КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет радіофізики, електроніки та комп'ютерних систем

Кафедра комп'ютерної інженерії

РОЗРОБКА КУРСУ ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ "СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ" ПО ВИКОРИСТАННЮ ПЛАТФОРМИ ENTITY FRAMEWORK CORE

Дипломна робота бакалавра

студента 4 року навчання

Спеціальність: 123 «Комп'ютерна інженерія»

Дмитра ПЕТРИКОВИЧА

Науковий керівник

канд. фіз.-мат. наук Сергій ЗАГОРОДНЮК,

доцент кафедри комп'ютерної інженерії

Рецензент

канд. фіз.-мат. наук Всеволод ДЕМИДОВ

доцент кафедри геоінформатики ННІ "Інститут геології"

До захисту допускаю: Завідувач кафедрою

Юрій БОЙКО

Ухвалено на засіданні кафедри "____" _____2022 р., протокол № ____

Київ 2022

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра містить: 77 с., 12 рис., 8 джерел.

В ході виконання роботи розроблено лабораторні роботи для дисципліни «Системне програмне забезпечення» або інших спецкурсів з програмування по створенню універсального програмного загального шаблону для взаємодії з користувачем. Курс складається з двох лабораторних робіт, які охоплюють основи роботи з платформами ASP.NET Core та Entity Framework Core.

В результаті роботи підготовлено навчальний курс, що знайомить студентів з сучасними методологіями роботи з засобами Entity Framework Core та розвиває у них основні навички для подальшого знайомства та роботи.

Застосовано технології .NET Core для абстрагування від конкретного СУБД як джерела даних, організовано надання можливості роботи з інформацією у об’єктному представленні.

Ключові слова: ASP.NET Core, Entity Framework Core, EF Core Курс з Системного програмування, База даних Microsoft SQL Server, Dependency Injection, Визначення відношення між сутностями EF Core, Міграція бази даних EF Core.

ВСТУП

Велике значення для вищих навчальних закладів має створення власних курсів по розробці програмного забезпечення. Вони дають змогу значно спростити робочі процеси, та більш доцільно використовувати ресурс студента. На теперішній час в розробці Web програм є багато проблем, вирішення яких потребує використання ресурсу розробника. Прикладами таких проблем є:

Ненадійність мережі передачі інформації - всі інтеграційні рішення передбачають в своєму складі передачу інформації між пристроями, доставка такої інформації є комплексним процесом який на кожному етапі внутрішньої обробки створює незначні затримки, і таких маленьких затримок може бути велика кількість, вони в свою чергу можуть стати причиною втрати важливої інформації.

Конкурентоспроможність в плані змін – додатки мають бути гнучкими при роботі в тандемі іншими додатками. Частіше всього зміни в одній системі тягнуть за собою наслідки в другій.

Мета: створити курсу з дисципліни «Системне програмне забезпечення», який ознайомить студентів з сучасним механізмом програмного доступу до баз даних на платформі Entity Framework Core.

Відкрити для студентів можливість взаємодіяти з базами даних водночас за традиційним механізмом, вивченим у попередніх курсах, і так само за новим механізмом, через процедуру створення об’єктного програмного представлення для бази даних MSSQL/MySQL. Порівняти переваги і недоліки обох механізмів.

Продемонструвати студентам можливість розробляти власні програмні бібліотеки доступу до баз даних, які можна використати у наступних завданнях, а також у курсових і дипломних проектах на основі платформи .NET.

Актуальність: Розробка програмного забезпечення вже давно зайняла своє місце в ринку професій які є гарним способом підтримки економіки нашої держави.

Розробники сервісів є цінними співробітниками в кожній компанії і якраз для таких розробників є гарні інструменти Entity Framework Core і ASP.NET Core. Однак існує дефіцит навчальних матеріалів в цій частині IT суспільства, і навіть ще гірше йдуть справи з Локалізацією таких знань в Україні.

Таким чином можна з упевненістю говорити, що актуальність теми полягає у розробці дієвого засобу як для стаціонарного навчання так і дистанційного, що буде гарним з практичної точки зору навчальним матеріалом для побудови комплексних програмних рішень на платформах .Net і EF Core.

Структура: Робота складається зі Вступу, трьох розділів, висновок, список використаних джерел і додаток.

Перший розділ: Огляд інструментів для роботи з базою даних Entity Framework Core опис і призначення окремих елементів.

Другий розділ: Огляд інструментів для розробки сервісів на платформі ASP.NET Core опис і призначення окремих елементів.

Третій розділ: Розробка лабораторних завдань які охоплюють матеріали зазначені в попередніх розділах.

ОСНОВНА ЧАСТИНА

РОЗДІЛ 1

«ОГЛЯД ІНСТРУМЕНТУ ДЛЯ РОБОТИ З БАЗОЮ ДАНИХ ENTITY FRAMEWORK CORE »

EF Core - це остання версія Entity Framework від Майкрософт. Він є легким, розширювальним та кросплатформенним інструментом для спрощення роботи з базою даних. Цей фреймворк є ORM (об'єктно-реляційною проекцією, Object-relational mapper) – технологія програмування, яка допомагає працювати з інформацією як з об'єктом. Фреймворк забирає на себе работу по перетворенню типів, під час роботи з базою даних.

1.1 Початок роботи з EF Core

1.1.1 Що може зробити ORM EF Core

- Відображати доменну модель на сутності в базі даних
- створювати бази даних і підтримувати схему відповідно до змін в моделі
- виконувати SQL запити до сутностей
- виконувати транзакції

1.1.2 Які засоби можна використати для встановлення Entity Framework Core.

1. Менеджер пакетів Nuget UI
2. Packet Manager Console
3. Command Line
4. Project.json

Модель

Способи розробки моделі бази даних

Code First – спочатку створюємо свою модель у вигляді класів, далі за допомогою EF Migrations створюємо базу.

Приклад:

Необхідно встановити

//Microsoft.EntityFrameworkCore.SqlServer

//Microsoft.EntityFrameworkCore.Tools

1.1.3 Створення моделі

За замовчуванням первинним ключем буде перша властивість моделі, так як є необхідність в його створенні. Побудуємо сутність **User**.

```
class UniqueUser
{
    public Guid UniqueUserId { get; set; }
    public int UniqueUserAge { get; set; }
    public string UniqueUserName { get; set; }
}
```

Основою EF Core для взаємодії з базою є класи:

DBContext – абстракція яка є оболонкою бази даних, і може бути використана для зчитування та запису екземплярів визначених там сутностей.

*Є комбінацією шаблонів **Unit of Work** та **Repository.**

DbSet/DbSet<TEntity> - набір об’єктів які зберігаються в базі(використовується для зчитування та зберігання екземплярів **TEntity,** так же до нього застосовуються стандартні **CRUD** операції та **LINQ** запити, в подальшому буде розглянуто)

DbContextOptionsBuilder – є програмним інтерфейсом для налаштування **DbContextOptions** та інших розширень, дозволяє конфігурувати параметри з'єднання з базою.

1.1.4 Налаштування DBContext

```
public class TestUniqueUserDbContext : DbContext
{
    public DbSet<UniqueUser> UniqueUsers { get; set; }

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.
            UseSqlServer(
                "Server=DESKTOP-F7TI41U\\SQLEXPRESS;Database=SusDb;Trusted_Connection=True"
            );
    }
}
```

Записуємо назву екземпляру сервера **`Server=DESKTOP-F7TI41U\\SQLEXPRESS`**, назви бази даних файл бази даних **`Database=TestDb`** та **`Trusted_Connection=True`** вказує на те що використовуватись буде використано Windows Authentication. І будь які user id = будуть проігноровані.

1.1.5 Створення міграції

Для того щоб створити міграцію необхідно в **Package Manager Console** ввести команду

```
PM> Add-Migration Initial
```

Де **Initial** це назва яку користувач **самостійно** надає конкретній міграції. Після цього буде автоматично створено каталог **Migrations** *Рис. 1.1.1.*

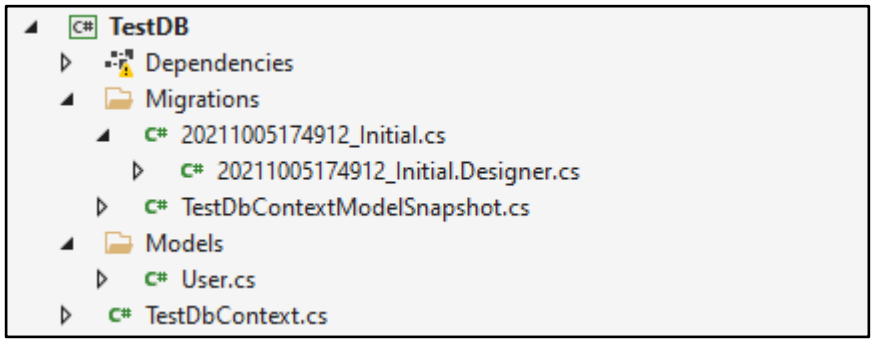

Рис. 1.1.1 – Структура тестової бази даних в контексті

```
public partial class Initial : Migration
{
    ~protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "UniqueUsers",
            columns: table => new
            {
                UniqueUserId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                UniqueUserAge = table.Column<int>(type: "int", nullable: false),
                UniqueUserName = table.Column<string>(type: "nvarchar(max)", nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_UniqueUsers", x => x.Id);
            });
    }

    ~protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "UniqueUsers");
    }
}
```

Автоматично створений файл міграції має вказану назву **Initial**.

Має в собі два методи, метод **Up** – для створення таблиці з усіма визначеними параметрами для скрипту створення таблички та **Down –** скрипт для відкату.

Для того щоб застосувати міграцію необхідно ввести команду

```
PM> Update-Database
```

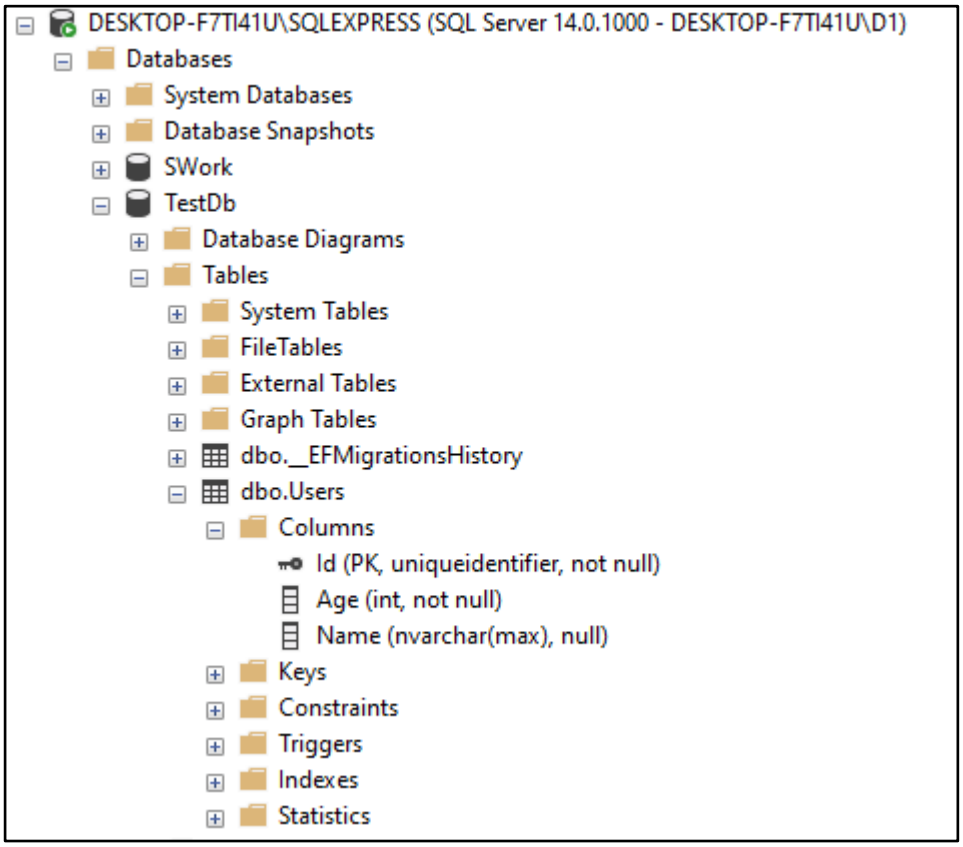

Рис. 1.1.2 – Структура тестової бази даних

В результаті створено базу TestDb з таблицею Users *Рис. 1.1.2*.

