**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

**ЛАБОРАТОРНАЯ РАБОТА №2**

по дисциплине: «Разработка приложений баз данных для информационных систем»

на тему: «Использование *ENTITY FRAMEWORK* и *LINQ* для работы с базами данных»

Выполнил: студент гр. ИТП-31

Демиденко Д. В.

Принял: ректор

Асенчик О. Д.

Гомель 2023

**Цель работы:** ознакомиться с возможностями *ENTITY* *FRAMEWORK* и получить навыки написания *LINQ* запросов к объектам, связанным с таблицами базы данных СУБД *MS* *SQL* сервер.

**Задание:**

1.1. Создать с использованием .*NET* *Core* *Entity* *Framework* Core консольное приложение, содержащее набор классов, моделирующих предметную область соответствующей своему варианту и ранее созданную и заполненной тестовыми данными задания базой *MS* *SQL* *Server*. Для этого необходимо создать:

• Классы, моделирующие не менее чем три таблицы базы данных согласно вашему варианту.

• Класс контекста данных.

1.2. Выполнить, используя объекты *Entity* *Framework* *Core* и *LINQ*:

1. Выборку всех данных из таблицы, стоящей в схеме базы данных нас стороне отношения «один» – 1 шт.

2. Выборку данных из таблицы, стоящей в схеме базы данных нас стороне отношения «один», отфильтрованные по определенному условию, налагающему ограничения на одно или несколько полей – 1 шт.

3. Выборку данных, сгруппированных по любому из полей данных с выводом какого-либо итогового результата (*min*, *max*, *avg*, *сount* или др.) по выбранному полю из таблицы, стоящей в схеме базы данных нас стороне отношения «многие» – 1 шт.

4. Выборку данных из двух полей двух таблиц, связанных между собой отношением «один-ко-многим» – 1 шт.

5. Выборку данных из двух таблиц, связанных между собой отношением «один-ко-многим» и отфильтрованным по некоторому условию, налагающему ограничения на значения одного или нескольких полей – 1 шт.

6. Вставку данных в таблицы, стоящей на стороне отношения «Один» – 1 шт.

7. Вставку данных в таблицы, стоящей на стороне отношения «Многие» – 1 шт.:

8. Удаление данных из таблицы, стоящей на стороне отношения «Один» – 1 шт.

9. Удаление данных из таблицы, стоящей на стороне отношения «Многие» – 1 шт.

10. Обновление удовлетворяющих определенному условию записей в любой из таблиц базы данных – 1 шт.

1.3. Разместить выполненный проект на *github*.

**Ход работы**

В ходе выполнения лабораторной работы было создано консольное приложение на версии .*NET* 7.0. Далее для работы с базой данных при помощи *ENTITY FRAMEWORK* были установлены необходимые *NuGET* пакеты:

1. *Microsoft.EntityFrameworkCore.Design ­*­– Предоставляет интерфейсы для связи между базой данных и *C*#.
2. *Microsoft.EntityFrameworkCore.SqlServer* – Предоставляет функционал для связи ORM фреймворка с базой данных *MS SQL Server.*
3. *Microsoft.EntityFrameworkCore.Tools* – Предоставляет список команд для работы с миграциями, переносом созданной базы в виде объектов в *SQL* и переносу *SQL* базы данных в объекты *C*#.

После установки всех необходимых пакетов приложение готово для работы с базой данных через *ENTITY FRAMEWORK*. В основе *ENTITY FRAMEWORK* лежит технология *ORM* (*object-relational mapping* – отображения данных на реальные объекты) которая позволят связать таблицы баз данных с *C*# объектами. Для связи объектов по определенным правилам существует три подхода.

*Code Convention* для работы нет необходимости что-либо настраивать собственноручно. Например, при создании объекта модели указывается поле *Id* и логично что данное поле должно иметь свойство *PRIMARY KEY*. ENTITY FRAMEWORK уже из названия поля понимает это и при переносе этой модели в базу данных устанавливает этому столбцу значения *PRIMARY KEY.* Это подхода вполне достаточно для простых баз данных с несложными связями. Для более тонкой настройки существует другие два подхода: через атрибуты или *Fluent API*.

При установлении свойств через атрибуты пользователь может указать сколько угодно атрибутов для каждого свойства модели. Например, атрибут *Table* для класса модели указывает как будет называться модель в базе данных. Атрибут *Key* для свойства модели указывает что данное поле будет первичным ключом в базе данных. Это подход позволяет более тонко настаивать ограничения и связи в базе данных. Но его минусом является то, что если база дынных содержит много таблиц, то для настройки связей между таблицами необходимо каждый раз переключатся между ними. Из-за этого очень легко запутаться в созданной структуре. Намного удобнее, когда все настройки моделей хранятся в одном классе. С проблемой и призвана справится такая технология как *Fluent API*.

Перед тек как перейти в технологии *Fluent API* необходимо понять, что сами по себе в отдельность модели никак не могут быть связаны с базой данных. Для этого в *ENTITY FRAMEWORK* есть специальный класс *DbContext* который предоставляет специальное *API* для подключения к базе данных и связи моделей в *C*# с таблицами в *SQL*. Именно в этом классе и прописываются ограничения при помощи *Fluent API.* Это делаете при помощи паттерна строитель и метод расширения. Плюсом данного подхода настройки моделей является то, что вся конфигурации хранится в одном файле.

