МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

***Циклова комісія програмування***

**Робочий план**

з курсу «Об’єктно-орієнтоване програмування»

Практична робота №15

Солом’яного Ярослава Сергійовича

*ПІБ студента*

студента групи **2П-18**

|  |  |  |
| --- | --- | --- |
| **Вид роботи** | **Дата** | **Оцінка** |
| *Пром. контроль* |  |  |
| *Залік* |  |  |

Викладач Марченко С. В.

Черкаси-2021

Завдання 1. Сформуйте звіт з детального конспекту та демонстрації роботи програмних проєктів з керівництва по Entity Framework Core.

EntityFramework Core являє собою об’єктно-орієнтовану технологію для роботи з базами даних.EFC являється ORM інструментом. Також EntityFramework має універсальне API для роботи з даними. Тому використання цього фреймворка є досить розповсюдженим, ця технологія використовується у крос-платформленій розробці і тому може підходити як для Windows Forms так і для ASP.NET CORE. Технологія являє собою поняття сутності, яка вимірює певний набір даних, яка поєднана з деяким об’єктом.

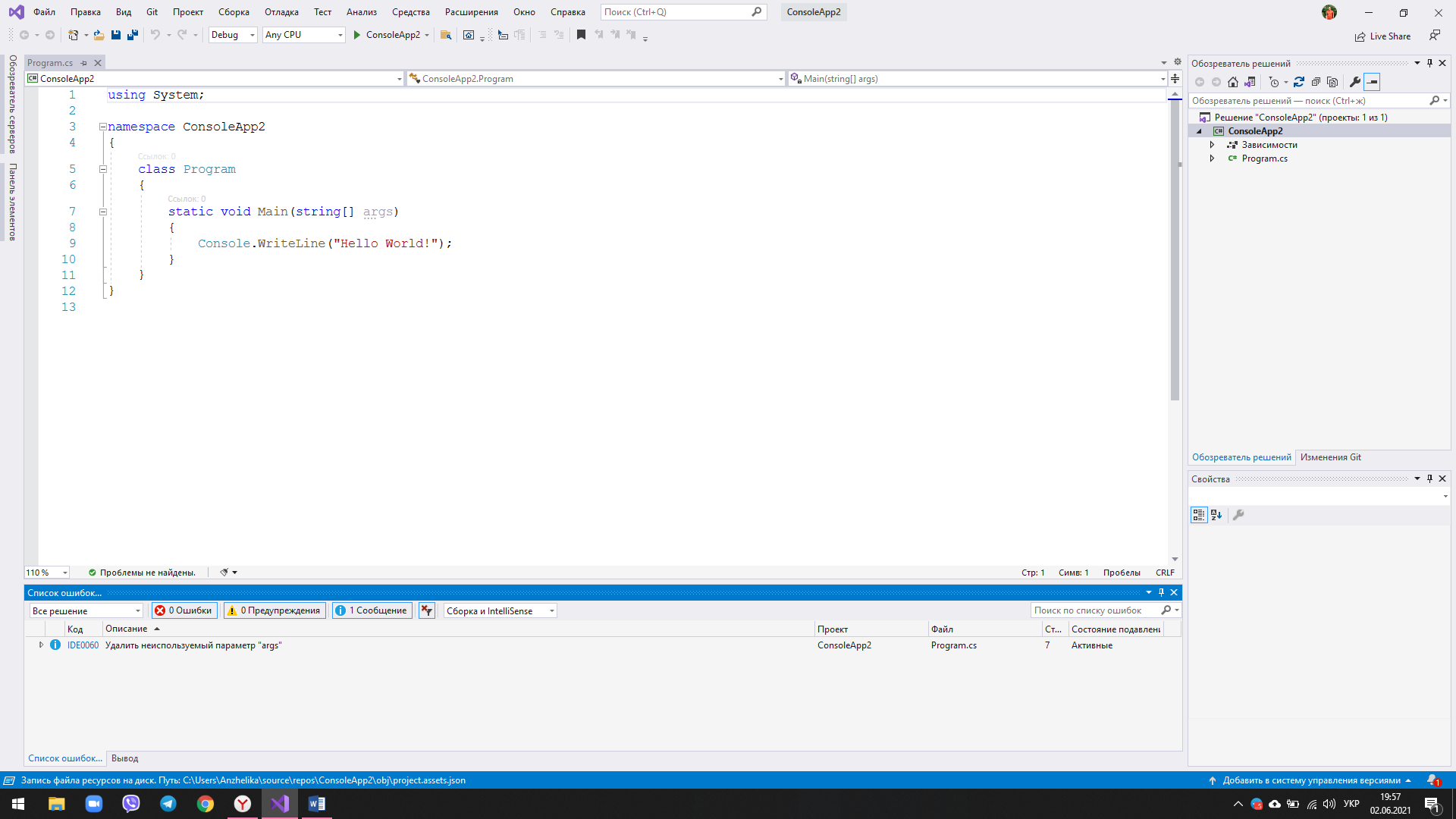


Рис 1. Створюємо консольний додаток

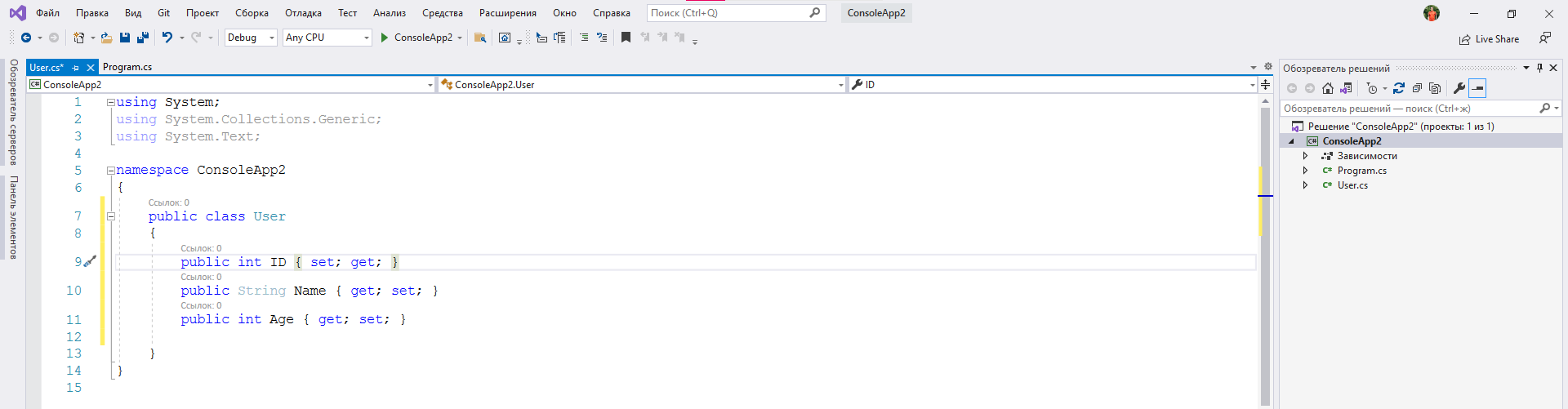


Рис 2. Створимо клас User.