Database-First – файли які створенні в **Code First** будуть автоматично згенеровані за допомогою команди.

```
PM> Scaffold-DbContext "Server=DESKTOPF7TI41U\SQLEXPRESS;Database=SWork;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer
```

Hard code – також ніхто не забороняє створити і базу і моделі окремо. Подібне до **Code First** реалізації.

1.1.6 CRUD команди

Create|Read|Update|Delete

Додавання

```
// INSERT INTO
using (var db = new SusDbContext())
{
    var user1 = new UniqueUser { UserName = "Mia", UserAge = 33 };
    var user2 = new UniqueUser { UserName = "Frank", UserAge = 26 };

    db.UniqueUsers.Add(user1);
    db.UniqueUsers.Add(user2);
    db.SaveChanges();
}
```

Зчитування

```
// Read
using (SusDbContext db = new SusDbContext())
{
    var users = db.UniqueUsers.ToList();
    Console.WriteLine("Instances after adding:");
    foreach (var uniqueUserin uniqueUsers)
    {
        Console.WriteLine($"Guid:{uniqueUser.Id} - Name:{uniqueUser.UserName} -
Age:{uniqueUser.UserAge}");
    }
}
```

Редагування

```
// SELECT FROM
using (SusDbContext db = new SusDbContext())
{
    // get first user
    var user = db.UniqueUsers.FirstOrDefault();
    if (user != null)
    {
        user.UserName = "Nans";
        user.UserAge = 21;
        db.SaveChanges();
    }

    Console.WriteLine("\nEdited lines:");
    var uniqueUsers = db.UniqueUsers.ToList();
    foreach (var uniqueUsers in uniqueUsers)
    {
        Console.WriteLine($"Guid:{u.Id} - Name:{u.UserName} - Age:{u.UserAge}");
    }
}
```

Видалення

```
// DELETE FROM
using (SusDbContext db = new SusDbContext())
{
    // get first user
    var UniqueUser = db.UniqueUsers.FirstOrDefault();
    if (user != null)
    {
        //delete
        db.UniqueUsers.Remove(user);
        //db.SaveChanges();
    }
    Console.WriteLine("\nTable after delete:");
    var users = db.UniqueUsers.ToList();
    foreach (User u in users)
    {
        Console.WriteLine($"Guid:{u.Id} - Name:{u.UserName} - Age:{u.UserAge}");
    }
}
```

1.1.7 Відслідковування запитів які надходять до бази даних

Це можливо зробити за допомогою програми **SQL Server Profiler, або програмно метом optionsBuilder.LogTo**. На початкових етапах буде корисно бачити в якому вигляді наші запити потрапляю в базу. На теперішньому етапі буде показано мінімальні і самі необхідні знання.

Етапи налаштування через **SQL Server Profiler**.

`Заходимо в програму` → `File` →`New Trace` → `Connect` → `Записуємо Trace Name` → `Run` *Рис. 1.1.3.*

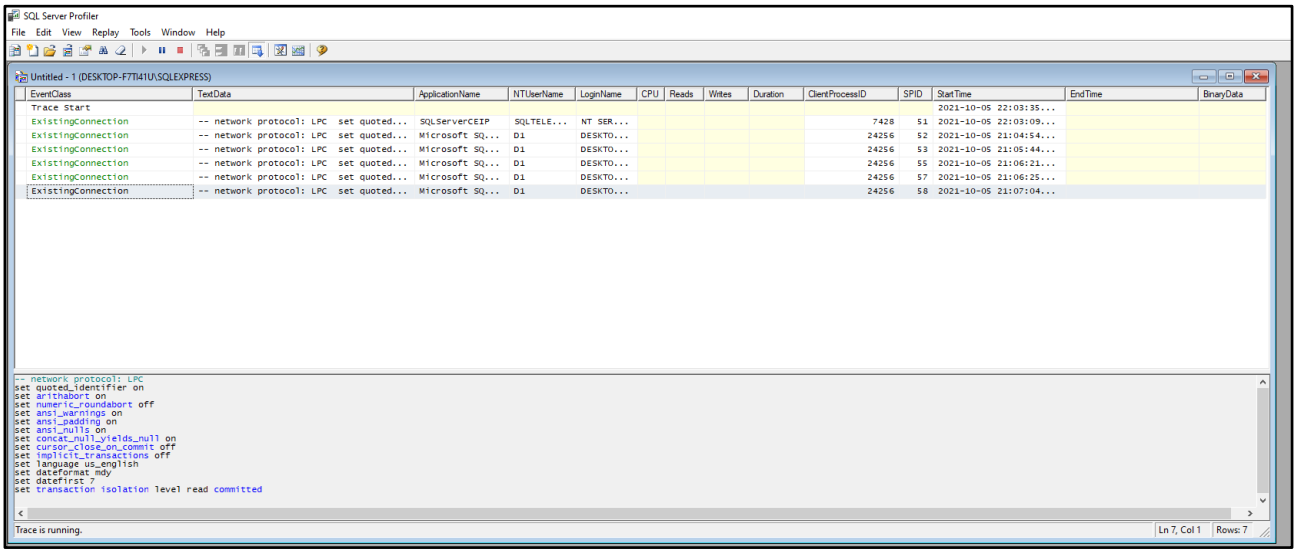

Рис. 1.1.3 – Зчитування логів з засобу Sql Server Profiler

Етапи налаштування через **LogTo**.

```
public class TestUniqueUserDbContext : DbContext
{
    public DbSet<UniqueUser> UniqueUsers { get; set; }

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
      {
        optionsBuilder
            .UseSqlServer(
                "Server=DESKTOP-F7TI41U\\SQLEXPRESS;Database=SusDb;Trusted_Connection=True"
            );
        optionsBuilder
            .LogTo(
                message => System.Diagnostics.Debug.WriteLine(message)
                , new[] { RelationalEventId.CommandExecuted }
            );
    }
}
```

Додає логування в вікно відлагодження, а також конкретизує для перегляду тільки команд *Рис. 1.1.4*.

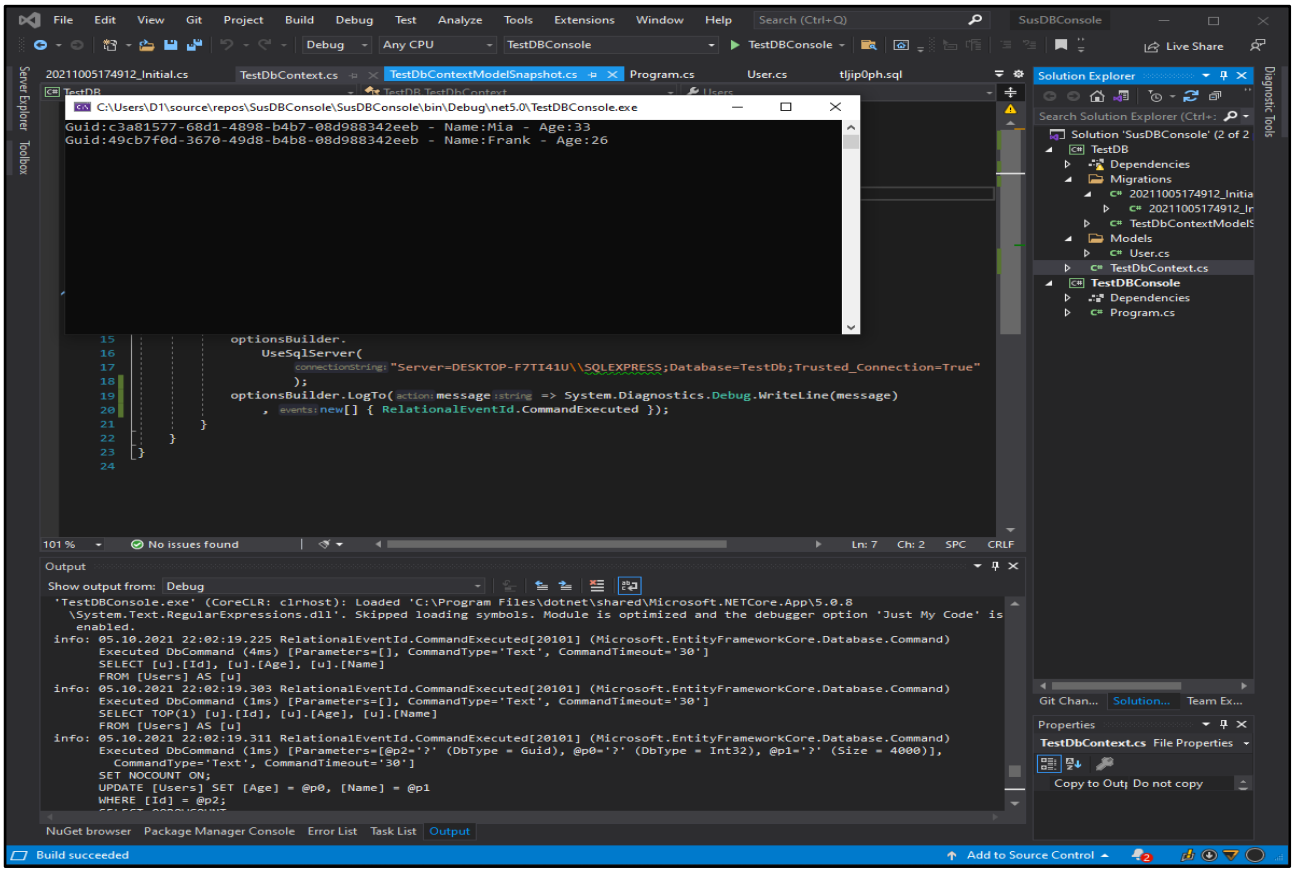

Рис. 1.1.4 – Зчитування логів з засобу консоль.

1.2 Проектування і конфігурація моделей

1.2.1 Fluent API та Data Annotation

EF Core використовує набір умов\правил для побудови таблиць на основі наданої моделі, такі маніпуляції виконуються за допомогою **Fluent API** та **Анотації даних – Data Annotation.**

Сутність (entity) – якщо опустити всі складні деталі то це клас який визначає правила, відношення, є проекцією нашого класу на базу даних. Представляє тип об’єкта який має зберігатись в базі даних. Кожна таблиця має представляти одну сутність.

Fluent API

Це набір методів які визначають відносини між класами і їхніми властивостями.

Перевизначається метод **OnModelCreating** і далі використовуємо **ModelBuilder** для конфігурації моделі. Це самий потужний метод конфігурації і дозволяє впроваджувати налаштування без модифікації класу сутності. Також **Fluent API** йде першим по приорітету, і таким чином перевизначає **анотацію даних.**

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUser>()
        .Property(b => b.UserName)
        .IsRequired(false); // Will override Required attribute from Data Annotation
}
```

В даному випадку застосована **IsRequired(false) –** тобто властивість Name не є обов'язковою для визначення. В БД буде визначення NULL.

Data Annotation

Налаштування класів за допомогою атрибутів. Іде далі по ієрархії після **Fluent API,** для застосування буде достатньо додати до властивості в моделі атрибут, в нашому випадку це **Required**, те саме що і **IsRequired()**

```
public class UniqueUser
{
    public int UniqueUserId { get; set; }
    [Required] //Required attribute
    public string UniqueUserName { get; set; }
}
```

1.2.2 Додавання та виключення сутностей з моделі.

Додавання

За замовчуванням всі типи які були додані в **DbSet**, будуть додані в модель. Але окрім цього, в модель будуть включатися і ті сутності посилання на які будуть в типах які вже потрапили до моделі.

Наприклад у нас буде визначено тип Blog і в ньому буде визначено ще один тип BlogDetails.