При работе с *ORM* существует два похода *Database-first* и *Code-first*. В первом походе сначала создается база данных, а потом при помощи *ENTITY FRAMEWORK* она переносится в *C*#. Другой подход – это тогда, когда программист уже создал все необходимые модели в *C*# и для того, чтобы не создавать такую же базу данных с нуля можно просто перенести все свои модели в базу данных при помощи *ENTITY FRAMWORK*.

Так как в предыдущей лабораторной работе уже была создана база данных то тут подойдет подход *Database-first.* Для переноса всех таблиц из базы данных в модели используется специальная команда *Scaffold-DbContext*. Которая в параметры принимает строку подключения к базе данных и провайдера, например *SQLServer.* После выполнения этой команды создаются классы моделей и класс контекста. Листинг этих классов указан в приложении А.

Далее для хранения запросов при помощи *LINQ* был создан класс *Queries*. Листинг данного класса указан в приложении А.

Первый запрос выполняет выборку данных из таблицы на стороне отношения один. Он достает данные из таблицы *Services*. Листинг данного метода *DisplayAllServices* указан в приложении А. Пример работы этого запроса указан на рисунке 1.

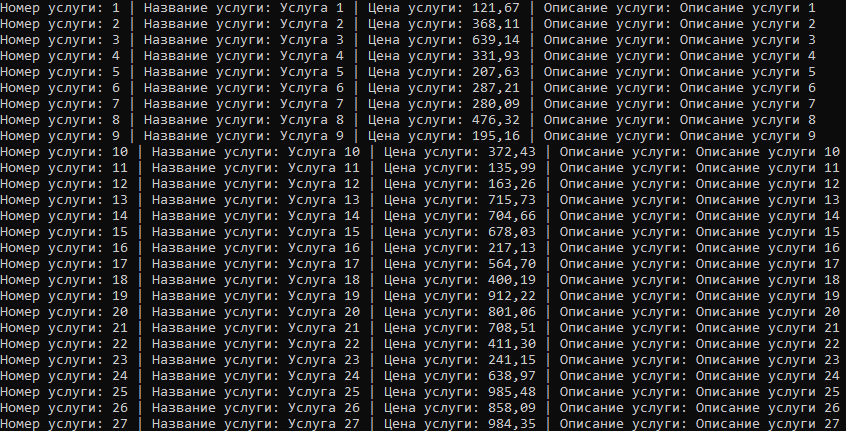


Рисунок 1 – Пример запроса на выборку данных из таблицы на стороне отношения один

Далее был реализован запрос на выборку данных из таблицы на стороне отношения один с последующей фильтрацией данных. Для этого был реализован метод *DisplayServicesAbovePrice* который выводит только те услуги, которые имеют стоимость выше 900. Листинг данного метода *DisplayServicesAbovePrice* указан в приложении А. Пример работы этого запроса указан на рисунке 2.

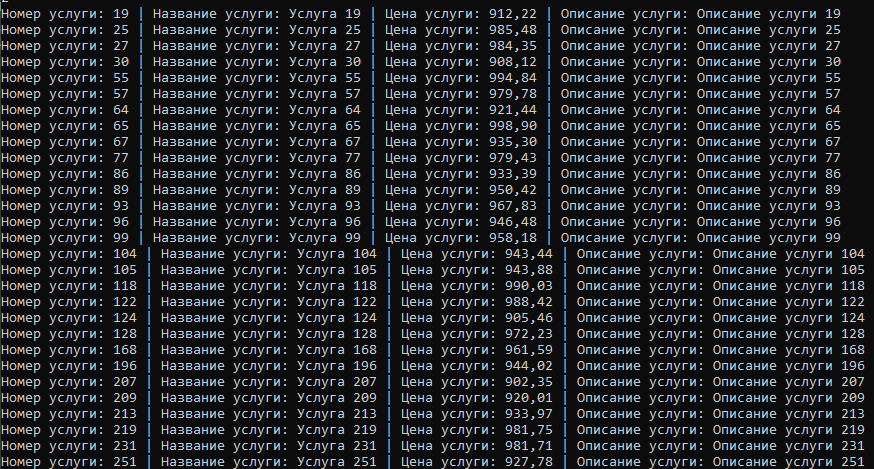


Рисунок 2 – Пример запроса на выборку данных из таблицы на стороне отношения один с фильтрацией данных

Далее был разработан запрос на выборку данных из таблицы стоящую на стороне отношения многие и агрегирующие полученный результат. В данном случае был разработан метод *DisplayGroupedApartments* который группирует квартиры по количеству комнат с выводом количества таких квартир, максимальной стоимости и средней площади. Листинг данного метода *DisplayGroupedApartments* указан в приложении А. Пример работы этого запроса указан на рисунке 3.