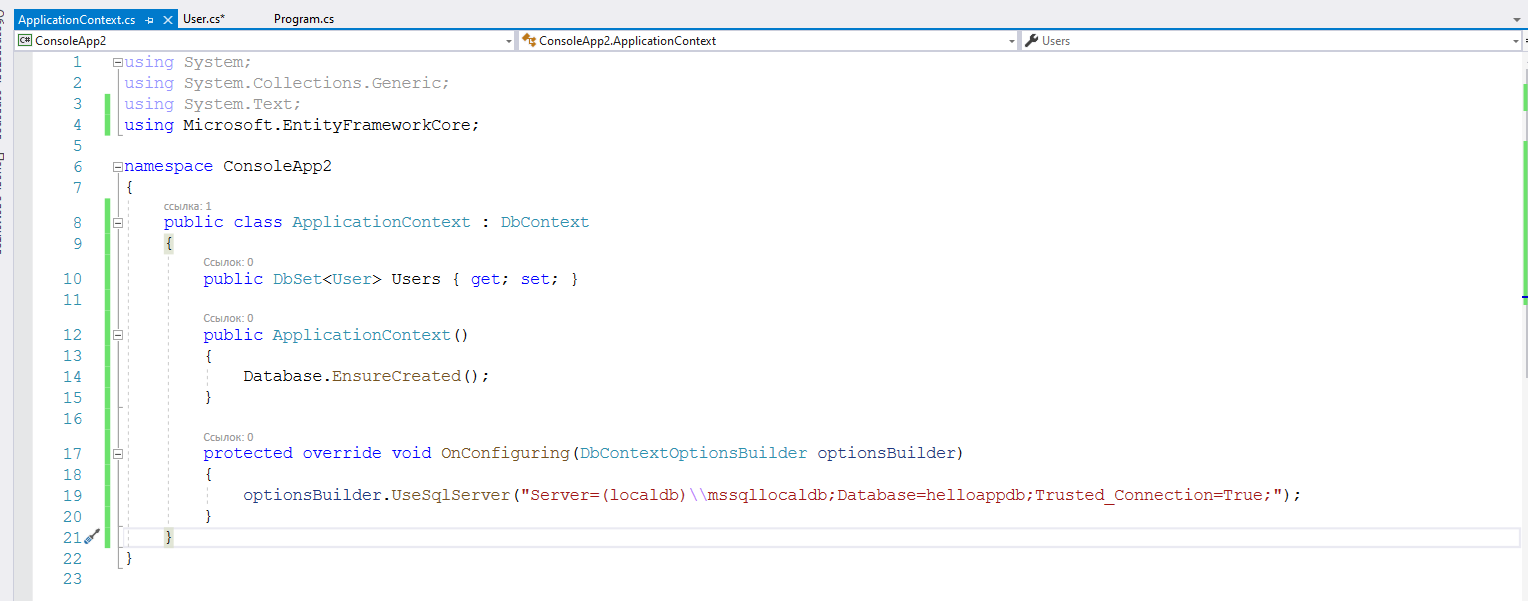


Рис 3. Створений клас ApplicationContext

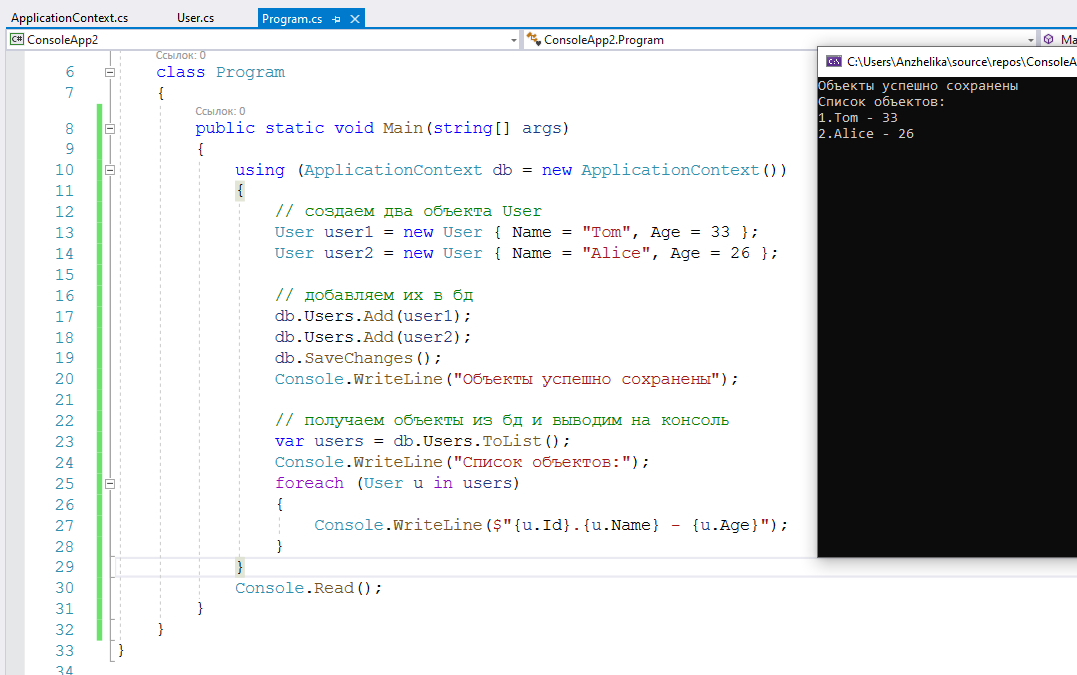


Рис 4. Створили об’єкти класу user та додали їх в нашу створену базу.

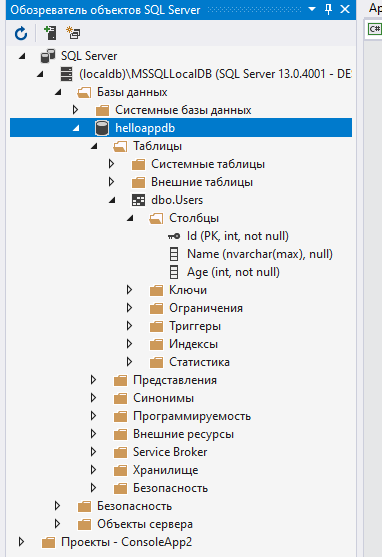


Рис 5. Обзор нашої бази даних в обзорі об’єктів SQL Server

Тепер створимо новий проект та добавимо в нього існуючу базу даних.

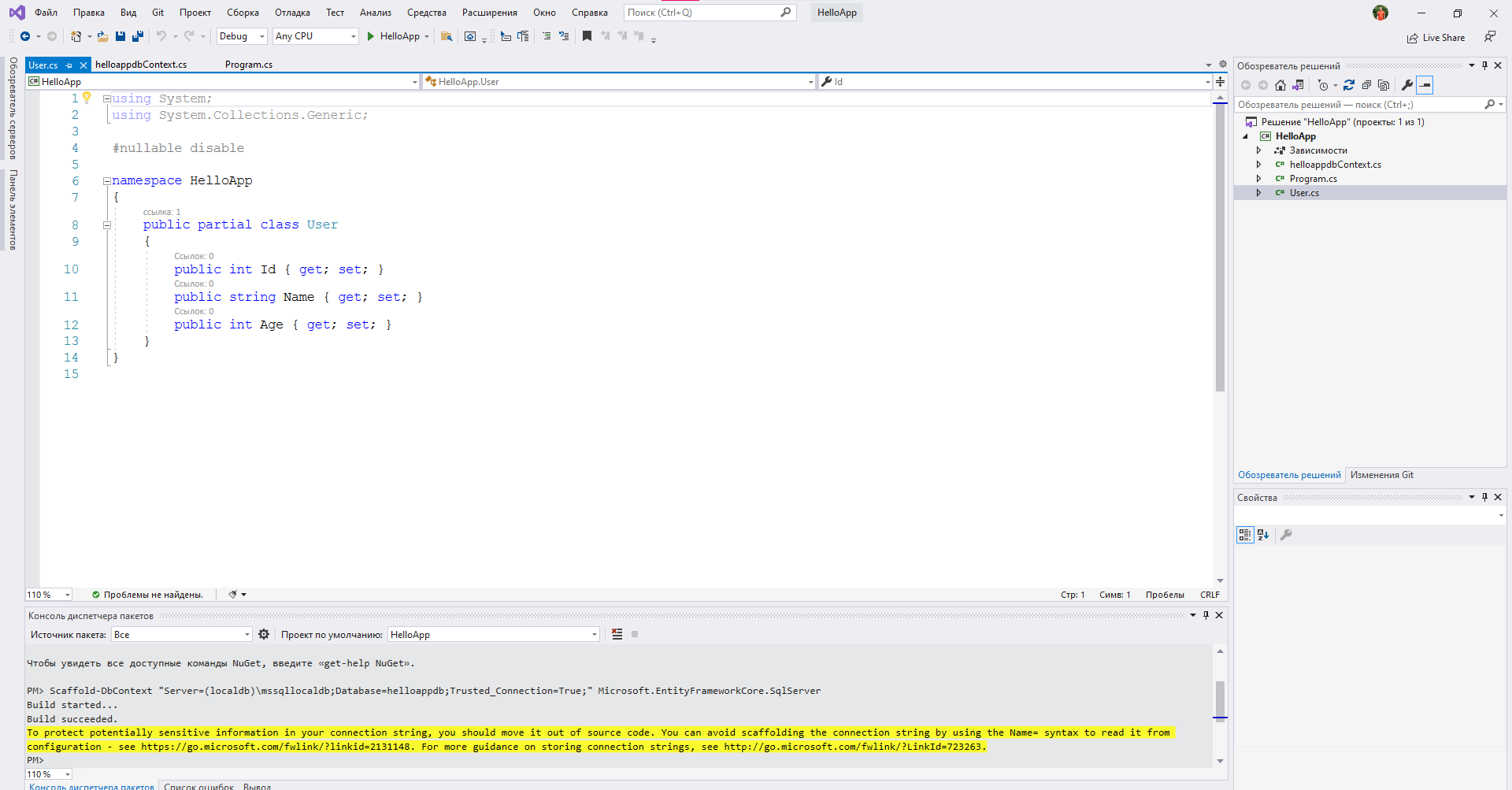


Рис 6. Добавили в проект клас з бази даних User

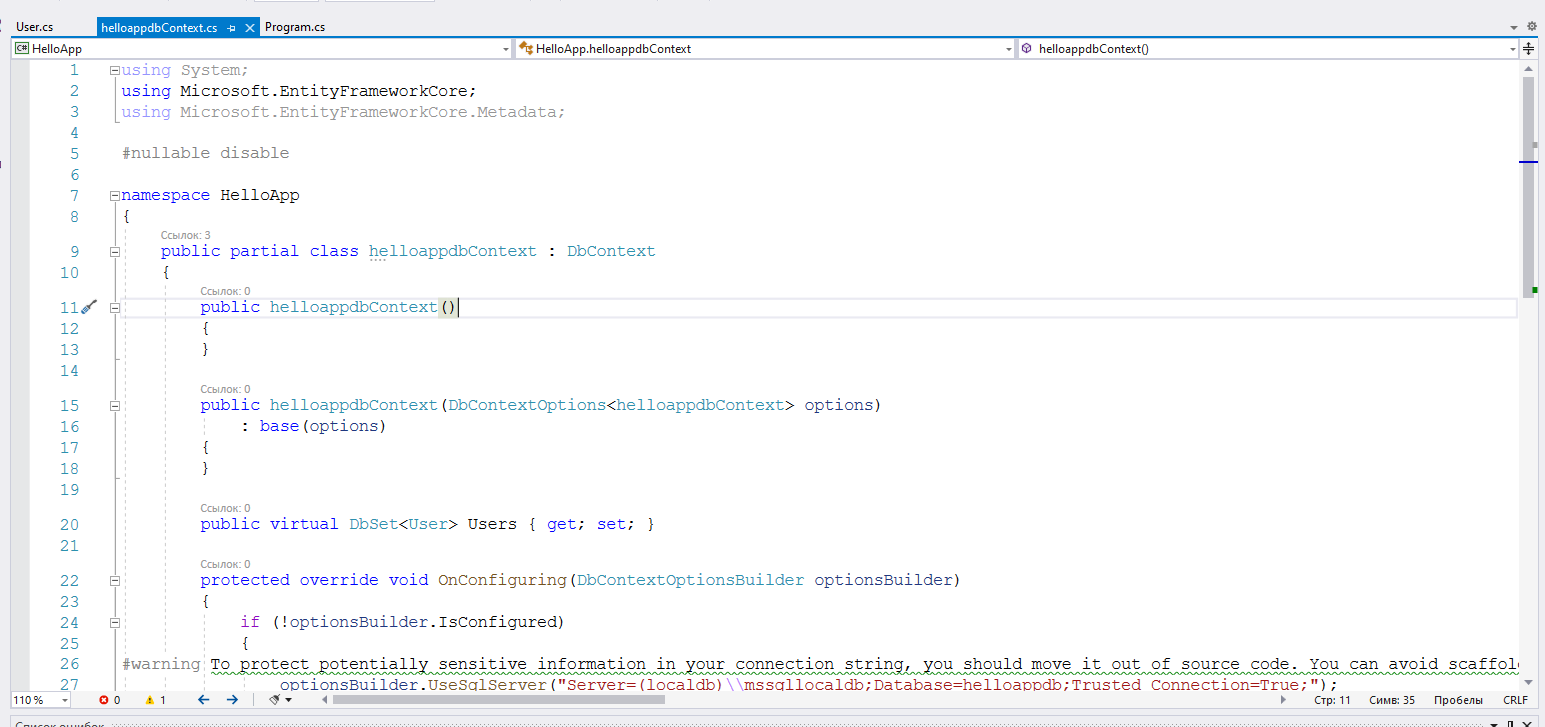
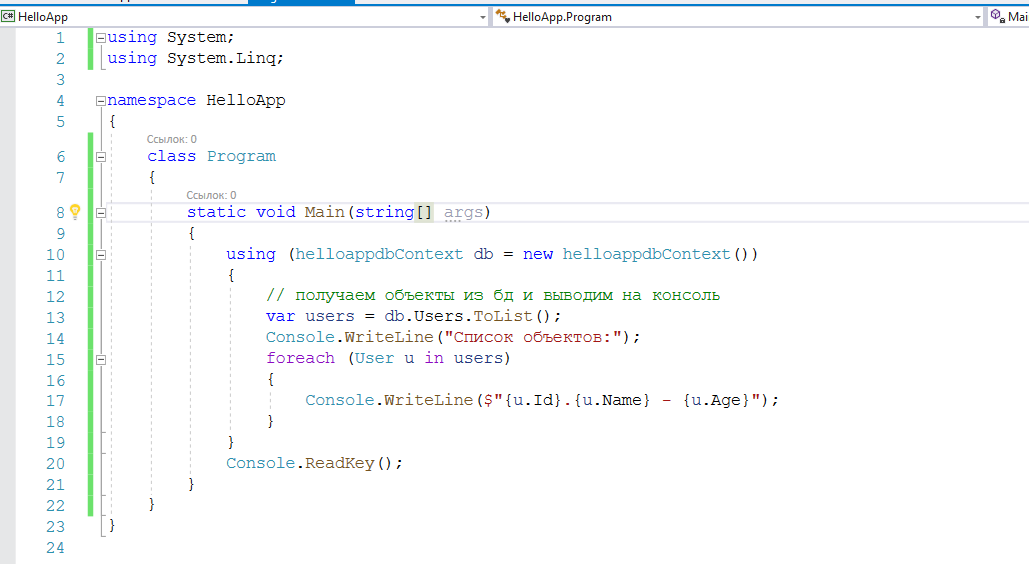