```
public class UniqueUserBlog
{
    public int UniqueUserBlogId { get; set; }
    public UniqueUserBlogDetails UniqueUserDetails { get; set; }
}
```

```
public class UniqueUserBlogDetails
{
    public int UniqueUserBlogDetailsId { get; set; }
    public string UniqueUserDetails { get; set; }
}
```

В DbSet буде включено тільки Blog, але в Базу Даних потрапляють обидві *Рис. 1.1.5*.

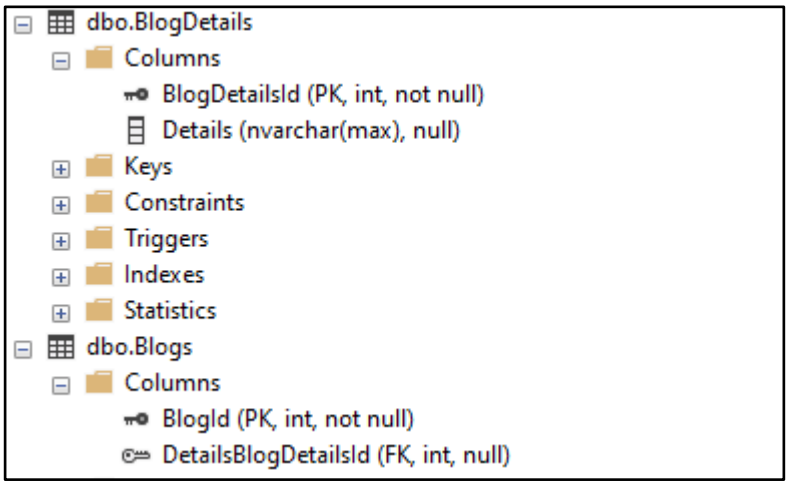

Рис. 1.1.5 – Вигляд бази даних Blog.

1.2.3 Виключення таблиць та їх елементів

За допомогою **Data Annotation**

```
public class UniqueUserBlog
{
    public int UniqueUserBlogId { get; set; }
    public UniqueUserBlogDetails UniqueUserDetails { get; set; }
    [NotMapped]
    public UniqueUser UniqueUserOwner { get; set; }
}
```

За допомогою **Fluent API**

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserBlog>().Ignore(b => b.Owner);
}
```

Також виключення можна застосувати не тільки до поля а і до всього класу.

За допомогою **Data Annotation**

```
[NotMapped]
public class UniqueUser
{
    public int UniqueUserId { get; set; }
    public string UniqueUserName { get; set; }
}
```

За допомогою **Fluent API**

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<UniqueUser>();
}
```

Даний приклад має виключно об’єктну реалізацію.

1.2.4 Ім’я таблиці та схеми

Такі маніпуляції дозволяє робити атрибут **Table** в його конструкторі є неіменований параметр **Name – ім’я таблиці**, та іменований **Scheme - ім’я схеми**. Схожі параметри має метод **ToTable.** Якщо не визначати поле **scheme,** то таблиця буде додана в схему за замовчуванням.

Data Annotation

```
[Table("usersUnique", Schema = "bloggingUnique")]
public class UniqueUser
```

```
{
    public int UniqueUserId { get; set; }
    public string UniqueUserName { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().ToTable("usersUnique", "bloggingUnique");
}
```

1.2.5 Ім’я властивості

Data Annotation

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().ToTable("usersUnique", "bloggingUnique");
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>().ToTable("usersUnique", "bloggingUnique");
    modelBuilder.Entity<User>().Property(u => u.UserId).HasColumnName("uniqueUserId");
}
```

Database realization

```
CREATE TABLE bloggingUnique.usersUnique (
    ...fields
);
```

1.2.6 Тип даних

Ще одна властивість атрибуту **Column** – **TypeName,** так можна визначити тип властивості, також є метод **HasColumnType**.

Data Annotation

```
public class UniqueUserBlog
{
    public int UniqueUserBlogId { get; set; }
    [Column(TypeName = "varchar(1500)")]
    public string BlogDetails { get; set; }

    [Column ( TypeName = " decimal(5, 2)" )]
    public decimal BlogRating { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserBlog>(
        eb =>
        {
            eb.Property(b => b.BlogDetails ).HasColumnType("varchar(1500)");
            eb.Property(b => b.BlogRating ).HasColumnType("decimal(5, 2)");
        });
}
```

Database realization

```
CREATE TABLE UniqueUserBlog (
    BlogDetails VARCHAR(1500) NULL,
    BlogRating  decimal(5, 2) NOT NULL,
);
```

1.2.7 Обмеження Required и MaxLength

За конвенцією, всі властивості які можуть бути null, будуть **Optional**, всі інші будуть **Required**, наприклад типи (int, decimal, bool) буде визначено як **Required**, а типи (int?, decimal?, bool?) будуть визначені **Optional**. Типи за посиланням же за замовчуванням завжди **Optional**. В C# 8 є можливість змінити це, а тобто застосувати **Nullable reference types**, і тепер string? - Optional a string – Required.

Data Annotation

```
public class UniqueUserBlog
{
    public int UniqueBlogId { get; set; }
    public int? UniqueBlogCounter { get; set; }

    [Required]
```

```
    public string UniqueBlogDetails { get; set; }
}
```

Fluent API

```
protectedoverride void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserBlog>()
        .Property(b => b.UniqueBlogDetails )
        .IsRequired();
}
```

Максимальний розмір контролюється атрибутом **MaxLength**, або методом **HasMaxLength.** Доповнимо модель цим прикладом.

Data Annotation

```
public class UniqueUserBlog
{
    public int UniqueBlogId { get; set; }
    public int? UniqueBlogCounter { get; set; }

    [MaxLength(1500)]
    public string UniqueBlogDetails { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserBlog>()
        .Property(b => b.UniqueBlogDetails )
        .HasMaxLength(1500)
        .IsRequired();
}
```

Database realization

```
CREATE TABLE UniqueUserBlog (
    BlogDetails VARCHAR(1500) NULL
);
```

1.2.8 Індекси

Індекси можна створювати за допомогою атрибуту **Index,** він має три параметри propertyName - ім’я властивості, IsUnique - чи буде значення поля унікальним та Name – ім’я самого індексу, так же в **Fluent API** присутні методи з аналогічними властивостями. За замовчуванням на Первинний ключ у любій таблиці ставиться кластеризований індекс.

Data Annotation

```
[Index(nameof(FirstName), IsUnique = true, Name = "Index_FUniqueUserName")]
public class UniqueUser
{
    public int UniqueUserId { get; set; }
    public string UniqueUserFirstName { get; set; }
    public string UniqueUserLastName { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserUser>()
        .HasIndex(b => b.UniqueUserFirstName)
        .IsUnique()
        .HasDatabaseName("Index_FUniqueUserName");

}
```

Database realization

```
CREATE INDEX Index_FUniqueUserName ON UniqueUserFirstName(UniqueUserFirstName);
```

Також можна складати індекси з декількох полів

Data Annotation

```
[Index(nameof(Id), nameof(UniqueUserBlogName))]
public class UniqueUserBlog
{
```

```
    public int UniqueUserBlogId { get; set; }
    public string UniqueUserBlogName { get; set; }
    public string UniqueUserBlogDescription { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUser>()
        .HasIndex(b => b.UniqueUserFirstName)
        .IsUnique()
        .HasDatabaseName("Index_FUniqueUserName");

    modelBuilder.Entity<UniqueUserBlog>()
        .HasIndex(b => new { b.UniqueUserId, b.UniqueUserBlogName })
        .HasFilter("[UniqueUserBlogName] IS NOT NULL");
}
```

Database realization

```
CREATE UNIQUE INDEX Index_FUniqueUserName ON UniqueUser(UniqueUserFirstName ASC, UniqueUserId ASC );
```

1.2.9 Ключі

За замовчуванням ключем стає поле яке має назву Id або <Ім’я класу>Id. Для установки використовують атрибут **key** або метод HasKey, за допомогою методу можна створювати також ключі які складаються з декількох властивостей.

Data Annotation

```
public class UniqueUserPhone
{
    [Key]
    public int UniqueUserPhoneId { get; set; }
    public string UniqueUserPhoneModel { get; set; }
    public string UniqueUserPhoneSeries { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueUserPhonePhone>()
        //.HasKey(p => p.UniqueUserPhoneId)
        .HasKey(b => new { b.UniqueUserPhoneId, b.UniqueUserPhoneModel }) // composite key
        .HasName("PK_UniqueUserPhonePhone_Id");
}
```

Database realization

```
CREATE TABLE UniqueUserPhone(
      UniqueUserPhoneId int NOT NULL PRIMARY KEY,
      UniqueUserPhoneModel varchar(255) NULL,
      UniqueUserPhoneSeries varchar(255) NULL
);
```

1.2.10 Відношення між моделями

Залежна сутність (dependent entity) – сутність в складі якої є **дочірній ключ (foreign key**), залежна ланка у відносинах.

Батьківська сутність(principal entity) - – сутність в складі якої є **первинний або альтернативний ключі (primary/alternate keys).**

Тобто виділяють головну, або батьківську сутність (principal/master), і залежну **(foreign key / child table).** Дочірня таблиця залежить від головної.

Суперключ (Superkey) – комбінація атрибутів(властивостей), яка є унікальним ідентифікатором для кожного рядку таблиці.(може бути як комбінацією так і одиничним)

Потенціальний ключ(candidate key) – ключ який задовольняє собою мінімальні умови

1) В кожному рядку є своє унікальне значення.

2) Має мінімальний можливий набір атрибутів, і не може бути скороченим.

3) Не може бути null

Первинний ключ (primary key) – є ідентифікатором рядків в таблиці, має задовольняти умов потенціального ключа.

Батьківський ключ (principal key) первинний ключ в батьківській таблиці.

Альтернативний ключ (alternate key) – якщо є декілька потенціальних ключів, то всі які не стали первинним є **альтернативними**.

Побічний ключ (foreign key) – властивість у залежній таблиці яка використовується для зберігання батьківського ключа.

Навігаційна властивість (navigation property) – властивість визначена в батьківській і/або в залежній яка є представленням пов’язаної таблиці.

Навігаційна властивість колекція (collection navigation property) – властивість яка посилається на багато пов’язаних з нею сутностей.

Навігаційна властивість посилання (reference navigation property) – посилається на одну сутність.

Зворотна навігаційна властивість (inverse navigation property) – відношення однієї властивості до другої яка знаходиться в протилежній сутності відносин.

Self-referencing relationship – сутність яка посилається сама на себе.

Приклад:

В данному прикладі показаний зв’язок один-до-багатьох між сутностями **Blog** і **Admin, Blog** є батьківською сутністю, а **Admin** є залежною.

У вигляді Сутностей EF

```
public class UniqueUserBlog // Principal Entity
{
    public int UniqueUserBlogId { get; set; } //Principal/Primary key
    public string UniqueUserBlogName { get; set; }
    public string UniqueUserBlogDescription { get; set; }
    public List<UniqueUserAdmin> UniqueUserAdmins { get; set; } // Collection navigation property
}

public class UniqueUserAdmin // Dependent Entity
{
    public int UniqueUserAdminId { get; set; } //Primary key and not Principal
    public string UniqueUserName { get; set; }
    public int UniqueUserBlogId { get; set; } //Foreign key
    public UniqueUserBlog UniqueUserBlog { get; set; } // Invert navigation property of Blog.Admins
&& reference n. p.
}
```

У вигляді SQL скрипту

```
CREATE TABLE [Blogs] (
    [UniqueUserBlogId] int NOT NULL IDENTITY,
    [UniqueUserBlogName] nvarchar(max) NULL,
    [UniqueUserBlogDescription] nvarchar(max) NULL,
    CONSTRAINT [PK_UniqueUserBlogs] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [UniqueUserAdmins] (
    [UniqueUserAdminId] int NOT NULL IDENTITY,
    [Name] nvarchar(max) NULL,
    [UniqueUserBlogId] int NOT NULL,
    CONSTRAINT [PK_UniqueUserAdmins] PRIMARY KEY ([AdminId]),
    CONSTRAINT [FK_Admins_Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blogs] ([BlogId]) ON
DELETE CASCADE
);
```

В сутності **Blogs**

первинний ключ - як PK_Blogs тобто PK_[Ім’я сутності]

В сутності **Admins**

первинний ключ - як PK_Admins тобто PK_[Ім’я сутності]

побічний ключ - FK_[Ім’я сутності]_[Ім’я батьківської сутності]_[ім’я первинного ключа батьківської сутності]

За замовчуванням:

Зв'язок буде створений коли в одному з типів буде присутня **навігаційна властивість,** яка в свою чергу не може бути склярним типом(тип представляється одним полем).