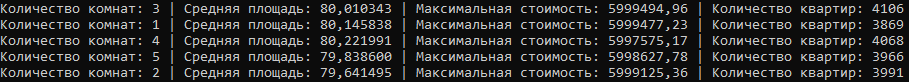


Рисунок 3 – Пример запроса на выборку данных из таблицы на стороне отношения многие с последующей агрегацией данных

Далее был разработан запрос на выборку данных, стоящую на стороне отношения один-многие с выводом данных из обоих таблиц. Для этого был реализован метод *DisplayServiceContracts* который выбирает данные из таблиц *Services* и *Contracts.* Для создания связи между двумя таблицами используется метод SelectMany. В результате на экран будет выведено название услуги из таблицы *Services* и дата заключения договора связанного с этой услугой. Листинг данного метода *DisplayServiceContracts* указан в приложении А. Пример работы этого запроса указан на рисунке 4.

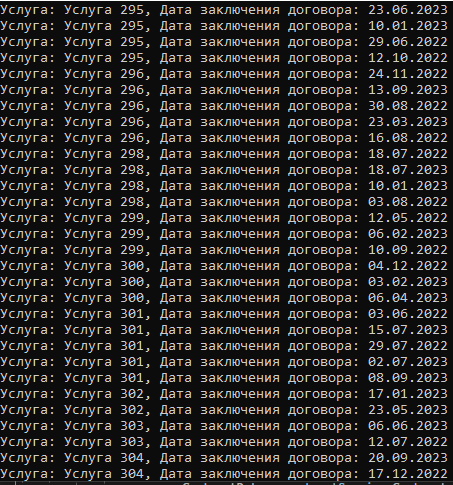


Рисунок 4 – Пример запроса на выборку данных из таблиц на стороне отношения один-многие

Далее был разработан запрос на выборку данных из таблиц, стоящих на стороне отношения один-многие с последующей фильтрацией данных. Для этого был разработан метод *DisplayApartmentsWithHighValueContracts*. В данном методе выборка производится из таблиц *Apartments* и *Contracts.* И фильтрация производится по полю *DealAmount* в таблице *Contracts* и по полю *ApartmentId* в таблице *Apartments*. В параметры метода передается цена договора и номер квартиры с которой связан договор*.* Листинг данного метода *DisplayApartmentsWithHighValueContracts* указан в приложении А. Пример работы этого запроса указан на рисунке 5.

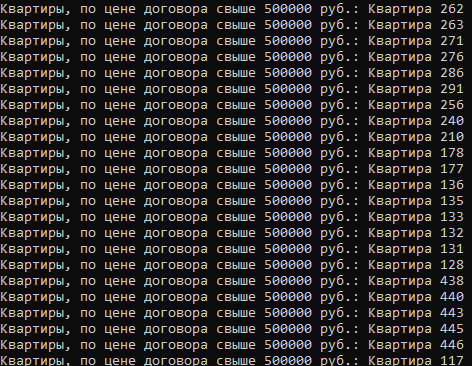


Рисунок 5 – Пример запроса на выборку данных из таблиц на стороне отношения один-многие с последующей фильтрацией данных

Далее был разработан запрос на добавления данных в таблицу, стоящую на стороне отношения один. Для этого был разработан метод *AddNewService* который добавляет данные в таблицу *Services* в качестве параметров метод принимает объект класса *Services*. Для добавления данных *SQL* базу в *ENTITY FRAMEWORK* достаточно добавить объект в нужный *DbSet* и у объекта контекста вызвать метод *SaveChanges*. Листинг данного метода *AddNewService* указан в приложении А. Пример работы этого запроса указан на рисунке 6.

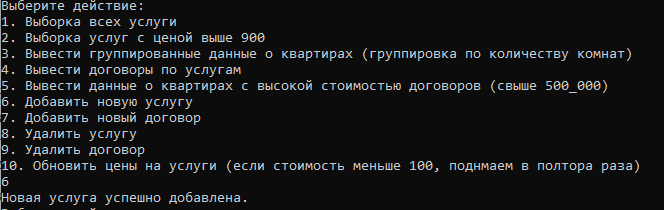


Рисунок 6 – Пример запроса на вставку данных в таблицу на стороне отношения один

После добавления нового объекта в *DbSet* и вызова метода *SaveChanges.* Добавленный данные сохранятся в *SQL* базе данных. Пример добавленных данных указан на рисунке 7.

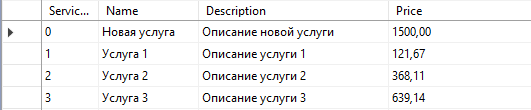


Рисунок 7 – Пример добавленных данных

Далее был разработан запрос на добавления данных в таблицу, стоящую на стороне отношения многие. Для этого был разработан метод *AddNewContract* который добавляет данные в таблицу *Contracts* в качестве параметров метод принимает объект класса *Contracts*. Если объект хранит в себе ссылки на другие объекты базы данных, то при добавлении можно просто указать *Id* нужного элемента в базе данных или указать новый объект и если его нет в базе данных, то он автоматически создастся. В данном случае объект *Contracts* хранит в себе ссылки на таблицы *Contracts* и *Sellers*. Листинг данного метода *AddNewContract* указан в приложении А. Пример работы этого запроса указан на рисунке 8.