Рис 7. Також клас з контекстом бази даних

Рис 8. Змінюємо головний клаc Program.cs щоб отримати дані

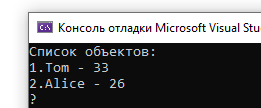


Рис 9.Результат роботи програми.

Лістинг 1. Код класу program.cs для додавання різних операцій редагування

using System;

using System.Linq;

namespace ConsoleApp2

{

public class Program

{

public static void Main(string[] args)

{

// Добавление

using (ApplicationContext db = new ApplicationContext())

{

User user1 = new User { Name = "Tom", Age = 33 };

User user2 = new User { Name = "Alice", Age = 26 };

// Добавление

db.Users.Add(user1);

db.Users.Add(user2);

db.SaveChanges();

}

// получение

using (ApplicationContext db = new ApplicationContext())

{

// получаем объекты из бд и выводим на консоль

var users = db.Users.ToList();

Console.WriteLine("Данные после добавления:");

foreach (User u in users)

{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

// Редактирование

using (ApplicationContext db = new ApplicationContext())

{

// получаем первый объект

User user = db.Users.FirstOrDefault();

if (user != null)

{

user.Name = "Bob";

user.Age = 44;

//обновляем объект

//db.Users.Update(user);

db.SaveChanges();

}

// выводим данные после обновления

Console.WriteLine("\nДанные после редактирования:");

var users = db.Users.ToList();

foreach (User u in users)

{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

// Удаление

using (ApplicationContext db = new ApplicationContext())

{

// получаем первый объект

User user = db.Users.FirstOrDefault();

if (user != null)

{

//удаляем объект

db.Users.Remove(user);

db.SaveChanges();

}

// выводим данные после обновления

Console.WriteLine("\nДанные после удаления:");

var users = db.Users.ToList();

foreach (User u in users)

{

Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");

}

}

Console.Read();

}

}

}

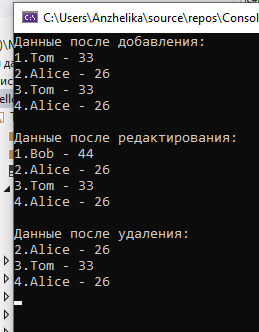


Рис 10.Результат роботи програми

Для взаємодії з базою даних для контексту даних повинна бути визначена конфігурація підключення. Для її установки можна застосовувати два способи:

Перевизначення у класу контексту даних методу OnConfiguring ()

Передача конфігурації в конструктор базового класу DbContext

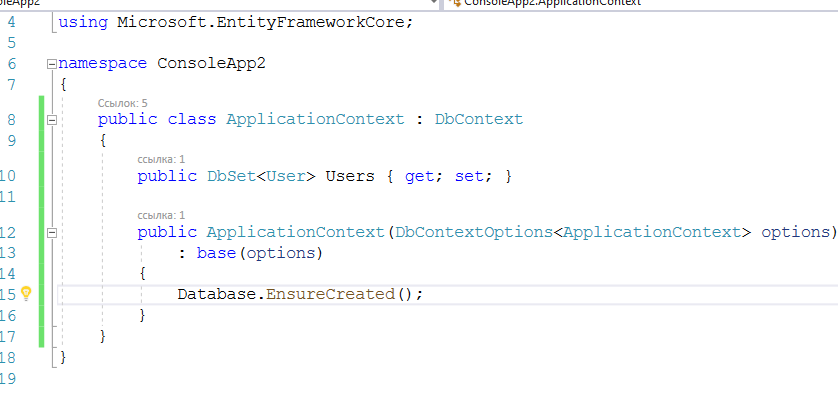


Рис 11. Використовуємо передачу конфігурації в конструктор базового класу DbContext



Рис 12.Class Program.cs

файл конфігурації

Обидва вище представлених способи цілком працюють, проте в тому визначенні, в якому вони представлені, вони мають один недолік - рядок підключення жорстко визначений в коді C#. І було б непогано, якби він був би визначений в якомусь зовнішньому файлі підключення, де ми його могли б поміняти без перекомпіляції програми.

Для цього додамо в проект новий елемент JSON File, який назвемо appsettings.json:

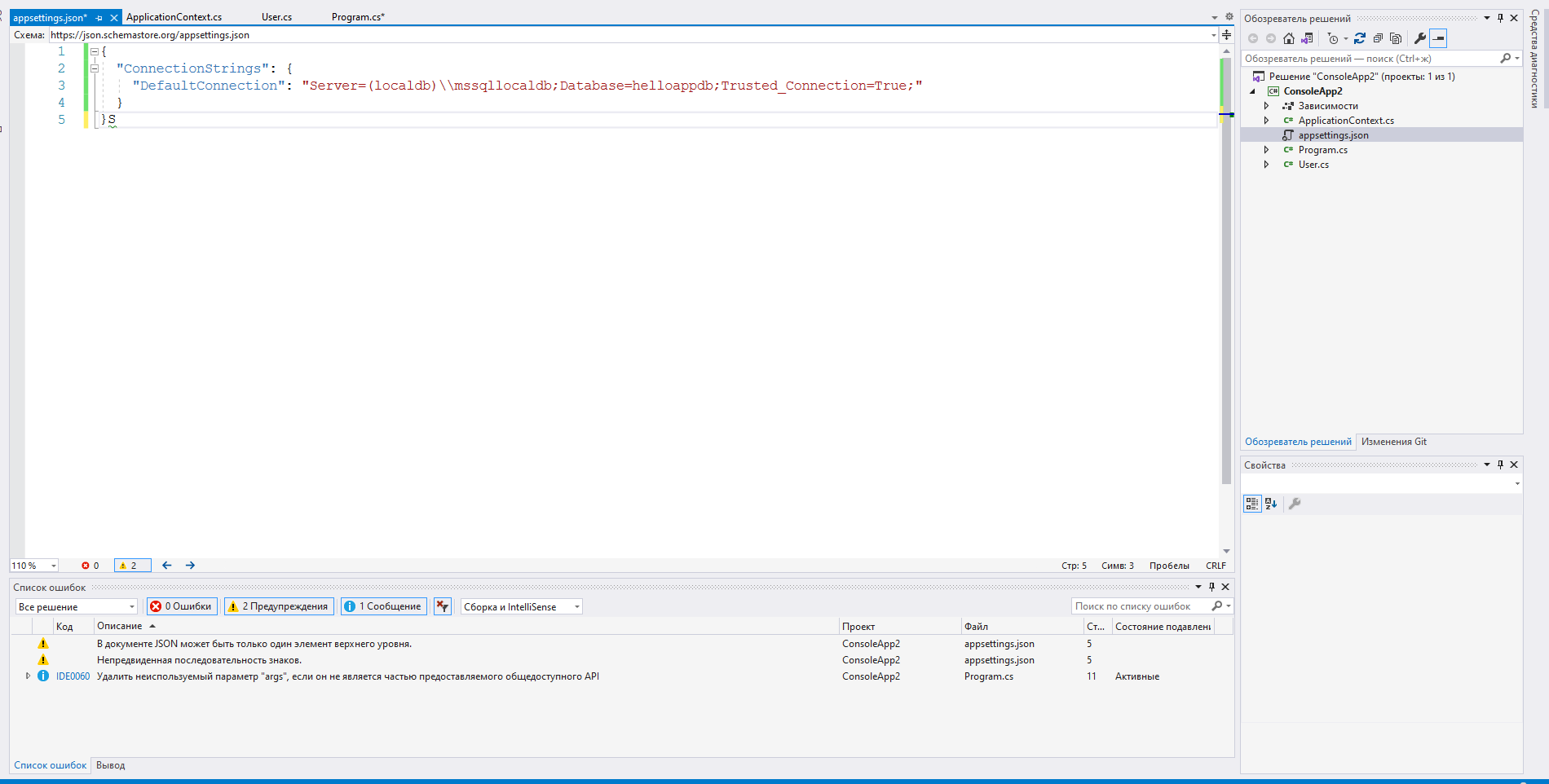


Рис 13. Створений JSON файл з конфігурацією підключення до БД

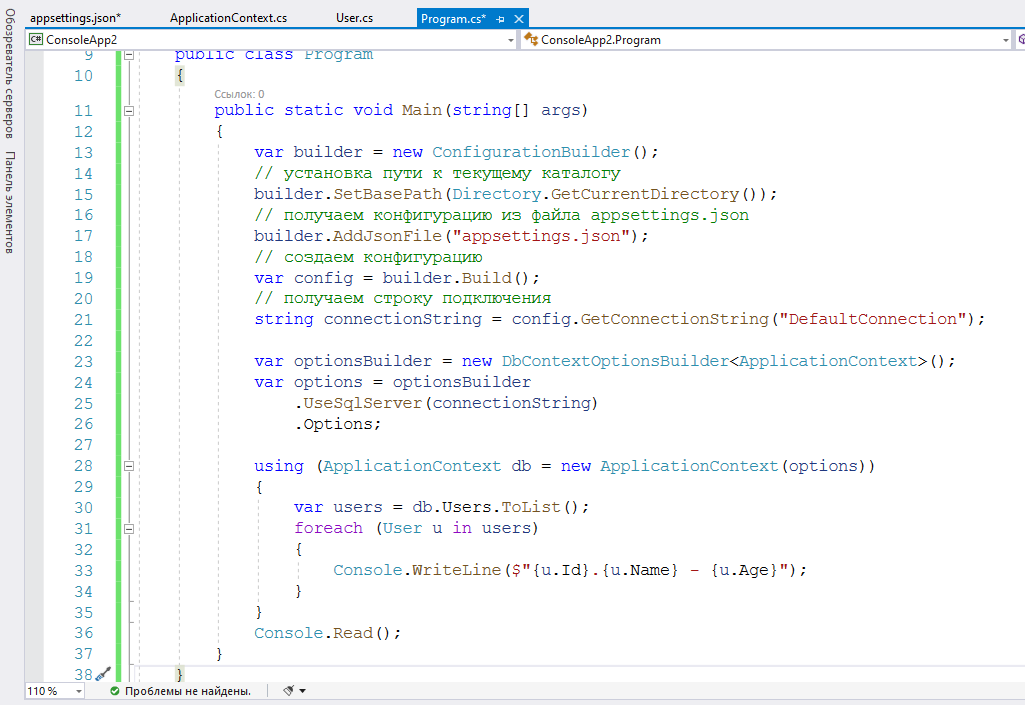


Рис 14. Тепер клас Program.cs отримує конфігурацію від JSON файла та підключається до БД