Зв’язки будуть завжди прив’язуватись до **первинного ключа** в **батьківській сутності**.

Побічний ключ і/або **навігаційну властивість** в залежній сутності **можна опустити**. В такому випадку їх буде згенеровано автоматично.

Також є можливість додати більше чим одну навігаційну властивість в зв'язку сутностей. Це можна зробити за допомогою атрибуту **InverseProperty.**

```
public class UniquePost
{
    public int UniqueArticlId { get; set; }
    public string UniqueName{ get; set; }
    public string UniqueContent { get; set; }

    public int UniqueAuthorUserId { get; set; }
    public UniqueUser UniqueAuthor { get; set; }

    public int UniqueContributorUserId { get; set; }
    public UniqueUser UniqueContributor { get; set; }
}

public class UniqueUser
{
    public string UniqueUserId { get; set; }
    public string UniqueFirstName { get; set; }
    public string UniqueLastName { get; set; }

    [InverseProperty("UniqueAuthor")]
    public List<UniquePost> UniqueAuthoredPosts { get; set; }

    [InverseProperty("UniqueContributor")]
    public List<UniquePost> UniqueContributedToPosts { get; set; }
}
```

1.2.11 Foreign Key

Data Annotation

```
public class UniqueBlogs
{
    public int UniqueBlogsId { get; set; }
    public string UniqueBlogsName { get; set; }
    public string UniqueBlogDescriptions { get; set; }
    public int UniqueBlogAdminsKey { get; set; }
    [ForeignKey("UniqueBlogAdminsKey")]
    public List<UniqueBlogAdmins>UniqueBlog Admins { get; set; }
}

public class UniqueBlogAdmin
{
    public int UniqueBlogAdminsId { get; set; }
    public string UniqueBlogName { get; set; }
    public int UniqueBlogAdminsKey { get; set; }
    public UniqueBlog UniqueBlog { get; set; }
}
```

Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueBlogAdmins>()
        .HasOne(p => p.UniqueBlogs)
        .WithMany(t => t.UniqueBlogAdminss)
        .HasForeignKey(p => p.UniqueBlogAdminsKey)
        .HasConstraintName("ForeignKey_Admin_Blog")
        .HasPrincipalKey(p => p.UniqueBlogBlogsId);
}
```

На прикладах видно що **Fluent API є** гнучкішим чим **Data Annotation**, ми можемо вибирати хто буде **Батьківським ключем**, а хто **побічним**, так же вибрано ім’я побічного ключа.

1.2.12 Відношення Один-до-одного

В даному випадку об’єкт однієї сутності можна співвідноситись тільки до одного із іншої сутності. Наприклад **один користувач – один профіль.**

Зазвичай це можна представити як одна велика таблиця, яку розбили на декілька маленьких, де одна зберігає дані які використовуються частіше, а в інших, дочірніх, будуть зберігатись дані які використовуються рідше.

```
public class UniqueUsers
{
    public int UniqueUserId { get; set; }
    public string UniqueUserLogins { get; set; }
    public string UniqueUserPasswords { get; set; }

    public UniqueUserProfile UniqueUserProfiles { get; set; }
}

public class UniqueUsersProfile
{
    public int UniqueUsersProfileId { get; set; }

    public string UniqueUserNames { get; set; }
    public int UniqueUserAges { get; set; }

    public int UniqueUserUsersId { get; set; }
    public User UniqueUserUsers { get; set; }
}
```

В даному випадку сутність **UniqueUser** є батьківською, а **UniqueUserProfile** дочірньою. EF Core вибирає так, якщо знайдено властивість ім’я якої збігається з одним з нижче перерахованих, то ця властивість буде вважатись залежним ключем.

- [navigation prop name][principal key prop name]
- [navigation prop name]Id
- [princpl entity name][princpl key prop name]
- [princpl entity name]Id

Також можна це детально конфігурувати.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<UniqueUser>()
        .HasOne(u => u.UniqueUserProfile)
        .WithOne(p => p.UniqueUser)
        .HasForeignKey<UniqueUserProfile>(p => p.UniqueUserId);
}
```

1.2.13 Відношення Багато-до-багатьох

Прикладом цього відношення може слугувати студенти і курси, один студент може мати багато курсів, а один курс може мати багато студентів, або статті та теги, одна стаття може мати багато тегів, а один тег може буди в багатьох статтях.

У Entity Framework Core 5 З'явилась можливість створювати подібні відношенні автоматично.

```
public class UniqueTag
{
    public int UniqueTagId { get; set; }
    public string UniqueTagName { get; set; }
    public List<UniqueArticle> UniqueArticles { get; set; } = new List<UniqueArticle>();
}
public class UniqueArticle
{
    public int UniqueArticleId { get; set; }
    public string UniqueArticleName { get; set; }
    public List<UniqueTag> UniqueTags { get; set; } = new List<UniqueTag>();
}
```

І цього буде достатньо щоб створити зв'язок багато-до-багатьох.

```
CREATE TABLE [UniqueArticles] (
    [UniqueArticlesId] int NOT NULL IDENTITY,
    [UniqueArticlesName] nvarchar(max) NULL,
    CONSTRAINT [PK_UniqueArticles] PRIMARY KEY ([Id])
);
GO

CREATE TABLE [UniqueTags] (
    [UniqueTagsId] int NOT NULL IDENTITY,
    [UniqueTagsName] nvarchar(max) NULL,
    CONSTRAINT [PK_UniqueTags] PRIMARY KEY ([Id])
);
GO

CREATE TABLE [TaggedArticles] (
    [UniqueArticlesId] int NOT NULL,
    [UniqueTagsId] int NOT NULL,
    CONSTRAINT [PK_TaggedArticles] PRIMARY KEY ([ArticlesId], [TagsId]),
    CONSTRAINT [FK_TaggedArticles_Articles_ArticlesId] FOREIGN KEY ([ArticlesId]) REFERENCES
[Articles] ([Id]) ON DELETE CASCADE,
    CONSTRAINT [FK_TaggedArticles_UniqueTags_TagsId] FOREIGN KEY ([TagsId]) REFERENCES [Tags] ([Id])
ON DELETE CASCADE
);
GO
```

У базі даних створено таблицю ArticleTag, а всередині Entity Framework буде створено Dictionary<string, object> (така конструкція має назву propery bag)

Також можна вносити додаткові конфігурації

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueTag>()
        .HasMany(c => c.UniqueArticles)
        .WithMany(s => s.UniqueTags)
        .UsingEntity(j => j.ToTable("UniqueTaggedArticles"));
}
```

До EF Core 5 треба робити більше маніпуляцій, а тобто додати до моделі сутність ArticleTag

```
public class UniqueArticle
{
    public int UniqueArticleId { get; set; }
    public string UniqueArticleTitle { get; set; }
    public string UniqueArticleContent { get; set; }

    public List<UniqueArticleTag> UniqueArticleTags { get; set; }
}

public class UniqueTag
{
    public string TagId { get; set; }

    public List<UniqueArticleTag> UniqueArticleTags { get; set; }
}

public class UniqueArticleTag
{
    public DateTime UniquePublicationDate { get; set; }

    public int UniqueArticleId { get; set; }
    public UniqueArticle UniqueArticle { get; set; }

    public string UniqueTagId { get; set; }
    public UniqueTag UniqueTag { get; set; }
}
```

також необхідно вносити додаткові налаштування через Fluent API.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<UniqueArticleTag>()
        .HasKey(t => new { t.UniqueArticleId, t.UniqueTagId });

    modelBuilder.Entity<UniqueArticleTag>()
        .HasOne(pt => pt.UniqueArticle)
        .WithMany(p => p.UniqueArticleTags)
        .HasForeignKey(pt => pt.UniqueArticleId);

    modelBuilder.Entity<UniqueArticleTag>()
        .HasOne(pt => pt.UniqueTag)
        .WithMany(t => t.UniqueArticleTags)
        .HasForeignKey(pt => pt.UniqueTagId);
}
```

Фактично ми робимо пару зав’язків один-до-багатьох, і отримуємо такий самий результат як і в попередньому випадку але з ручним налаштуванням.

1.2.14 Відношення Один-до-багатьох

Найчастіше використовують саме цей вид зв’язку, прикладом може бути Блог і статті, один блог має декілька статей, або команда і гравці – в одній команді декілька гравців.

```
public class UniqueTeam
{
    public int UniqueTeamId { get; set; }
    public string UniqueTeamName { get; set; }
    public List<UniqueTeamPlayer> UniqueTeamPlayers { get; set; } // игроки команды
}

public class UniquePlayer
{
    public int UniquePlayerId { get; set; }
    public string UniquePlayerName { get; set; }

    public int UniqueTeamId { get; set; }
    public UniqueTeam UniqueTeam { get; set; }
}
```

1.3 Запити Linq та Fluent Interface

Linq to Entities – технологія яка використовується для вилучення інформації з бази даних. Конкретно дана реалізація запитів транслюється в визначений запит **SQL**, як і обговорено раніше база даних сприймає тільки запити на **SQL.**

Також запити можна будувати через **методи розширення.**

В якості моделі буде використано попередній приклад, і заповнимо її наступними даними.

```
using (var db = new TestUniqueUserDbContext())
{

    var red = new UniqueTeam { Name = "Red" };
    var green = new UniqueTeam { Name = "Green" };
    db.UniqueTeams.AddRange(red, red);

    var tom = new UniquePlayer { Name = "Tom", Birthday = new DateTime(2000, 6, 21), UniqueTeam = red
};
    var denis = new UniquePlayer { Name = "Denis", Birthday = new DateTime(2003, 6, 11), UniqueTeam =
red };
    var frank = new UniquePlayer { Name = "Frank", Birthday = new DateTime(2002, 1, 12), UniqueTeam =
red };
    var chloe = new UniquePlayer { Name = "Chloe", Birthday = new DateTime(2001, 2, 25), UniqueTeam =
green };
    var zoe = new UniquePlayer { Name = "Zoe", Birthday = new DateTime(2001, 2, 25), UniqueTeam =
green };
    var terry = new UniquePlayer { Name = "Terry", Birthday = new DateTime(2001, 2, 25), UniqueTeam =
green };
    db.UniquePlayers.AddRange(tom, denis, frank, chloe, zoe, terry);
    db.SaveChanges();
}
```

Виходить 2 команди і 6 гравців.

