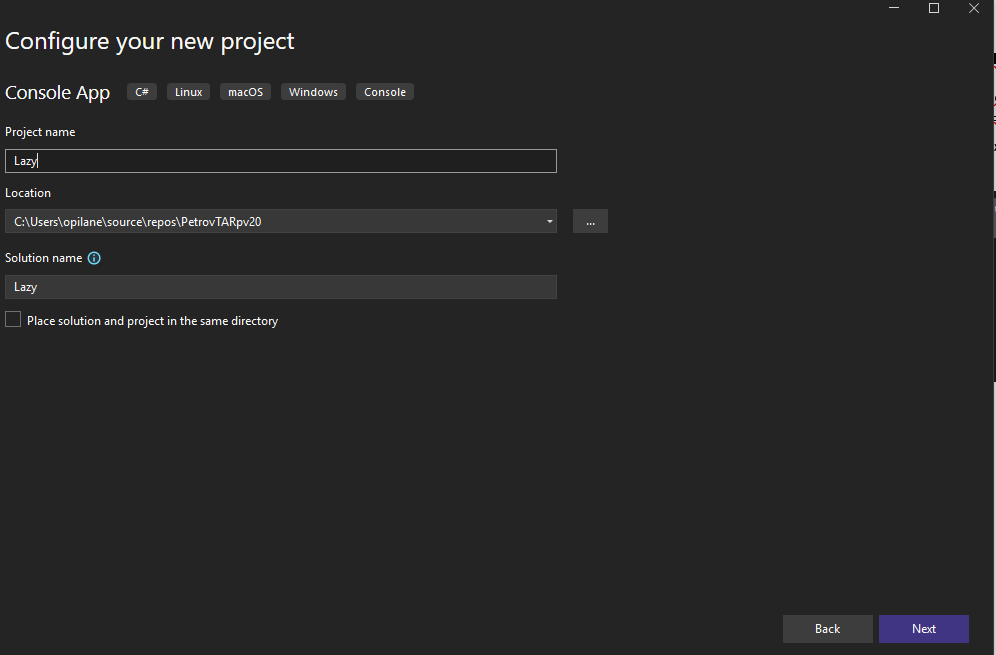
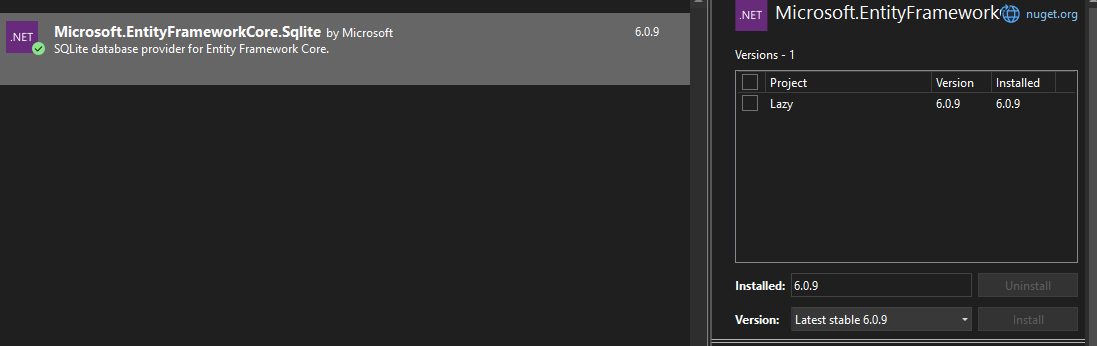
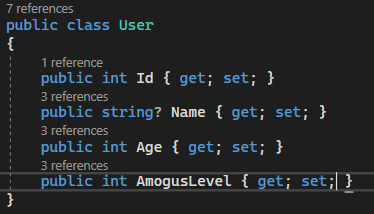
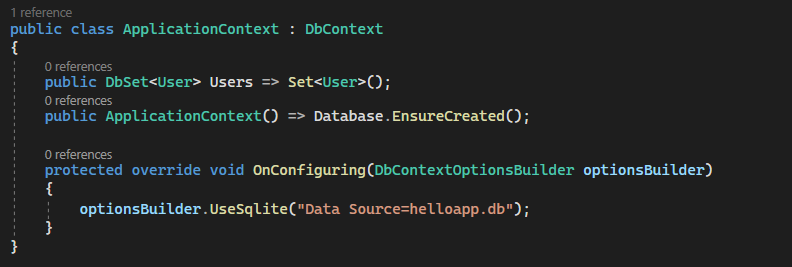
**Первое приложение на EF Core**

**Entity Framework** - Entity Framework представляет ORM-технологию (object-relational mapping - отображения данных на реальные объекты) от компании Microsoft для доступа к данным.

Короче поехали делать приложение:

****После того, как создан проект, добавляем в него это:

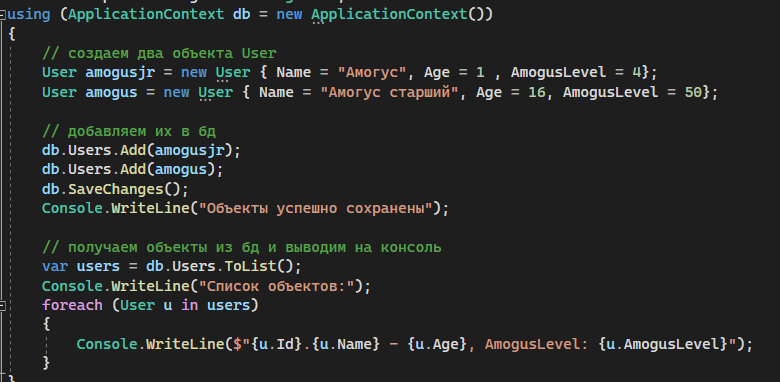
Создаем класс User

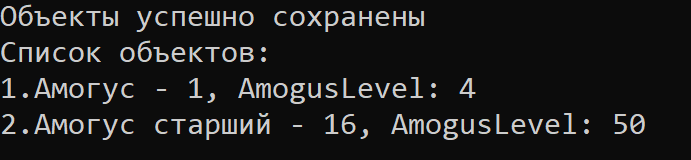
Дальше создаем класс ApplicationContext, который будет взаимодействовать с базой данных.

**DbContext**: определяет контекст данных, используемый для взаимодействия с базой данных

**DbSet/DbSet<TEntity>**: представляет набор объектов, которые хранятся в базе данных

**DbContextOptionsBuilder**: устанавливает параметры подключения

Теперь идем в Program.cs и добавляем следующий код:

И получаем следующий результат:

## Подключение к существующей базе данных

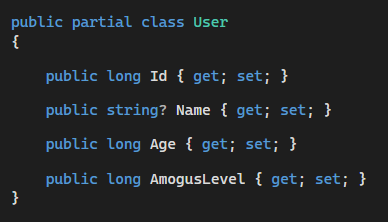
Попробуем также создать класс по существующей БД, для этого создаем еще одно приложение и проделываем все те же самые этапы, кроме того, что добавляем дополнительный пакет **Microsoft.EntityFrameworkCore.Tools** который необходим, чтобы создать классы по сущ. БД.