Також існує цікавий спосіб, який називається логування, він дозволяє нам логувати інформацію

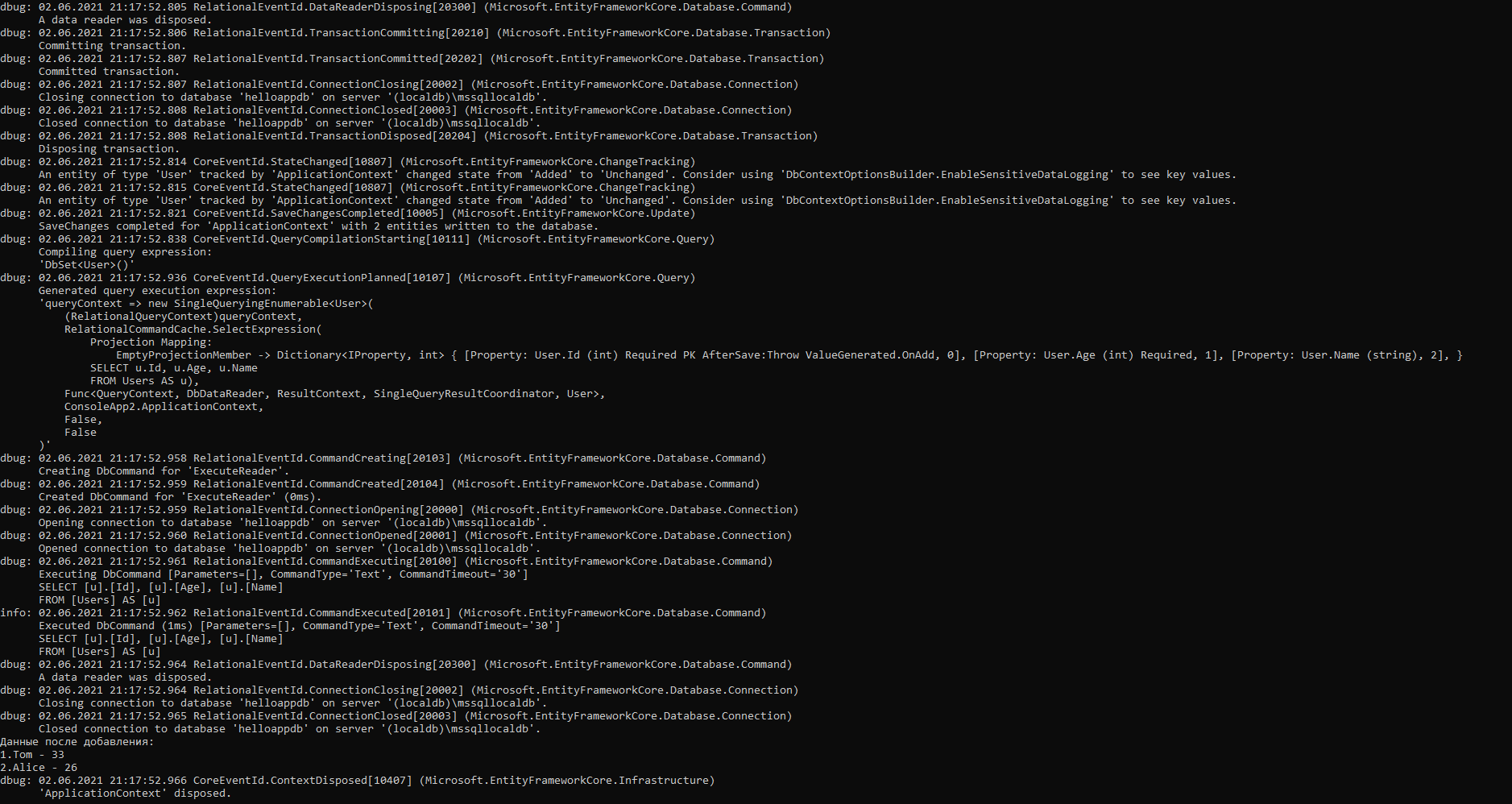


Рис 15. Приклад підключення та показу бази даних

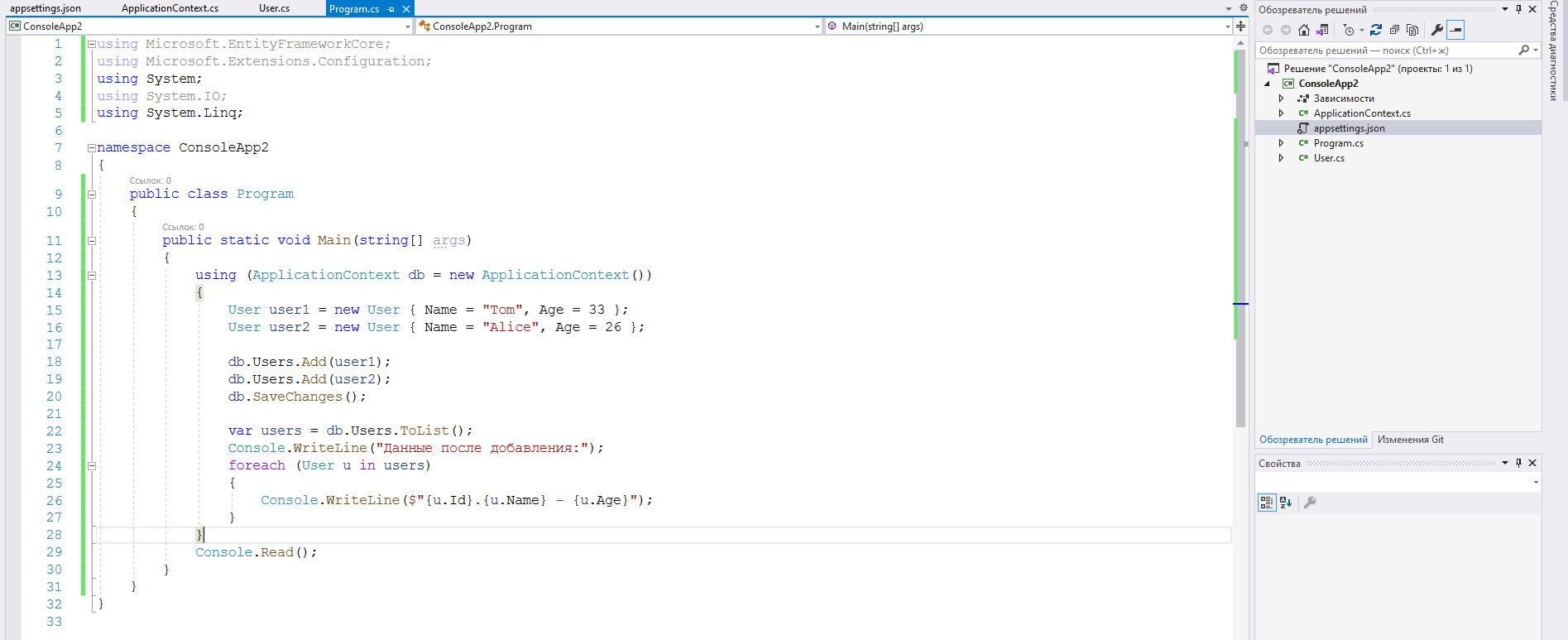


Рис 16. Class Program.cs для логування 

Рис 17. Class ApplicationContext який дозволяє нам логувати інформацію за допомогою методу .LogTo

Метод LogTo () має ряд перевантажених версій, які приймають різну кількість параметрів. Так, ми можемо передати в LogTo рівень логгірованія у вигляді одного з значень перерахування LogLevel:

Trace: використовується для виведення найбільш деталізованих повідомлень. Подібні повідомлення можуть нести важливу інформацію про програму та його будову, а тому цей рівень краще використовувати при розробці, але ніяк не при публікації

Debug: для виведення інформації, яка може бути корисною в процесі розробки і налагодження програми

Information: рівень повідомлень, що дозволяє просто відстежити потік виконання програми

Warning: використовується для виведення повідомлень про несподівані події, наприклад, помилки, які не впливають не зупиняють виконання програми, але в той же час повинні бути іследовать

Error: для виведення інформації про помилки та винятки, які виникли при поточній операції і які не можуть бути оброблені

Critical: рівень критичних помилок, які вимагають негайної реакції - помилками операційної системи, втратою даних в бд, переповнення пам'яті диска і т.д.

None: висновок інформації в лог не застосовується

Провайдери логування.Створимо свій провайдер, створимо новий клас.

Клас провайдера логгірованія повинен реалізувати інтерфейс ILoggerProvider. У цьому інтерфейсі визначено два методи:

CreateLogger: створює і повертає об'єкт Логгер. Для створення Логгер використовується шлях до файлу, який передається через конструктор

Dispose: управляє звільнення ресурсів. В даному випадку порожня реалізація

Сам логгер представлений об'єктом ILogger. Цей інтерфейс визначає три методи:

BeginScope: цей метод повертає об'єкт IDisposable, який представляє деяку область видимості для Логгер. В даному випадку нам цей метод не важливий, тому повертаємо значення null

IsEnabled: повертає значення true або false, які вказує, чи доступний логгер для використання. Тут можна здати різну логіку. Зокрема, в цей метод передається об'єкт LogLevel, і ми можемо, наприклад, задіяти логгер в залежності від значення цього об'єкта. Але в даному випадку просто повертаємо true, тобто логгер доступний завжди.