1.2.1 Побудова запису

Запит будується за допомогою ключових слів from in where…

Побудова запиту через Linq

```
using (var db = new TestUniqueUserDbContext())
    {
        var uniquePlayers = (from UniquePlayer in db.UniquePlayers.Include(p => p.UniqueTeam)
            where uniquePlayer.UniqueTeam.UniqueTeamName == "Red"
            select player).ToList();

        foreach (var uniquePlayer in uniquePlayers)
            Console.WriteLine($"{uniquePlayer.UniquePlayerName}
({uniquePlayer.UniquePlayerBirthday.Date}) - {uniquePlayer.UniquePlayerTeam?.Name}");
    }
```

Побудова запиту через Fluent interface

```
using (var db = new TestUniqueUserDbContext())
{
    var uniquePlayers = db.UniquePlayers.FirstOrDefault(p => p.UniquePlayerName == "Denis");

    if (player != null)
        Console.WriteLine($"{uniquePlayer.UniquePlayerName} - {uniquePlayer.UniquePlayerTeam} -
{uniquePlayer.UniquePlayerId}");
}
```

В конкретному випадку ми виконуємо запит всіх гравців команди з в якій ім’я Red. Синтаксис є дуже схожим до запитів SQL. У випадку **Linq** ми вибираємо всіх гравців з таблиці Players, потім виконуємо фільтрацію через оператор **where**, і в кінці оператор **select** повертає вибрані значення в результуючу вибірку. Такі ж маніпуляції ми виконуємо і через метод розширення **where.**

Обидва приклади транслюються в однин і той же запит SQL

```
SELECT [p].[Id], [p].[Birthday], [p].[Name], [p].[TeamId], [t].[Id], [t].[Name]
FROM [Players] AS [p]
LEFT JOIN [Teams] AS [t] ON [p].[TeamId] = [t].[Id]
WHERE [t].[Name] = N'Red'
```

Якщо виконати цей запит то має показати наступну інформацію.

```
Tom (21.06.2000) - Red
Denis (11.06.2003) - Red
Frank (12.01.2002) - Red
```

РОЗДІЛ 2

«ОГЛЯД ІНСТРУМЕНТУ ДЛЯ РОЗРОБКИ ПРОГРАМ НА ПЛАТФОРМІ ASP.NET CORE»

2.1 Початок роботи з Web-Api

2.1.1 Основні проблеми в розробці Web-Api сервісу

- **Ненадійність мережі передачі інформації -** Всі інтеграційні рішення передбачають в своєму складі передачу інформації між пристроями, а вони в свою чергу між собою можуть містити великий шлях із мережевих пристроїв, каналів зв'язку і навіть проходити через декілька країн. Доставка такої інформації є комплексним процесом який на кожному етапі внутрішньої обробки створює незначні затримки, і таких маленьких затримок може бути велика кількість, вони в свою чергу можуть стати причиною втрати важливої інформації ^(все ж таки теперішнє суспільство трішки збалуване в плані якості подачі послуг програмного продукту, і якщо користувач стикнеться з незначним дискомфортом в його роботі, то це може послужить його відмовою від програми, або погрозою розправи в адрес розробників, або ще багато чого цікавого… Навіть здогадуватися страшно яку розправу готують в своїй уяві розгнівані користувачі, покоління добра і миру як не як).

- **Низька швидкість передачі інформації –** ситуація аналогічна до попереднього пункту, але проблеми попереднього пункту хаотичні, і складно піддаються контролю, а теперішній пункт в більшості випадків статичний а тобто його можна **автоматизувати** і звести не комфортність роботи з ним до мінімум.

- **Розбіжність між програмами –** інтеграційні рішення мають враховувати в собі присутність розбіжностей(платформи, формат даних, мови програмування)

- **Конкурентоспроможність в плані змін –** програми мають бути гнучкими при роботі в тандемі іншими програмами. Частіше всього зміни в одній системі тягнуть за собою наслідки в другій. Тому гарною практикою буде зменшувати їхню взаємодію.

А тепер перерахуємо можливі інструменти вирішення даних проблем.

- **Передача файлів (File Transfer) -** перша система створює файл, а друга читає його.

- **Спільна база даних (Shared Database) -** декілька додатків використовують спільну базу даних.

- **Віддалений виклик процедури (Remote Procedure Invocation) –** програма надає доступ до частини свого функціоналу методом виклику віддалених процедур.

- **Обмін повідомленнями –** Програма розміщує повідомлення в спільному каналі, який потім зачитується другим додатком.

Тобто що можна витягнути із цього короткого огляду, це те що в курсі буде розглядатися методи передачі інформації між додатками, які від тепер будуть називатись **сервісами.**

Основною платформоою на якій буде будуватись наш сервіс буде ASP.NET Core — це розширювана структура для створення служб, до яких можна отримати доступ у різних програмах з різних платформ, таких як Web, Windows, Linux, Android, ios та інші. Він працює за таким же принципом, як і веб-додаток ASP.NET MVC, за винятком що він надсилає дані як відповідь замість їх відображення.

ASP.NET Core підтримує створення REST (Representational State Transfer (передача представлення стану)) веб-API служб. Для обробки запитів веб-API використовує контролери. Контролери у веб-API — це класи, які походять від ControllerBase.

2.1.2 Основні типи запитів

GET: загрузити вказаний ресурс

POST: передача інформації користувача на серевер

PUT: залити інформацію на сервер

DELETE: видалити на стороні серверу

2.1.3 Класс контроллера

Веб-API складається з одного або кількох класів контролерів, які є похідними від ControllerBase. Шаблон проекту веб-API надає початковий WeatherForecastController:

```
[ApiController]
[Route("/SampleController")]
public class SampleController : ControllerBase
{
```

Процес розробки контроллеру є дещо схожим до розробки звичайних ASP додатків, наслідуємо від класу ControllerBase, і далі працюємо з атрибутами. Першим атрибутом що ми споглядаємо є ApiController, який додає до контролера додаткові функції.

2.1.4 Основні функції **ApiController**

Автоматична відповідь HTTP 400 - змушує автоматично виконуватись валідацію моделі, в подальшому можлива автоматичне повернення помилки 400. Тобто наступна перевірка не буде необхідною

```
if (!SampleModelState.IsValid)
{
    return BadRequest(SampleModelState);
}
```

Можливість використання **Binding source parameter inference -** визначає місце з якого параметри будуть забиратись. Єдиним який буде розглянуто, але не єдиним який існує буде розглянуто атрибут **FromBody -** він вказує на те що параметр або властивість мають бути взяті з тіла запиту**.**

```
[HttpPost("[action]", Name = "AddBasketItem")]
[ProducesResponseType(typeof(Basket), (int)HttpStatusCode.OK)]
public async Task<ActionResult<BasketItem>> AddBasketItem([FromBody] BasketItem basketItem)
```

Далі показано приклад із практичного завдання. Атрибут HttpGet означає те що для конкретного методу буде використано шаблон GET. Ми вказуємо тип який буде повернуто у відповіді а тобто Basket, і статус коди, У випадку наступного методу або відповідь NotFound, або відповідь OK і результат. Відповіді ми відправляємо через влаштовані в клас ControllerBase методи NotFound і OK.

```
[HttpGet("[action]", Name = "GetBasketItem")]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
[ProducesResponseType(typeof(Basket), (int)HttpStatusCode.OK)]
public async Task<ActionResult<Basket>> GetBasketItemById(Guid id)
{
    BasketItem resultBasketItem = null;
    await _dbContextFactory.UseDbContextAsync(async dbContext =>
    {
        resultBasketItem = await dbContext.BasketItems.FirstOrDefaultAsync(b => b.Id ==
id).ConfigureAwait(false);
    }).ConfigureAwait(false);
    if (resultBasketItem != null)
        return Ok(resultBasketItem);

    _logger.LogError($"BasketItem with Id:{id}, not found");
    return NotFound();
}
```

Статус коди які може повертати сервіс

- 1xx інформаційні. Теперішня робота серверу.
- 2xx Успіх
- 3xx перенаправлення
- 4xx помилка зі сторони клієнта
- 5xx помилка серверу

2.1.5 Використання Options

Шаблон options використовує класи для забезпечення строго типізованого доступу до груп пов'язаних параметрів. Коли налаштування конфігурації виділені за сценарієм на окремі класи, програма дотримується двох важливих принципів розробки програмного забезпечення:

Принцип розділення інтерфейсу (ISP) або інкапсуляція: сценарії (класи), залежать тільки від параметрів конфігурації.

Розділення проблем: налаштування для різних частин програми не залежать і не пов'язані одне з одним.

Опції також забезпечують механізм валідації даних конфігурації. В сервіс же впроваджувати їх можна за допомогою механізму Dependency Injection. Передаються вони у вигляді інтерфейсу IOption<T>, де T є наша опція.

Для прикладу взято клас AppSettingsOptions. Клас з основними налаштуваннями сервісу.

```
public class AppSettingsOptions
```

```
{
    public DbConnectionSettings DbConnectionSettings { get; set; }
}
```

```
public class DbConnectionSettings
{
    public string SampConnectionString { get; set; }
}
```

За стандартом у проекті створено файл appsetting.Development.json, там і будуть зберігатись опції.

```
"AppSettings":
{
  "DBConnectionSettings":
  {
    "SampConnectionString": "Server= SQLEXPRESS;Database=BasketCatalog;Trusted_Connection=True"
  }
}
```

