers. Додайте до директорії Controllers клас контролера StudentController.cs:

```
using Microsoft.AspNetCore.Mvc;
using webapp.Domain;
using webapp.Dao;

namespace webapp.Controllers;

public class StudentController : Controller
{
    //Обробник виводу головної сторінки
    public IActionResult GetAll()
    {
        List<Student> studentList = NHibernateDAOFactory
            .getInstance().getStudentDAO().GetAll();
        return View(studentList);
    }

    //Обробник додання студента
    [HttpPost]
    public IActionResult Add(
        [Bind("FirstName, LastName, Sex, Year")] Student student)
    {
        NHibernateDAOFactory.getInstance()
            .getStudentDAO().SaveOrUpdate(student);
        return RedirectToAction("GetAll");
    }

    //Обробник відкриття форми редагування студента
    [Route("EditForm/{id}")]
    public IActionResult EditForm(long id)
    {
        Student student = NHibernateDAOFactory
            .getInstance().getStudentDAO().GetById(id);
        return View(student);
    }

    //Обробник редагування студента
    [HttpPost]
    public IActionResult Edit(
        [Bind("Id, FirstName, LastName, Sex, Year")] Student student)
    {
        Student studentToUpdate = NHibernateDAOFactory
            .getInstance().getStudentDAO().GetById(student.Id);
        studentToUpdate.FirstName = student.FirstName;
        studentToUpdate.LastName = student.LastName;
        studentToUpdate.Sex = student.Sex;
        studentToUpdate.Year = student.Year;
        NHibernateDAOFactory.getInstance()
            .getStudentDAO().SaveOrUpdate(studentToUpdate);
        return RedirectToAction("GetAll");
    }

    //Обробник видалення студента
    [Route("Delete/{id}")]
    public IActionResult Delete(long id)
    {
        Student student = NHibernateDAOFactory
            .getInstance().getStudentDAO().GetById(id);
        NHibernateDAOFactory.getInstance()
            .getStudentDAO().Delete(student);
        return RedirectToAction("GetAll");
    }

}
```

8. Створіть в корені проекту директорію Views. В директорії Views створіть директорію Shared. Додайте в директорію Shared файл-шаблон Layout.cshtml:

```
<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="utf-8" />
 <title>@ViewData["Title"]</title>
 <link rel="stylesheet" href="~/css/university.css" />
</head>

<body>
 <div class="header">
  Chernihiv politechnic national university
 </div>
 <div class="main">

  @RenderBody()
 </div>
 <div class="footer">
  &copy; 2023 CPNU
 </div>
</body>

</html>
```

9. Створіть в директорії Views директорію Student (ім’я директорії має співпадати з ім’ям класу-контролера). Додай до директорії Student файл-відображення GetAll.cshtml:

```
@using webapp.Domain
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model IEnumerable<Student>
@{
 Layout = "Layout";
 ViewData["Title"] = "Student list";
}
<h1>Student list</h1>
<table>
 <thead>
  <tr>
   <td>
    @Html.DisplayNameFor(model => model.FirstName)
   </td>
   <td>
    @Html.DisplayNameFor(model => model.LastName)
   </td>
   <td>
    @Html.DisplayNameFor(model => model.Sex)
   </td>
   <td>
    @Html.DisplayNameFor(model => model.Year)
   </td>
   <td>
    Actions
   </td>
  </tr>
 </thead>
 <tbody>
```

```
  @foreach (var item in Model)
  {
   <tr>
    <td>
     @Html.DisplayFor(modelItem => item.FirstName)
    </td>
    <td>
     @Html.DisplayFor(modelItem => item.LastName)
    </td>
    <td>
     @Html.DisplayFor(modelItem => item.Sex)
    </td>
    <td>
     @Html.DisplayFor(modelItem => item.Year)
    </td>
    <td>
     <a asp-controller="Student" asp-action="EditForm"
      asp-route-id="@item.Id">Edit</a> |
     <a asp-controller="Student" asp-action="Delete"
      asp-route-id="@item.Id">Delete</a>
    </td>
   </tr>
  }
  <tr>
   <form action="/Student/Add" method="post">
    <td>
     <input name="FirstName" type="text" />
    </td>
    <td>
     <input name="LastName" type="text" />
    </td>
    <td>
     <select name="Sex">
      <option value="M">Male</option>
      <option value="F">Female</option>
     </select>
    </td>
    <td>
     <input name="Year" type="number" min="1" max="200" />
    </td>
    <td>
     <input name="Add" type="submit" value="Add student" />
    </td>
   </form>
  </tr>
 </tbody>
</table>
```

10. Додай до директорії Student файл-відображення EditForm.cshtml:

```
@using webapp.Domain
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model Student
@{
 Layout = "Layout";
 ViewData["Title"] = "Edit Student";
}
<h1>Edit Student</h1>
<form action="/Student/Edit" method="post">
 <input asp-for="Id" type="hidden" />
 <div>
  <label asp-for="FirstName"></label>
 </div>
```

```
 <div>
  <input asp-for="FirstName" type="text" />
 </div>
 <div>
  <label asp-for="LastName"></label>
 </div>
 <div>
  <input asp-for="LastName" type="text" />
 </div>
 <div>
  <label asp-for="Sex"></label>
 </div>
 <div>
  <select asp-for="Sex">
   <option value="M">Male</option>
   <option value="F">Female</option>
  </select>
 </div>
 <div>
  <label asp-for="Year"></label>
 </div>
 <div>
  <input asp-for="Year" type="number" min="1" max="200" />
 </div>
 <div>
  <input name="Edit" type="submit" value="Edit student" />
 </div>
 <div>
  <a asp-controller="Student" asp-action="GetAll">Back to student list</a>
 </div>
</form>
```

11. Для покращення зовнішнього вигляду застосунку необхідно застосувати каскадні таблиці стилів CSS. За замовчуванням статичні файли проекту повинні знаходитися в директорії wwwroot. В корені проекту створіть директорію wwwroot. В директорії wwwroot створіть директорію css і додайте до неї файл university.css:

```
table {
      border-collapse: collapse;
      font-family: Tahoma, Geneva, sans-serif;
}
table td {
      padding: 15px;
}
table thead td {
      background-color: #54585d;
      color: #ffffff;
      font-weight: bold;
      font-size: 13px;
      border: 1px solid #54585d;
}
table tbody td {
      color: #636363;
      border: 1px solid #dddfe1;
}
table tbody tr {
      background-color: #f9fafb;
}
table tbody tr:nth-child(odd) {
      background-color: #ffffff;
}
```

```
.header {
      padding: 60px;
      text-align: center;
      background: #1248ad;
      color: white;
      font-size: 30px;
}

.footer {
      padding: 20px;
      text-align: center;
      background: #1248ad;
      color: white;
      font-size: 20px;
}

.main {
      padding: 50px;
      font-size: 20px;
}
```

12. Змініть вміст класу Program.cs як показано нижче:

```
using webapp.Dao;
internal class Program
{
    private static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllersWithViews();

        var app = builder.Build();

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();
        var applicationLifetime = app.Services
            .GetRequiredService<IHostApplicationLifetime>();
        applicationLifetime.ApplicationStopping.Register(OnShutdown);
        app.MapControllerRoute(
            name: "default",
            pattern: "{controller=Student}/{action=GetAll}");

        app.Run();
    }

    //Обробник зупинки застосунку
    public static void OnShutdown()
    {
        NHibernateDAOFactory.getInstance().destroy();
    }
}
```

13. Відправте застосунок на виконання (Run -> Start Debugging або Run -> Run Without Debugging), перевірте працездатність, виправте помилки. Зовнішній вигляд застосунку представлений на рисунку 4.1.

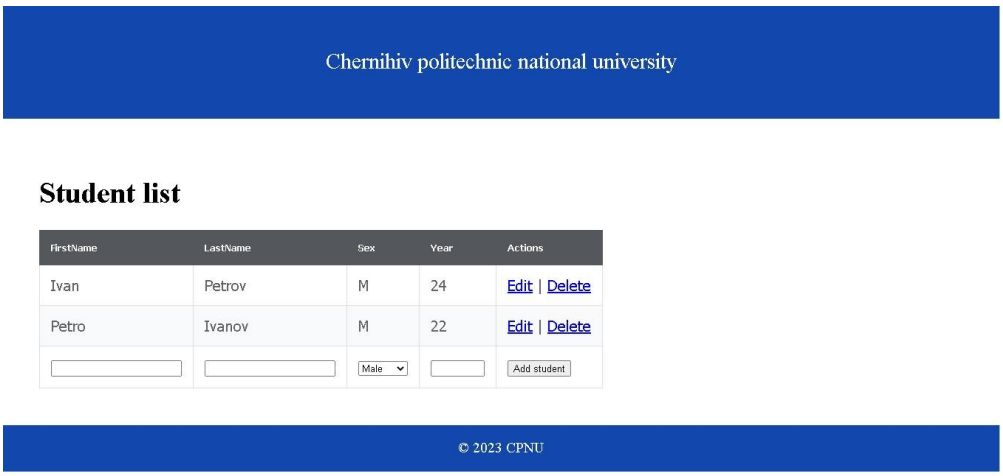

Рисунок 4.1 – Зовнішній вигляд застосунку

14. Зупиніть застосунок. Відкрийте клас NHibernateDAOFactory.cs і змініть метод BuildSchema як показано нижче. Це потрібно для того, що при запуску застосунку не створювати кожного разу схему бази даних (при створенні схеми бази даних всі дані зникають).

```
private static void BuildSchema(Configuration config)
{
    new SchemaExport(config).Create(true, false);
}
```

15. Відправте застосунок на виконання і переконайтеся, що дані не зникли з бази даних.

4.4 Завдання до лабораторної роботи

Створити застосунок, описаний в ході виконання лабораторної роботи. Замінити об’єкт предметної області «Студент» на об’єкт предметної області відповідно до варіанту з таблиці 4.1 та внести необхідні зміни в код застосунку. Номер варіанта визначається за останньою цифрою залікової книжки.

Таблиця 4.1 – Варіанти завдань

Номер варіанта	Предметна область	Об’єкт предметної області
0	Аеропорт	Пасажир
1	Піцерія	Замовлення
2	Крамниця	Товар
3	Супермаркет	Постачальник
4	Відділ поліції	Порушник
5	Лікарня	Пацієнт
6	Кінотеатр	Сеанс
7	Корабель	Пасажир
8	Підприємство	Співробітник
9	Бібліотека	Книга

4.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;
- номер і назва лабораторної роботи;
- мета лабораторної роботи;
- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

4.6 Контрольні питання

1. За що відповідає відображення в шаблоні MVC?
2. За що відповідає контролер в шаблоні MVC?
3. Які елементи синтаксису Razor вам відомі?
4. Які управляючі структури Razor ви знаєте?
5. Які директиви Razor вам відомі?
6. Чим явні вирази Razor відрізняються від неявних?
7. Чим явний перехід в Razor блоці коду відрізняється від неявного?
8. Які підходи до передачі даних в відображення ви знаєте?
9. Чим об’єкт ViewData відрізняється від об’єкта ViewBag?
10. Для чого використовуються макети в ASP.NET Core?

4.7 Завдання до захисту лабораторної роботи

1. Реалізувати пошук доменних об’єктів по будь-якому полю з використанням запитів Native SQL.
2. Реалізувати пошук доменних об’єктів по будь-якому полю з використанням запитів за критерієм.
3. Реалізувати пошук доменних об’єктів по будь-якому полю з використанням запитів HQL.
4. Додати логіку по відображенню TOP 10 доменних об’єктів з використання запитів Native SQL.
5. Додати логіку по відображенню TOP 10 доменних об’єктів з використання запитів за критерієм.
6. Додати логіку по відображенню TOP 10 доменних об’єктів з використання запитів HQL.
7. Додати серверну валідацію для полів вводу для доменного об’єкту. Виводити помилку валідації біля поля вводу, якщо воно містить помилку.
8. Додати функцію збереження даних в JSON файл і пересилання збереженого файлу за вказаною электронною адресою.
9. Додати функцію збереження даних в XML файл і пересилання збереженого файлу за вказаною электронною адресою.
10. Додати функцію збереження даних в CSV файл і пересилання збереженого файлу за вказаною электронною адресою.

5 ЛАБОРАТОРНА РОБОТА № 5 СТВОРЕННЯ REST-СЕРВІСІВ НА БАЗІ ПЛАТФОРМИ DOTNET CORE

5.1 Мета роботи

Ознайомитися з технологіями створення REST-сервісів на базі платформи Dotnet Core. Отримати практичні навички по роботі з технологіями для створення REST-сервісів на базі платформи Dotnet Core.

5.2 Теоретичні відомості

5.2.1 Основи REST сервісів

REST визначає ряд архітектурних принципів проектування web-сервісів, орієнтованих на системні ресурси, включаючи способи обробки і передачі станів ресурсів по протоколу HTTP різноманітними застосунками, що написані на різних мовах програмування. На даний момент REST є однією з найпоширеніших моделей проектування web-сервісів і він, фактично, справив настільки великий вплив на Web, що практично витіснив дизайн інтерфейсів, заснований на SOAP і WSDL через більш простий стиль проектування.