Для реверса БД, выполняется следующая команда:

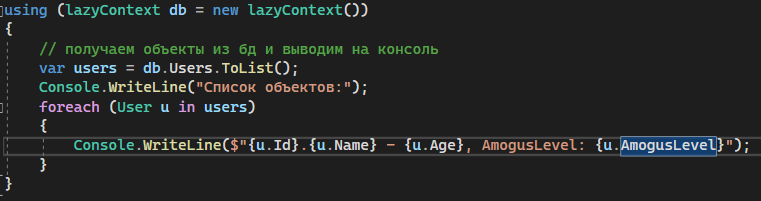
|  |  |
| --- | --- |
|  | Scaffold-DbContext "строка подключения" провайдер\_бд |

В моем же случае, это будет

|  |
| --- |
| Scaffold-DbContext "Data Source=C:\Users\opilane\source\repos\PetrovTARpv20\Lazy\Lazy\bin\Debug\net6.0\lazy.db" Microsoft.EntityFrameworkCore.Sqlite |

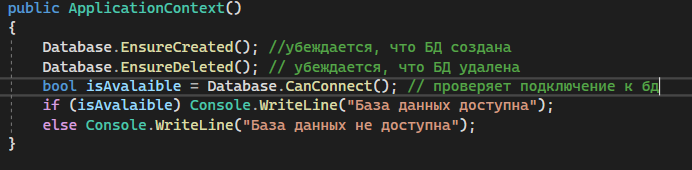
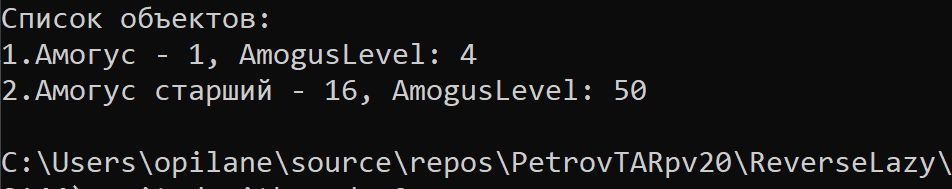
После выполнения этой команды будет добавлен класс User

А также будет добавлен класс контекста данных, который будет называться по имени базы данных плюс суффикс "Context":

И после генерации данных классов мы сможем работать с базой данных. Для этого изменим код файла Program:

И в итоге получаем:

## Управление базой данных

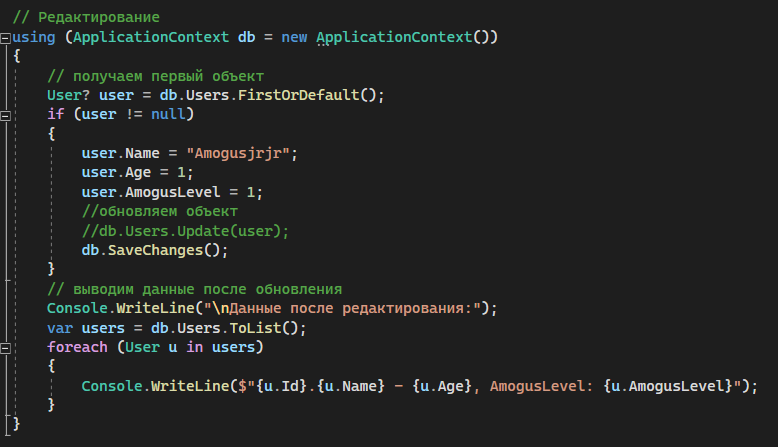
 Также у этих методов есть асинхронные версии (с суффиксом Async)

## Основные операции с данными. CRUD

## Добавление

## Удаление

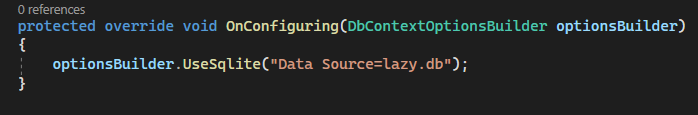
## Редактирование

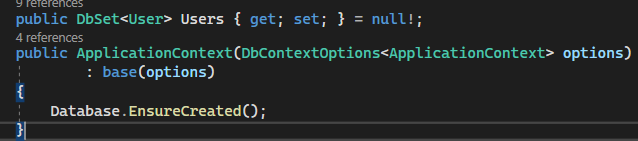


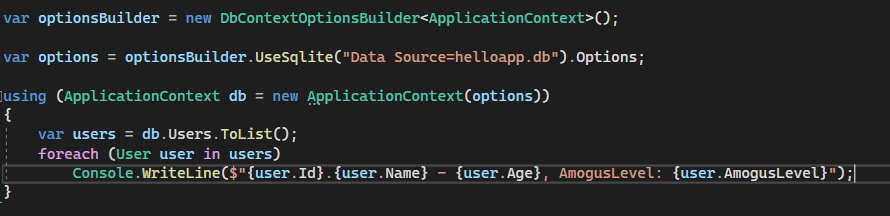
И, конечно же, у всех этих методов есть их асинхронные версии, а также версии для множества объектов( AddRange(); RemoveRange(); UpdateRange(); )

## Конфигурация подключения

Есть 2 способа конфигурации подключения.

1. Мы уже использовали, OnConfiguring()
2. Передача конфигурации в конструктор базового класса DbContext

Второй способ предполагает передачу в конструктор базового класса объекта DbContextOptions, который инкапсулирует параметры конфигурации. Для применения этого способа изменим класс контекста следующим образом:

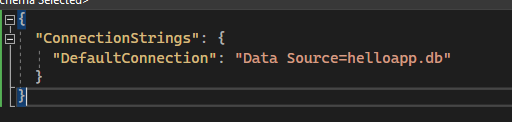
Тогда мы могли бы использовать класс контекста следующим образом:

Здесь опять же применяется метод UseSqlServer класса DbContextOptionsBuilder для создания конфигурации по той же строке подключения. Только результат этой операции - объект DbContextOptions затем передается в контекст данных. А контекст данных далее передает этот параметр в конструктор базового класса.

## Файл конфигурации

Оба выше представленных способа вполне работают, однако в том определении, в котором они представлены, они имеют один недостаток - строка подключения жестко определена в коде C#. И было бы неплохо, если бы она была бы определена в каком-нибудь внешнем файле подключения, где мы ее могли бы поменять без перекомпиляции приложения.

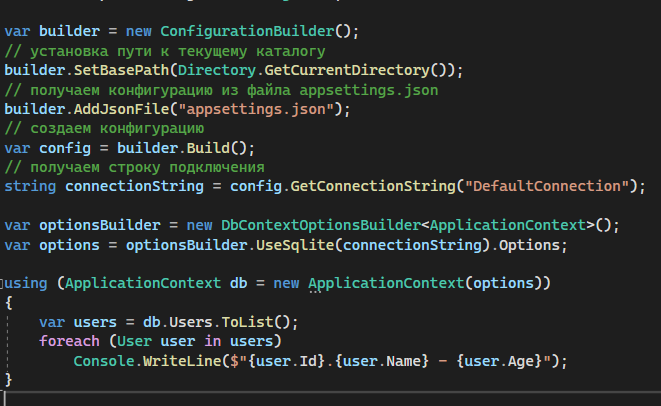
Для этого добавим в проект новый элемент JavaScript JSON Configuration File, который назовем appsettings.json:



Здесь опять же определена наша строка подключения.

Чтобы ее использовать, нам надо добавить в проект через Nuget пакет: Microsoft.Extensions.Configuration.Json. Этот пакет специально предназначен для работы с конфигурацией в формате json.