Далі необхідно зареєструвати опції в методі ConfigureService

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<AppSettingsOptions>(_configuration.GetSection("AppSettings"));
```

РОЗДІЛ 3

«РОЗРОБКА ПРАКТИЧНОЇ ЧАСТИНИ »

Лабораторна робота №1

Тема: Знайомство з Entity Framework Core

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Магазину.

В проекті створити сутності:

- Користувач
- кошик товарів
- Товар

Відношення між сутностями:

- Користувач може мати одну корзину (один до одного)
- кошик може мати безліч товарів(один до багатьох)

Обмеження і додаткові умови:

- Кожна сутність повинна мати ключем тип Guid.
- Сутність користувач повинна мати обов’язкове поле з ім’ям користувача і обмеженням в 140 символів. Використати метод **HasMaxLength** або атрибут **MaxLength**.

- В кожній сутності необхідно передбачити поля DateTime які уособлюють час зміни і створення сутності в них необхідно передбачити заповнення поля за замовчування використовуючи властивості **Fluent API** або **Data Annotation**.

1. Розробка проекту та завантаження необхідних бібліотек для роботи з фреймворком.

Процес створення проекту.

при створенні проекту вибираємо в пошуку Class Library *Рис. 3.1.1*.

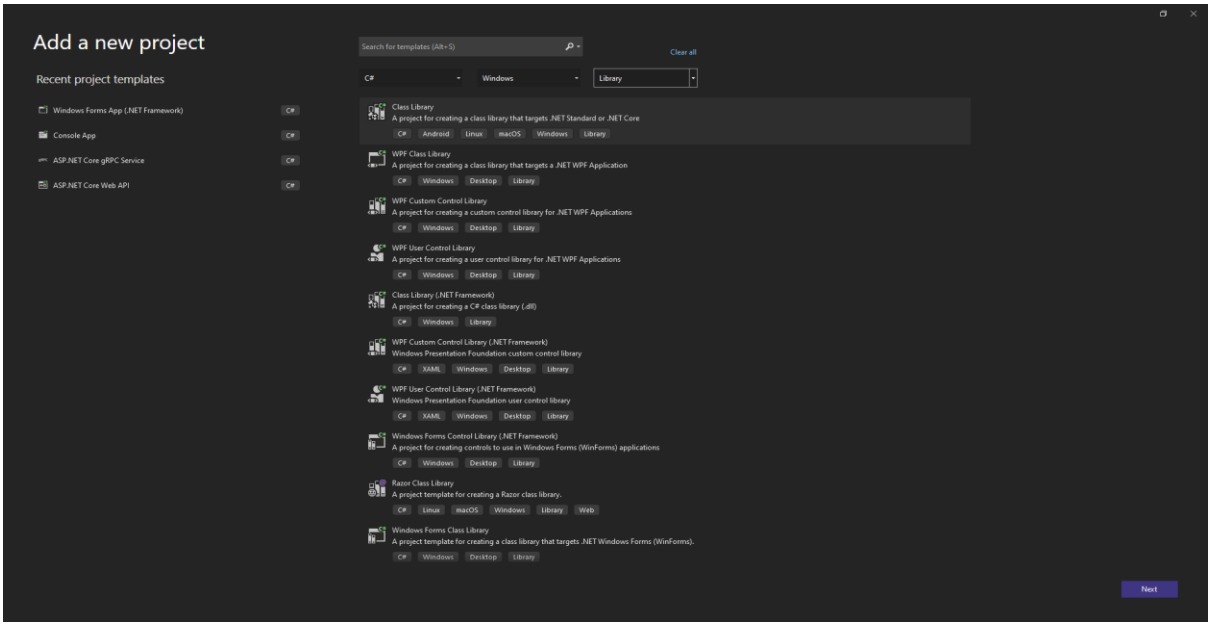

Рис. 3.1.1 – Створення проекту Class Library.

Процес інсталяції пакетів EF Core.

Необхідно знайти в пункт меню Manage Nuget Packages *Рис. 3.1.2.*

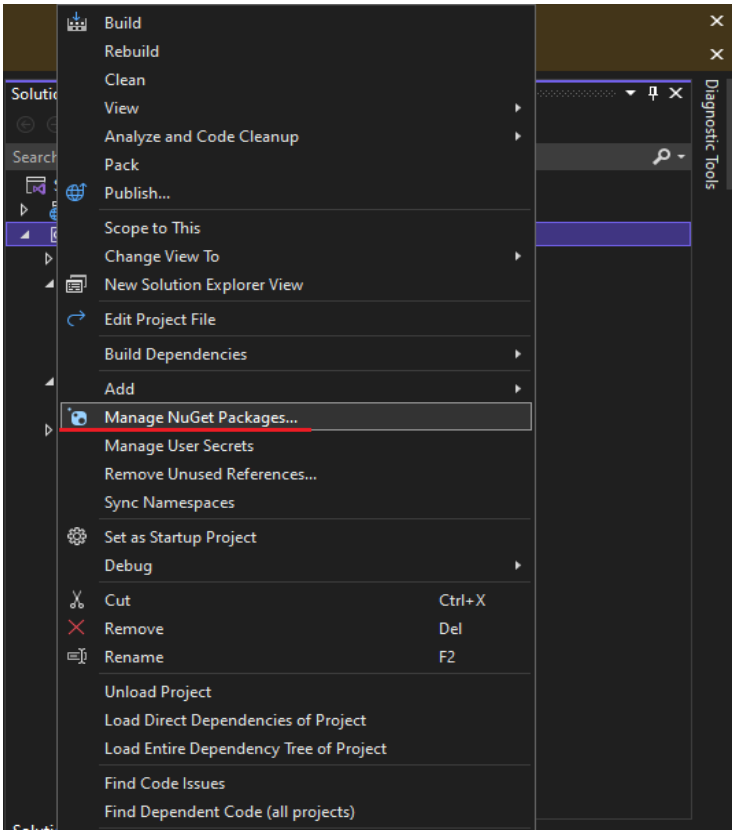

Рис. 3.1.2 – Додавання необхідного nuget пакету.

Перейти в Browse виконати пошук потрібного фреймворку, після чого натиснути на Install *Рис. 3.1.3*.

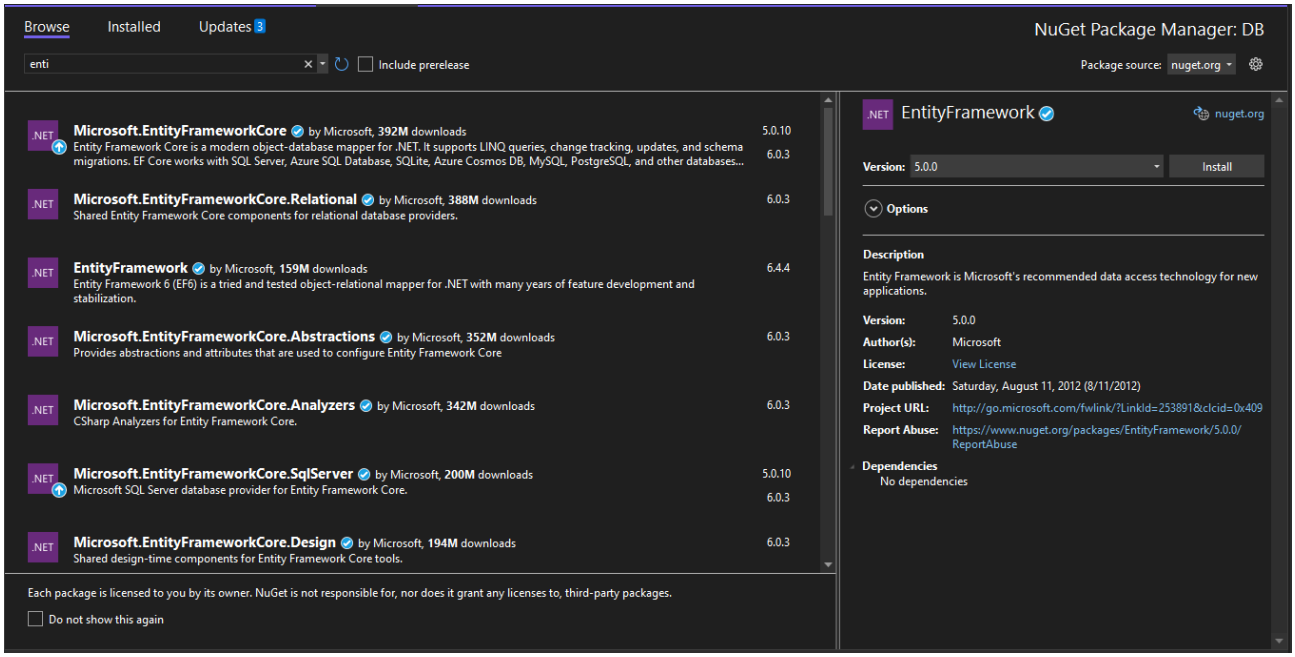

Рис. 3.1.3 – Додавання необхідного nuget пакету (пошук).

Список пакетів з якими потрібно буде працювати.

```
~Microsoft.EntityFrameworkCore
~Microsoft.EntityFrameworkCore.SqlServer
~Microsoft.EntityFrameworkCore.Tools.
```

2. Визначення об’єктного представлення моделей.

кошик товарів

```
public class UniqueBasket
{
    [Key]
    public Guid UniqueBasketId { get; set; }

    public DateTime CreatedAtUtc { get; set; }

    public DateTime ModifiedAtUtc { get; set; }
    public long CreatorPhone { get; set; }
    public bool IsSystem { get; set; }

    public virtual ICollection<BasketItem> BasketItems { get; set; }
}
```

Товар

```
    public class UniqueBasketItem
    {
        [Key]
        public Guid UniqueBasketItemId { get; set; }

        public DateTime CreatedAtUtc { get; set; }

        public DateTime ModifiedAtUtc { get; set; }
        public string UniqueBasketItemBarcode { get; set; }
        public string UniqueBasketItemName { get; set; }
        public string UniqueBasketItemComment { get; set; }
        public double UniqueBasketItemPrice { get; set; }

        public Guid UniqueBasketItemBasketId { get; set; }

        [ForeignKey("UniqueBasketItemBasketId")]
        public virtual UniqueBasketItemBasket UniqueBasketItemBasket { get; set; }
    }
```

Користувач

```
    public class UniqueUser
    {
        [Key]
        public Guid UniqueUserId { get; set; }
        [Required]
        public string UniqueUserName { get; set; }
        [Required]
        public int UniqueUserPhone { get; set; }
        public DateTime CreatedAtUtc { get; set; }
        public DateTime ModifiedAtUtc { get; set; }
        public UniqueUserBasket UniqueUserBasket { get; set; }

        public Guid? UniqueUserBasketId { get; set; }
    }
```

3. Розробка класу Context

В класі визначаємо поля DBSet під моделі які хочемо додати в базу даних, і виконуємо додаткову конфігурацію сутностей.

```
public class BasketCatalogDbContext : DbContext
{
    public BasketCatalogDbContext(DbContextOptions<BasketCatalogDbContext> options)
        : base(options)
    {
    }

    public virtual DbSet<UniqueBasket> UniqueBaskets { get; set; }

    public virtual DbSet<UniqueBasketItem> UniqueBasketItems { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {

    }

}
```

4. Визначення відношень між сутностями:

- Користувач може мати одну корзину (один до одного)

В методі OnModelCreating визначаємо такі відношення оскільки ми визначили поля UserId і BasketId як optional.

```
    modelBuilder.Entity<UniqueUser>().HasOne(t => t.UniqueBasket)
        .WithOne(t => t.UniqueUser)
        .HasForeignKey<UniqueBasket>(t => t.UniqueUserId);

    modelBuilder.Entity<UniqueBasket>().HasOne(t => t.UniqueUser)
        .WithOne(t => t.UniqueBasket)
        .HasForeignKey<UniqueUser>(t => t.UniqueBasketId);
```

- кошик може мати безліч товарів (один до багатьох)

Відношення встановлюються автоматично.

5. Визначення обмежень і додаткових умов:

- Кожна сутність повинна мати ключем тип Guid.

```
public class UniqueBasketItem
    {
        [Key]
        public Guid UniqueBasketItemId { get; set; }
```

Сутність користувач повинна мати обов'язкове поле з ім'ям користувача і обмеженням в 140 символів. Використати метод **HasMaxLength** або атрибут **MaxLength**.

```
        entity.Property(e => e.UniqueBasketItemName)
            .HasMaxLength(140)
            .IsRequired();
```

- В кожній сутності необхідно передбачити поля DateTime які уособлюють час зміни і створення сутності в яких необхідно передбачити заповнення поля за визначенням використовуючи властивості **Fluent API** або **Data Annotation**.

```
    entity.Property(e => e.CreatedAtUtc)
        .HasColumnType("smalldatetime")
        .HasDefaultValueSql("(getutcdate())");
```

```
    entity.Property(e => e.ModifiedAtUtc)
        .HasColumnType("smalldatetime")
        .HasDefaultValueSql("(getutcdate())");
```

6. Створення міграції

Для того щоб створити міграцію необхідно в **Package Manager Console** ввести команду

```
PM> Add-Migration Initial
```

Де **Initial** це назва яку користувач **самостійно** надає конкретній міграції.

Після цього буде автоматично створено каталог **Migrations** *Рис. 3.6.1.*

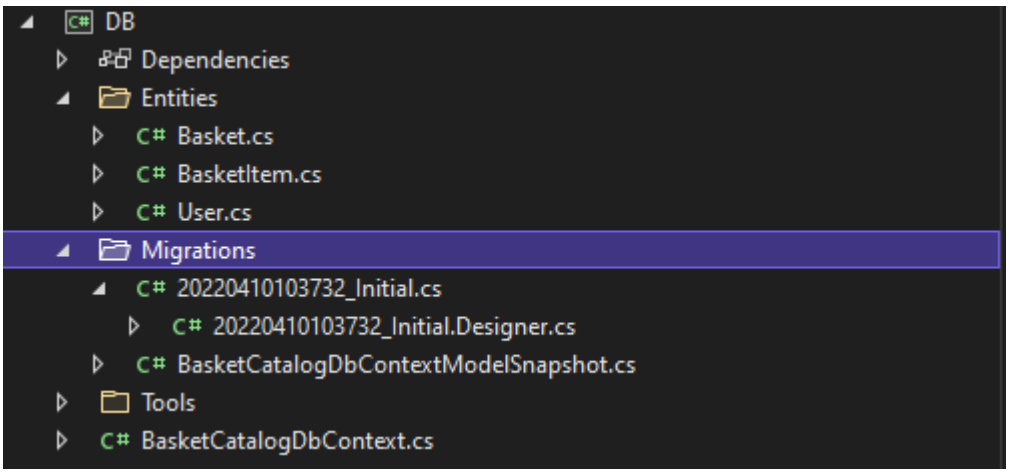

Рис. 3.6.1 – Представлення бази даних в контексті проекту.

Для того щоб застосувати міграцію необхідно ввести команду

```
PM> Update-Database
```

В результаті створено базу TestDb з таблицею Users *Рис. 3.6.2*.

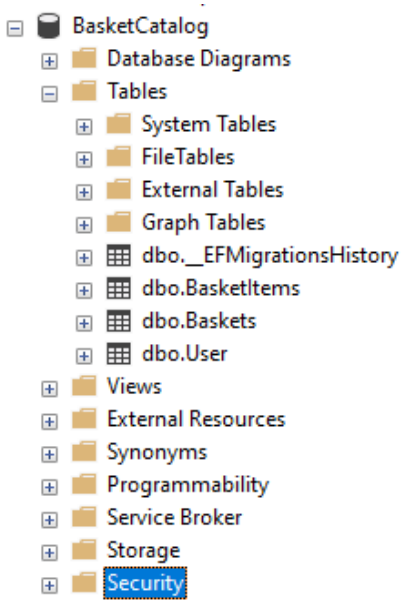

Рис. 3.6.2 – Представлення бази даних.

7. Демонстрація роботи з сутностями

Розробити консольну програму і продемонструвати додавання і зчитування даних в моделі BasketItem, Basket, User.

Додавання:

```
using var db = new BasketCatalogDbContext();
// Add
var userMia = new User
{
    Id = Guid.NewGuid(),
    Name = "Mia",
    Phone = 14555,
    UniqueBasket = new UniqueBasket()
    {
        Id = Guid.NewGuid(),
        UniqueBasketItems = new List<UniqueBasketItem>()
        {
            new ()
            {
                Id = Guid.NewGuid(),
                Name = "Cabbage",
                Comment = "It is Cabbage",
                Price = 11.50
            },
            new ()
            {
                Id = Guid.NewGuid(),
                Name = "Raspberry",
                Comment = "It is Raspberry not Rasberry",
                Price = 14.50
```

```
                }
            }
        }
    };

    db.UniqueUser.Add(userMia);
    db.SaveChanges();
```

Зчитування:

```
    var miaUser = db.UniqueUser.FirstOrDefault(p => p.Name == "Mia");

    Console.WriteLine($"Guid:{miaUser?.Id} - Name:{miaUser?.Name} - Age:{miaUser?.Phone} \n");
    Console.WriteLine($"BasketGuid:{miaUser?.Basket.Id} --\n");
    foreach (var basketItem in miaUniqueUser?.UniqueBasket?.UniqueBasketItems)
    {
        Console.WriteLine($"Guid:{basketItem.UniqueBasketId} Name:{basketItem.UniqueBasketName}
Price:{basketItem.Price}");
    }
```

Порівняння результату з даними в базі *Рис. 3.7.1 - 3.7.2.*

Консольний додаток

```
Guid:8c881ced-c5e7-476c-ab20-60871c09bc60 - Name:Mia - Age:14555

BasketGuid:0e6f4dd7-5c99-48fb-a7e2-18a3e0f2baa0 --

Guid:0e6f4dd7-5c99-48fb-a7e2-18a3e0f2baa0 Name:Cabbage Price:11,5
Guid:0e6f4dd7-5c99-48fb-a7e2-18a3e0f2baa0 Name:Raspberry Price:14,5
```

Рис. 3.7.1 – Демонстрація результату в консолі.

База даних

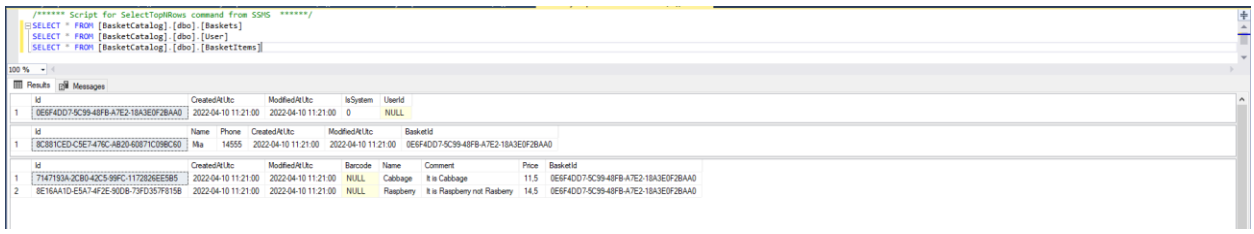

Рис. 3.7.2 – Демонстрація результату в базі даних.

Лабораторне завдання:

Необхідно отримати у викладача варіант і виконати наступні пункти.

1. Визначення об’єктного представлення моделей.

2. Розробка класу Context.

3. Визначення відношень між сутностями.

4. Визначення обмежень і додаткових умов.

5. Створення міграції.

6. Демонстрація роботи з сутностями

Примітка: Виконавець може самостійно вносити зміни в архітектурі бази даних.

Наприклад: Відношення (один до багатьох, товар - кошик) може не підійти для магазину оскільки тоді доведеться створювати безліч однакових товарів, тому прийняте рішення внести перейти на більш комфортний варіант (багато до багатьох, товар - кошик). Такі маніпуляції необхідно обґрунтувати у звіті.

Лабораторна робота №2

Тема: Знайомство з ASP.NET Core Web-Api

Завдання: Необхідно розробити проекті ASP.NET Web-Api під'єднати його до попереднього проекту База даних магазин.

В проекті створити контролери (CRUD представлення таких сутностей):

- Користувач

Обмеження і додаткові умови

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати методи роботи з базою даних.

1. Початкову конфігурації.

Створити рішення в тому самому солюшині що і бібліотека з представленням бази даних.

В проекті за визначенням буде знаходитись небагато файлів, Program, Startup, файли конфігурації і дефолтний контролер.

Розглянемо Клас Startup. В ньому знаходиться метод ConfigureServices – реєструє додаткові сервіси які використовуються в сервісі. Всі методи починаються зі слова Add. В нашому випадку зареєстровано компоненти для роботи контролерів і Свагеру.

```
public void ConfigureServices(IServiceCollection services)
{
```

```
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "BasketCatalog", Version = "v1" });
    });
}
```

метод Configure – вирішує те як сервіс буде опрацьовувати запит.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
```

2. Впровадження класу реєстрації контейнерів DI

Клас буде мати назву DiExtensions в ньому будуть окремо, методи для окремої реєстрації служб.

Метод для реєстрації опцій.

```
public static IServiceCollectionAddCommonConfigurations(thisIServiceCollection services,
IConfiguration configuration)
{
    return services;
}
```

Метод для реєстрації DbContext

```
public static IServiceCollectionAddDbContext(this IServiceCollection services, IConfiguration
configuration)
{
    var connstr =
configuration.GetSection("AppSettings:DBConnectionSettings:ConnectionString").Value;

    //services.AddDbContext<BasketCatalogDbContext>(options =>
            //options.UseSqlServer(connstr, builder =>
            {
                builder.EnableRetryOnFailure(3, TimeSpan.FromSeconds(1), null);
                //builder.MigrationsAssembly("DB");
            }),
        ServiceLifetime.Transient);


    return services;
```

```
}
```

В проект використовується база даних MS SqlServer, але за бажанням можна її замінити. Треба звернути увагу що в реєстрації контексту використано стратегію EnableRetryOnFailures, вона додає додаткову обробку - якщо при виконанні запиту виникне тимчасова помилка, то він почекає вказаний час тобто TimeSpan.FromSeconds(1) одну секунду, і повторить спробу, таких спроб буде 3.

Метод для реєстрації інших сервісів.

```
public static IServiceCollection AddServices(this IServiceCollection services)
{
    services.AddTransient(typeof(DB.Tools.IDbContextFactory<>), typeof(DbContextFactory<>));

    return services;
}
```

```
public static IServiceCollection AddServices(this IServiceCollection services)
{
    services.AddTransient(typeof(DB.Tools.IDbContextFactory<>), typeof(DbContextFactory<>));

    return services;
}
```

Всі методи викликаємо в класі ConfigureServices

```
public void ConfigureServices ( IServiceCollection services )
{

    services.AddDbContext( _configuration );
    Console.WriteLine($"DbContext Registered ");

    services.AddCommonConfigurations(_configuration);
    Console.WriteLine($"Configuration Registered");

    var appSettings = _configuration.GetSection("AppSettings").Get<AppSettingsOptions>();

    services. AddControllers(  ) ;
    services. AddSwaggerGen( c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "BasketCatalog", Version = "v1" });
```

```
    });


    services.AddServices();
}
```

3. Реєстрація контейнеру UserRepository

Створюємо інтерфейс IUserRepository і його реалізацію UserRepository

Інтерфейс

```
public interface IUserRepository
{
    Task<UserResponse> AddUser(AddUserRequest request);
    Task RemoveUser(Guid userId);
    Task<UserResponse> GetUser(int phone);
    Task UpdateUser(UpdateUserRequest request);
}
```

Реалізація

```
public class UserRepository: IUserRepository
{
    private readonly ILogger<UserRepository> _logger;
    private readonly DB.BasketCatalogDbContext _dbContext;
    public UserRepository(ILogger<UserRepository> logger
        , DB.BasketCatalogDbContext dbContext)
    {
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        _dbContext = dbContext ?? throw new ArgumentNullException(nameof(dbContext));
    }

    public async Task<UserResponse> AddUser(AddUserRequest request)
    {
        var user = await _dbContext.User.FirstOrDefaultAsync(p => p.Phone == request.Phone);

        if (user == null)
        {
            user = new User()
            {
                Id = Guid.NewGuid(),
                Name = request.Name,
                Phone = request.Phone
            };
            await _dbContext.User.AddAsync(user);
            await _dbContext.SaveChangesAsync();
        }

        return new UserResponse(user);
    }
    public async Task RemoveUser(Guid userId)
```

```
    {
        var user = await _dbContext.User.FirstOrDefaultAsync(p => p.Id == userId);

        if (user is not null)
        {
            _dbContext.User.Remove(user);
            await _dbContext.SaveChangesAsync();
        }

    }
    public async Task<UserResponse> GetUser(int phone)
    {
        var user = await _dbContext.User.FirstOrDefaultAsync(p => p.Phone == phone);

        if (user is null)
            throw new Exception("user is not exist");

        return new UserResponse(user);

    }

    public async Task UpdateUser(UpdateUserRequest request)
    {
        var user = await _dbContext.User.FirstOrDefaultAsync(p => p.Id == request.Id);

        if (user is null)
            throw new Exception("user is not exist");

        user.Phone = request.Phone;
        user.Name = request.Name;
        user.ModifiedAtUtc = DateTime.Now;
         await _dbContext.SaveChangesAsync();
    }
```

Далі реєструємо репозиторій. Необхідно в клас AddService додати ще одну строку.

```
public static IServiceCollection AddServices(this IServiceCollection services)
{
    services.AddTransient(typeof(DB.Tools.IDbContextFactory<>), typeof(DbContextFactory<>));

    services.AddScoped<IUserRepository, UserRepository>();

    return services;
}
```

4.Розробка класу контролера

```
[Route("api/v1/[controller]")]
[ApiController]
public class UserController : ControllerBase
{
    private readonly ILogger<UserController> _logger;
    private readonly IUserRepository _repository;

    public UserController(IUserRepository repository)
```

```
    {
        _repository = repository;
    }

}
```

```
[Route("api/v1/[controller]")]
[ApiController]
public class UserController : ControllerBase
{
    private readonly ILogger<UserController> _logger;
    private readonly IUserRepository _repository;

    public UserController(IUserRepository repository)
    {
        _repository = repository;
    }

}
```

```
[HttpPost("[action]", Name = "AddUser")]
[ProducesResponseType(typeof(UserResponse), (int)HttpStatusCode.OK)]
public async Task<ActionResult<UserResponse>> AddUser([FromBody] AddUserRequest request)
{
    try
    {
        var resp = await _repository.AddUser(request);
        return Ok(resp);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

```
[HttpPost("[action]", Name = "AddUser")]
[ProducesResponseType(typeof(UserResponse), (int)HttpStatusCode.OK)]
public async Task<ActionResult<UserResponse>> AddUser([FromBody] AddUserRequest request)
{
    try
    {
        var resp = await _repository.AddUser(request);
        return Ok(resp);
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

Метод RemoveUser

```
[HttpDelete("{userId}", Name = "RemoveUser")]
[ProducesResponseType(typeof(void), (int)HttpStatusCode.OK)]
public async Task<IActionResult> RemoveUser(Guid userId)
{
    try
    {
        await _repository.RemoveUser(userId);
        return Ok();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

Метод GetUser

```
[HttpDelete("{userId}", Name = "RemoveUser")]
[ProducesResponseType(typeof(void), (int)HttpStatusCode.OK)]
public async Task<IActionResult> RemoveUser(Guid userId)
{
    try
    {
        await _repository.RemoveUser(userId);
        return Ok();
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

Метод UpdateUser

```
[HttpPost()]
[ProducesResponseType(typeof(void), (int)HttpStatusCode.OK)]
public async Task<IActionResult> UpdateUser(UpdateUserRequest request)
{
    try
    {
        await _repository.UpdateUser(request);
        return Ok();
    }
```

```
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);
    }
}
```

Лабораторне завдання:

Необхідно отримати у викладача варіант і виконати наступні пункти.

1. Створити проект і під’єднати до нього бібліотеку з минулої лабораторної роботи.

2. Додати репозиторії з представленням зазначених в завданні таблиць.

3. Додати контролери для виведенням зазначеної в завданні інформації.

4. Провести тестування всіх створених контролерів і додати у звіт.