Log: цей метод призначений для виконання логгірованія. Він приймає п'ять параметрів:

LogLevel: рівень деталізації поточного повідомлення

EventId: ідентифікатор події

TState: деякий об'єкт стану, який зберігає повідомлення

Exception: інформація про виключення

formatter: функція форматування, яка за допомогою двох попередніх параметрів дозволяє отримати власне повідомлення для логгірованія

У поточній реалізації проводиться запис в текстовий файл і висновок на консоль, але природно дії по логгірованію можуть бути і іншими

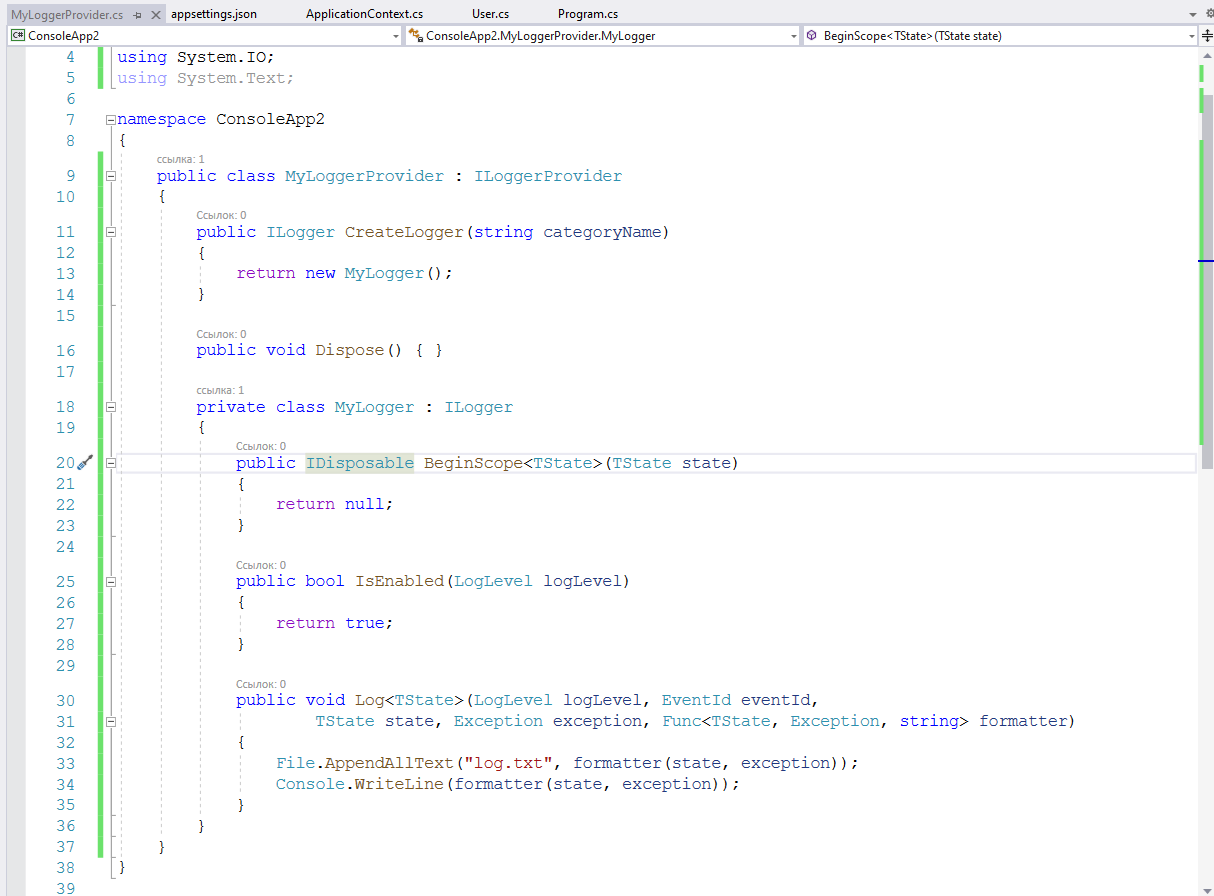


Рис 18.Class MyLoggerProvider який виводить текст у log.txt

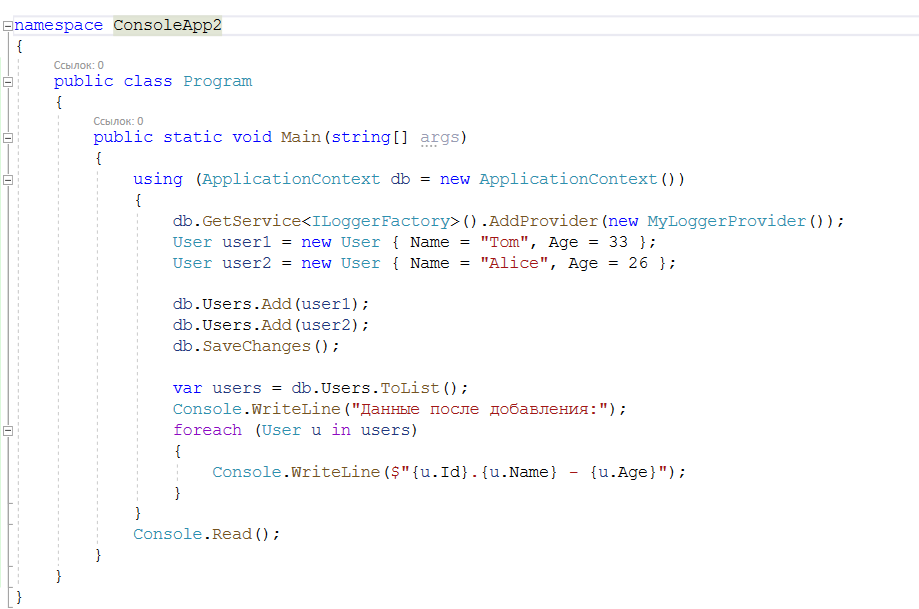


Рис 19.Class Program.cs

За допомогою методу GetService <ILoggerFactory> () отримуємо вбудований сервіс ILoggerFactory, якому через метод AddProvider () передаємо наш провайдер MyLoggerProvider.

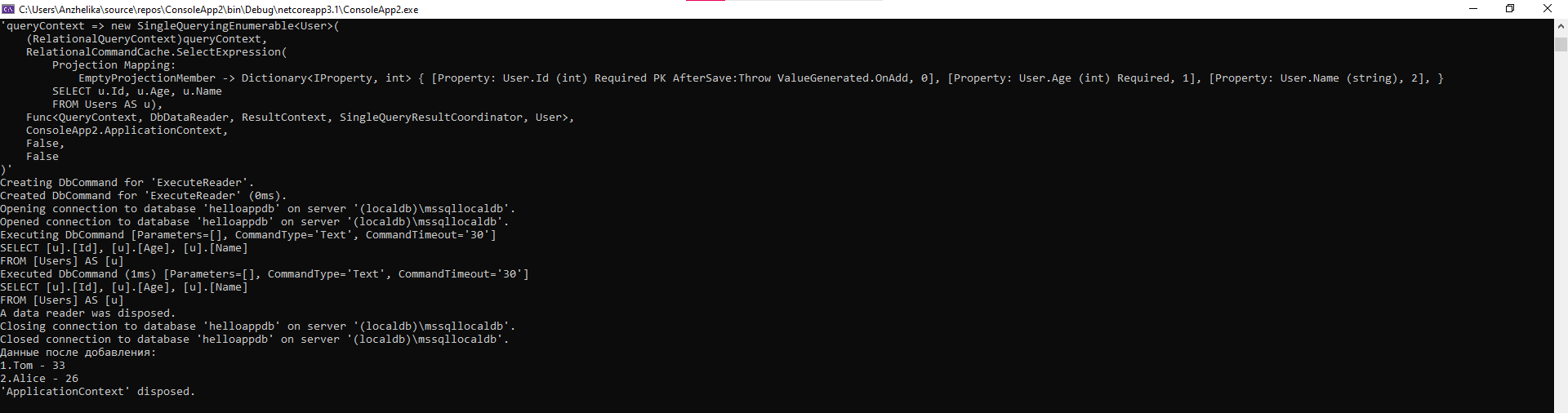


Рис 20.Результат виконання програми з детальними операціями

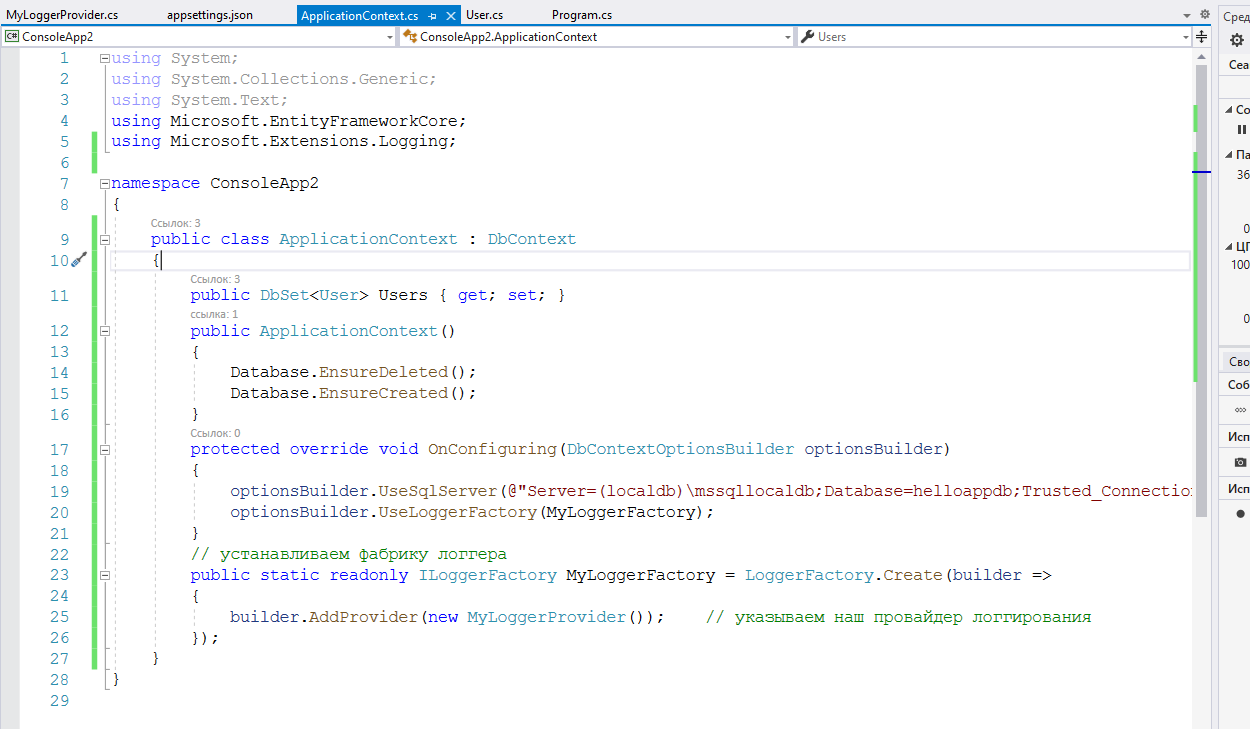
Глобальна установка логгірованія в контексті даних 