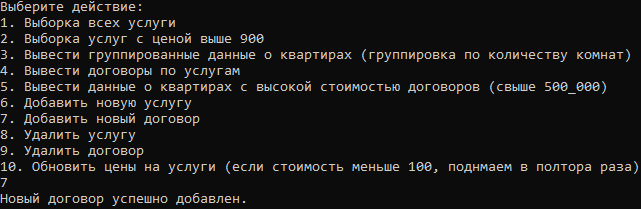


Рисунок 8 – Пример запроса на вставку данных в таблицу на стороне отношения многие

После добавления нового объекта в *DbSet* и вызова метода *SaveChanges.* Добавленный данные сохранятся в *SQL* базе данных. Пример добавленных данных указан на рисунке 9.

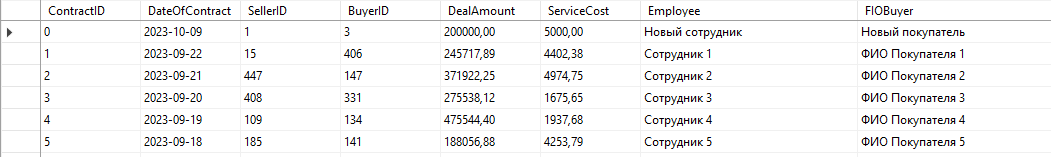


Рисунок 9 – Пример добавленных данных

Далее был реализован запрос для удаления данных из таблицы, стоящей на стороне отношения многие. Для этого был реализован метод *DeleteService,* который удаляет данные из таблицы *Services* и в качестве параметра принимает объект класса *Services.* Для удаления данных достаточно вызвать метод *Remove* для объекта *DbSet* и у объекта класса контекста вызвать метод *SaveChanges.* Листинг данного метода *DeleteService* указан в приложении А. Пример работы этого запроса указан на рисунке 10.

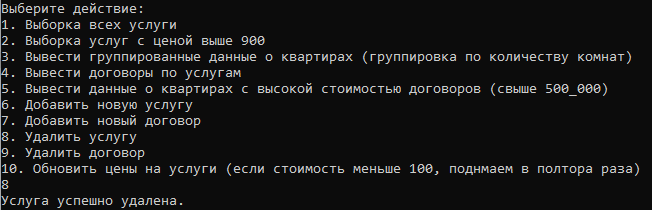


Рисунок 10 – Пример удаления данных на стороне отношения один

После удаления объекта в *DbSet* и вызова метода *SaveChanges.* Удаление данные сохранятся в *SQL* базе данных. Пример удаления данных указан на рисунке 11.

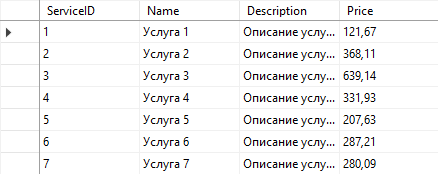


Рисунок 12 – Пример удаления данных на стороне отношения один

Далее был реализован запрос на удаления данных из таблицы, стоящей на стороне отношения многие. Для этого был реализован метод *DeleteContract,* который удаляет данные в таблице *Contracts* в качестве параметров метод принимает объект класса *Contracts*. Если этот класс содержит ссылки на другие объекты базы данных, то они удаляются согласно правилам, заданным при помощи атрибутов или *Fluent API.* Например, если удаления стоит каскадное, то удалятся сама запись в базе данных и все связанные с ней записи из других таблиц. Листинг данного метода *DeleteContract* указан в приложении А. Пример работы этого запроса указан на рисунке 13.

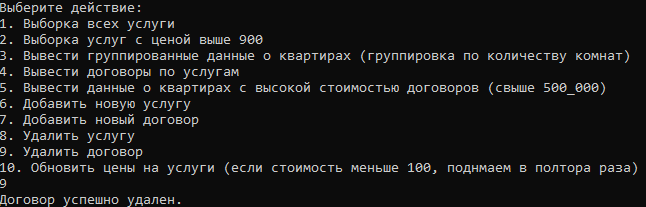


Рисунок 13 – Пример удаления данных на стороне отношения многие

После удаления объекта в *DbSet* и вызова метода *SaveChanges.* Удаление данные сохранятся в *SQL* базе данных. Пример удаления данных указан на рисунке 14.

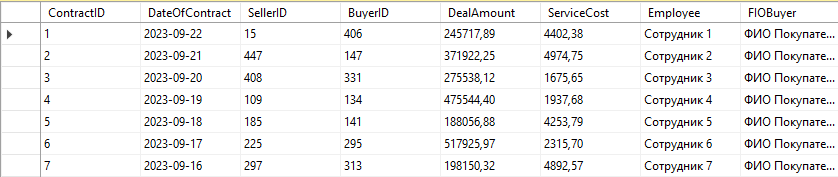


Рисунок 14 – Пример удаления данных на стороне отношения многие

Далее был разработан запрос на обновления данных в таблице. Для этого был разработан метод *UpdateServicePrices,* который принимает в качестве параметров цену и увеличивает её в полтора раза там, где цена меньше 100. Для сохранения изменений в базе данных достаточно изменить определенный параметр в *DbSet* и вызвать метод *SaveChanges.* Листинг данного метода *UpdateServicePrices* указан в приложении А. Пример работы этого запроса указан на рисунке 14.