После добавления файла в Visual Studio для его копирования в каталог приложения в окне свойств необходимо установить для опции Copy to Output Directory значение "Copy if newer" (или "Copy always")

После добавления файла и заполнения, добавим следующий код

Для создания конфигурации применяется класс ConfigurationBuilder. Метод AddJsonFile() добавляет все настройки из файла конфигурации. С помощью метода Build() создается объект конфигурации, из которого мы можем получить строку подключения:

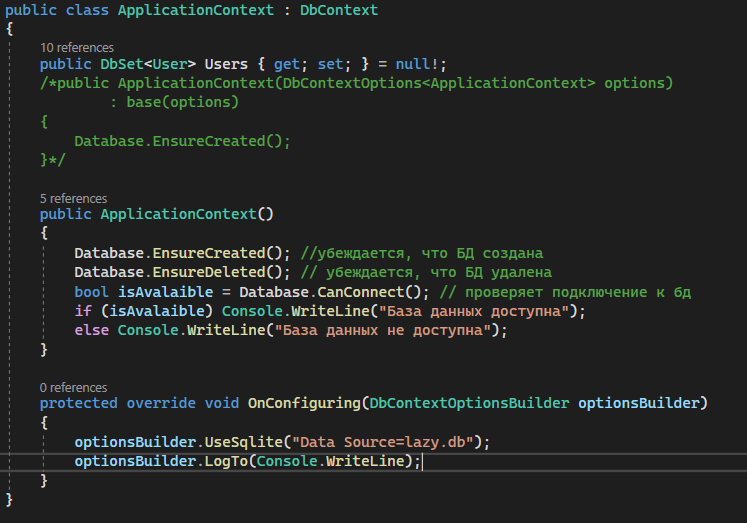
Для получения строки подключения используется ее имя - "DefaultConnection", которое указано в appsettings.json.

В остальном работа с контекстом данных будет протекать также.

## Логгирование операций

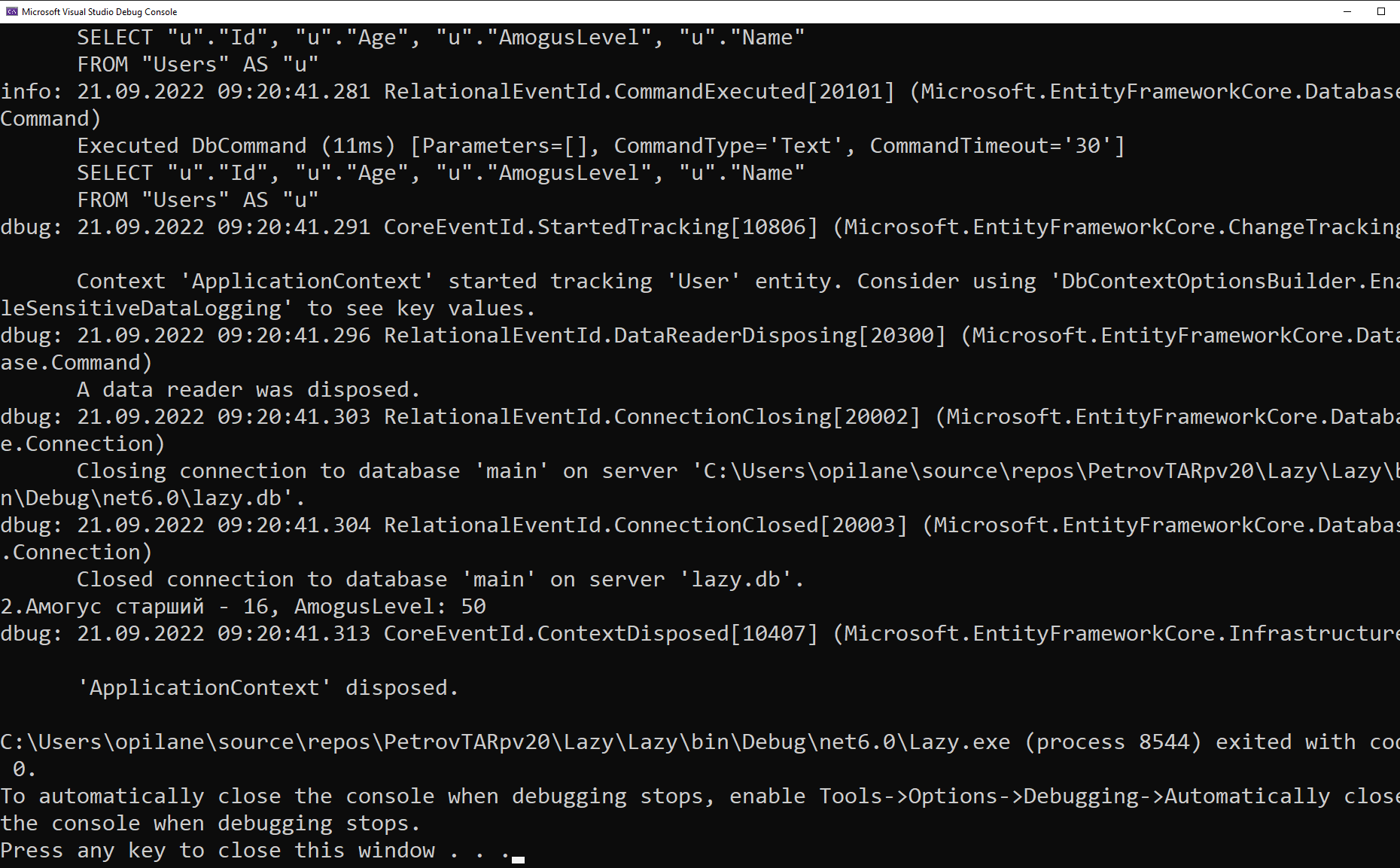
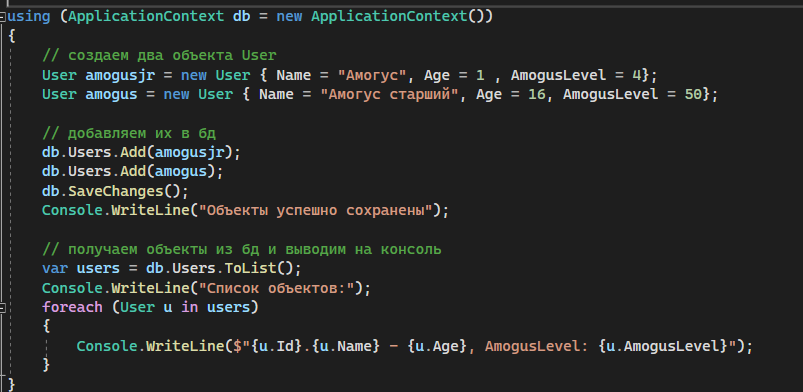
Логгирование позволяет нам получить информацию о выполняемых в Entity Framework операциях. Причем использовать как встроенные возможности, так и создать и встроить свою инфраструктуру логгирования. Рассмотрим оба варианта и начнем со встроенных возможностей.

Для логгирования информации можно использовать метод LogTo(). Он применяется при конфигурации класса контекста данных.

Например, выведем всю информацию об операциях с базой данных на консоль. Допустим, у нас будут следующая модель:

В методе OnConfiguring() у передаваемого в качестве параметра объекта DbContextOptionsBuilder вызывается метод LogTo(), в который передается делегат Action<string> - то есть некоторое действие, которое принимает один параметр типа string и и ничего не возвращает. Именно такое действие представляет традиционный метод Console.WriteLine(), который выводит строку на консоль.