Рис 21. Class ApplicatonContext

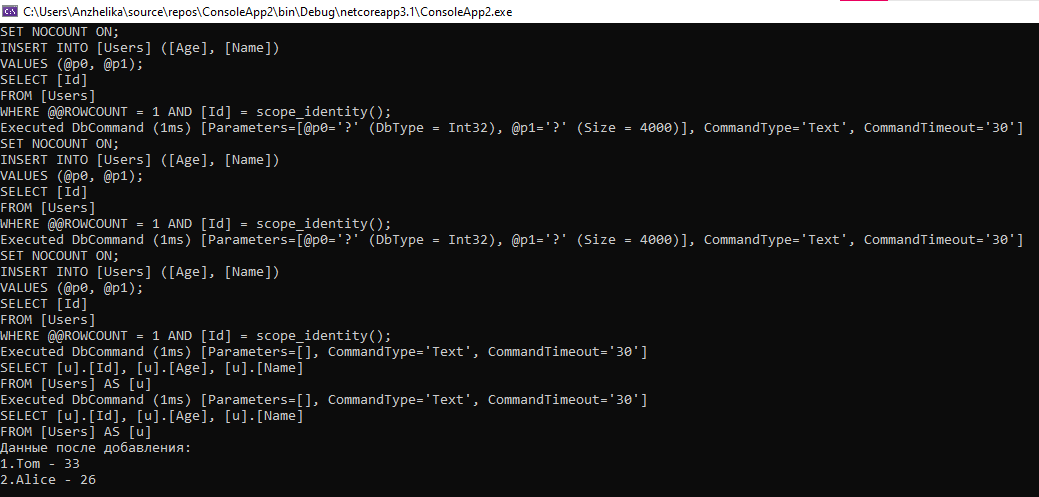


Рис 20.Логування в контексті даних

ГЛАВА 2. Створення моделей в Entity Framework Core

Моделі, Fluent API і анотації даних

Fluent API представляє набір методів, які визначають зіставлення між класами і їх властивостями і таблицями та їх стовпцями. Як правило, функціонал Fluent API задіюється при перевизначенні методу OnModelCreating:

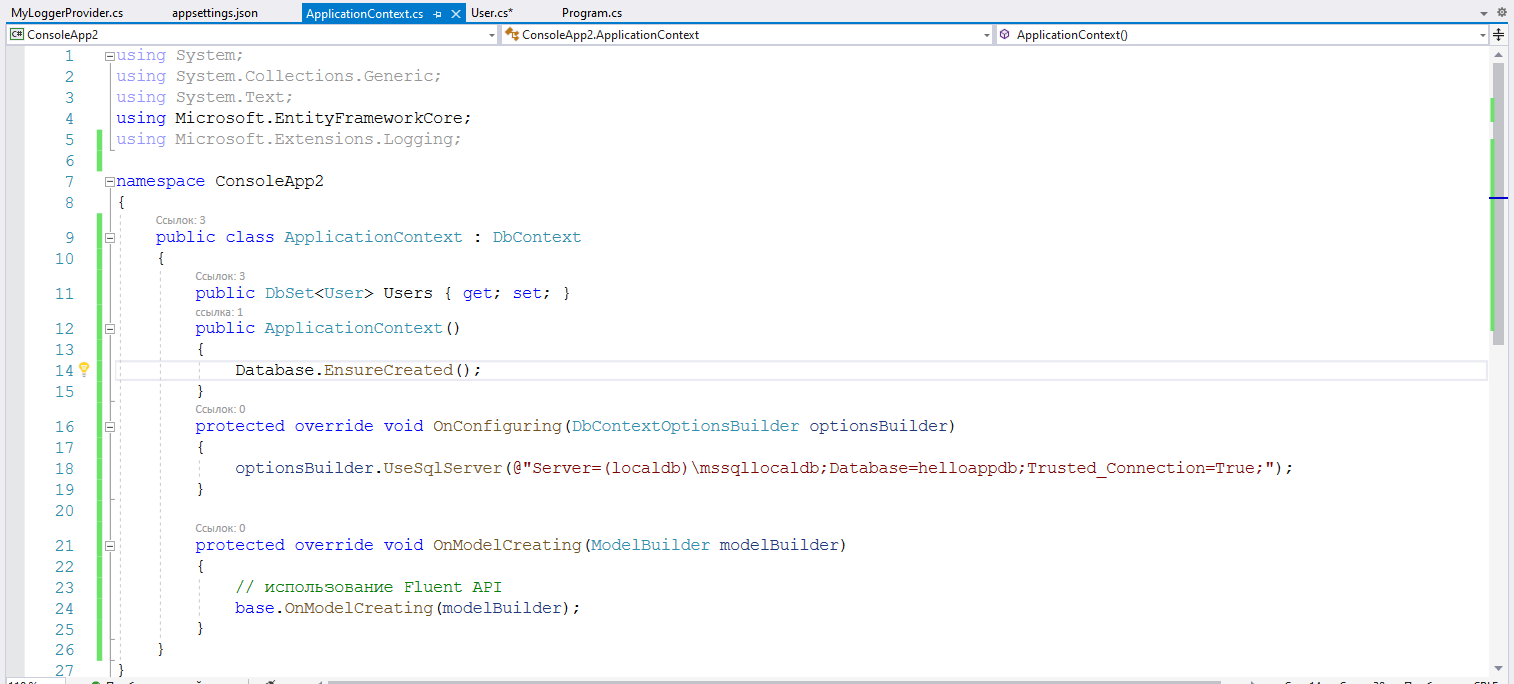


Рис 21. Class ApplicationContext

За допомогою методу OnModelCreating використовуємо Fluent API

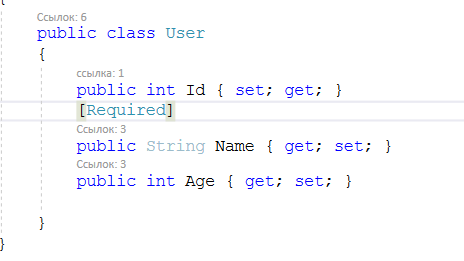


Рис 22. Анотація Required, яка вказує що змінна Name обов’язково повинна містити текст.