Висновок

В результаті виконання кваліфікаційної роботи створено тематичний факультативний курс лабораторних робіт, який ознайомлює студентів з сучасним механізмом програмного доступу до баз даних - платформою Entity Framework Core. В курсі лабораторних робіт продемонстровані основні можливості і функціонал цієї програмної платформи, поширеної серед розробників усього світу.

Практичні завдання курсу побудовані таким чином, що студенти мають можливість взаємодіяти з базами даних водночас у традиційний спосіб, який студенти вивчали раніше і у спосіб, новий для студентів. Традиційним способом є звертатись напряму до служби MSSQL і виконувати запити до конкретної бази даних мовою запитів TransactSQL. Новим способом є виконати формалізовану процедуру розробки об’єктного представлення для такої ж бази даних MSSQL і надалі взаємодіяти з цієї базою даних вже не напряму, а через розроблений програмний об’єкт.

Порівняння двох перелічений механізмів доступу добре демонструє, що новий механізм є більш гнучким і безпечним. Зокрема, дизайн безпеки доступу до бази даних можна забезпечити, якщо реалізувати додатковими багаторівневими програмними перевірками. На основі опрацьованого матеріалу студенти можуть розробляти програмні бібліотеки доступу до баз даних, які можуть водночас використовувати у наступних практичних завданнях, а також у перспективі в інших масштабних проектах, які студенти зможуть розробляти на платформі .NET.

Перелік використаних джерел

1) Джон П Сміт Entity Framework Core in Action: Навчальний посібник / Д. П. Сміт, 2019 - С.191-302.

2) Ендрю Лок ASP.NET Core in Action, Second Edition: Навчальний посібник / Е. Лок, 2020 - 557 с.

3) Цезар де ла Торре .NET Microservices: Architecture for Containerized .NET Applications: Навчальний посібник/ Цезар де ла Торре, Б. Вагнер, М. Русо - С.18-91.

4) Шаблоны интеграции корпоративных приложений: Книга посібник/ Хоп Г., Вульф Б., Браун К., Д'Круз К. Ф., Фаулер М., Неввіл Ш., Реттіг М. Дж., Саймон Д., 2007 - С. 43-77.

5) Паттерны проектирования на платформе .NET: Книга посібник/ С. Тепляков 2015 С. 231-305.

6) Джеффри Рихтер CLR via C# Программирование на платформе microsoft .NET FRAMEWORK 4.5 на языке C#: Книга посібник/ Д. Ріхтер, 2013 - С. 28-94.

7) Microsoft asp.net core documentation: Електроний ресурс / https://docs.microsoft.com/en-us/aspnet/core

8) Посилання: Microsoft ef core documentation: Електроний ресурс/ https://docs.microsoft.com/en-us/ef/core/

Додатки

Додаток А

Варіанти лабораторної роботи номер 1

Варіант 1

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Школа.

В проекті створити сутності:

- Учитель
- Клас
- Учні

Відношення між сутностями:

- Учитель може мати багато учнів так і учні можуть мати багато вчителів (багато до багатьох)
- Один клас може мати багато учнів (один до багатьох)

Обмеження і додаткові умови:

- Кожна сутність повинна мати ключем тип int.
- Сутність Учня і учителя повинна мати обов’язкове поле з ім’ям і обмеженням в 140 символів. Використати метод **HasMaxLength** або атрибут **MaxLength**.

Завдання на додатковий бал: Необхідно додати сутність перекус в кожному перекусі можуть брати участь кілька учнів.

Варіант 2

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Готель.

В проекті створити сутності:

- Кімнати
- Постояльці
- Покоївки
- Зміни покоївок
- Період проживань

Відношення між сутностями:

- В один період проживання може бути декілька гостей і одна кімната.
- В одну зміну може працювати декілька покоївок які обслуговують одну кімнату.

Обмеження і додаткові умови:

- Сутність кімната повинна мати ключем тип **int.** Сутності Зміни покоївок і Період проживань ключем мають бути **Guid.** Всі інші сутності на розсуд виконавця лабораторну роботу.
- В сутності зміни необхідно передбачити поля DateTime час початку і кінця змін.
- В сутності період проживання також необхідно передбачити час початку терміну проживання і час кінця.

Варіант 3

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Лікарня.

В проекті створити сутності:

- Палата
- Постояльці
- Медсестри
- Період лікування

Відношення між сутностями:

- В один період лікування може бути один гість за ним має піклуватись одна медсестра.
- В одній палаті може лікуватись декілька хворих.
- Одна медсестра може піклуватись за декількома хворими.

Обмеження і додаткові умови:

- Сутність палата повинна мати ключем тип **int.** Сутності період лікування ключем мають бути **Guid.** Всі інші сутності на розсуд виконавця лабораторну роботу.
- В сутностях період лікування необхідно передбачити поля DateTime час початку і кінця періоду лікування.

Завдання на додатковий бал: додати можливість постояльцам вступати у групи по інтересам, наприклад книжковий клуб. Один клуб може мати багато постояльців, і один постоялець може бути в багатьох клубах.

Варіант 4

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Фабрика.

В проекті створити сутності:

- Керівник цеху
- Цех
- Механіки

Відношення між сутностями:

- Керівник цеху може мати у підлеглих багато механіків. Механіки можуть працювати на більше чим одного керівника цеху.
- Один цех може мати багато механіків. Такі і механіки можуть працювати на багато цехів.

Обмеження і додаткові умови:

- Кожна сутність повинна мати ключем тип int.
- Сутність Механіка і начальника повинна мати обов’язкове поле з ім’ям і обмеженням в 140 символів. Використати метод **HasMaxLength** або атрибут **MaxLength**.

Завдання на додатковий бал: Додати можливість працювати по змінам.

Варіант 5

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Ресторан.

В проекті створити сутності:

- Столик
- Гості
- Офіціант
- Зміни офіціантів
- Час перебування гостей

Відношення між сутностями:

- В один період проживання може бути декілька гостей і один столик.
- В одну зміну може працювати декілька офіціантів які обслуговують один столик.

Обмеження і додаткові умови:

- Сутність столик повинна мати ключем тип **int.** Сутності Зміни офіціантів і Час перебування гостей ключем мають бути **Guid.** Всі інші сутності на розсуд виконавця лабораторну роботу.
- В сутності зміни необхідно передбачити поля DateTime час початку і кінця змін.
- В сутності час перебування гостей також необхідно передбачити час початку терміну проживання і час кінця.
- **Завдання на додатковий бал:** Зробити можливим щоб гості могли замовляти бронь столика і могли це робити на більше чим один столик.

Варіант 6

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Бібліотека.

В проекті створити сутності:

- Книги
- Читачі
- Бібліотекарі
- Зміна бібліотекаря
- Бронювання книг

Відношення між сутностями:

- Читач може забронювати декілька книг. Одну книгу можуть забронювати різні читачі на різний період
- Бібліотекар може мати декілька змін. На одній зміні може працювати кілька бібліотекарів.

Обмеження і додаткові умови:

- Сутність книги повинна мати ключем тип **int.** Сутності зміни бібліотекаря і бронювання ключем мають бути **Guid.** Всі інші сутності на розсуд виконавця лабораторну роботу.
- В сутностях бронювання книг і зміна бібліотекаря необхідно передбачити поля DateTime час початку і кінця періоду.

Завдання на додатковий бал: додати можливість розподіляти книги по жанрах, і додати список уподобань читача.

Варіант 7

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Охорона.

В проекті створити сутності:

- Керівник охорони
- Сектор
- Охоронець

Відношення між сутностями:

- Керівник охорони може мати у підлеглих багато охоронців. Охоронці можуть працювати на більше чим одного керівника охорони.
- Один сектор може мати багато охорони. Такі і охоронці можуть працювати на багато секторів.
- **Обмеження і додаткові умови:**
- Кожна сутність повинна мати ключем тип int.
- Сутність Охоронця і начальника повинна мати обов’язкове поле з ім’ям і обмеженням в 140 символів. Використати метод **HasMaxLength** або атрибут **MaxLength**.

Завдання на додатковий бал: Додати можливість працювати по змінам.

Варіант 8

Завдання: Необхідно розробити проект Class Library яка буде об’єктним представленням бази даних Санаторій.

В проекті створити сутності:

- Кімнати
- Постояльці
- Консьєрж
- Зміни покоївок
- Період проживань

Відношення між сутностями:

- В один період проживання може бути декілька гостей і одна кімната.
- В одну зміну може працювати декілька покоївок які обслуговують одну кімнату.

Обмеження і додаткові умови:

- Сутність кімната повинна мати ключем тип **int.** Сутності Зміни консьєржів і Період проживань ключем мають бути **Guid.** Всі інші сутності на розсуд виконавця лабораторну роботу.
- В сутності зміни необхідно передбачити поля DateTime час початку і кінця змін.
- В сутності період проживання також необхідно передбачити час початку терміну проживання і час кінця.

Завдання на додатковий бал: Зробити можливим, щоб гості могли заселятись до декількох кімнат в один період проживання, а консьєржі могли прибирати декілька кімнат в одну зміну.

Додаток Б

Варіанти лабораторної роботи номер 2

Варіант 1

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Школа.

В проекті створити контролери (CRUD представлення таких сутностей):

- Учитель
- Клас
- Учні

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити можливість переміщення учня з одного класу в інший.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 2

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Готель.

В проекті створити контролери (CRUD представлення таких сутностей):

- Кімнати
- Постояльці
- Покоївки
- Зміни покоївок
- Період проживань

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити обмеження яке не дозволить додавати більше чим 10 покоївок на одну зміну.
- Створити метод для виведення чеків за прожива. Вивести його в окремий контролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 3

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Лікарня.

В проекті створити контролери (CRUD представлення таких сутностей):

- Палата
- Постояльці
- Медсестри
- Період лікування

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Додати методи контролю над числом постояльців в одній палаті, і числом постояльців якими може опікуватись одна медсестра. Вивести їх в окремий контролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 4

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Фабрика.

В проекті створити контролери (CRUD представлення таких сутностей):

- Керівник цеху
- Цех
- Механіки

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Додати методи контролю над числом постояльців в одній палаті, і числом постояльців якими може опікуватись одна медсестра. Вивести їх в окремий контролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 5

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Ресторан.

В проекті створити контролери (CRUD представлення таких сутностей):

- Столик
- Гості
- Офіціант
- Зміни офіціантів
- Час перебування гостей

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити обмеження яке не дозволить забронювати більше чим 10 столиків за визначенням. Вивести в окремий контролер.
- Додати можливість замінити офіціанта. Також вивести цей функціонал в окремий конотролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 6

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Ресторан.

В проекті створити контролери (CRUD представлення таких сутностей):

- Книги
- Читачі
- Бібліотекарі
- Зміна бібліотекаря
- Бронювання книг

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити обмеження яке не дозволить взяти більше чим 10 книг. Вивести функціонал в окремий конотролер.
- Додати можливість виводити історію прочитаних книг. Також вивести функціонал в окремий конотролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 7

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Охорона.

В проекті створити контролери (CRUD представлення таких сутностей):

- Керівник охорони
- Сектор
- Охоронець

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити контроль заміни Охоронця на секторі (але якщо чергування уже почалось то заборонити). Вивести функціонал в окремий конотролер.
- Передбачити обмеження охоронців на один сектор. Також вивести функціонал в окремий конотролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```

Варіант 8

Завдання: Необхідно розробити проектASP.NET Web-Api під’єднати його до попереднього проекту База даних Охорона.

В проекті створити контролери (CRUD представлення таких сутностей):

- Кімнати
- Постояльці
- Консьєрж
- Зміни покоївок
- Період проживань

Обмеження і додаткові умови:

- Передбачити обробку помилок і виведення відповідних http статус кодів.
- Використати в проекті патерн репозиторій в якому прописати логіку роботи з базою даних.
- Передбачити обмеження яке не дозволить забронювати більше чим 10 кімнат за визначенням. Вивести в окремий контролер.
- Додати можливість замінити консьєржа. Також вивести функціонал в окремий конотролер.

Завдання на додатковий бал: Розробити власну модель помилки в якій буде передаватись код і текст. Додати в кожну модель Response. Приклад:

```
{
   ErrorCode: 1
   ErrorText:" You can`t add more students to the cafeteria!"
}
```