Для тестирования пусть у нас определена следующая программа:

И при запуске приложения на консоль будет выведена детальная информация по всем операциям: 

## Настройка логгирования

### Уровень логгирования

Метод LogTo() имеет ряд перегруженных версий, которые принимают разное количество параметров. Так, мы можем передать в LogTo уровень логгирования в виде одного из значений перечисления LogLevel:

* Trace: используется для вывода наиболее детализированных сообщений. Подобные сообщения могут нести важную информацию о приложении и его строении, поэтому данный уровень лучше использовать при разработке, но никак не при публикации
* Debug: для вывода информации, которая может быть полезной в процессе разработки и отладки приложения
* Information: уровень сообщений, позволяющий просто отследить поток выполнения приложения
* Warning: используется для вывода сообщений о неожиданных событиях, например, ошибках, которые не влияют не останавливают выполнение приложения, но в то же время должны быть иследованы
* Error: для вывода информации об ошибках и исключениях, которые возникли при текущей операции и которые не могут быть обработаны
* Critical: уровень критических ошибок, которые требуют немедленной реакции - ошибками операционной системы, потерей данных в бд, переполнение памяти диска и т.д.
* None: вывод информации в лог не применяется

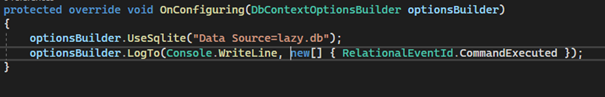
По умолчанию EntityFramework Core использует значение Debug, но можно указать какое-нибудь другое значение:

### Конкретизация сообщений

Каждое сообщение в логе ассоциировано с определенным идентификатором события. По сути идентификаторы представляют тип возникающих событий

* SqlServerEventId: описывает сообщения, специфические для провайдера для MS SQL Server
* CoreEventId: описывает сообщения, общие для всех провайдеров Entity Framework Core
* RelationalEventId: описывает сообщения, общие для всех провайдеров для реляционных баз данных

Поскольку каждый класс идентификатора имеет довольно много полей, которые представляют опеделенное сообщение, я не буду подробно расписывать все эти поля. Посмотрим на простом примере, как мы можем конкретизировать сообщения - например, нам надо вывести только выполняемые команды SQL. В этом случае мы можем воспользоваться RelationalEventId и его переменной CommandExecuted, которая представляет окончание выполнения команды:



#### 

### Категории сообщений

Другим способом фильтрации сообщений представляет использование категорий, которые представлены классом DbLoggerCategory и который позволяет задать нужные категории логгирования:

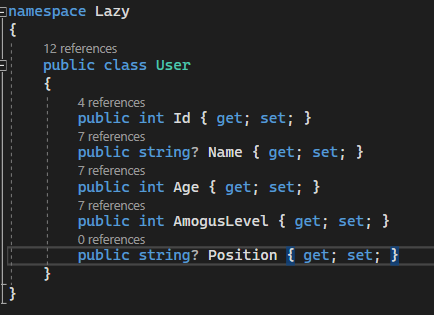
* Database.Command: категория для выполняемых команд, позволяет получить выполняемый код SQL
* Database.Connection : категория для операций подключения к БД
* Database.Transaction : категория для транзакций с бд
* Migration: категория для миграций
* Model: категория для действий, совершаемых при привязке модели
* Query: категория для запросов за исключением тех, что генерируют исполняемый код SQL
* Scaffolding: категория для действий, выполняемых в поцессе обратного инжиниринга (то есть когда по базе данных генерируются классы и класс контекста)
* Update: категория для сообщений вызова DbContext.SaveChanges()
* Infrastructure: категория для всех остальных сообщений

## Управление схемой БД и миграции

Если мы меняем модели в Entity Framework, которые входят в контекст данных, например, добавляем в нее какие-то новые свойства или удаляем некоторые свойства, то необходимо, чтобы база данных также применяла эти изменения. Например, в прошлых темах был создан класс User, который описывал пользователя.

А для работы с базой данных использовался контекст данных

Допустим, мы хотим добавить в класс User новое свойство, например:



Так как модель User изменилась, то нам надо привести в соответствие соответствующую таблицу в БД. В зависимости от используемого провайдера бд конкретное сообщение может отличаться, но суть будет одна - столбца Position в таблице нет. Модель изменилась, и база данных с таблицами уже ей не соответствует.

В зависимости от конкретной ситуации можно использовать ряд подходов для этого. Рассмотрим их.

### Ручное изменение базы данных

В самых простых случаях мы можем написать sql-скрипт для добавления столбцов или таблиц, либо же даже можем изменить таблицы вручную с помощью различных программ, которые позволяют в режиме дизайнера редактировать таблицы.

### Database.EnsureCreated и Database.EnsureDeleted

Если нам не важны данные в БД и мы хотим ее просто пересоздать для соответствия новой структуре классов, то через контекст данных можно вызывать метод Database.EnsureDeleted для удаления и затем метод Database.EnsureCreated для создания бд. Например, в коде самого контекста данных (обычно в конструкторе):

### Миграция

Миграция по сути предствляет план перехода базы данных от старой схемы к новой. Как использовать миграции?

Для создания миграции в окне Package Manager Console вводится следующая команда: Add-Migration название\_миграции

Название миграции представляет произвольное название, главное чтобы все миграции в проекте имели разные названия.

После создания миграции ее надо выполнить с помощью команды:

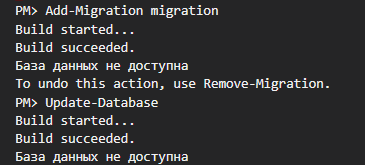
Update-Database

Если планируется использовать миграции, то лучше их использовать сразу при создании базы данных.

Для использования миграций в Visual Stuido необходимо добавить в проект через менеджер Nuget пакет Microsoft.EntityFrameworkCore.Tools.

Например, определим модели и так, как показано выше.

Также стоит отметить, что при самом первом применении миграции по отношению к БД SQLite Entity Framework пытается создать ее заново, однако если создаваемые таблицы в ней уже есть, то мы столкнемся с ошибкой. Поэтому следует убедиться, что по используемому пути нет файла базы данных с подобным именем. При последующих применениях миграции EF будет использовать бд, созданную при первой миграции.

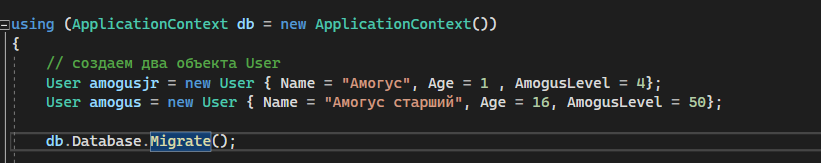
Теперь для создания и выполнения миграции перейдем в Visual Studio к окну Package Manager Console. Вначале введем команду

После выполнения миграции по указанному в методе optionsBuilder.UseSqlite("Data Source=D:\\lazy.db") пути будет сгенерированная база данных. В случае с бд SQLite, для которой указан относительный путь (например, "Data Source=lazy.db"), файл бд генерируется в папке проекта. Для других провайдеров - MS SQL Server, MySQL и т.д. бд генерируется на сервере бд в соответствии со строкой подключения.

Следует отметить, что кроме основных таблиц (в случае выше таблицы Users) база данных также будет содержать дополнительную таблицу \_EFMigrationsHystory, которая будет хранить информацию о миграциях.

Если мы хотим изменить пользователя еще раз, то просто нужно создать новую миграцию и применить её.

### Метод Migrate

В некоторых случаях, например, в приложениях с локальной базой данных (SQLite в UWP), мы можем выполнять миграции в процессе выполнения приложения. Для этого определен метод Database.Migrate() или его асинхронный двойник - Database.MigrateAsync(), который можно вызвать через объект контекста:

Стоит учитывать, что перед вызовом этого метода не следует вызывать метод EnsureCreated, который обходит миграции при создании базы данных, что вызывает ошибку при выполнении метода Migrate.