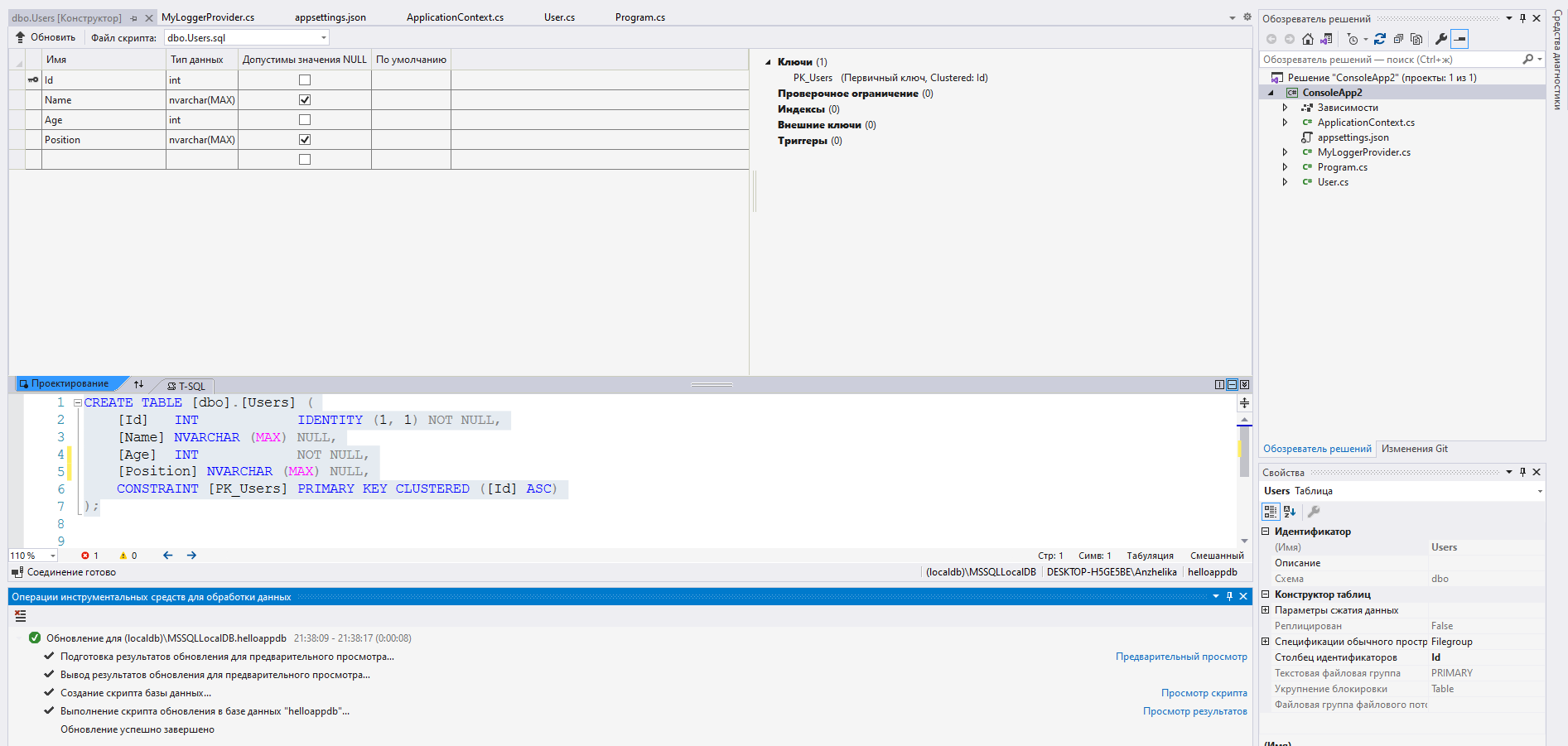


Рис 23. Добавляємо новий стопчик в базу даних та оновлюємо її

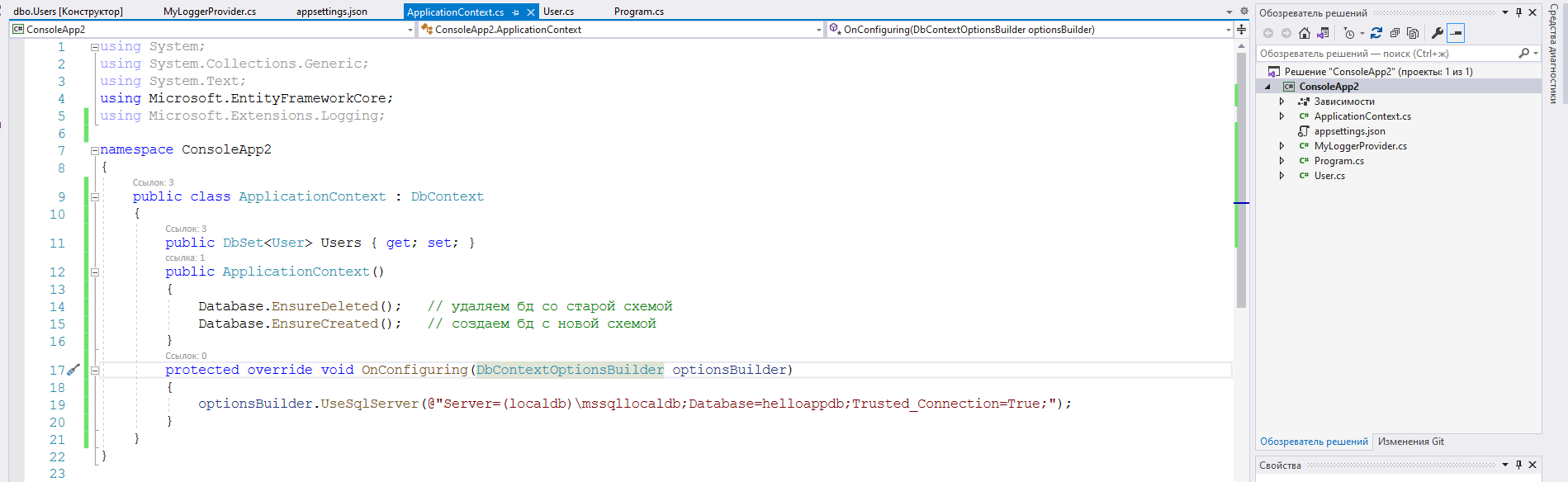


Рис 24. Інший спосіб це удалити стару базу, та створити нову на основі класу User

Якщо ми запустимо код на виконання, отримаємо помилку:

PM> Add-Migration InitialCreate

Build started...

Build succeeded.

Unable to create an object of type 'ApplicationContext'. For the different patterns supported at design time, see https://go.microsoft.com/fwlink/?linkid=851728

Щоб її не отримати нам потрібно створити новий клас SampleContextFactory

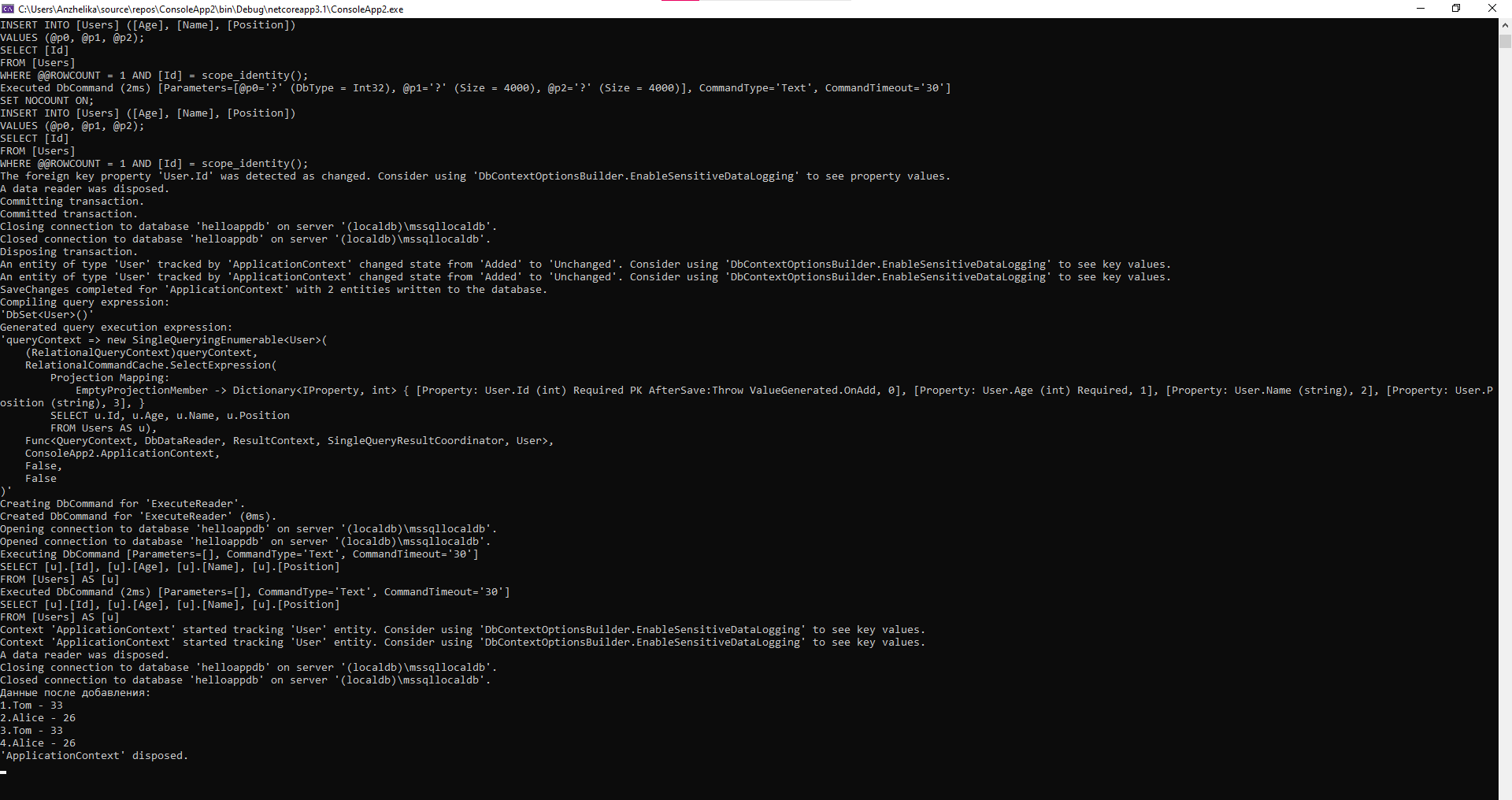


Рис 25.Результат виконання програми

Клас SampleContextFactory застосовує інтерфейс IDesignTimeDbContextFactory, який Типізуються типом контексту даних - в даному випадку клас ApplicationContext. Даний інтерфейс містить один метод CreateDbContext (), який повинен повертати створений об'єкт контексту даних.

В даному випадку також отримуємо конфігурацію з файлу appsettings.json і витягаємо з її рядок підключення і таким чином створюємо контекст.

Хоча цей клас формально ніде не викликається і ніяк не використовується, фактично він викликається інфраструктурою Entity Framework при створенні міграції.

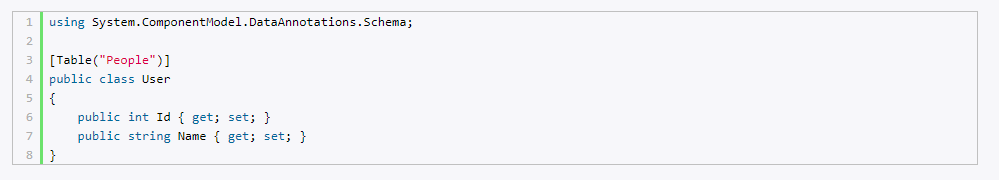


Рис 26. Атрибут Table дозволяє переопреділяти зіставлення з таблицею



Рис 27.Аналогічне переопреділення через метод toTable через Fluent API

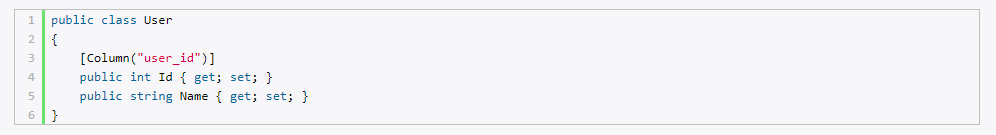


Рис 28. Атрибут Column переопреділяє зіставлення

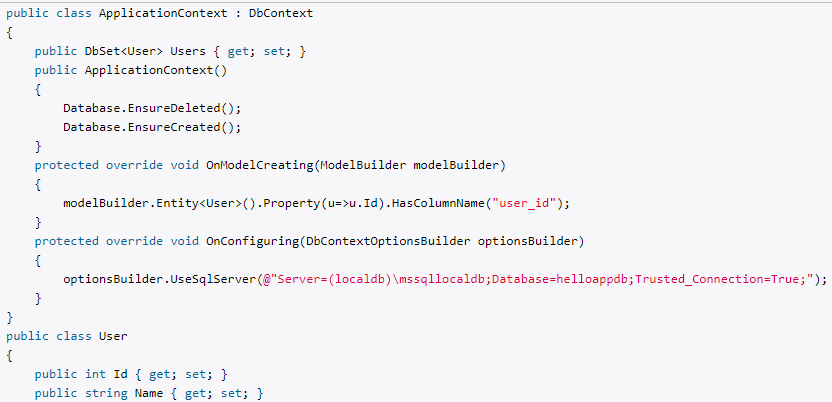


Рис 29.Аналогічне зіставлення за допомогою HasColumnName в Fluent API

Налаштування ключів..

За замовчанням ключ встановлюється значення яке називається Id або назва класу.

Для встановлення властивості в качестві первичного ключа за допомогою анотації [key]

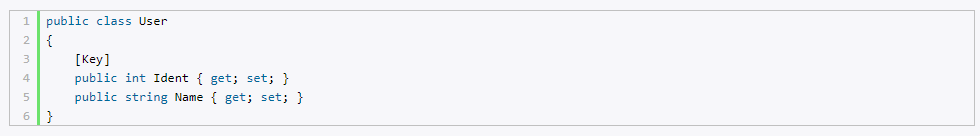


Рис 30.Анотація [Key]



Рис 31. Для конфігурації через Fluent API використовується метод HasKey.