Технологія REST не привернула великої уваги в 2000 році, коли Рой Філдінг (Roy Fielding) вперше представив її Каліфорнійському університеті в Ірвайні у своїй дисертації «Архітектурні стилі і дизайн мережевих архітектур програмного забезпечення», де аналізувався набір принципів архітектури програмного забезпечення, що використовував Web в якості платформи розподілених розрахунків. Однак сьогодні, через багато років, виникла і продовжує розвиватися велика кількість інфраструктур для REST. В чистому вигляді (в якому вона приваблює настільки щільну увагу) конкретна реалізація web-сервісів REST притримується чотирьох базових принципів проектування:

- явне використання методів протоколу HTTP;
- незбереження станів;
- ідентифікація ресурсів з використанням URI;
- передача даних в XML, JSON або в обох форматах даних.

Далі розглянемо кожен з принципів більш детально.

5.2.2 Зв'язок REST з протоколом HTTP

Однією з ключових характеристик web-сервісів RESTful є явне використання методів протоколу HTTP, що визначені в RFC 2616. Наприклад, HTTP GET визначається як метод генерації даних, що використовується клієнтським застосунком для отримання ресурсів та даних з web-сервера, або виконання запиту в надії на те, що web-сервер знайде і поверне набір відповідних ресурсів. REST пропонує розробникам використовувати HTTP-методи явно відповідно з визначенням протоколу.

Цей основний принцип проектування REST встановлює однозначну відповідність між операціями create, read, update і delete (CRUD) і HTTP-методами. Згідно цієї відповідності для створення ресурсу на сервері використовується метод POST, для отримання ресурсу використовується метод GET, для зміни стану ресурсу чи для його оновлення використовується метод PUT і для видалення ресурсу використовується метод DELETE.

Недоліком проектування багатьох Web API є використання HTTP-методів не за прямим призначенням. Наприклад, GET запит в HTTP зазвичай визначає один конкретний ресурс, або ж рядок запиту містить ряд параметрів, що визначають критерії пошуку сервером набору відповідних ресурсів. Саме так описаний метод GET в HTTP/1.1 RFC. Однак, часто зустрічаються неприваблівні Web API, що використовують HTTP GET для виконання різного роду транзакцій на сервері (наприклад для додання записів в базу даних). В таких випадках запит GET використовується некоректно, або, принаймні, не використовується в REST-стилі (RESTfully). Якщо Web API використовує GET для запуску віддалених процедур, запит може виглядати наступним чином:

```
GET /adduser?name=Robert HTTP/1.1
```

Це є невдалим дизайном, оскільки вищезгаданий метод підтримує операцію, що змінює стан, шляхом виконання HTTP-запиту GET. У випадку успішного виконання запиту в сховищі даних буде доданий новий користувач. Проблема тут є семантичною. Метод GET повинен використовуватися для повернення ресурсів, а не для додання запису в базу даних. З точки зору передбачуваного використання та з точки зору HTTP/1.1-сумісних web-сервісів таке використання методу GET є ненадійним. Але крім семантики ще однією проблемою є те, що метод GET використовується різними засобами web-кешування і пошуковими механізмами, які можуть виконувати ненавмисні зміни сервера шляхом простого обходу посилань, якщо метод GET використовувати для видалення, зміни і додання записів в базу даних або для зміни яких-небудь станів на стороні сервера.

Простим вирішенням даної проблем є передача імен і значень параметрів URI в тілі запиту і використання замість методу GET методу POST. Приклад запиту на додавання нового ресурсу після виконання вищезазначених змін буде виглядати наступним чином:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

Це приклад RESTful-запиту: HTTP-запит POST використовується коректно, а тіло запиту містить корисне навантаження. Після цього клієнтський застосунок може отримати доданий ресурс, використовуючи

новий URI, що вказує, що ресурс розміщений в каталозі /users:

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Це вірне використання запиту GET, оскільки він слугує тільки для отримання даних. GET – це операція, що повинна бути вільною від побічних ефектів. Дана властивість ще відома під назвою ідемпотентність.

Аналогічний рефакторинг треба виконати в тих випадках, коли HTTP-запит GET підтримує операцію update:

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

Після рефакторингу даного запиту запит на зміну даних буде виглядати наступним чином:

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0"?>
<user>
  <name>Bob</name>
</user>
```

Використання запиту PUT для заміни початкового ресурсу забезпечує значно більш прозорий інтерфейс, що сумісний з принципами REST і з визначенням HTTP-методів. Запит PUT з прикладу є явним в тому сенсі, що він вказує на ресурс, що оновлюється, визначаючи його в URI запиті і передає нове значення ресурсу від клієнта на сервер в тілі запиту PUT, замість того, щоб передавати атрибути ресурсу в вигляді слабко зв’язаного набору імен і значень параметрів в URI запиту.

Загальноприйнятим підходом, що відповідає рекомендаціям REST по явному застосуванню HTTP-методів є використання в URI іменників замість дієслів. В web-сервісі RESTful дієслова POST, GET, PUT і DELETE вже визначені протоколом. В ідеалі для реалізації узагальненого інтерфейсу і явного виклику операцій клієнтськими застосунками web-сервіс не повинен визначати додаткові команди чи віддалені процедури, наприклад /adduser чи /updateuser. Цей загальний принцип можна застосовувати і до тіла HTTP-запиту, яке призначене для передачі стану ресурсу, а не для імен методів, що викликаються.

5.2.3 Збереження стану REST сервісів

Для задоволення постійно зростаючих потреб до продуктивності web-сервіси REST повинні бути масштабованими. Для формування топології сервісів, яка дозволяє за необхідності перенаправляти запити з одного сервера на інший з метою зменшення загального часу реакції на виклик web-сервісу, зазвичай застосовують кластери серверів з можливістю розподілення навантаження і аварійного переключення на резерв, проксі-сервери і шлюзи. Використання проміжних серверів для покращення масштабованості

потребує, щоб клієнти web-сервісів REST відправляли повні самодостатні запити, що містять всі необхідні для їх виконання дані, щоб компоненти на проміжних серверах могли перенаправити, маршрутизувати і розподіляти навантаження без локального збереження стану між запитами.

При обробці повного самодостатнього запиту серверу не потрібно вивільняти стан чи контекст застосунку. Застосунок (чи клієнт) web-сервісу REST додає до HTTP-заголовку і тіла запиту всі параметри, контекст і дані, що необхідні серверному компоненту для генерування відповіді. В цьому відношенні незбереження стану (statelessness) покращує продуктивність web-сервісу і спрощує дизайн і реалізацію серверних компонентів, оскільки відсутність стану на сервері усуває необхідність синхронізації сеансів даних з зовнішнім застосунком.

На рисунку 5.1 показаний сервіс, що зберігає стан, в якому застосунок може запросити наступну сторінку в багатосторінковому наборі результатів, вважаючи, що сервер зберігає послідовність переходів застосунку по цьому набору результатів. В цій моделі зі збереженням станів сервіс нарощує і зберігає змінну previousPage, щоб бути здатним відповідати на запити на отримання наступних сторінок.

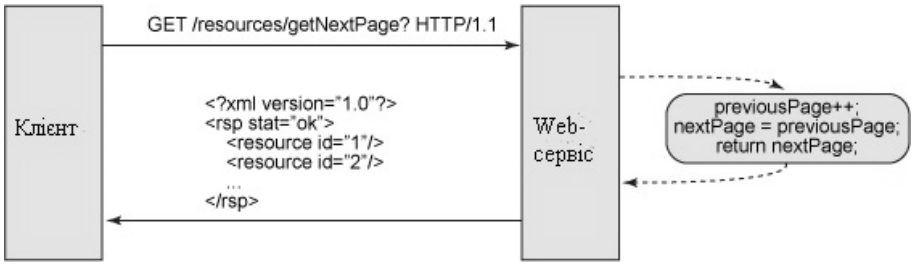

Рисунок 5.1 – Модель зі збереженням стану

Подібні сервіси, що зберігають стан, виходять складними. В протилежність цьому підходу, серверні компоненти, що не зберігають стан, більш прості в проектуванні, написанні і розподіленні між серверами зі збалансованим навантаженням. Сервіс, що не зберігає стан, не тільки краще працює, але і покладає основну відповідальність за збереження станів на клієнтський застосунок. В web-сервісі RESTful сервер відповідає за генерацію відповідей і за надання інтерфейсу, що дозволяє клієнтському застосунку самостійно зберігати свій стан. Наприклад, при запиті багатосторінкового набору результатів клієнтський застосунок повинен включити в запит номер конкретної сторінки, а не просто робити запит на наступну сторінку (див. рисунок 5.2).

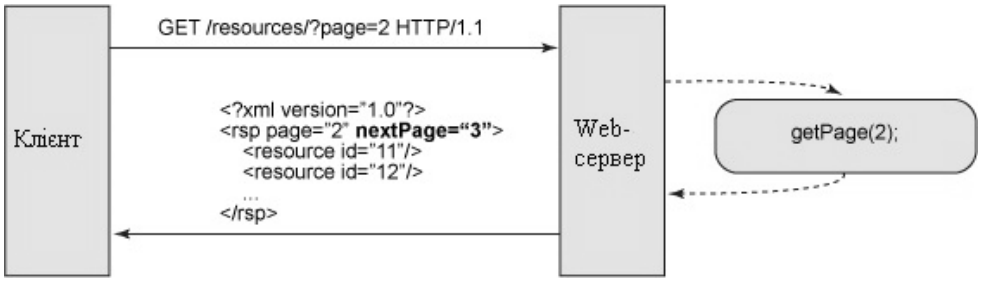

Рисунок 5.2 – Модель без збереження стану

Web-сервіс, що не зберігає стан, генерує відповідь, що містить посилання на номер наступної сторінки в наборі, що дозволяє клієнту самостійно подбати про збереження цього значення. В самому загальному вигляді цей аспект моделі web-сервісів RESTful можна розподілити на дві сфери відповідальності, що пояснюють суть функціонування сервісів, що не зберігають стан:

Сервер генерує відповіді, що містять посилання на інші ресурси для навігації застосунків по пов’язаним ресурсам. Такі відповіді містять вбудовані посилання. Аналогічним чином при запиті батьківського чи контейнерного ресурсу типова RESTful-відповідь може містити посилання на потомків батьківського елементу або на підлеглі ресурси., щоб зберігати зв'язок з ними.

Сервер генерує відповіді, що містять інформацію про те, чи підлягають вони кешуванню з метою підвищення продуктивності за рахунок зменшення кількості запитів ресурсів, що дублюються і повної відмови від деяких запитів. Для цього сервер додає до відповіді HTTP-заголовки Cache-Control і Last-Modified.

Клієнтський застосунок по заголовку Cache-Control відповіді визначає можливість кешування ресурсу. Також клієнтський застосунок читає заголовок Last-Modified відповіді і повертає значення дати в заголовку If-Modified-Since для відправки на сервер запиту про зміну ресурсу. Такий запит, так званий Conditional GET, використовує обидва заголовки. Якщо ресурс з вказаного часу не змінювався, відповіддю сервера є стандартний код 304 (Not Modified) і ресурс, що було запитано, не відправляється. Код відповіді 304 HTTP означає, що клієнтський застосунок може спокійно використовувати кешовану локальну копію ресурсу в якості самої останньої його версії, фактично опускаючи наступні запити GET до того часу, поки ресурс не буде змінений.

Клієнтський застосунок відправляє повні запити, які можуть оброблюватися незалежно від інших запитів. Це потребує від клієнтського застосунку використання в повному обсязі HTTP-заголовків, визначених інтерфейсом web-сервісу, і відправки повних представлень ресурсів в тілі запиту. Клієнтський застосунок відправляє запити, які практично нічого не знають про попередні запити, про існування сеансу на сервері, про здатність серверу додавати контекст в запит і про стан застосунку, що зберігається між запитами.

Така сумісна робота клієнтського застосунку і сервера дуже важлива для відмови від збереження стану в web-сервісах RESTful. Її результатом є ріст продуктивності за рахунок зменшення трафіку і мінімізації стану серверного застосунку.

5.2.4 Ідентифікація ресурсів з використанням URI

З точки зору звернення до ресурсів з клієнтського застосунку надані URI визначають, наскільки інтуїтивним буде web-сервіс REST і чи буде він використовуватися так, як припускав розробник.

URI-адреса web-сервісу REST повинна бути інтуїтивно зрозумілою. URI треба розглядати як деякий самодокументований інтерфейс, який майже не потребує пояснень або звернення до розробника для його розуміння і для отримання відповідних ресурсів. Тому структура URI повинна бути простою, передбачуваною і зрозумілою.

Одним із способів досягнути такого рівня зручності використання – побудова URI по аналогії зі структурою каталогів. Такого роду URI є ієрархічними, такими, що виходять із одного кореневого шляху, розгалуження якого відображають основні функції сервісу. Відповідно до цього визначення, URI – це не просто рядок зі слешами в якості розділових знаків, а скоріше дерево з вище лежачими і нижче лежачими гілками, з'єднаними в вузлах. Наприклад, в сервісі обговорення різних тем, можна визначити структурований набір URI наступного вигляду:

```
http://www.myservice.org/discussion/topics/{topic}
```

Корінь /discussion має нижче стоячий вузол /topics. Нижче розташовуються назви тем (наприклад, gossip, technology і т. д.), кожна з яких вказує на свою гілку обговорення. В рамках даної структури можна легко викликати гілки обговорення простим введенням будь-чого після /topics/.

В деяких випадках каталогоподібна структура особливо добре підходить для шляхів до ресурсів. В якості прикладу можна назвати ресурси впорядковані по даті. Для них дуже добре підходить ієрархічний синтаксис. Наступний приклад інтуїтивно зрозумілий, оскільки заснований на правилах:

```
http://www.myservice.org/discussion/2008/12/10/{topic}
```

Перший фрагмент шляху – чотири цифри року, другий – дві цифри дня і третій – дві цифри місяця. Подібне пояснення може показатися дещо спрощеним, але це саме той рівень простоти, який нам потрібен. Люди і комп'ютери можуть легко генерувати подібні структуровані URI, оскільки вони засновані на правилах. Вказування фрагментів шляху у відповідних позиціях відповідно синтаксису робить URI уніфікованими, оскільки існує закономірність їх створення:

```
http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}
```

Приведемо деякі додаткові рекомендації, на які слід звернути увагу при обдумуванні структури URI для web-сервісів RESTful:

1. Необхідно приховувати розширення файлів серверних сценаріїв (.jsp, .php, .asp), якщо такі використовуються, що, можна було виконувати перенесення застосунків на інші технології без зміни URI.
2. Необхідно використовувати тільки малі літери.
3. Необхідно заміняти пробіли дефісами чи знаками підкреслення (чимось одним).
4. Треба максимально уникати використання рядків запиту.
5. Замість використання коду 404 Not Found для URI, що вказує на

неповний шлях, завжди необхідно надавати в якості відповіді ресурс чи сторінку за замовчуванням.

6. URI також повинен бути статичним, щоб при зміні ресурсів чи реалізації сервісу, посилання залишалося тим самим. Це дозволяє зберігати закладки. Також важливо, щоб взаємозв'язки між ресурсами, закодованими в URI, залишалися незалежними від способу вказування місця розташування ресурсів в сховищі.

5.2.5 Формати передачі даних

Відображення ресурсу зазвичай показує поточний стан ресурсу (або його атрибутів) на момент запиту його клієнтським застосунком. Відображення ресурсів в цьому сенсі є просто знімками станів в певний момент часу. Ці відображення повинні бути такими ж простими, як відображення запису бази даних, що складається з зв'язків між іменем колонок і XML-тегами, де значення елементів в XML містить значення рядків. Якщо система має модель даних, то відповідно цьому визначенню відображення ресурсу є знімком стану атрибутів одного з об'єктів моделі даних системи. Це ті об'єкти, які буде обслуговувати web-сервіс REST.

Останній набір обмежень, тісно пов'язаний з дизайном web-сервісів RESTful, відноситься до формату даних, якими обмінюються застосунок і сервіс при роботі в режимі запит-відповідь. Тут особливо важлива простота, читабельність і зв'язаність. Об'єкти моделі зазвичай якимось чином пов'язані і ці відношення між об'єктами (ресурсами) моделі даних повинні відображатися в способі їх відображення для передачі клієнтському застосунку. В сервісі обговорень приклад відображень зв'язаних ресурсів може включати в себе кореневу тему обговорення і її атрибути, а також вбудовані посилання на відповіді, відправлені в цю тему.

```
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

Також, щоб надати клієнтським застосункам можливість робити запит на конкретний найбільш підходящий їх тип вмісту, необхідно проектувати сервіс так, що він використовував вбудований HTTP-заголовок Accept, значення якого є MIME типом. Деякими загальновживаними MIME-типами, що використовуються в RESTful-сервісах є JSON (application/json), XML (application/xml), XHTML (application/xhtml+xml) та інші.

Це дозволить використовувати сервіс клієнтським застосункам, написаним на різних мовах, працюючих на різних платформах і приладах.

Використання MIME-типів і HTTP-заголовка Accept є механізмом узгодження вмісту (content negotiation), що дозволяє клієнтським застосункам обирати підходящий для них формат даних і мінімізувати зв'язність даних

між сервісом і застосунком, що його використовує.

5.2.6 HTTP коди

В стандарті HTTP описано більш ніж 70 статус кодів. Гарним підходом є використання хоча б основних статус кодів.

200 – ОК – успішний запит. Якщо клієнт зробив запит на деякі дані, то вони знаходяться в заголовку і/або тілі повідомлення.

201 – ОК – в результаті успішного виконання запиту був створений новий ресурс.

204 – ОК – ресурс успішно видалений.

304 – Not Modified – клієнт може використовувати дані з кеша.

400 – Bad Request – запит інвалідний або не може бути оброблений.

401 – Unauthorized – запит потребує аутентифікацію користувача.

403 – Forbidden – сервер зрозумів запит, але відмовляється його обробити або доступ заборонений.

404 – Not found – ресурс не знайдений.

500 – Internal Server Error – розробники API повинні намагатися уникати таких помилок. Ці помилки повинні бути виловлені в глобальному catch блоці, але не повинні бути повернені у відповіді.

Чим більшим є набір кодів, який буде використовуватися, тим більш зрозумілим буде API, яке створюється. Але треба врахувати, що деякі коди браузери оброблюють по-різному. Наприклад, деякі браузери отримавши код відповіді 307 одразу виконують перенаправлення, а деякі дозволяють обробити таку ситуацію і відмінити дію. Перш ніж використовувати той чи інший код, необхідно цілком розуміти, як він буде оброблюватися клієнтською стороною.

5.2.7 Параметри пошуку ресурсів

Щоб спростити використання сервісів, що відповідають за повернення будь-яких даних і додатково зробити їх найбільш продуктивними, необхідно використовувати в якості параметрів запиту параметри для сортування, фільтрації, вибору полів і пагінації.

Для виконання фільтрації необхідно використовувати унікальний параметр запиту для кожного поля. Це дозволяє обмежити кількість даних, що виводиться, що оптимізує час обробки запиту. Наприклад, щоб вивести всі червоні книжки, необхідно виконати запит:

```
GET /books?color=red
```

Сортування реалізується подібно фільтрації. Наприклад, щоб вивести всі книжки, відсортовані по року публікації по спаданню і по назві по зростанню необхідно виконати наступний запит:

```
GET /books?sort=-year,+name
```

Для того, щоб підтримати можливість завантаження списку ресурсів,

які повинні відображатися на певній сторінці застосунку, в REST API повинен бути передбачений функціонал пагінації. Реалізується він з допомогою знайомих нам по SQL параметрів limit і offset. Наприклад:

```
GET /books?offset=10&limit=5
```

Окрім того гарним тоном є вивід посилання на попередню і наступну , першу і останню сторінки в хедері Link. Наприклад:

```
Link: <http://localhost/api/books?offset=15&limit=5>; rel=”next”,
<http://localhost/api/books?offset=50&limit=3>; rel=”last”,
<http://localhost/api/books?offset=0&limit=5>; rel=”first”,
<http://localhost/api/books?offset=5&limit=5>; rel=”prev”
```

Рекомендовано також повертати загальну кількість ресурсів в хедері X-Total-Count.

Для більш зручного використання сервісу, для економії трафіку можна надати можливість управляти форматом виводу даних, що реалізується наданням можливості вибору полів ресурсу, які повинен повернути REST сервіс. Наприклад, якщо необхідно отримати тільки id книжок і їх кольори, необхідно виконати наступний запит:

```
GET /books?fields=id,color
```

5.2.8 Обробка виключень

При виникненні помилкових ситуацій необхідно виводити відформатовану і зрозумілу інформацію. Це відноситься в першу чергу до статус кодів в HTTP відповіді. Помилки в сервісах найчастіше відносяться до двох типів:

- 4xx – помилки клієнта;
- 5xx – помилки сервера.

У випадку помилки клієнта, наприклад, помилка валідації якого-небудь з параметрів запиту, в тілі відповіді рекомендовано передавати корисну інформацію про помилку: повідомлення, опис, код (наприклад в форматі JSON). У випадку помилки сервера відправляти бажано відправляти додаткову інформацію в тілі відповіді, наприклад у випадках коли сервер не доступний.

Поганим тоном є вивід всього стек трейсу виключення. Рекомендовано для кожної виключної ситуації мати свій код. В подальшому для виводу інформації про помилку можна буде додавати посилання на документацію, в якій даний код буде виступати унікальним ідентифікатором. Наприклад:

```
{
   "code" : 1234,
   "message" : "Something bad happened :(",
   "description" : "More details about the error here",
   “moreInfo”: “http:/localhost/api/v2/errors/1234”
}
```

5.2.9 Атрибути ASP.NET Core Web API

ASP.NET Core Web API є фреймворком для розробки REST-сервісів з використанням протоколу HTTP. Він дозволяє розробникам створювати API з урахуванням потреб користувачів та використовувати різноманітні формати даних, такі як JSON та XML.

Нижче наведено кілька найбільш використовуваних атрибутів ASP.NET Core Web API з прикладами їх використання.

[HttpGet] – цей атрибут використовується для визначення методу контролера, який повертає результат GET-запиту:

```
[HttpGet]
public ActionResult<IEnumerable<string>> Get()
{
    return new string[] { "value1", "value2" };
}
```

[HttpPost] – цей атрибут використовується для визначення методу контролера, який обробляє POST-запити:

```
[HttpPost]
public void Post([FromBody] string value)
{
    //обробка POST-запиту
}
```

[HttpPut] – цей атрибут використовується для визначення методу контролера, який оновлює існуючий ресурс.

```
[HttpPut("{id}")]
public void Put(int id, [FromBody] string value)
{
    //оновлення ресурсу з ідентифікатором id
}
```

[HttpDelete] – цей атрибут використовується для визначення методу контролера, який видаляє існуючий ресурс.

```
[HttpDelete("{id}")]
public void Delete(int id)
{
    //видалення ресурсу з ідентифікатором id
}
```

[HttpHead] – дозволяє визначити метод HTTP HEAD для запиту, який повинен бути оброблений контролером.

```
[HttpHead]
public IActionResult Get()
{
    //обробка HEAD-запиту
}
```

[HttpPatch] – дозволяє визначити метод HTTP PATCH для запиту, який повинен бути оброблений контролером.

```
[HttpPatch("{id}")]
public IActionResult Update(int id, [FromBody] JsonPatchDocument<MyModel>
patchDoc)
{
    //обробка PATCH-запиту
}
```

[HttpOptions] - дозволяє визначити метод HTTP OPTIONS для запиту, який повинен бути оброблений контролером.

```
[HttpOptions]
public IActionResult Get()
{
    //обробка OPTION-запиту
}
```

[Route] – цей атрибут використовується для визначення маршруту URL для методу контролера.

```
[HttpGet]
[Route("api/values")]
public ActionResult<IEnumerable<string>> Get()
{
    return new string[] { "value1", "value2" };
}
```

[FromBody] – цей атрибут використовується для вказівки параметра методу контролера, який повинен бути взятий з тіла запиту.

```
[HttpPost]
public IActionResult Create([FromBody] MyModel model)
{
    if (model == null)
    {
        return BadRequest();
    }

    // do something with the model
    // ...

    return Ok();
}
```

[FromForm] – цей атрибут використовується для зв'язування даних запиту з параметрами дії контролера з форми HTML.

```
[HttpPost]
public IActionResult AddUser([FromForm] string name, [FromForm] int age)
{
    //додати користувача з ім'ям та віком
    return Ok();
}
```

[FromHeader] – цей атрибут використовується для зв'язування даних запиту з параметрами дії контролера з заголовків запиту.

```
[HttpGet]
public IActionResult GetUser([FromHeader(Name = "Authorization")] string
token)
{
```

```
    //отримати користувача з токеном авторизації
    return Ok();
}
```

[FromQuery] – цей атрибут використовується для зв'язування даних запиту з параметрами дії контролера з рядка запиту.

```
[HttpGet]
public IActionResult GetUsers([FromQuery] int page, [FromQuery] int pageSize)
{
    //отримати список користувачів для сторінки та розміру сторінки
    return Ok();
}
```

[FromRoute] – цей атрибут використовується для зв'язування даних запиту з параметрами дії контролера з шляху запиту.

```
[HttpGet("{id}")]
public IActionResult GetUserById([FromRoute] int id)
{
    //отримати користувача за ідентифікатором
    return Ok();
}
```

[Consumes] – цей атрибут вказує, що тип контенту, що передається, має відповідати заданому формату.

```
[Consumes("application/json")]
public IActionResult AddStudent(Student student)
{
    // ...
}
```

[Produces] – вказує, що тип контенту, який повертається, має відповідати заданому формату.

```
[Produces("application/json")]
public IActionResult GetStudent(int id)
{
    // ...
}
```

[ResponseCache] – вказує, що відповідь на запит може бути закешована на стороні клієнта або сервера.

```
[ResponseCache(Duration = 60)]
public IActionResult GetStudents()
{
    // ...
}
```

5.3 Порядок виконання лабораторної роботи

В даній лабораторній роботі був створений набір сервісів для взаємодії з реляційною базою даних з використанням технології Fluent Nhibernate. Взаємодія буде відбуватися з однією табличкою бази даних «Студент». На відміну від попередньої лабораторної роботи, в якій для прошарку

відображення була використана технологія ASP.NET Core MVC, в даній лабораторній роботі нема прошарку відображення, а тестування набору сервісів буде з використанням Postman.