Чтобы задействовать этот метод, необходимо подключить пространство имен Microsoft.EntityFrameworkCore

### Миграции в консоли

Если разработка осуществляется не в Visual Studio, то для миграций мы можем выполнять соответствующие команды в консоли. Для создания миграции:

dotnet ef migrations add InitialCreate

Для выполнения миграции:

dotnet ef database update

Для создания скрипта sql по миграции применяется следующая команда:

dotnet ef migrations script

С передачей названия миграции:

dotnet ef migrations script InitialCreate

### Объединение миграций

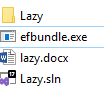
Начиная с версии 6.0 Entity Framework позволяет создавать бандлы миграций - объединение миграций в виде исполняемого файла. Для создания бандла миграций надо в Visual Studio в окне Package Manager Console выполнить команду:

Bundle-Migration

А если использовуется .NET CLI, то в консоли надо перейти к папке проекта и выполнить команду

dotnet ef migrations bundle

После выполнения этих команд в папке решения будет сгенерирован файл efbundle (в Windows он будет иметь расширение exe). Запустим его.



И после запуска бандла будут последовательно применяться добавленные в бандл миграции.

# Создание модели в Entity Framework Core

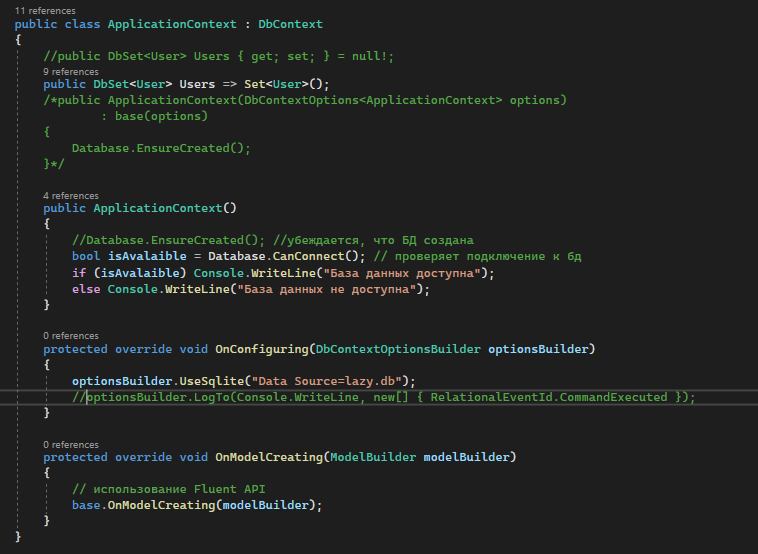
## Fluent API и аннотации данных

Модель в Entity Framework представляет набор всех сущностей и связей между ними, которыми управляет контекст данных. Все сущности, с которыми работает Entity Framework Core и которые хранятся в базе данных, определяются в C# в виде классов. При этом Entity Framework применяет ряд условностей для сопоставления классов с таблицами. Например, названия столбцов должны соответствовать названиям свойств и т.д. В этом случае Entity Framework сможет сопоставить столбцы таблицы и свойства классов.

Однако с помощью таких механизмов, как Fluent API и аннотации данных мы можем добавить дополнительные правила конфигурации, либо переопределить используемые условности.

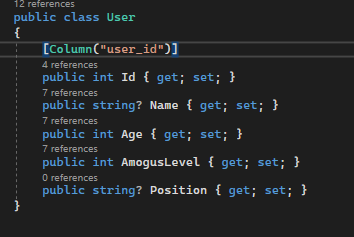
### Fluent API

Fluent API представляет набор методов, которые определяют сопоставление между классами и их свойствами и таблицами и их столбцами. Для использования функционала Fluent API переопределяется метод OnModelCreating():



### Аннотации

Аннотации представляют настройку классов сущностей с помощью атрибутов. Большинство подобных атрибутов располагаются в пространстве System.ComponentModel.DataAnnotations, которое нам надо подключить перед использованием аннотаций. Например:



В данном случае атрибут Column представляет аннотацию, которая указывает, что свойство Id будет сопоставляться со столбцом "user\_id" (а не Id, как бы было по умолчанию).

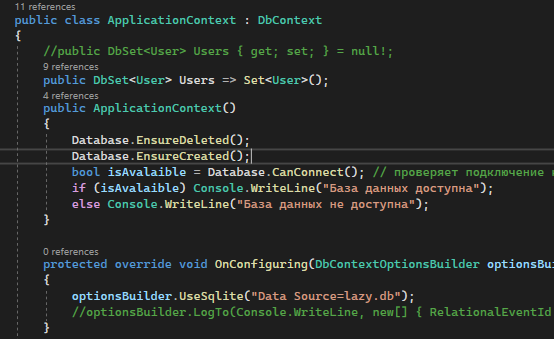
Таким образом, мы можем использовать три подхода к определению модели:

* Условности (conventions)
* Fluent API
* Аннотации данных

## Определение моделей

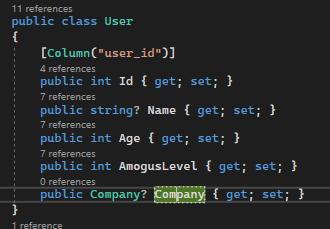
### Включение сущностей в модель

По умолчанию все типы сущностей, для которых определены в контексте данных наборы DbSet, включаются в модель и в дальнейшем сопоставляются с таблицами в базе данных. Например:



## Свойства сущности

По умолчанию модель включает все свойства сущности, которые определены как публичные и которые открыты для записи и чтения. Например,



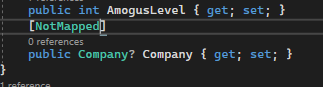
Класс User имеет три публичных свойства, поэтому при чтении или записи в базу данных Entity Framework будет автоматически сопоставлять столбцы из таблицы с этими свойствами по имени. Но такое поведение не всегда необходимо. Иногда требуется, наоборот, исключить определенное свойство, чтобы для него не создавался столбец в таблице.

Исключение с помощью Fluent API производится через метод Ignore():



Здесь свойство Company исключается из модели, и для него не будет создаваться столбец в таблице Users.

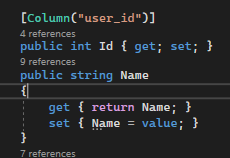
Исключение с помощью аннотаций данных:



В обоих случаях при миграции будет создана таблица Users, которая не будет содержать столбца для свойства Address, и оно не будет участвовать в сопоставлениях при операциях с бд

### Использование полей класса

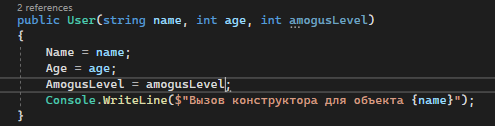
В примере выше применялись автосвойства, которые представляют сокращенную версию свойств без полноценных блоков get и set. Однако свойства не обязательно должны представлять именно автосвойства. Для хранения значений они могут использовать поля класса и иметь полноценные блоки get и set. Например:



## Конструкторы сущностей

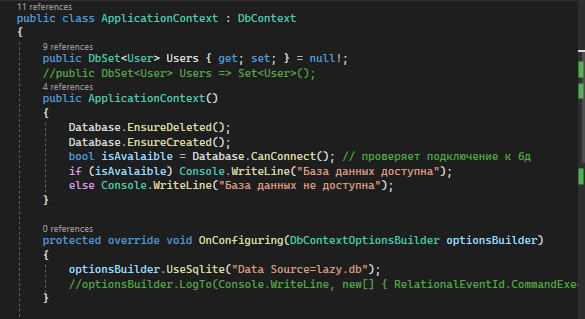
Когда EF Core создает объект сущности, например, при после получения данных из БД, он вначале вызывае конструктор по умолчанию, который не имеет параметров, и затем передает каждому свойству полученные из бд значения.