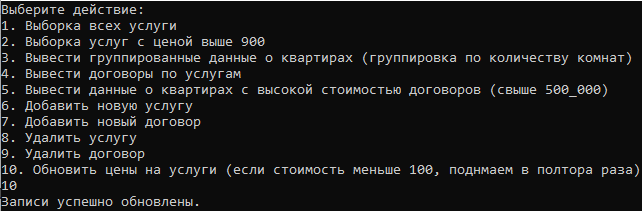


Рисунок 14 – Пример обновления данных в базе данных

После обновления объекта в *DbSet* и вызова метода *SaveChanges.* Обновление данные сохранятся в *SQL* базе данных. Пример обновлённых данных указан на рисунке 15.

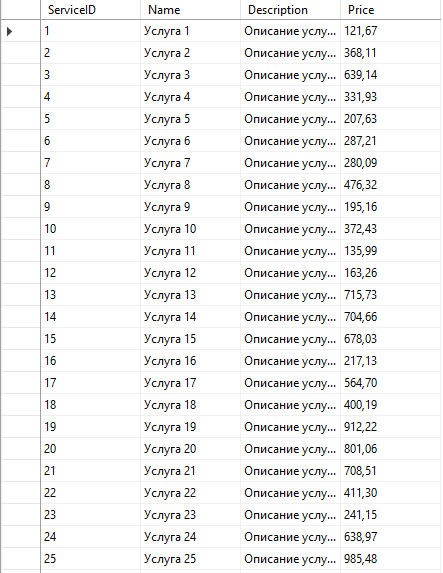


Рисунок 15 – Пример обновления данных с заданным условием

После выполнения лабораторной работы созданные проект был добавлен в локальный *git* репозиторий а потом перенесен в *GitHub* репозиторий своего аккаунта. Чтобы ознакомится с созданным проектом можно по ссылке https://github.com/ddemidenko/RPBDIS/tree/master*.*

**Вывод**: в ходе выполнения лабораторной работы была изучена такая технология как *ENTITY FRAMEWORK* которая позволяет работать с базой данный при помощи объектов *C*#. Также для работы с *ORM* была использована такая технология как *LINQ* которая позволяет удобно работать с данными. В ходе выполнения лабораторной работы база данных созданная в первой лабораторной работе была перенесена в *C*#. Были изучены основный функции *ENTITY FRAMEWORK* для агрегации, фильтрации и группировки данных при помощи *LINQ* запросов.

**ПРИЛОЖНИЕ А**

Листинг класса *Apartments*

namespace RealEstateAgency

{

public partial class Apartments

{

public Apartments()

{

Contracts = new HashSet<Contracts>();

Sellers = new HashSet<Sellers>();

}

public int ApartmentId { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public int? NumberOfRooms { get; set; }

public decimal? Area { get; set; }

public bool? SeparateBathroom { get; set; }

public bool? HasPhone { get; set; }

public decimal? MaxPrice { get; set; }

public string AdditionalPreferences { get; set; }

public virtual ICollection<Contracts> Contracts { get; set; }

public virtual ICollection<Sellers> Sellers { get; set; }

}

}

Листинг класса *Contracts*

namespace RealEstateAgency

{

public partial class Contracts

{

public Contracts()

{

ContractServices = new HashSet<ContractServices>();

}

public int ContractId { get; set; }

public DateTime? DateOfContract { get; set; }

public int? SellerId { get; set; }

public int? BuyerId { get; set; }

public decimal? DealAmount { get; set; }

public decimal? ServiceCost { get; set; }

public string Employee { get; set; }

public string Fiobuyer { get; set; }

public virtual Apartments Buyer { get; set; }

public virtual Sellers Seller { get; set; }

public virtual ICollection<ContractServices> ContractServices { get; set; }

}

}

Листинг класса *ContractServices*

namespace RealEstateAgency

{

public partial class ContractServices

{

public int ContractServiceId { get; set; }

public int? ContractId { get; set; }

public int? ServiceId { get; set; }

public virtual Contracts Contract { get; set; }

public virtual Services Service { get; set; }

}

}

Листинг класса *Sellers*

namespace RealEstateAgency

{

public partial class Sellers

{

public Sellers()

{

Contracts = new HashSet<Contracts>();

}

public int SellerId { get; set; }

public string FullName { get; set; }

public string Gender { get; set; }

public DateTime? DateOfBirth { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string PassportData { get; set; }

public int? ApartmentId { get; set; }

public string ApartmentAddress { get; set; }

public decimal? Price { get; set; }

public string AdditionalInformation { get; set; }

public virtual Apartments Apartment { get; set; }

public virtual ICollection<Contracts> Contracts { get; set; }

}

}

Листинг класса *Services*

namespace RealEstateAgency

{

public partial class Services

{

public Services()

{

ContractServices = new HashSet<ContractServices>();