1. Створіть в директорії з проектами (в якості прикладу у всіх лабораторних роботах використовується директорія D:\csexamples, але директорія з проектами може знаходитись в будь-якому зручному місці) директорію для нового проекту з назвою restapi (або з будь-якою іншою назвою). Відкрийте Visual Studio Code і змініть директорію проекту на D:\csexamples\restapi (File->Open Folder). Відкрийте термінал в Visual Studio Code, якщо він не з'явився автоматично (Terminal->New Terminal).

2. Зараз необхідно створити нову базу даних в postgres з назвою universityrestapi. В терміналі введіть команду:

```
psql --username=postgres --password=Welcome1
```

Ім'я користувача і пароль для postgresql залежить від налаштувань СУБД і може бути іншим.

В командному рядку postgres введіть команду:

```
create database universityrestapi;
```

3. В терміналі Visual Studio Code введіть команду exit для того, щоб вийти з postgres. Введіть команду для створення нового порожнього web-застосунку в директорії D:\csexamples\restapi:

```
dotnet new web
```

4. В даній лабораторній роботі будуть використовуватися додаткові бібліотеки для роботи з технологією Fluent Nhibernate, а також бібліотеки для роботи з REST API і Swagger, які необхідно додати до проекту шляхом виконання наступних команд:

```
dotnet add package fluentnhibernate
dotnet add package npgsql
dotnet add package Microsoft.AspNetCore.OpenApi
dotnet add package Swashbuckle.AspNetCore
```

5. Створіть в корені проекту директорію Domain. Створіть в директорії Domain класи EntityBase.cs, Student.cs та StudentMap.cs:

```
//Базовий доменний клас
namespace restapi.Domain
{
    public abstract class  EntityBase
    {
        public virtual long Id { get; set; }
    }
}

//Доменний клас студента
namespace restapi.Domain
{
    public class Student : EntityBase
    {
        public virtual string? FirstName { get; set; }
```

```
        public virtual string? LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

    }
}

//Клас відображення студента
using FluentNHibernate.Mapping;

namespace restapi.Domain
{
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
        }
    }
}
```

6. Створіть в корені проекту директорію Dao. Додайте до директорії Dao інтерфейси і класи шаблону DAO IGenericDAO.cs, IGenericDAO.cs, IStudentDAO.cs, StudentDAO.cs, DAOFactory.cs, NHibernateDAOFactory.cs:

```
//Інтерфейс IGenericDAO
namespace restapi.Dao
{
    public interface IGenericDAO<T>
    {
        T Merge(T item);

        T GetById(long id);

        List<T> GetAll();

        void Delete(T item);
    }
}

//Клас GenericDAO
using NHibernate;
using ISession = NHibernate.ISession;

namespace restapi.Dao
{
    public class GenericDAO<T> : IGenericDAO<T>
    {
        protected ISession session;

        public GenericDAO() { }

        public GenericDAO(ISession session)
        {
            this.session = session;
        }
```

```
        public T Merge(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            T resultItem = (T)session.Merge(item);
            transaction.Commit();
            return resultItem;
        }

        public T GetById(long id)
        {
            return session.Get<T>(id);
        }

        public List<T> GetAll()
        {
            return new List<T>(session.CreateCriteria(typeof(T)).List<T>());
        }

        public void Delete(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.Delete(item);
            transaction.Commit();
        }
    }
}


//Інтерфейс IStudentDAO
using restapi.Domain;

namespace restapi.Dao
{
    public interface IStudentDAO:IGenericDAO<Student>
    {

    }
}

//Клас StudentDAO
using ISession = NHibernate.ISession;
using restapi.Domain;

namespace restapi.Dao
{
    public class StudentDAO : GenericDAO<Student>, IStudentDAO
    {

        public StudentDAO(ISession session) : base(session) { }

    }
}

//Клас DAOFactory
namespace restapi.Dao
{
    abstract public class DAOFactory
    {
        public abstract IStudentDAO getStudentDAO();

        public abstract void destroy();

    }
```

```
}

//Клас NHibernateDAOFactory
using System.Reflection;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;
using ISession = NHibernate.ISession;

namespace restapi.Dao
{
    public class NHibernateDAOFactory : DAOFactory
    {
        private static ISessionFactory factory;

        private static DAOFactory instance;

        private ISession session;

        private IStudentDAO studentDAO;

        public static DAOFactory getInstance()
        {
            if (null == instance)
            {
                ISession session = openSession("127.0.0.1",
                    Convert.ToInt32("5432"), "universityrestapi",
                    "postgres", "Welcome1");
                instance = new NHibernateDAOFactory(session);
            }
            return instance;
        }

        public NHibernateDAOFactory(ISession session)
        {
            this.session = session;
        }

        public override IStudentDAO getStudentDAO()
        {
            if (null == studentDAO)
            {
                studentDAO = new StudentDAO(session);
            }
            return studentDAO;
        }

        public override void destroy()
        {
            session.Close();
        }

        private static ISession openSession(String host, int port,
            String database, String user, String password)
        {
            ISession session = null;
            Assembly mappingsAssemly = Assembly.GetExecutingAssembly();
            if (null == factory)
            {
                factory = Fluently.Configure()
                   .Database(PostgreSQLConfiguration
                   .PostgreSQL82.ConnectionString(c => c
```

```
                .Host(host)
                .Port(port)
                .Database(database)
                .Username(user)
                .Password(password)))
                .Mappings(m => m.FluentMappings
                .AddFromAssembly(mappingsAssemly))
                .ExposeConfiguration(BuildSchema)
                .BuildSessionFactory();
            }
            session = factory.OpenSession();
            return session;
        }

        private static void BuildSchema(Configuration config)
        {
            new SchemaExport(config).Create(true, true);
        }
    }
}
```

7. Створіть в корені проекту директорію Controllers. Додайте до директорії Controllers клас контролера StudentController.cs:

```
using Microsoft.AspNetCore.Mvc;
using restapi.Domain;
using restapi.Dao;

namespace restapi.Controllers
{
    [ApiController]
    [Route("student")]
    public class StudentController : ControllerBase
    {

        //Сервіс для виводу всіх студентів
        [HttpGet]
        [Consumes("application/json")]
        [Produces("application/json")]
        public IList<Student> GetAllStudents()
        {
            IList<Student> students = NHibernateDAOFactory
                .getInstance().getStudentDAO().GetAll();
            return students;
        }

        //Сервіс для додавання нового студента
        [HttpPost]
        [Consumes("application/json")]
        [Produces("application/json")]
        public Student AddStudent(Student student)
        {
            Student resultStudent = NHibernateDAOFactory
                .getInstance().getStudentDAO().Merge(student);
            return resultStudent;
        }

        //Сервіс для редагування студента
        [HttpPut]
        [Consumes("application/json")]
        [Produces("application/json")]
        public Student UpdateStudent(Student student)
        {
```

```
            Student resultStudent = NHibernateDAOFactory
                .getInstance().getStudentDAO().Merge(student);
            return resultStudent;
        }

        //Сервіс для видалення студента
        [HttpDelete]
        [Consumes("application/json")]
        [Produces("application/json")]
        [Route("{id}")]
        public string Delete(long id)
        {
            string result;
            Student student = NHibernateDAOFactory
                .getInstance().getStudentDAO().GetById(id);
            if (null != student)
            {
                NHibernateDAOFactory.getInstance()
                    .getStudentDAO().Delete(student);
                result = "Student was successfully removed.";
            }
            else
            {
                result = "Nothing was removed.";
            }

            return result;
        }

    }
}
```

8. Змініть вміст класу Program.cs як показано нижче:

```
using restapi.Dao;

internal class Program
{
    private static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();
        var app = builder.Build();
        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }
        app.UseHttpsRedirection();
        app.UseAuthorization();
        app.MapControllers();
        var applicationLifetime = app.Services
            .GetRequiredService<IHostApplicationLifetime>();
        applicationLifetime.ApplicationStopping.Register(OnShutdown);
        app.Run();
    }

    //Метод для обробки зупинки застосунку
    public static void OnShutdown()
    {
        NHibernateDAOFactory.getInstance().destroy();
```

```
    }
}
```

9. Відправте застосунок на виконання (Run -> Start Debugging або Run -> Run Without Debugging), виправте помилки. В браузері відрийте http://localhost:5187/swagger/index.html. Порт (5187) може відрізнятися і залежить від налаштувань системи. Хост і порт застосунку можна знайти в консольному виводі старту застосунку на вкладці DEBUG CONSOLE:

```
Now listening on: http://localhost:5187
```

Сторінка Swagger з сервісами для доменного обєкта «Студент» показана на рисунку 5.3.

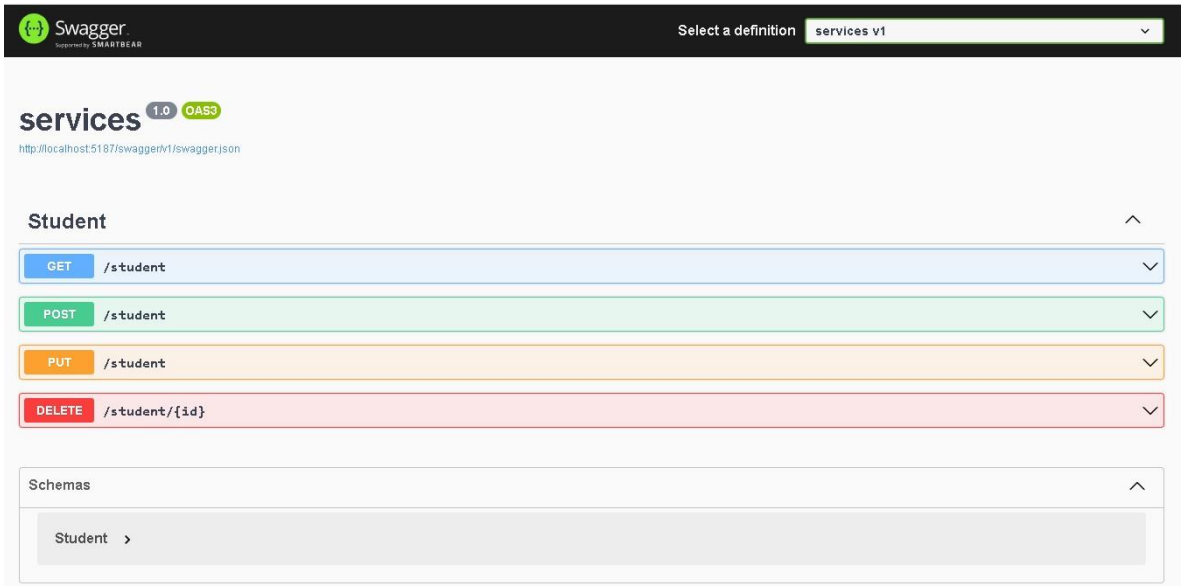

Рисунок 5.3 - Сторінка Swagger з сервісами для доменного обєкта «Студент»

10. Swagger може використовуватися в якості інструменту для виконання запитів до сервісів. Розкрийте кожен сервіс, перегляньте, яким чином сервіси відображаються в Swagger. Здійсніть виклик методу GET. Для цього розкрийте елемент GET, натисніть кнопку Try it out, а потім натисніть кнопку Execute. Перегляньте відповідь сервісу. Сервіс поверне порожній список студентів, але це спричинить генерацію схеми бази даних.

11. Зупиніть застосунок. Відкрийте клас NHibernateDAOFactory.cs і змініть метод BuildSchema як показано нижче. Це потрібно для того, що при запуску застосунку не створювати кожного разу схему бази даних (при створенні схеми бази даних всі дані зникають).

```
private static void BuildSchema(Configuration config)
{
    new SchemaExport(config).Create(true, false);
}
```

12. Swagger є хорошим інструментом для взаємодії з сервісами, але біль досконалим інструментом є Postman. Він дозволяє створювати проекти з різнотипними запитами до сервісів. Встановіть Postman. На момент створення даних методичних вказівок його можна було завантажити за

посиланням https://www.postman.com/downloads/. Відправте застосунок на виконання (Run -> Start Debugging або Run -> Run Without Debugging).

13. Відкрийте Postman. Створіть нову колекцію сервісів (File -> New -> Collection) з іменем Student collection.

14. Додайте до колекції Student collection новий запит GET. Назвіть його Get All Students, задайте URL адресу http://{{domain}}/student. Відкрите вкладку Body і впевніться, що на цій вкладці вибраний елемент none, як показано на рисунку 5.4.