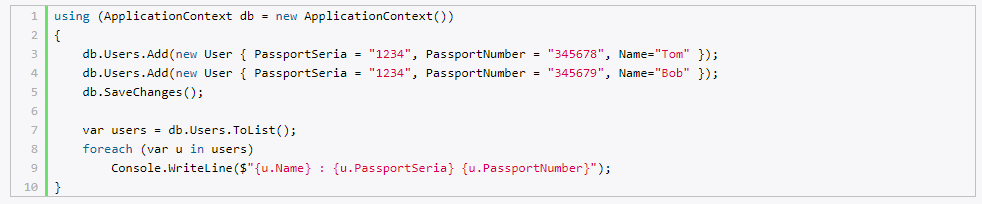


Рис 32. Використання зіставного ключа, який створюється лише за допомогою Fluent API



Рис 33. Для альтернативного ключа використовується метод .HasAlternativeKey

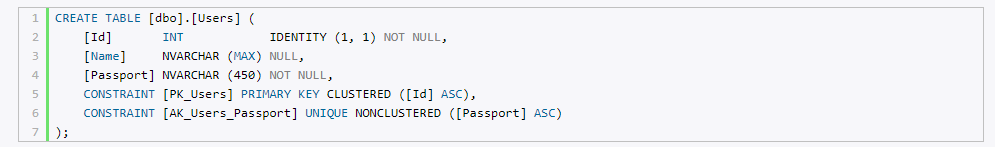


Рис 34. SQL – скрипт, де серія та номер паспорта будуть альтернативними ключами

Настройки Індексів.Індексація



Рис 35. Використання методу HasIndex

Для того щоб збільшити швидкодію праці бази даних також використовується індексація.

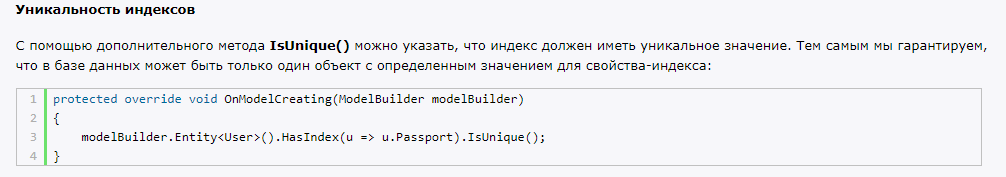


Рис 36. IsUnique метод, який перевіряє індекс на унікальність.



Рис 37. Перевірка індексів для пари свойств

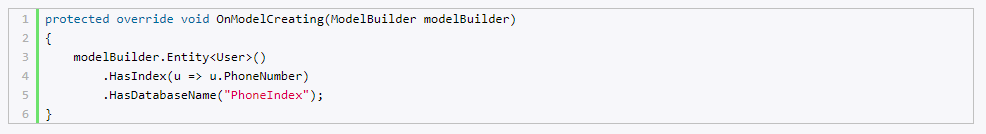
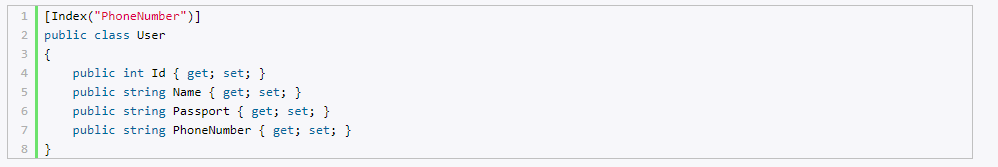


Рис 38. За допомогою методу HasDatabaseName можна задати деякому індексу, ім’я.



Рис 38. Фільтр індексів



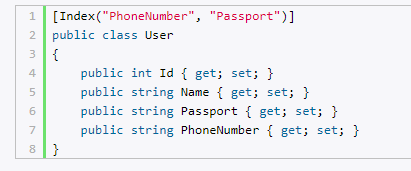
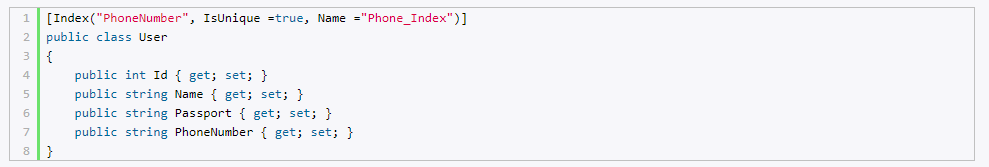
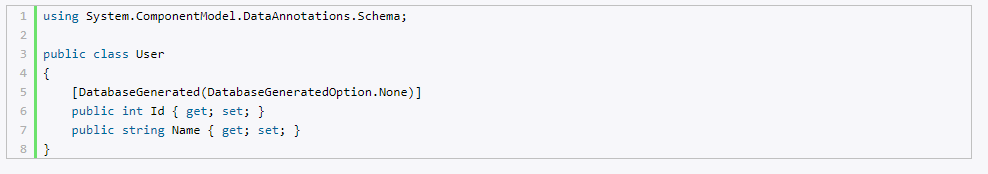
 

Рис 39,40,41. Налаштування індексів через атрибут

генерація ключів

За замовчуванням для властивостей первинних ключів, які представляють типи int або GUID і які мають значення за замовчуванням, генерується значення при вставці в базу даних. Для всіх інших властивостей значення за замовчуванням не генерується.



### Рис 42. За допомогою DatabaseGeneratedAttribute ми можемо відключити автоматичне генерування при додавані в базу нового значення

### 

Рис 43. Відключення автоматичного генерування за допомогою Fluent API

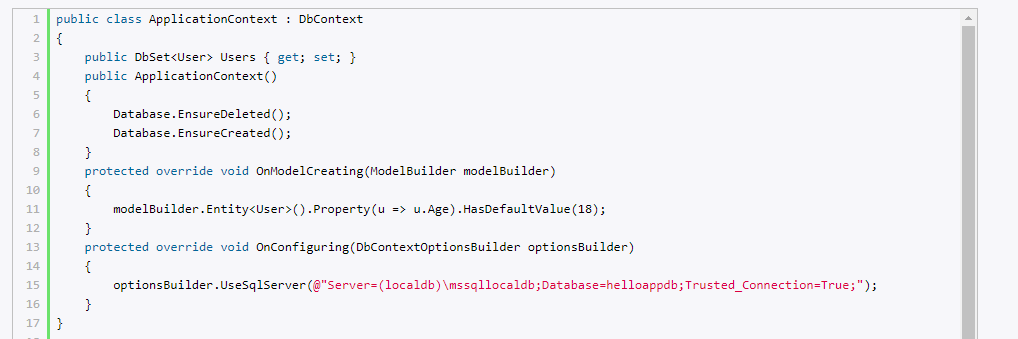


Рис 44. Також за допомогою методу HasDefaultValue ми можемо встановити значення за замовчуванням.

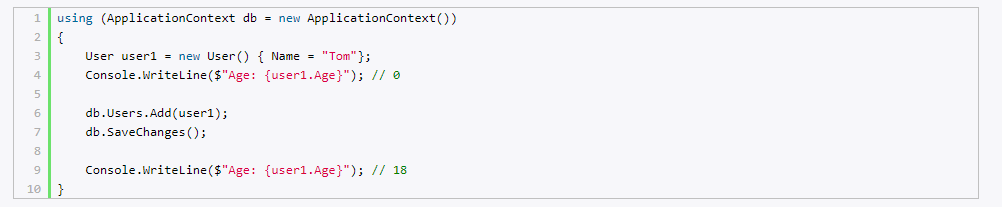


Рис 45. Якщо в цьому випадку ми не вкажемо значення за значенням, то воно буде рівне 18



Рис 46.За допомогою SQL запросу ми можемо також встановити значення за замовчуванням, але з викликом HasDefaultValueSql



Рис 47. Встановлення значення name за допомогою Fluent API та стовпців Firstname and LastName.

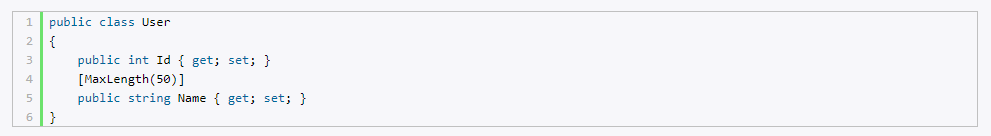
Рис 48. За допомогою свойства MaxLength можемо встановити дліну поля.



Рис 49. Встановлення максимальної кількості символів в назві за допомогою FluentAPI

За замовчуванням провайдер певної СУБД вибирає для стовпців типи даних на основі типів даних властивостей сутності.

Якщо в якості цільової СУБД використовується MS SQL Server, то типи SQL Serverа і C # зіставляються наступним чином:

int: int

bit: bool

char: string

date: DateTime

datetime: DateTime

datetime2: DateTime

decimal: decimal

float: double

money: decimal

nchar: string

ntext: string

numeric: decimal

nvarchar: string

real: float

smallint: short

text: string

tinyint: byte

varchar: string

Наприклад, за замовчуванням Entity Framework для властивості з типом string буде використовувати стовпець з типом nvarchar.

У провадеров для інших СУБД система зіставлення типів може відрізнятися.

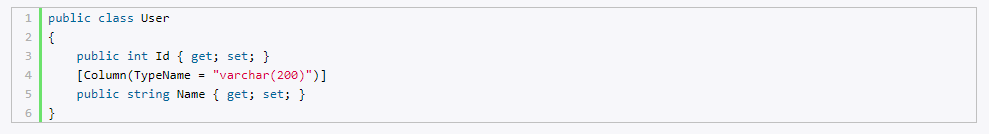


Рис 50. За допомогою атрибута Column можна задати тип даних для бд.



Рис 51. У Fluent API для такого завдання застосовується метод HasColumnType.

За допомогою Fluent API ми можемо поставити численні настройки для моделей і їх властивостей. Однак, якщо налаштувань дуже багато, то вони можуть обтяжувати клас контексту. Однак в Entity Framework Core ми можемо винести конфігурацію моделей в окремі класи.

Для винесення зміни у поза необхідно створити клас конфігурації, який реалізує інтерфейс EntityTypeConfiguration <T>.



Рис 52. Метод OnModelCreating



Рис 53. Тут конфігурація визначається для кожного типу в окремому методі, який в якості параметра приймає об'єкт EntityTypeBuilder <T>. Потім метод передається в виклик modelBuilder.Entity <T> () для відповідної моделі.



Рис 54. Для ініціалізації БД при конфігурації використовуємо метод .HasData().

Далі по ланцюжку викликається метод HasData (), який власне і визначає початкові дані. В даному випадку це набір з трьох об'єктів User. При цьому для кожного об'єкту необхідно встановити значення первинного ключа - в даному випадку значення властивості Id. Причому незалежно від того, генерує чи база даних для даних автоматично індентіфікатор чи ні, нам в будь-якому випадку його треба встановити - це основне обмеження при ініціалізації БД початковими даними.

При цьому слід враховувати, що ініціалізація початковими даними буде виконуватися тільки в двох випадках:

При виконанні міграції. (При створенні міграції додаються дані автоматично включаються в скрипт міграції)

При виклику методу Database.EnsureCreated (), який створює БД при її відсутності

У разі вище в конструкторі застосовується метод Database.EnsureCreated (), тому при створенні контексту даних

ApplicationContext db = new ApplicationContext ();

буде автоматично проводитися ініціалізація бд початковими даними.