}

public int ServiceId { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public decimal? Price { get; set; }

public virtual ICollection<ContractServices> ContractServices { get; set; }

}

}

Листинг класса *ApartmentView*

namespace RealEstateAgency

{

public partial class ApartmentView

{

public int ApartmentId { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public int? NumberOfRooms { get; set; }

public decimal? Area { get; set; }

public bool? SeparateBathroom { get; set; }

public bool? HasPhone { get; set; }

public decimal? MaxPrice { get; set; }

public string AdditionalPreferences { get; set; }

}

}

Листинг класса *ContractView*

namespace RealEstateAgency

{

public partial class ContractView

{

public int ContractId { get; set; }

public DateTime? DateOfContract { get; set; }

public string SellerFullName { get; set; }

public string BuyerName { get; set; }

public decimal? DealAmount { get; set; }

public decimal? ServiceCost { get; set; }

}

}

Листинг класса *ServiceView*

namespace RealEstateAgency

{

public partial class ServiceView

{

public int ServiceId { get; set; }

public string ServiceName { get; set; }

public string ServiceDescription { get; set; }

public decimal? ServicePrice { get; set; }

}

}

Листинг класса *RealEstateAgencyContext*

using System;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata;

// Code scaffolded by EF Core assumes nullable reference types (NRTs) are not used or disabled.

// If you have enabled NRTs for your project, then un-comment the following line:

// #nullable disable

namespace RealEstateAgency

{

public partial class RealEstateAgencyContext : DbContext

{

public RealEstateAgencyContext()

{

}

public RealEstateAgencyContext(DbContextOptions<RealEstateAgencyContext> options)

: base(options)

{

}

public virtual DbSet<ApartmentView> ApartmentView { get; set; }

public virtual DbSet<Apartments> Apartments { get; set; }

public virtual DbSet<ContractServices> ContractServices { get; set; }

public virtual DbSet<ContractView> ContractView { get; set; }

public virtual DbSet<Contracts> Contracts { get; set; }

public virtual DbSet<Sellers> Sellers { get; set; }

public virtual DbSet<ServiceView> ServiceView { get; set; }

public virtual DbSet<Services> Services { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseSqlServer("Server=.\\sqlexpress;Database=RealEstateAgency;Trusted\_Connection=True;");

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<ApartmentView>(entity =>

{

entity.HasNoKey();

entity.ToView("ApartmentView");

entity.Property(e => e.AdditionalPreferences).HasMaxLength(1000);

entity.Property(e => e.ApartmentId).HasColumnName("ApartmentID");

entity.Property(e => e.Area).HasColumnType("decimal(10, 2)");

entity.Property(e => e.Description).HasMaxLength(1000);

entity.Property(e => e.MaxPrice).HasColumnType("decimal(10, 2)");

entity.Property(e => e.Name).HasMaxLength(255);

});

modelBuilder.Entity<Apartments>(entity =>

{

entity.HasKey(e => e.ApartmentId)

.HasName("PK\_\_Apartmen\_\_CBDF57444B3534CB");

entity.Property(e => e.ApartmentId)

.HasColumnName("ApartmentID")

.ValueGeneratedNever();

entity.Property(e => e.AdditionalPreferences).HasMaxLength(1000);

entity.Property(e => e.Area).HasColumnType("decimal(10, 2)");

entity.Property(e => e.Description).HasMaxLength(1000);

entity.Property(e => e.MaxPrice).HasColumnType("decimal(10, 2)");

entity.Property(e => e.Name).HasMaxLength(255);

});

modelBuilder.Entity<ContractServices>(entity =>

{

entity.HasKey(e => e.ContractServiceId)

.HasName("PK\_\_Contract\_\_55E1AE33F404A6CB");

entity.Property(e => e.ContractServiceId)

.HasColumnName("ContractServiceID")

.ValueGeneratedNever();

entity.Property(e => e.ContractId).HasColumnName("ContractID");

entity.Property(e => e.ServiceId).HasColumnName("ServiceID");

entity.HasOne(d => d.Contract)

.WithMany(p => p.ContractServices)

.HasForeignKey(d => d.ContractId)

.HasConstraintName("FK\_\_ContractS\_\_Contr\_\_412EB0B6");

entity.HasOne(d => d.Service)

.WithMany(p => p.ContractServices)

.HasForeignKey(d => d.ServiceId)

.HasConstraintName("FK\_\_ContractS\_\_Servi\_\_4222D4EF");

});

modelBuilder.Entity<ContractView>(entity =>

{

entity.HasNoKey();

entity.ToView("ContractView");

entity.Property(e => e.BuyerName).HasMaxLength(255);

entity.Property(e => e.ContractId).HasColumnName("ContractID");

entity.Property(e => e.DateOfContract).HasColumnType("date");

entity.Property(e => e.DealAmount).HasColumnType("decimal(10, 2)");

entity.Property(e => e.SellerFullName).HasMaxLength(255);

entity.Property(e => e.ServiceCost).HasColumnType("decimal(10, 2)");

});

modelBuilder.Entity<Contracts>(entity =>

{

entity.HasKey(e => e.ContractId)

.HasName("PK\_\_Contract\_\_C90D3409C859D437");

entity.Property(e => e.ContractId)

.HasColumnName("ContractID")

.ValueGeneratedNever();

entity.Property(e => e.BuyerId).HasColumnName("BuyerID");

entity.Property(e => e.DateOfContract).HasColumnType("date");

entity.Property(e => e.DealAmount).HasColumnType("decimal(10, 2)");

entity.Property(e => e.Employee).HasMaxLength(255);

entity.Property(e => e.Fiobuyer)