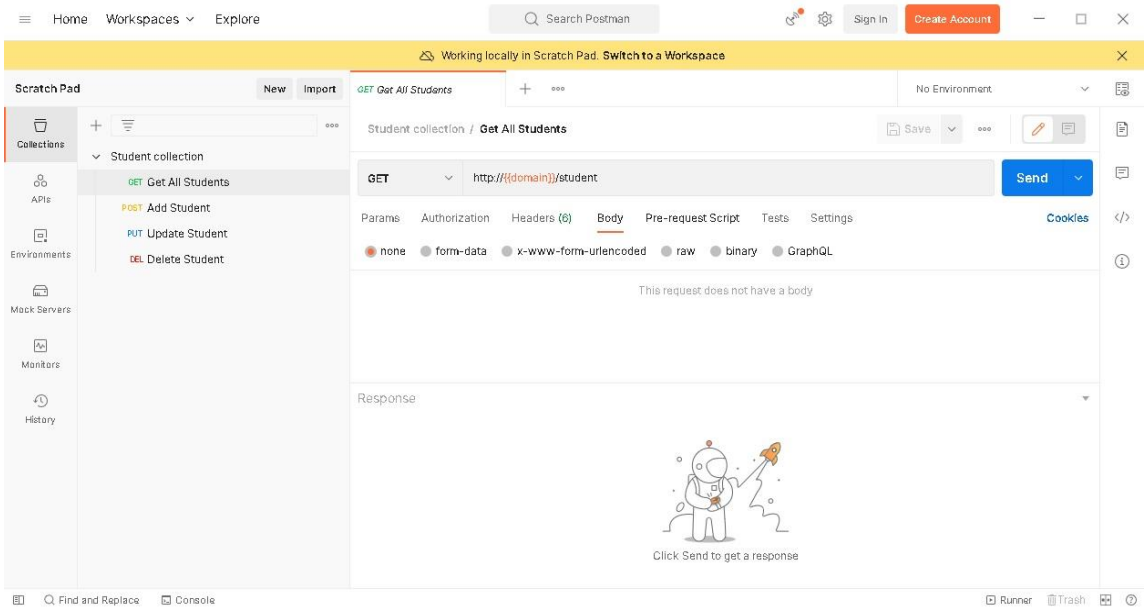

Рисунок 5.4 – Конфігурація сервісу Get All Students в Postman

15. Натисніть на Student collection та на вкладці Variables задайте значення змінної {{domain}} localhost:5187, як показано на рисунку 5.5. Порт (5187) може відрізнятися і залежить від налаштувань системи.

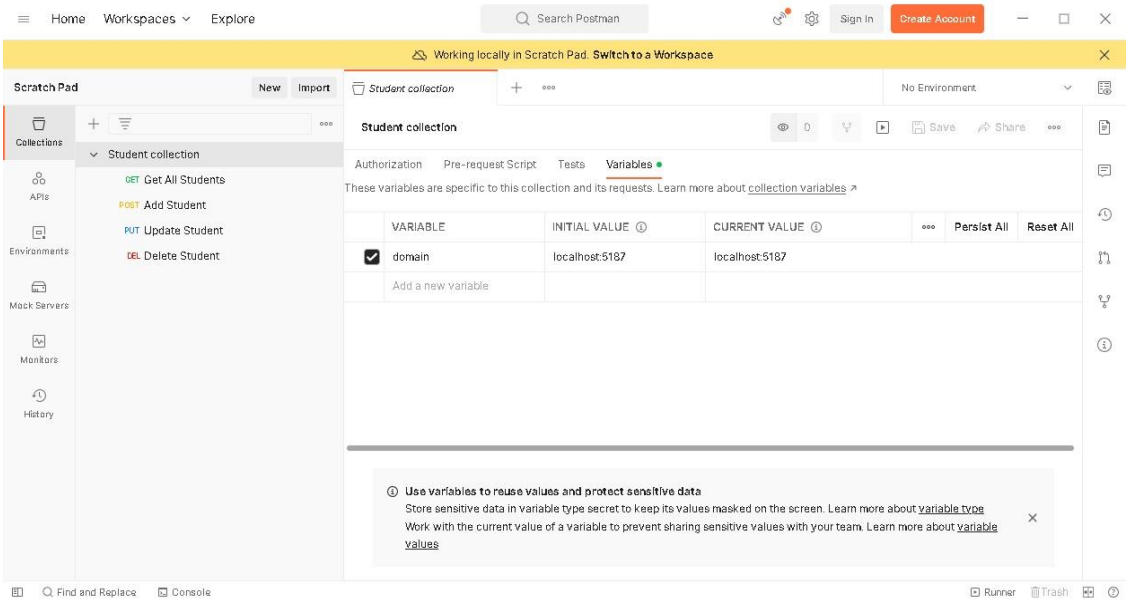

Рисунок 5.5 – Вкладка Variables для встановлення значення змінних

16. Додайте до колекції Student collection новий запит POST. Назвіть його Add Student, задайте URL адресу http://{{domain}}/student. Відкрите вкладку Body, оберіть значення raw, з випадаючого списку оберіть JSON, додайте в поле вводу JSON:

```
{
    "firstName": "New First Name",
    "lastName": "New Last Name",
    "sex": "M",
    "year": 1990
}
```

Конфігурація сервісу Add Student показана на рисунку 5.6.

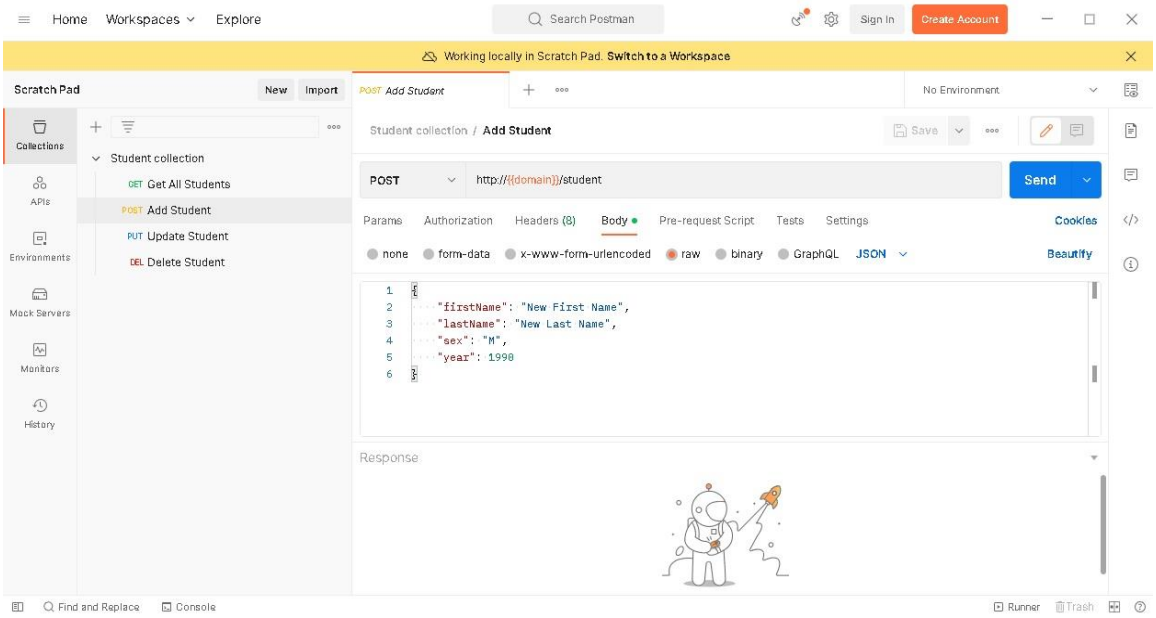

Рисунок 5.6 – Конфігурація сервісу Add Student в Postman

17. Додайте до колекції Student collection новий запит PUT. Назвіть його Update Student, задайте URL адресу http://{{domain}}/student. Відкрите вкладку Body, оберіть значення raw, з випадаючого списку оберіть JSON, додайте в поле вводу наступний JSON:

```
{
    "firstName": "New Updated First Name",
    "lastName": "New Updated Last Name",
    "sex": "M",
    "year": 1990,
    "id": 9
}
```

Конфігурація сервісу Update Student показана на рисунку 5.7.

18. Додайте до колекції Student collection новий запит DELETE. Назвіть його Delete Student, задайте URL адресу http://{{domain}}/student/6. Відкрите вкладку Body і впевніться, що на цій вкладці вибраний елемент none, як показано на рисунку 5.8.

19. Додайте декілька студентів з використанням сервісу Add Student, отримайте список студентів з використанням сервісу Get All Students,

виконайте оновлення студента з використання сервісу Update Student (необхідно змінити значення поля id в body (id=9) на існуюче значення), виконайте видалення студента з використанням сервісу Delete Student (необхідно змінити значення id в URL запиту (id=6) на існуюче значення).

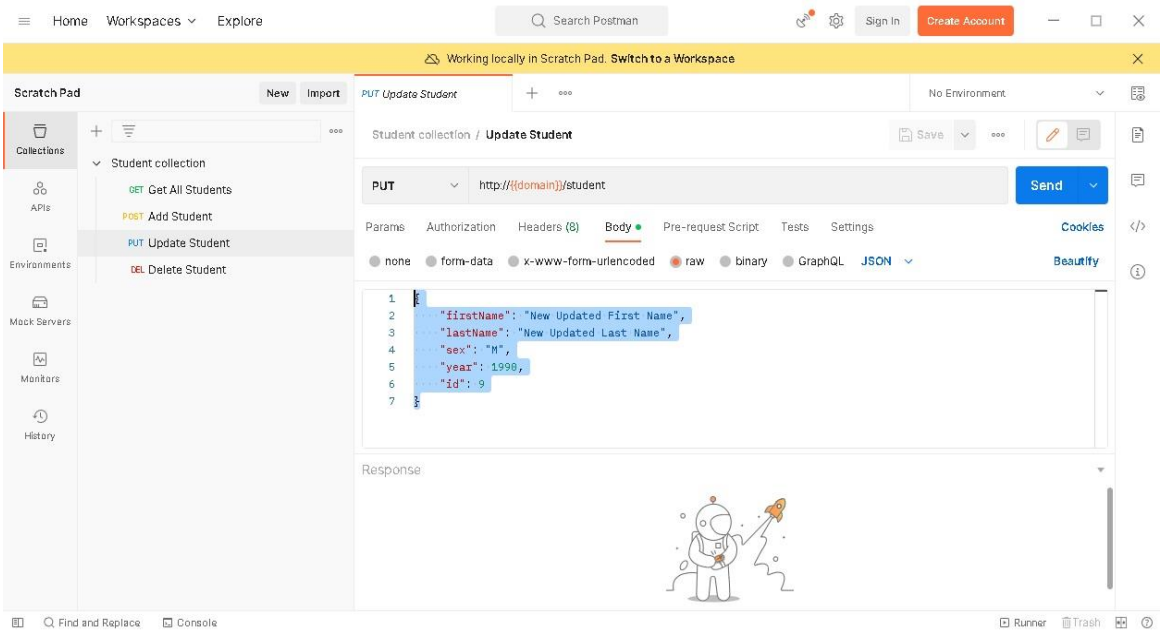

Рисунок 5.7 – Конфігурація сервісу Update Student в Postman

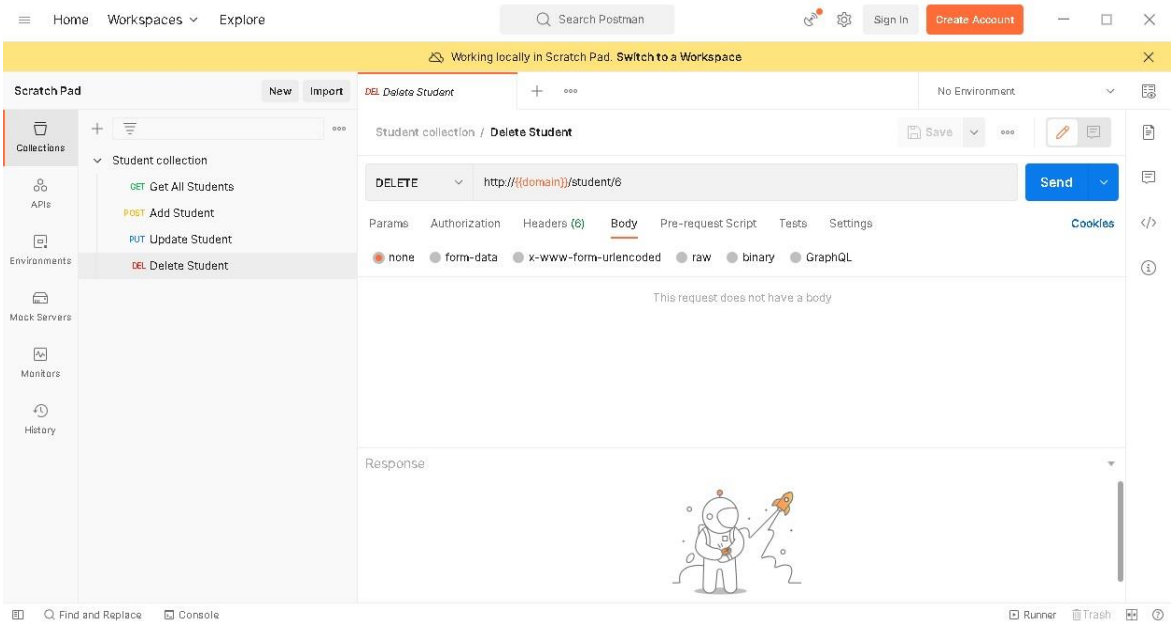

Рисунок 5.8 - Конфігурація сервісу Delete Student в Postman

5.4 Завдання до лабораторної роботи

Створити застосунок, описаний в ході виконання лабораторної роботи. Замінити об’єкт предметної області «Студент» на об’єкт предметної області відповідно до варіанту з таблиці 5.1 та внести необхідні зміни в код застосунку. Номер варіанта визначається за останньою цифрою залікової книжки.