Если EF Core находит конструктор с параметрами, где названия и типы параметров соответствуют устанавливаемым свойствам, то вместо установки свойств EF передает полученные из БД значения параметрам конструктора. При этом между параметрами и свойствами должно быть соответствие по типу и имени за тем исключением, что названия могут отличаться по регистру, например, свойство Name и параметр name. Рассмотрим на примере. Допустим, у нас есть следующая сущность User:

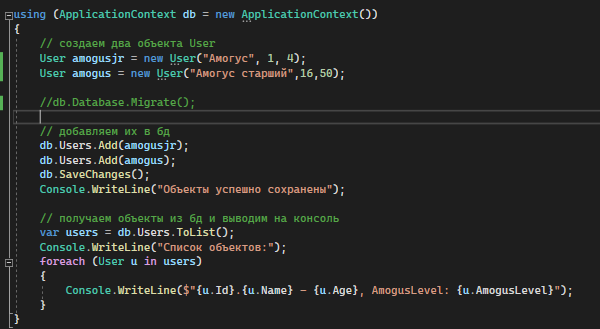


Класс User имеет три свойства и через конструктор устанавливает два из них.

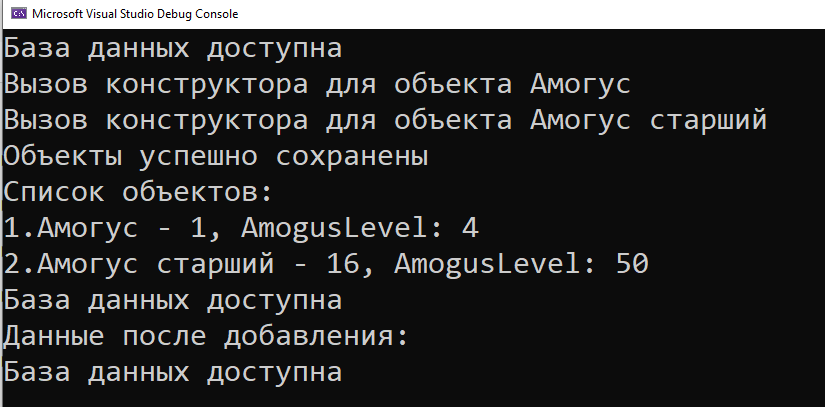
Пусть у нас будет стандартный контекст данных:



И, допустим, в программе создаем несколько объектов User, добавляем их в БД и получаем обратно из БД:



Здесь при получении данных при выполнении метода db.Users.ToList() EF Core будет вызывать для каждой полученной строки из таблицы объект User, вызывая его конструктор с двумя параметрами. Для наглядности в примере выше разделы операции добавления и получения по разным объектам контекста. В итоге мы получим следующий консольный вывод:

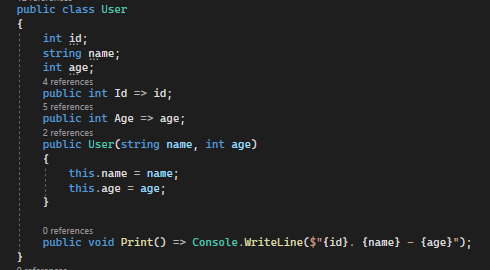


Здесь надо учитывать несколько моментов:

* Необязательно для всех свойств определять в конструкторе свои параметры. Например, свойство Id не устанавливается в конструкторе. Те свойства, для которых в конструкторе не определено параметров, устанавливаются напрямую, как в общем случае.
* Параметры и свойства должны соответствовать по имени и типу за исключением регистра имени.
* Конструкторы могут иметь любой модификатор доступа, в том числе, private.
* EF Core НЕ устанавливает таким обазом навигационные свойства, которые представляют другие сущности и имеют конструктор

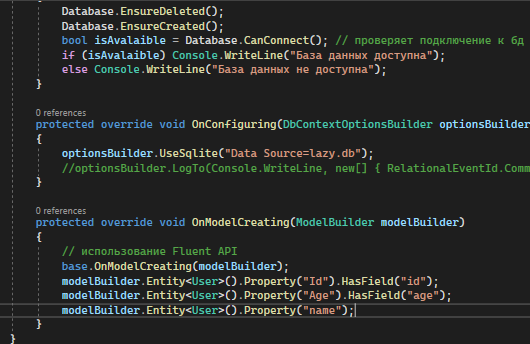
## Использование полей сущности

Кроме свойств Entity Framework также может использовать поля класса (в том числе приватные) для сопоставления со столбцами. Например, возьмем следующую сущность:



Здесь в классе User определено три поля. Все они приватные, недоступные извне. Кроме того, есть два свойства для чтения, которые возвращают значения полей. Два поля - name и age устанавливаются только через конструктор. Третье поле - id, как мы ожидаем, будет устанавливаться при добавлении объекта сущности в базу данных.

Настроим класс контекста для применения этих полей:



Для сопоставления полей со свойствами и столбцами применяется Fluent API. Чтобы использовать поле в качестве свойства при сопоставлении со столбцами применяется метод Property(), в который передается название поля:

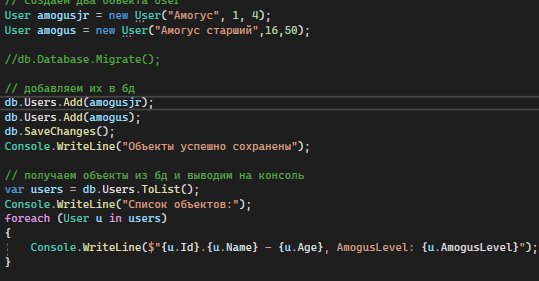


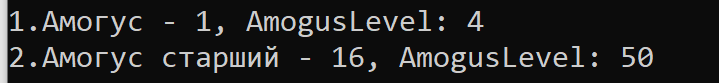
То есть в данном случае мы говорим, что мы хотим, чтобы поле "name" выступало в качестве свойства сущности и сопоставлялось со столбцом в таблице бд.

С помощью метода HasField() устанавливается поле, которое используется для свойства. Так, в выражении



Для свойства Id будет использоваться поле id

В программе мы можем создать объект User и добавить в бд: 



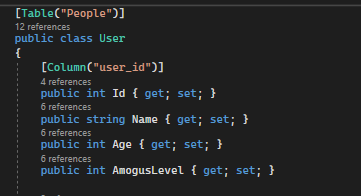
## Сопоставление таблиц и столбцов

### Сопоставление таблиц

Каждая сущность по умолчанию сопоставляется с таблицей, которая называется по имени свойства DbSet<T> в контексте данных, представляющего данную сущность. Если в контексте данных подобного свойства не определено, то для названия таблицы используется имя класса сущности.

#### Атрибут Table

Атрибут Table позволяет переопределить сопоставление с таблицей по имени:



Теперь сущность User будет сопоставляться с таблицей "People".

#### Метод ToTable

Аналогичное переопределение можно произвести через Fluent API с помощью метода ToTable():



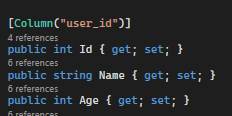
С помощью дополнительного параметра schema можно определить схему, к которой будет принадлежать таблица: 

### Сопоставление столбцов

По умолчанию каждое свойство сопоставляется с одноименным столбцом.

#### Атрибут Column

Атрибут Column переопределяет сопоставление:



Теперь свойство Id будет сопоставляться со столбцом "user\_id".

#### Метод HasColumnName

Также сопоставление можно переопределить в Fluent API с помощью метода HasColumnName:



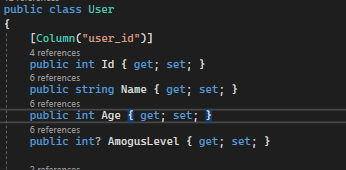
## Обязательные и необязательные свойства

### Обязательные свойства

По умолчанию свойство является необязательным к установке, если оно допускает значение null. Это свойства, которые представляют nullable-типы, например, string?, int? и т.д. Хотя мы также можем настроить эти свойства как обязательные.