.HasColumnName("FIOBuyer")

.HasMaxLength(255);

entity.Property(e => e.SellerId).HasColumnName("SellerID");

entity.Property(e => e.ServiceCost).HasColumnType("decimal(10, 2)");

entity.HasOne(d => d.Buyer)

.WithMany(p => p.Contracts)

.HasForeignKey(d => d.BuyerId)

.HasConstraintName("FK\_\_Contracts\_\_Buyer\_\_3C69FB99");

entity.HasOne(d => d.Seller)

.WithMany(p => p.Contracts)

.HasForeignKey(d => d.SellerId)

.HasConstraintName("FK\_\_Contracts\_\_Selle\_\_3B75D760");

});

modelBuilder.Entity<Sellers>(entity =>

{

entity.HasKey(e => e.SellerId)

.HasName("PK\_\_Sellers\_\_7FE3DBA12300466D");

entity.Property(e => e.SellerId)

.HasColumnName("SellerID")

.ValueGeneratedNever();

entity.Property(e => e.AdditionalInformation).HasMaxLength(1000);

entity.Property(e => e.Address).HasMaxLength(500);

entity.Property(e => e.ApartmentAddress).HasMaxLength(500);

entity.Property(e => e.ApartmentId).HasColumnName("ApartmentID");

entity.Property(e => e.DateOfBirth).HasColumnType("date");

entity.Property(e => e.FullName).HasMaxLength(255);

entity.Property(e => e.Gender).HasMaxLength(10);

entity.Property(e => e.PassportData).HasMaxLength(255);

entity.Property(e => e.Phone).HasMaxLength(20);

entity.Property(e => e.Price).HasColumnType("decimal(10, 2)");

entity.HasOne(d => d.Apartment)

.WithMany(p => p.Sellers)

.HasForeignKey(d => d.ApartmentId)

.HasConstraintName("FK\_\_Sellers\_\_Apartme\_\_38996AB5");

});

modelBuilder.Entity<ServiceView>(entity =>

{

entity.HasNoKey();

entity.ToView("ServiceView");

entity.Property(e => e.ServiceDescription).HasMaxLength(1000);

entity.Property(e => e.ServiceId).HasColumnName("ServiceID");

entity.Property(e => e.ServiceName).HasMaxLength(255);

entity.Property(e => e.ServicePrice).HasColumnType("decimal(10, 2)");

});

modelBuilder.Entity<Services>(entity =>

{

entity.HasKey(e => e.ServiceId)

.HasName("PK\_\_Services\_\_C51BB0EAFDD30054");

entity.Property(e => e.ServiceId)

.HasColumnName("ServiceID")

.ValueGeneratedNever();

entity.Property(e => e.Description).HasMaxLength(1000);

entity.Property(e => e.Name).HasMaxLength(255);

entity.Property(e => e.Price).HasColumnType("decimal(10, 2)");

});

OnModelCreatingPartial(modelBuilder);

}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

}

}

Листинг класса *Program*

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace RealEstateAgency

{

internal class Program

{

static void Main(string[] args)

{

while (true)

{

Console.WriteLine("Выберите действие:");

Console.WriteLine("1. Выборка всех услуги");

Console.WriteLine("2. Выборка услуг с ценой выше 900");

Console.WriteLine("3. Вывести группированные данные о квартирах (группировка по количеству комнат)");

Console.WriteLine("4. Вывести договоры по услугам");

Console.WriteLine("5. Вывести данные о квартирах с высокой стоимостью договоров (свыше 500\_000)");

Console.WriteLine("6. Добавить новую услугу");

Console.WriteLine("7. Добавить новый договор");

Console.WriteLine("8. Удалить услугу");

Console.WriteLine("9. Удалить договор");

Console.WriteLine("10. Обновить цены на услуги (если стоимость меньше 100, поднмаем в полтора раза)");

int choice;

if (int.TryParse(Console.ReadLine(), out choice))

{

switch (choice)

{

case 1:

DisplayAllServices();

break;

case 2:

DisplayServicesAbovePrice();

break;

case 3:

DisplayGroupedApartments();

break;

case 4:

DisplayServiceContracts();

break;

case 5:

DisplayApartmentsWithHighValueContracts();

break;

case 6:

AddNewService();

break;

case 7:

AddNewContract();

break;

case 8:

DeleteService();

break;

case 9:

DeleteContract();

break;

case 10:

UpdateServicePrices();

break;

default:

Console.WriteLine("Неверный выбор. Пожалуйста, выберите снова.");

break;

}

}

else

{

Console.WriteLine("Неверный ввод. Пожалуйста, выберите снова.");

}

}

}

static void DisplayAllServices()

{

using (var context = new RealEstateAgencyContext()) //Выборка всех данных из таблицы Services

{

var allServices = context.Services.ToList();

foreach (var service in allServices)

{

Console.WriteLine($"Номер услуги: {service.ServiceId} | Название услуги: {service.Name} | Цена услуги: {service.Price} | Описание услуги: {service.Description}");

}

}

}

static void DisplayServicesAbovePrice()