Таблиця 5.1 – Варіанти завдань

Номер варіанта	Предметна область	Об’єкт предметної області
0	Аеропорт	Пасажир
1	Піцерія	Замовлення
2	Крамниця	Товар
3	Супермаркет	Постачальник
4	Відділ поліції	Порушник
5	Лікарня	Пацієнт
6	Кінотеатр	Сеанс
7	Корабель	Пасажир
8	Підприємство	Співробітник
9	Бібліотека	Книга

5.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;
- номер і назва лабораторної роботи;
- мета лабораторної роботи;
- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

5.6 Контрольні питання

1. Що таке REST-сервіс?
2. Які базові принципи побудови REST-сервісів ви знаєте?.
3. Як REST-сервіси пов’язані з протоколом HTTP?
4. Які рекомендації існують для REST сервісів з приводу URI?
5. Які рекомендації існують для REST сервісів з приводу пагінації і сортування?
6. Які рекомендації існують для REST сервісів з приводу обробки виключень?
7. Які типи вхідних і вихідних даних підтримуються в REST-сервісах і яким чином їх можна задати?
8. Які способи передачі параметрів в REST-сервіси існують?
9. Які атрибути для передачі параметрів є в ASP.NET Core Web API?
10. Які атрибути ASP.NET Core Web API ви знаєте?

5.7 Завдання до захисту лабораторної роботи

1. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів Native SQL. Рядок пошуку повинен передаватися через URI. Формат вихідних даних XML.
2. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів Native SQL. Рядок пошуку повинен передаватися через тіло запиту. Формат вихідних даних JSON.

3. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів Native SQL. Рядок пошуку повинен передаватися через заголовок запиту. Формат вихідних даних XML.

4. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів за критерієм. Рядок пошуку повинен передаватися через URI. Формат вихідних даних XML.

5. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів за критерієм. Рядок пошуку повинен передаватися через тіло запиту. Формат вихідних даних JSON.

6. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів за критерієм. Рядок пошуку повинен передаватися через заголовок запиту. Формат вихідних даних XML.

7. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів HQL. Рядок пошуку повинен передаватися через параметри запиту. Формат вихідних даних XML.

8. Створити сервіс пошуку доменних об’єктів по будь-якому полю з використанням запитів HQL. Рядок пошуку повинен передаватися через тіло запиту. Формат вихідних даних CSV.

9. Створити сервіс відправки електронних листів. Текст листа повинен передаватись через тіло запиту, адреса електронної пошти повинна передаватись через заголовок запиту, всі інші параметри для відправки листа повинні передаватись через параметри запиту.

10. Створити сервіс для авторизації та аутентифікації користувачів з використанням JWT-токенів.

6 ЛАБОРАТОРНА РОБОТА № 6 ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ЗАСТОСУНКІВ В DOTNET CORE

6.1 Мета роботи

Ознайомитися з технологіям автоматизованого тестування застосунків в dotnet core і особливостями їх використання. Отримати практичні навички по роботі з технологіями автоматизованого тестування застосунків в dotnet core.

6.2 Теоретичні відомості

Модульне тестування, або unit-тестування (англ. unit testing) – процес у програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми.

Ідея полягає в тому, щоб писати тести для кожної нетривіальною функції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергове зміна коду до регресії, тобто до появи помилок у вже протестованих місцях програми, а також полегшує виявлення та усунення таких помилок. Мета модульного тестування - ізолювати окремі частини програми і показати, що окремо ці частини працездатні.

Як і будь-яка технологія тестування, модульне тестування не дозволяє відловити всі помилки програми. Справді, це випливає з практичної неможливості трасування всіх можливих шляхів виконання програми, за винятком найпростіших випадків. Крім того, відбувається тестування кожного з модулів окремо. Це означає, що помилки інтеграції, системного рівня, функцій, виконуваних в декількох модулях не будуть визначені. Крім того, дана технологія марна для проведення тестів на продуктивність. Таким чином, модульне тестування більш ефективно при використанні в поєднанні з іншими методиками тестування.

Для отримання вигоди від модульного тестування потрібно строго слідувати технології тестування у всьому процесі розробки програмного забезпечення. Потрібно зберігати не тільки записи про всі проведених тестах, але і про всі зміни вихідного коду у всіх модулях. З цією метою слід використовувати систему контролю версій ПЗ. Таким чином, якщо пізніша версія ПЗ не проходить тест, який був успішно пройдений раніше, буде нескладним звірити вихідний код варіантів і усунути помилку. Також необхідно переконатися в незмінному відстеженні і аналізі невдалих тестів. Ігнорування цієї вимоги призведе до лавиноподібного збільшення невдалих тестових результатів.

6.2.1 Бібліотеки модульного тестування в .NET

На даний момент найбільш поширеними інструментами модульного тестування платформи. NET є бібліотека NUnit і Microsoft Unit Testing Framework.

NUnit - відкрите середовище unit-тестування програм для. NET. Вона була перенесена з мови Java (бібліотека JUnit). Перші версії NUnit були написані на J #, але потім весь код був переписаний на C # з використанням таких нововведень. NET, як атрибути.

Існують також відомі розширення оригінального пакета NUnit, велика частина з них також з відкритим вихідним кодом. NUnit.Forms доповнює NUnit засобами тестування елементів користувацького інтерфейсу Windows Forms. NUnit.ASP виконує ту ж задачу для елементів інтерфейсу в ASP.NET.

Приклад модульного тесту з використанням бібліотеки NUnit наведено нижче:

```
using NUnit.Framework;

[TestFixture]
public class ExampleTestOfNUnit
{
    [Test]
    public void TestMultiplication()
    {
        Assert.AreEqual(4, 2 * 2, "Множення");
    }
}
```

Як видно з прикладу модульного тесту NUnit, тестовий клас позначений атрибутом [TestFixture] який вказує, що в даному класі будуть міститися тестові методи. Тестові методи позначаються атрибутом [Test], а в середині тестового методу використовується статичний клас Assert з набором спеціальних методів, для порівняння результатів, що повертаються тестовим методом з необхідними результатами.

6.2.2 Unit Testing Framework від Microsoft

Visual Studio Unit Testing Framework - інструмент модульного тестування для платформи. NET, вбудований в середовище розробки Visual Studio. Щоб визначити, що клас є тестовим, необхідно позначити його атрибутом [TestClass]. Якщо клас позначений цим атрибутом, то він може містити в собі тестові методи. Зазвичай тестовий клас називають так само, як і той, що тестується, тільки з префіксом Test.

У тестовому класі можуть міститися тестові методи і зазвичай для всіх методів класу, що тестується, які повертають значення, створюється окремий тестовий метод. Тестовий метод зазвичай називають, так само як і метод, що тестується, тільки з префіксом Test.

Крім тестових методів у тестовому класі можуть бути методи ініціалізації та очищення. Метод ініціалізації позначається атрибутом [TestInitialize] і дозволяє провести ініціалізацію необхідних змінних перед виконанням тестового методу. Метод очищення позначається атрибутом [TestCleanup] і дозволяє очистити результати виконання тесту, наприклад, очистити файл, видалити зайві записи з бази даних, привласнити змінним значення за замовчуванням.

Крім методів ініціалізації та очищення на рівні тесту, в тестуючому класі можуть бути присутніми методи ініціалізації та очищення рівня класу. Ці методи викликаються один раз. Методи ініціалізації рівня класу викликається один раз перед викликом першого тесту, а метод очищення рівня класу викликається після виконання останнього тесту. Метод ініціалізації рівня класу позначається атрибутом [ClassInitialize], а метод очищення рівня класу позначається атрибутом [ClassCleanup]. Нижче наведено приклад тестового класу.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Test
{
  [TestClass]
  public class TestClass
  {
    public TestClass(){}

    //Тестовий контекст
    private TestContext testContextInstance;

    public TestContext getTestContext(){
      return testContextInstance;
    }

    public void setTestContext(TestContext testContext){
      this.testContextInstance = testContextInstance;
    }

    [ClassInitialize]
    public static void TestClassInitialize(TestContext testContext) {
      /*Код буде виконуватися до виклику першого тесту*/
    }

    [ClassCleanup]
    public static void TestClassCleanup(){
      /*Код буде виконуватися після виклику останнього тесту*/
    }

    [TestInitialize]
    public void TestInitialize(){
      /*Код буде виконуватися перед виконанням кожного тесту*/
    }

    [TestCleanup]
    public void TestCleanup(){
      /*Код буде виконуватися після виконання кожного тесту*/
    }

    [TestMethod]
    public void TestMethod(){
      /*Код тесту*/
    }
  }
}
```

У тестовому класі для перевірки значень, що повертаються методи, що тестуються, використовуються статичні методи класу Assert.

Список всіх методів класу Assert наведено в таблиці 6.1.

Таблиця 6.1. Методи класу Assert для тестування

Назва методу	Опис
Assert.AreEqual	Перевіряє, чи рівні задані значення. Помилка виникає, якщо значення нерівні.
Assert.AreNotEqual	Перевіряє, щоб задані значення не були рівні. Помилка виникає, якщо значення рівні.
Assert.AreSame	Перевіряє, чи вказують посилання на один і той же об'єкт. Помилка виникає, якщо посилання вказують на різні об'єкти.
Assert.AreNotSame	Перевіряє, щоб посилання не вказували на один і той же об'єкт. Помилка виникає, якщо посилання вказують на один і той же об'єкт.
Assert.Fail	Помилка виникає без будь-яких перевірок.
Assert.IsTrue	Перевіряє, чи є передане значення true. Помилка виникає якщо значення false.
Assert.IsFalse	Перевіряє, чи є передане значення false. Помилка виникає якщо значення є true.
Assert.IsNull	Перевіряє, чи є передане значення null. Помилка виникає, якщо значення не null.
Assert.IsNotNull	Перевіряє, щоб передане значення не було null. Помилка виникає, якщо передане значення null.
Assert.IsInstanceOfType	Перевіряє, чи є переданий об'єкт екземпляром заданого класу. Помилка виникає, якщо значення не є екземпляром заданого класу.
Assert.IsNotInstanceOfType	Перевіряє, щоб переданий об'єкт не був екземпляром заданого класу. Помилка виникає, якщо об'єкт є екземпляром заданого класу.

6.3 Порядок виконання лабораторної роботи

1. Створіть в директорії з проектами (в якості прикладу у всіх лабораторних роботах використовується директорія D:\csexamples, але директорія з проектами може знаходитись в будь-якому зручному місці) директорію для нового проекту з назвою integrationtest (або з будь-якою іншою назвою). Відкрийте Visual Studio Code і змініть директорію проекту на D:\csexamples\integrationtest (File->Open Folder). Відкрийте термінал в Visual Studio Code, якщо він не з’явився автоматично (Terminal->New Terminal).

2. Зараз необхідно створити нову базу даних в postgres з назвою universitytest. В терміналі введіть команду:

```
psql --username=postgres --password=Welcome1
```

Ім’я користувача і пароль для postgresql залежить від налаштувань СУБД і може бути іншим.

В командному рядку postgres введіть команду:

```
create database universitytest;
```

3. В терміналі Visual Studio Code введіть команду exit для того, щоб вийти з postgres.

4. Створіть в директорії проекту дві нові директорії з іменами orm і orm.test. В директорії orm буде міститися проект з класами DAO, що були створені в лабораторній роботі №3, а в директорії orm.test буде міститися проект з тестовими класами.

5. В терміналі Visual Studio Code введіть команду для створення нового проекту в директорії D:\csexamples\integrationtest\orm. В терміналі поточна директорі буде D:\csexamples\integrationtest тому перед виконанням команди створення нового проекту необхідно перейти в директорію D:\csexamples\integrationtest\orm.