Свойство является обязательным, если оно не допускает значение null.

Например, возьмем следующую модель:



В данном случае свойство Name не представляет nullable-тип, поэтому оно рассматривается как обязательное (как и свойство Age). А свойство AmogusLevel представляет nullable-тип - string?, соответственно является необязательным. Поэтому для этой сущности в SQLite будет сгенерирована следующая таблица:

CREATE TABLE "Users" (

    "Id"    INTEGER NOT NULL,

    "Name"  TEXT NOT NULL,

    "AmogusLevel"   TEXT,

    "Age"   INTEGER NOT NULL,

    CONSTRAINT "PK\_Users" PRIMARY KEY("Id" AUTOINCREMENT)

);

Здесь мы видим, что столбец Company допускает значение NULL, а столбец Name - не допускает благодаря установке атрибута NOT NULL. Хотя здесь приведен пример бд SQLite, но для других систем баз данных будет действовать аналогичная логика.

### Атрибут Required

Атрибут Required указывает, что данное свойство обязательно для установки, то есть будет иметь определение NOT NULL в БД, даже если оно представляет nullable-тип:



А столбец Name в базе данных будет определен как NOT NULL.

Если мы не установим свойство Name у объекта User и попытаемся добавить этот объект в бд, то получим во время выполнения исключение типа Microsoft.EntityFrameworkCore.DbUpdateException

### Метод IsRequired

То же самое можно сделать и через Fluent API с помощью метода IsRequired():

## Настройка ключей

По умолчанию в качестве ключа используется свойство, которое называется Id или [имя\_класса]Id.

Для установки свойства в качестве первичного ключа с помощью аннотаций применяется атрибут [Key]

Для конфигурации ключа с Fluent API применяется метод HasKey():



Дополнительно с помощью Fluent API можно настроить имя ограничения, которое задается для первичного ключа. Для этого применяется метод HasName()

### Составные ключи

С помощью Fluent API можно создать составной ключ из нескольких свойств:

Составной ключ можно создать только с помощью Fluent API.

### Альтернативные ключи

Альтернативные ключи представляют свойства, которые также, как и первичный ключ, должны иметь уникальное значение. В то же время альтернативные ключи не являются первичными. На уровне базы данных это выражается в установке для соответствующих столбцов ограничения на уникальность.

Для установки альтернативного ключа используется метод HasAlternateKey():



В данном случае свойство Passport (серия и номер паспорта) будет альтернативным ключом.

Альтернативные ключи также могут быть составными:

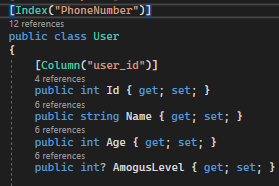


## Настройка индексов

Для увеличения производительности поиска в базе данных применяются индексы. По умолчанию индекс создается для каждого свойства, которое используется в качестве внешнего ключа. Однако Entity Framework также позволяет создавать свои индексы.

### Настройка индексов с помощью атрибутов

Для создания индекса можно использовать атрибут [Index]. Например:



Первый и обязательный параметр атрибута указывает на свойство (или набор свойств), с которым будет ассоциирован индекс. В данном случае это свойство PhoneNumber.

Но также он может принимать набор свойств, для которых создается индекс. В этом случае названия свойств просто перечисляются через запятую:



### Настройка индексов с помощью Fluent API

Для создания индекса через Fluent API применяется метод HasIndex()

#### Уникальность индексов

С помощью дополнительного метода IsUnique() можно указать, что индекс должен иметь уникальное значение. Тем самым мы гарантируем, что в базе данных может быть только один объект с определенным значением для свойства-индекса:



#### Уникальность индексов

С помощью дополнительного метода IsUnique() можно указать, что индекс должен иметь уникальное значение. Тем самым мы гарантируем, что в базе данных может быть только один объект с определенным значением для свойства-индекса

#### Составные индексы

Также можно определить индексы сразу для нескольких свойств:

#### Имя индекса

Для установки имени индекса применяется метод HasDatabaseName(), в который передается имя индекса

#### Фильтры индексов

Некоторые системы управления базами данных позволяют определять индексы с фильрами или частичные индексы, которые позволяют выполнять индексацию только по ограниченному набору значений, что увеличивает производительность и уменьшает использование дискового простанства. И EntityFramework Core также позволяет создавать подобные индексы. Для этого применяется метод HasFilter(), в который передается sql-выражение, которое определяет условие фильтра.

## Генерация значений свойств и столбцов

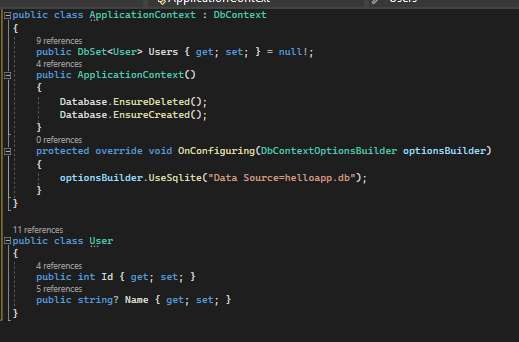
Если при добавлении или обновлении нового объекта у него уже установлено значение для свойства, Entity Framework использует это значение при вставке или обновлении в таблицу. Если для свойства явным образом не установлено значение, то для свойства устанавливается значение по умолчанию (null для nullable-типов, 0 для int, Guid.Empty для Guid и т.д.).

В зависимости от используемого провайдера базы данных, значения для свойств могут генерироваться на стороне клиента с помощью EF, либо же генерироваться уже на стороне базы данных при добавлении. Если значение генерируется базой данных, тогда при добавлении объекта в контекст EF может назначить временное значение. Это временное значение будет заменено значением, сгенерированным базой данных при вызове метода SaveChanges().

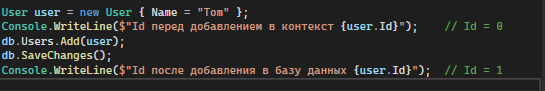
### Генерация ключей

По умолчанию для свойств первичных ключей, которые представляют типы int или GUID и которые имеют значение по умолчанию, генерируется значение при вставке в базу данных. Для всех остальных свойств значения по умолчанию не генерируется.

Например, пусть у нас имеет следующая модель:



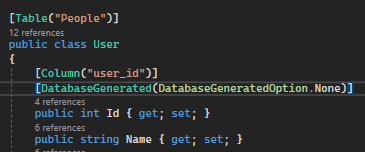
То после добавления в базу данных мы сможем получить сгенерированный Id:



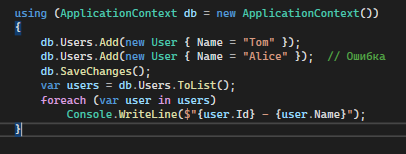
### Атрибут DatabaseGeneratedAttribute

Атрибут DatabaseGeneratedAttribute представляет аннотацию, которая позволяет изменить поведение базы данных при добавлении или изменении.

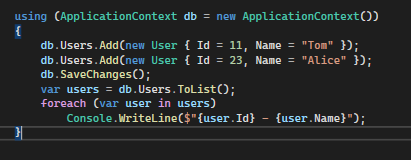
Например, мы хотим отключить автогенерацию значения при добавлении:



И если теперь мы попробуем добавить объект без установленного Id, то EF в качесте временного значения будет использовать значение по умолчанию, то есть Id=0. В итоге при добавление более одного объекта в бд мы получим ошибку:



В этом случае нам надо будет устанавливать Id:



Если мы хотим, чтобы база данных, наоборот, сама генерировала значение, то в атрибут надо передавать значение DatabaseGeneratedOption.Identity

Но в данном случае для свойства Id это значение избыточно, так как значение генерируется по умолчанию.