{

using (var context = new RealEstateAgencyContext()) //Выборка данных из таблицы, отфильтрованную по определенному условию, налагающему ограничения на одно поле

{

decimal priceThreshold = 900.0m;

var selectedServices = context.Services

.Where(service => service.Price > priceThreshold)

.ToList();

foreach (var service in selectedServices)

{

Console.WriteLine($"Номер услуги: {service.ServiceId} | Название услуги: {service.Name} | Цена услуги: {service.Price} | Описание услуги: {service.Description}");

}

}

}

static void DisplayGroupedApartments()

{

using (var context = new RealEstateAgencyContext()) //Выборка данных, сгруппированных по любому из полей данных с выводом какого-либо итогового результата по выбранным полям из таблицы Apartments

{

var groupedApartments = context.Apartments

.GroupBy(apartment => apartment.NumberOfRooms)

.Select(group => new

{

NumberOfRooms = group.Key,

AverageArea = group.Average(apartment => apartment.Area),

MaxPrice = group.Max(apartment => apartment.MaxPrice),

Count = group.Count()

})

.ToList();

foreach (var group in groupedApartments)

{

Console.WriteLine($"Количество комнат: {group.NumberOfRooms} | Средняя площадь: {group.AverageArea} | Максимальная стоимость: {group.MaxPrice} | Количество квартир: {group.Count}");

}

}

}

static void DisplayServiceContracts()

{

using (var context = new RealEstateAgencyContext()) //Выборка данных из двух полей двух таблиц, связанных между собой отношением «один-ко-многим»

{

var serviceContracts = context.Services

.SelectMany(service => service.ContractServices, (service, contractService) => new

{

ServiceName = service.Name,

ContractDate = contractService.Contract.DateOfContract.GetValueOrDefault().ToShortDateString()

})

.ToList();

foreach (var entry in serviceContracts)

{

Console.WriteLine($"Услуга: {entry.ServiceName}, Дата заключения договора: {entry.ContractDate}");

}

}

}

static void DisplayApartmentsWithHighValueContracts()

{

using (var context = new RealEstateAgencyContext()) //Выборка данных из двух таблиц Apartments и Contracts, связанных между собой отношением «один-ко-многим» и отфильтрованным по некоторому условию, налагающему ограничения на значения одного или нескольких полей

{

var apartmentsWithHighValueContracts = context.Apartments

.Where(apartment =>

context.Contracts.Any(contract =>

contract.BuyerId == apartment.ApartmentId &&

contract.DealAmount > 500000))

.ToList();

foreach (var apartment in apartmentsWithHighValueContracts)

{

Console.WriteLine($"Квартиры, по цене договора свыше 500000 руб.: {apartment.Name}");

}

}

}

static void AddNewService()

{

using (var context = new RealEstateAgencyContext()) //Вставка данных в таблицу Services, стоящей на стороне отношения «Один»

{

var newService = new Services

{

Name = "Новая услуга",

Description = "Описание новой услуги",

Price = 1500.00m

};

context.Services.Add(newService);

context.SaveChanges();

Console.WriteLine("Новая услуга успешно добавлена.");

}

}

static void AddNewContract()

{

using (var context = new RealEstateAgencyContext()) //Вставка данных в таблицу Contracts, стоящей на стороне отношения «Многие»

{

int sellerId = context.Sellers.First().SellerId;

var newContract = new Contracts

{

DateOfContract = DateTime.Now,

SellerId = sellerId,

BuyerId = 3,

DealAmount = 200000.00m,

ServiceCost = 5000.0m,

Employee = "Новый сотрудник",

Fiobuyer = "Новый покупатель"

};

context.Contracts.Add(newContract);

context.SaveChanges();

Console.WriteLine("Новый договор успешно добавлен.");

}

}

static void DeleteService()

{

using (var context = new RealEstateAgencyContext()) //Удаление данных из таблицы Services, стоящей на стороне отношения «Один»

{

var serviceToDelete = context.Services.FirstOrDefault(s => s.Name == "Новая услуга");

if (serviceToDelete != null)

{

context.Services.Remove(serviceToDelete);

context.SaveChanges();

Console.WriteLine("Услуга успешно удалена.");

}

else

{

Console.WriteLine("Услуга не найдена.");

}

}

}

static void DeleteContract()

{

using (var context = new RealEstateAgencyContext()) //Удаление данных из таблицы Contracts, стоящей на стороне отношения «Многие»

{

var contractToDelete = context.Contracts.FirstOrDefault(c => c.ContractId == 0);

if (contractToDelete != null)

{

context.Contracts.Remove(contractToDelete);

context.SaveChanges();

Console.WriteLine("Договор успешно удален.");

}

else

{

Console.WriteLine("Договор не найден.");

}

}

}

static void UpdateServicePrices()

{

using (var context = new RealEstateAgencyContext()) //Обновление удовлетворяющих определенному условию записей в таблице Services, если цена ниже 100, тогда увечиваем её в полтора раза

{

var servicesToUpdate = context.Services.Where(s => s.Price < 100);

foreach (var service in servicesToUpdate)

{

if (service.Price.HasValue)

{

service.Price \*= 1.5m;

}

}

context.SaveChanges();

Console.WriteLine("Записи успешно обновлены.");

}

}

}

}