```
dotnet new classlib
```

Видаліть клас Class1.cs, який створився за замовчуванням в директорії D:\csexamples\integrationtest\orm після виконання команди створення нового проекту.

6. В даній лабораторній роботі будуть використовуватися додаткові бібліотеки для роботи з технологією Fluent Nhibernate, які необхідно додати до проекту шляхом виконання наступних команд:

```
dotnet add package fluentnhibernate
dotnet add package npgsql
```

7. Створіть в директорії проекту D:\csexamples\integrationtest\orm директорію Domain. Створіть в директорії Domain класи EntityBase.cs, Student.cs та StudentMap.cs:

```
//Базовий доменний клас
namespace restapi.Domain
{
    public abstract class  EntityBase
    {
        public virtual long Id { get; set; }
    }
}

//Доменний клас студента
namespace restapi.Domain
{
    public class Student : EntityBase
    {
        public virtual string? FirstName { get; set; }

        public virtual string? LastName { get; set; }

        public virtual char Sex { get; set; }

        public virtual int Year { get; set; }

    }
}

//Клас відображення студента
using FluentNHibernate.Mapping;

namespace restapi.Domain
```

```
{
    public class StudentMap : ClassMap<Student>
    {
        public StudentMap()
        {
            Table("Students");
            Id(x => x.Id).GeneratedBy.Native();
            Map(x => x.FirstName);
            Map(x => x.LastName);
            Map(x => x.Sex);
            Map(x => x.Year);
        }
    }
}
```

8. В директорії проекту D:\csexamples\integrationtest\orm директорію Dao. Додайте до директорії Dao інтерфейси і класи шаблону DAO IGenericDAO.cs, IGenericDAO.cs, IStudentDAO.cs, StudentDAO.cs, DAOFactory.cs, NHibernateDAOFactory.cs:

```
//Інтерфейс IGenericDAO
namespace orm.Dao
{
    public interface IGenericDAO<T>
    {
        T Merge(T item);

        T GetById(long id);

        List<T> GetAll();

        void Delete(T item);

        T GetPersistentObject(T nonPersistentObject);

    }
}

//Клас GenericDAO
using NHibernate;
using NHibernate.Criterion;
using ISession = NHibernate.ISession;

namespace orm.Dao
{
    public class GenericDAO<T> : IGenericDAO<T>
    {
        protected ISession session;

        public GenericDAO() { }

        public GenericDAO(ISession session)
        {
            this.session = session;
        }

        public T Merge(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            T resultItem = (T)session.Merge(item);
            transaction.Commit();
            return resultItem;
```

```
        }

        public T GetById(long id)
        {
            return session.Get<T>(id);
        }

        public List<T> GetAll()
        {
            return new List<T>(session
                .CreateCriteria(typeof(T)).List<T>());
        }

        public void Delete(T item)
        {
            ITransaction transaction = session.BeginTransaction();
            session.Delete(item);
            transaction.Commit();
        }

        public T GetPersistentObject(T nonPersistentObject)
        {
            ICriteria criteria = session.CreateCriteria(typeof(T))
                .Add(Example.Create(nonPersistentObject));
            IList<T> list = criteria.List<T>();
            return list[0];
        }
    }
}

//Інтерфейс IStudentDAO
using orm.Domain;

namespace orm.Dao
{
    public interface IStudentDAO:IGenericDAO<Student>
    {

    }
}

//Клас StudentDAO
using ISession = NHibernate.ISession;
using orm.Domain;

namespace orm.Dao
{
    public class StudentDAO : GenericDAO<Student>, IStudentDAO
    {

        public StudentDAO(ISession session) : base(session) { }

    }
}

//Клас DAOFactory
namespace orm.Dao
{
    abstract public class DAOFactory
    {
        public abstract IStudentDAO getStudentDAO();

        public abstract void destroy();
```

```
    }
}

//Клас NHibernateDAOFactory
using System.Reflection;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;
using ISession = NHibernate.ISession;

namespace orm.Dao
{
    public class NHibernateDAOFactory : DAOFactory
    {
        private static ISessionFactory factory;

        private static DAOFactory instance;

        private ISession session;

        private IStudentDAO studentDAO;

        public static DAOFactory getInstance()
        {
            if (null == instance)
            {
                ISession session = openSession("127.0.0.1",
                    Convert.ToInt32("5432"), "universitytest",
                    "postgres", "Welcome1");
                instance = new NHibernateDAOFactory(session);
            }
            return instance;
        }

        public NHibernateDAOFactory(ISession session)
        {
            this.session = session;
        }

        public override IStudentDAO getStudentDAO()
        {
            if (null == studentDAO)
            {
                studentDAO = new StudentDAO(session);
            }
            return studentDAO;
        }

        public override void destroy()
        {
            session.Close();
        }

        private static ISession openSession(String host, int port,
            String database, String user, String password)
        {
            ISession session = null;
            Assembly mappingsAssemly = Assembly.GetExecutingAssembly();
            if (null == factory)
            {
                factory = Fluently.Configure()
                   .Database(PostgreSQLConfiguration
```

```
                    .PostgreSQL82.ConnectionString(c => c
                    .Host(host)
                    .Port(port)
                    .Database(database)
                    .Username(user)
                    .Password(password)))
                    .Mappings(m => m.FluentMappings
                    .AddFromAssembly(mappingsAssemly))
                    .ExposeConfiguration(BuildSchema)
                    .BuildSessionFactory();
            }
            session = factory.OpenSession();
            return session;
        }

        private static void BuildSchema(Configuration config)
        {
            new SchemaExport(config).Create(true, true);
        }
    }
}
```

9. В терміналі Visual Studio Code перейдіть в директорію D:\csexamples\integrationtest\orm.test і введіть команду для створення нового проекту MS Unit Testing Framework:

```
dotnet new mstest
```

Видаліть класи UnitTest1.cs і Usings.cs, що створилися за замовчуванням.

10. Додайте в тестовий проект посилання на проект, який необхідно протестувати (команду необхідно виконувати с директорії D:\csexamples\integrationtest\orm.test):

```
dotnet add reference ../orm/orm.csproj
```

11. Створіть в корені тестового проекту (D:\csexamples\integrationtest\orm.test) директорію Dao. Додайте до директорії Dao клас TestGenericDAO:

```
using orm.Domain;
using orm.Dao;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace orm.test.Dao
{
    [TestClass()]
    public abstract class TestGenericDAO<T> where T : EntityBase
    {
        protected TestContext testContextInstance;
        /** DAO that will be tested */
        protected IGenericDAO<T> dao = null;

        /** First entity that will be used in tests */
        protected T entity1 = null;
        /** Second entity that will be used in tests */
        protected T entity2 = null;
        /** Third entity that will be used in tests */
        protected T entity3 = null;

        protected T entity4 = null;
```

```
public TestGenericDAO()
{
}

public TestContext TestContext
{
    get
    {
        return testContextInstance;
    }
    set
    {
        testContextInstance = value;
    }
}

/*Getting dao this test case works with*/
public IGenericDAO<T> getDAO()
{
    return dao;
}

/*Setting dao this test case will work with*/
public void setDAO(IGenericDAO<T> dao)
{
    this.dao = dao;
}

[ClassCleanup]
public static void ClassCleanup()
{
    NHibernateDAOFactory.getInstance().destroy();
}

[TestInitialize]
public void TestInitialize()
{
    Assert.IsNotNull(dao,
        "Please, provide IGenericDAO implementation in constructor");
    createEntities();
    Assert.IsNotNull(entity1, "Please, create object for entity1");
    Assert.IsNotNull(entity2, "Please, create object for entity2");
    Assert.IsNotNull(entity3, "Please, create object for entity3");
    checkAllPropertiesDiffer(entity1, entity2);
    checkAllPropertiesDiffer(entity1, entity3);
    checkAllPropertiesDiffer(entity2, entity3);
    saveEntitiesGeneric();
}

[TestCleanup]
public void TestCleanup()
{
    try
    {
        if ((entity1 = dao.GetById(entity1.Id)) != null)
        {
            dao.Delete(entity1);
        }
    }
    catch (Exception)
    {
        Assert.Fail("Problem in cleanup method");
    }
```

```
        try
        {
            if ((entity2 = dao.GetById(entity2.Id)) != null)
                dao.Delete(entity2);
        }
        catch (Exception)
        {
            Assert.Fail("Problem in cleanup method");
        }
        try
        {
            if ((entity3 = dao.GetById(entity3.Id)) != null)
            {
                dao.Delete(entity3);

            }
        }
        catch (Exception)
        {
            Assert.Fail("Problem in cleanup method");
        }
        try
        {
            if ((entity4 = dao.GetById(entity4.Id)) != null)
            {
                dao.Delete(entity4);

            }
        }
        catch (Exception)
        {
            Assert.Fail("Problem in cleanup method");
        }
        entity1 = null;
        entity2 = null;
        entity3 = null;
        entity4 = null;
    }

    public void TestMergeGeneric()
    {
        dao.Merge(entity4);
        T savedObject = getPersistentObject(entity3);
        Assert.IsNotNull(savedObject,
            "Dao method Merge should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity3);
    }

    public void TestGetByIdGeneric()
    {
        T foundObject = null;
        // Should not find with inexistant id
        try
        {
            long id = DateTime.Now.ToFileTime();
            foundObject = dao.GetById(id);
            Assert.IsNull(foundObject,
                "Should return null if id is inexistent");
        }
        catch (Exception)
        {
            Assert.Fail("Should return null if object not found");
        }
        // Getting all three entities
```

```
    getEntityGeneric(entity1.Id, entity1);
    getEntityGeneric(entity2.Id, entity2);
    getEntityGeneric(entity3.Id, entity3);
}

public void TestGetAllGeneric()
{
    List<T> list = getListOfAllEntities();
    Assert.IsTrue(list.Contains(entity1),
        "After dao method GetAll list should contain entity1");
    Assert.IsTrue(list.Contains(entity2),
        "After dao method GetAll list should contain entity2");
    Assert.IsTrue(list.Contains(entity3),
        "After dao method GetAll list should contain entity3");
}

public void TestDeleteGeneric()
{
    try
    {
        dao.Delete((T)null);
        Assert.Fail("Should not delete entity will null id");
    }
    catch (Exception)
    {
    }
    // Deleting second entity
    try
    {
        dao.Delete(entity2);
    }
    catch (Exception)
    {
        Assert.Fail("Deletion should be successful of entity2");
    }

    // Checking if other two entities can be still found
    getEntityGeneric(entity1.Id, entity1);
    getEntityGeneric(entity3.Id, entity3);

    // Checking if entity2 can not be found
    try
    {
        T foundEntity = null;
        foundEntity = dao.GetById(entity2.Id);
        Assert.IsNull(foundEntity,
            "After deletion entity should not be found with id "
                + entity2.Id);
    }
    catch (Exception)
    {
        Assert.Fail("Should return null if entity was deleted");
    }

    List<T> list = getListOfAllEntities();
    Assert.IsTrue(list.Contains(entity1),
        "After dao method GetAll list should contain entity1");
    Assert.IsTrue(list.Contains(entity3),
        "After dao method GetAll list should contain entity3");
}

protected abstract void createEntities();

protected abstract void checkAllPropertiesDiffer(
```

```
    T entityToCheck1, T entityToCheck2);

protected abstract void checkAllPropertiesEqual(
    T entityToCheck1, T entityToCheck2);

protected void saveEntitiesGeneric()
{
    T savedObject = null;
    try
    {
        dao.Merge(entity1);
        savedObject = getPersistentObject(entity1);
        Assert.IsNotNull(savedObject,
            "Dao method Merge should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity1);
        entity1 = savedObject;
    }
    catch (Exception)
    {
        Assert.Fail("Fail to save entity1");
    }
    try
    {
        dao.Merge(entity2);
        savedObject = getPersistentObject(entity2);
        Assert.IsNotNull(savedObject,
            "Dao method Merge should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity2);
        entity2 = savedObject;
    }
    catch (Exception)
    {
        Assert.Fail("Fail to save entity2");
    }

    try
    {
        dao.Merge(entity3);
        savedObject = getPersistentObject(entity3);
        Assert.IsNotNull(savedObject,
            "Dao method Merge should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity3);
        entity3 = savedObject;
    }
    catch (Exception)
    {
        Assert.Fail("Fail to save entity3");
    }
}

protected T getPersistentObject(T nonPersistentObject)
{
    return dao.GetPersistentObject(nonPersistentObject);
}

protected void getEntityGeneric(long id, T entity)
{
    T foundEntity = null;
    try
    {
        foundEntity = dao.GetById(id);
        Assert.IsNotNull(foundEntity,
            "Service method should return entity if successfull");
        checkAllPropertiesEqual(foundEntity, entity);
```

```
            }
            catch (Exception)
            {
                Assert.Fail("Failed to get entity with id " + id);
            }
        }

        protected List<T> getListOfAllEntities()
        {
            List<T> list = null;
            // Should get not null and not empty list
            try
            {
                list = dao.GetAll();
            }
            catch (Exception)
            {
                Assert.Fail(
                        "Should be able to get entities were added before");
            }
            Assert.IsNotNull(list,
                "Dao method should return list of entities if successfull");
            Assert.IsFalse(list.Count == 0,
                "Dao method should return not empty list if successfull");
            return list;
        }
    }

}
```

12. Додайте до директорії Dao (D:\csexamples\integrationtest\orm.test) клас TestStudentDAO:

```
using orm.Dao;
using orm.Domain;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace orm.test.Dao
{
    [TestClass]
    public class TestStudentDAO : TestGenericDAO<Student>
    {
        protected IStudentDAO studentDAO = null;

        public TestStudentDAO() : base()
        {
            IStudentDAO studentDAO =
                NHibernateDAOFactory.getInstance().getStudentDAO();
            setDAO(studentDAO);
        }

        protected override void createEntities()
        {
            entity1 = new Student();
            entity1.FirstName = "Elon";
            entity1.LastName = "Mask";
            entity1.Sex = 'M';
            entity1.Year = 1971;

            entity2 = new Student();
            entity2.FirstName = "Jeff";
            entity2.LastName = "Bezos";
```