### Fluent API

Отключение автогенерации значения для свойства с помощью Fluent API:



### Значения по умолчанию

Для свойств, которые не представляют ключи и для которых не устанавливается значения, используются значения по умолчанию. Например, для свойств типа int это значение 0. С помощью метода HasDefaultValue() можно переопределить значение по умолчанию, которое будет применяться после добавления объекта в базу данных:

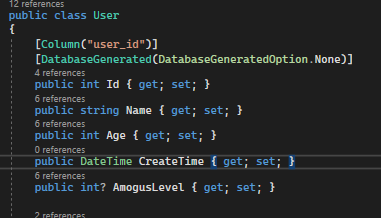


В этом случае, если мы не укажем значение для свойства Age, то ему будет присвоено значение 18

### HasDefaultValueSql

Метод HasDefaultValueSql() также определяет генерацию значения по умолчанию, только само значение устанавливается на основе кода SQL, который передается в этот метод.

Например, пусть в классе пользователя будет свойство CreatedAt, которое представляет дату занесения пользователя в базу данных:



Для генерации значения этого свойства в базе данных можно вызывать специальные функции, которые применяются в той или иной СУБД. Например, в MS SQL Server/T-SQL это функция GETDATE(), в SQLite это функции DATETIME()/DATE() и т.д. Например:



В метод HasDefaultValueSql() передается SQL-выражение, которые вызывается при добавлении объекта User в базу данных. Поскольку в данном случае используется база данных SQLite, то в качестве SQL-выражения передается вызов функции DATETIME('now') - "now" здесь указывает, что мы хотим получить текущую дату.

### Вычисляемые столбцы

Столбцы могут иметь значение, которое вычисляется на основании остальных столбцов. Например, пусть модель User имеет свойства для хранения имени и фамилии:





## Ограничения свойств

### Установка ограничений

С помощью метода HasCheckConstraint() можно установить ограничение для столбца. На уровне базы данных это приведет к установке для столбца атрибута CHECK, который задает ограничение.

В метод HasCheckConstraint() передается название столбца и sql-выражение, которое выполняет проверку корректности передаваемого значения. Например, у пользователя есть возраст, который должен укладываться в некоторые допустимые рамки. К примеру, возраст не может быть меньше 0 и больше 120: 

Соответственно мы не сможем добавить в бд объект User, у которого значение свойства Age будет меньше 0 или больше 120.

В качестве третьего параметра в HasCheckConstraint, передается делегат Action, который принимает объект CheckConstraintBuilder для настройки ограничия. В частности, мы можем задать имя ограничения:



### Ограничения по длине

Как правило, по умолчанию для строкового свойства в таблице создается столбец для хранения строки неограниченной длины. Используя аннотации данных или Fluent API, мы можем ограничить строку по длине.

Ограничение максимальной длины применяется только к строкам и к массивам, например, byte[].

#### Атрибут MaxLength

В аннотациях данных ограничение по длине устанавливается с помощью атрибута MaxLength:



Стоит отметить, что данное ограничение будет действовать только для тех систем баз данных, которые поддерживают данную возможность. Например, для бд SQLite это не будет иметь никакого значения. А в случае с бд MS SQL Server столбец Name в базе данных будет иметь тип nvarchar(50) и тем самым иметь ограничение по длине.

Надо отметить, что также есть атрибут [MinLength], который устанавливает минимальную длину, но он на определение таблицы не влияет.

#### Метод HasMaxLength

В Fluent API ограничение по длине устанавливается с помощью метода HasMaxLength():

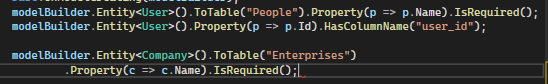


## Конфигурация моделей

С помощью атрибутов и Fluent API для сущостей и их свойств можно установить многочисленные настройки. Однако, если настроек очень много, то они могут утяжелять класс контекста и сущностей. В этом случае Entity Framework Core позволяет вынести конфигурацию сущностей в отдельные классы.

Для вынесения конфигурации во вне необходимо создать класс конфигурации, реализующий интерфейс EntityTypeConfiguration<T>.

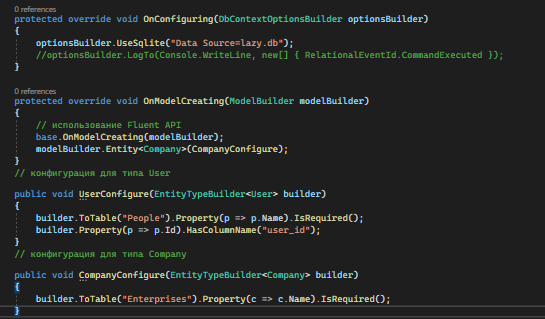
К примеру, пусть у нас есть следующий класс контекста и моделей:



Вся конфигурация здесь определена в методе OnModelCreating(). В принципе он не содержит много кода, однако при наличии гораздо большего количества сущностей и более изощренной их конфигурации с помощью Fluent API данный метод мог бы сильно раздуться в размерах.

Теперь конфигурация моделей вынесена в отдельные классы. А для добавления конкретных конфигураций в контекст используется метод modelBuilder.ApplyConfiguration(), которому передается нужный объект конфигурации. В итоге по своему действию первый и второй варианты контекста будут идентичны.

В качестве альтернативы мы могли бы использовать еще один вариант. Вместо выделения отдельных классов конфигураций определить конфигурацию в виде отдельных методов в том же классе контекста данных:



Здесь конфигурация определяется для каждого типа в отдельном методе, который в качестве параметра принимает объект EntityTypeBuilder<T>. Затем метод передается в вызов modelBuilder.Entity<T>() для соответствующей модели.

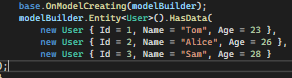
## Инициализация БД начальными данными

Иногда необходимо, чтобы при первом обращении база данных уже содержала некоторые данные. И Entity Framework Core позволяет инициализировать базу данных при ее создании некоторыми начальными данными. Благодаря этому к моменту первого использования базы данных она уже будет содержать начальные данные, которые мы сможем тут же использовать. И нам не потребуется вручную или программно добавлять в БД нужные нам данные.

Для инициализации БД при конфигурации определенной модели вызывается метод HasData(), в который передаются добавляемые данные:



Например, инициализируем БД набором данных:



У объекта ModelBuilder, который передается в OnModelCreating в качестве параметра, вызывается метод Entity<T>(). Этот метод типизируется типом, для которого будут добавляться начальные данные. То есть в данном случае данные будут добавляться в таблицу, где хранятся объекты User. Поэтому Entity типизируется типом User.

Далее по цепочке вызывается метод HasData(), который собственно и определяет начальные данные. В данном случае это набор из трех объектов User. При этом для каждого объекта необходимо установить значение первичного ключа - в данном случае значение свойства Id. Причем вне зависимости, генерирует ли база данных для данных автоматически индентификатор или нет, нам в любом случае его надо установить - это основное ограничение при инициализации БД начальными данными.

При этом следует учитывать, что инициализация начальными данными будет выполняться только в двух случаях:

* При выполнении миграции. (При создании миграции добавляемые данные автоматически включаются в скрипт миграции)
* При вызове метода Database.EnsureCreated(), который создает БД при ее отсутствии

В случае выше в конструкторе применяется метод Database.EnsureCreated(), поэтому при создании контекста данных будет автоматически производиться инициализация бд начальными данными.