```
            entity2.Sex = 'M';
            entity2.Year = 1964;

            entity3 = new Student();
            entity3.FirstName = "Bill";
            entity3.LastName = "Gates";
            entity3.Sex = 'M';
            entity3.Year = 1955;

            entity4 = new Student();
            entity4.FirstName = "Mark";
            entity4.LastName = "Zuckerberg";
            entity4.Sex = 'M';
            entity4.Year = 1984;
        }

        protected override void checkAllPropertiesDiffer(
            Student entityToCheck1, Student entityToCheck2)
        {
            Assert.AreNotEqual(entityToCheck1.FirstName,
                entityToCheck2.FirstName, "Values must be different");
            Assert.AreNotEqual(entityToCheck1.LastName,
                entityToCheck2.LastName, "Values must be different");
            Assert.AreNotEqual(entityToCheck1.Year,
                entityToCheck2.Year, "Values must be different");
        }

        protected override void checkAllPropertiesEqual(
            Student entityToCheck1, Student entityToCheck2)
        {
            Assert.AreEqual(entityToCheck1.FirstName,
                entityToCheck2.FirstName, "Values must be equal");
            Assert.AreEqual(entityToCheck1.LastName,
                entityToCheck2.LastName, "Values must be equal");
            Assert.AreEqual(entityToCheck1.Sex,
                entityToCheck2.Sex, "Values must be equal");
            Assert.AreEqual(entityToCheck1.Year,
                entityToCheck2.Year, "Values must be equal");
        }

        [TestMethod]
        public void TestMergeStudent()
        {
            base.TestMergeGeneric();
        }

        [TestMethod]
        public void TestGetByIdStudent()
        {
            base.TestGetByIdGeneric();
        }

        [TestMethod]
        public void TestGetAllStudent()
        {
            base.TestGetAllGeneric();
        }

        [TestMethod]
        public void TestDeleteStudent()
        {
            base.TestDeleteGeneric();
        }
    }
```

```
}
```

13. Далі необхідно створити рішення (solution), яке необхідне для правильної взаємодії тестового проекту з проектом, що тестується. В терміналі Visual Studio Code перейдіть в директорію D:\csexamples\integrationtest і виконайте наступну команду:

```
dotnet new sln
```

14. Додайте до рішення проекти orm і orm.test:

```
dotnet sln add ./orm/orm.csproj
dotnet sln add ./orm.test/orm.test.csproj
```

15. Для виконання тестів в Visual Studio Code додайте плагін .NET Core Test Explorer, що зображений на рисунку 6.1.

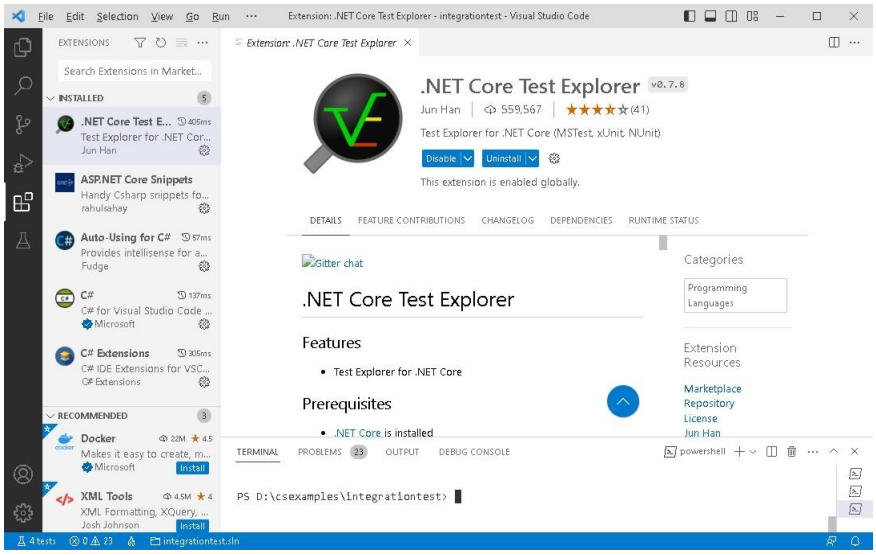

Рисунок 6.1 - Плагін .NET Core Test Explorer

16. В налаштуваннях плагіна .NET Core Test Explorer (Settings -> Extensions -> .NET Core Test Explorer) встановіть значення ./orm.test в поле Test Project Path, як показано на рисунку 6.2.

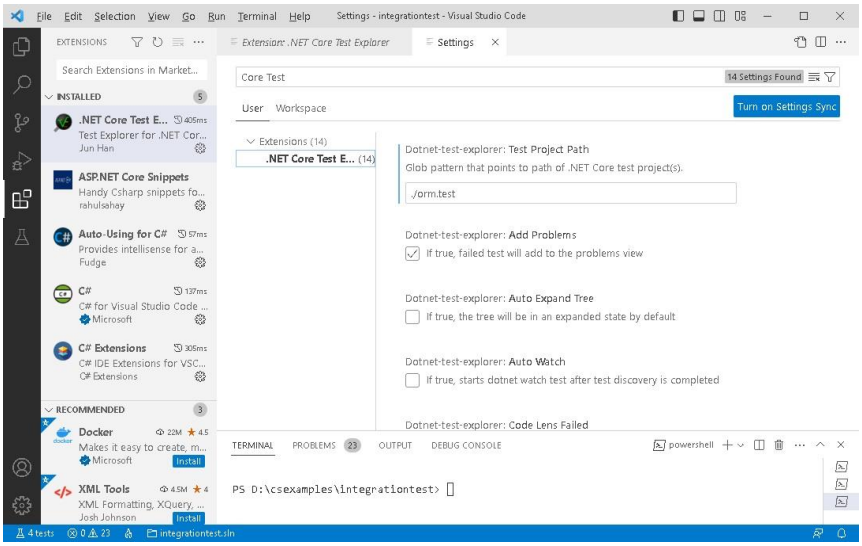

Рисунок 6.2 – Конфігурація плагіна .NET Core Test Explorer

17. Перейдіть на вкладку Testing і натисніть кнопку Refresh. В списку повинні з’явитися тести з класу TestStudentDAO. Натисніть кнопку Run All Tests. Результат виконання тестів показаний на рисунку 6.3.

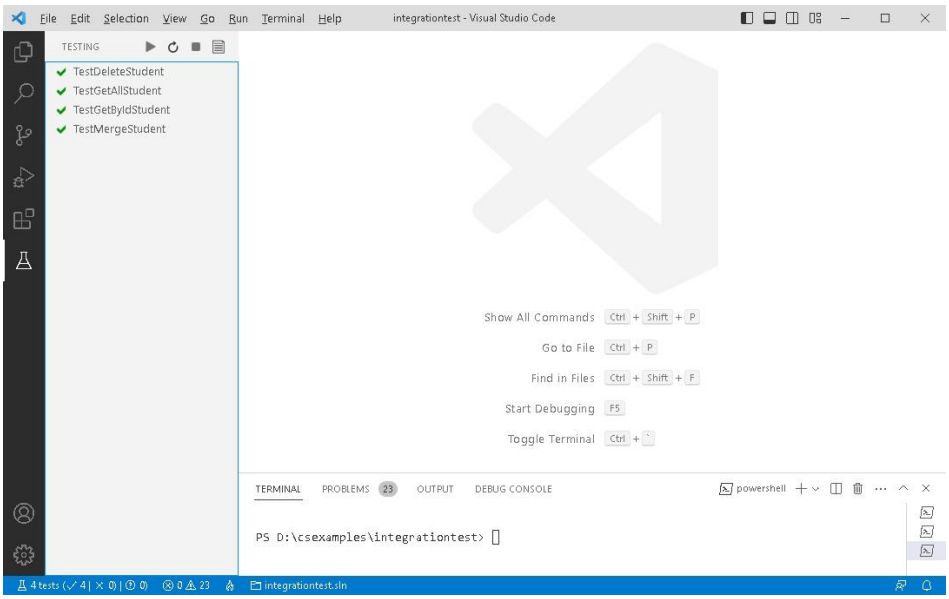

Рисунок 6.3 – Результат виконання тестів

6.4 Завдання до лабораторної роботи

Створити набір тестових класів, описаний в ході виконання лабораторної роботи. Замінити об’єкт предметної області «Студент» на об’єкт предметної області відповідно до варіанту з таблиці 6.2 та внести необхідні зміни в проект. Номер варіанта визначається за останньою цифрою залікової книжки.

Таблиця 6.2 – Варіанти завдань

Номер варіанта	Предметна область	Об’єкт предметної області
0	Аеропорт	Пасажир
1	Піцерія	Замовлення
2	Крамниця	Товар
3	Супермаркет	Постачальник
4	Відділ поліції	Порушник
5	Лікарня	Пацієнт
6	Кінотеатр	Сеанс
7	Корабель	Пасажир
8	Підприємство	Співробітник
9	Бібліотека	Книга

6.5 Що повинно бути у звіті

- прізвище та ім’я виконавця лабораторної роботи;
- номер і назва лабораторної роботи;
- мета лабораторної роботи;

- короткі теоретичні відомості на одну сторінку;
- хід виконання (лістинг програми, знімки екранів програми);
- висновки про виконану роботу.

6.6 Контрольні питання

1. Що таке модульне тестування?
2. Що таке інтеграційне тестування?
3. Які бібліотеки модульного тестування ви знаєте?
4. Яка структура модульного тесту в MS Testing Framework?
5. Яка структура модульного тесту в NUnit?
6. Чим модульний тест NUnit відрізняється від модульного тесту MS Testing Framework?
7. Які методи класу Assert MS Testing Framework ви знаєте?
8. Які атрибути використовуються в MS Unit Testing Framework для визначення тестового класу, тестового методу, методів ініціалізації і очистки класу і методів тестування?
9. Які атрибути використовуються в NUnit для визначення тестового класу, тестового методу, методів ініціалізації і очистки класу і методів тестування?
10. Які атрибути використовуються в xUnit для визначення тестового класу, тестового методу, методів ініціалізації і очистки класу і методів тестування?

6.7 Завдання до захисту лабораторної роботи

1. Написати тест з використанням MS Unit Testing Framework, що тестує метод, який повертає TOP 10 об’єктів (відповідно до варіанту завдання), з використанням запитів Native SQL.
2. Написати тест з використанням MS Unit Testing Framework, що тестує метод, який повертає TOP 10 об’єктів (відповідно до варіанту завдання), з використанням запитів по критерію.
3. Написати тест з використанням MS Unit Testing Framework, що тестує метод, який повертає TOP 10 об’єктів (відповідно до варіанту завдання), з використанням запитів за прикладом.
4. Написати тест з використанням MS Unit Testing Framework, що тестує метод, який повертає TOP 10 об’єктів (відповідно до варіанту завдання), з використанням запитів HQL.
5. Написати тест з використанням MS Unit Testing Framework, що тестує метод пошук доменних об’єктів (відповідно до варіанту завдання) по будь якому полю чи частині поля. Для пошуку використати мову запитів HQL.
6. Написати тест з використанням MS Unit Testing Framework, що тестує метод пошук доменних об’єктів (відповідно до варіанту завдання) по будь якому полю чи частині поля. Для пошуку використати запити по критерію.
7. Написати тест з використанням MS Unit Testing Framework, що

тестує метод пошук доменних об’єктів (відповідно до варіанту завдання) по будь якому полю чи частині поля. Для пошуку використати запити за прикладом.

8. Написати тест з використанням MS Unit Testing Framework, що тестує метод пошук доменних об’єктів (відповідно до варіанту завдання) по будь якому полю чи частині поля. Для пошуку використати запити Native SQL.

9. Написати тест з використанням NUnit, що тестує метод, який повертає TOP 10 об’єктів (відповідно до варіанту завдання), з використанням запитів Native SQL.

10. Написати тест з використанням Selenium, який здійснює тестування веб-застосунку, створеного в лабораторній роботі №4.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Dustin Metzgar .NET Core in Action. Manning, 2018.- 288 p.
2. Pierre Henri Kuaté, Christian Bauer, Gavin King, Tobin Harris. NHibernate in Action 1st Edition. Manning, 2009.- 400 p.
3. Neha Shrivastava, Rishabh Verma .NET Core 2.0 By Example. Packt Publishing, 2018.- 458 p.
4. Andrew Lock. ASP.NET Core in Action, 2nd ed. Manning, 2021.- 832 p.
5. Samuele Resca. Hands-On RESTful Web Services with ASP.NET Core 3: Design production-ready, testable, and flexible RESTful APIs for web applications and microservices. Packt Publishing, 2019.- 510 p.
6. Christian Horsdal Gammelgaard. Microservices in .NET, Second Edition 2nd Edition. Manning, 2021.- 328 p.
7. Ayobami Adewole. C# and .NET Core Test Driven Development: Dive into TDD to create flexible, maintainable, and production-ready .NET Core applications. Packt Publishing, 2018 .- 300 p.
8. Robert C Martin. Clean Code : A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.- 464 p.
9. Hohpe Gregor, Woolf Bobby. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, 2012.- 741 p.
10. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch. Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition. Addison-Wesley Professional, 1994.- 416 p.
11. Система дистанційного навчання «moodle» [Електронний ресурс]. – Режим доступу : http://eln.stu.cn.ua/course/... Java та C# технології прикладного програмування.