Sisukord

[Log To 7](#_Toc115250449)

[**Настройка логгирования** 9](#_Toc115250450)

[Конкретизация сообщений 10](#_Toc115250451)

[MySQL SERVER 17](#_Toc115250452)

[Fluent API 21](#_Toc115250453)

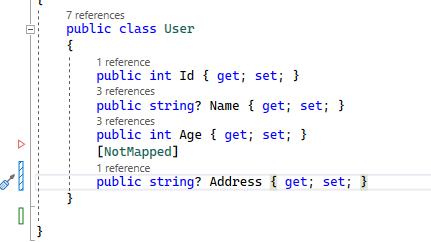
[Аннотации 21](#_Toc115250454)

[Включение сущностей в модель 22](#_Toc115250455)

[Ссылочные nullable-типы и DbSet 22](#_Toc115250456)

[Включение сущностей в модель без DbSet 23](#_Toc115250457)

[Свойства сущности 26](#_Toc115250458)

[ 27](#_Toc115250459)

[Использование полей класса 28](#_Toc115250460)

[Конструкторы сущностей 29](#_Toc115250461)

[Сопоставление таблиц 33](#_Toc115250462)

Создаем обычное консольное приложение



Устанавливаем эти пакеты через Nuget



создаем класс User который станет основой для нашей таблицы Users



Создаем класс контекста наших данных для того чтобы потом иметь возможность взаимодействовать с бд



создаем и добавляем пользователей в бд и выводим их на экран



результат



просмотр нашей бд



немного видоизменяються наши основные классы



добавляем helloappContext то, есть контекст нашей бд



пользователей в бд выводим их на экран



результат



проверка создана ли бд и если нт он создает ее



результат



бд была удалена и создана заново и была проверена на возможность подключения



результат



добавление редактирование и удаление из бд



Результат



Способ взаимодействия с бд через конструктор и connectionString



Результат



Второй способ предполагает передачу в конструктор базового класса объекта DbContextOptions



Тогда мы могли бы использовать класс контекста так



Результат



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



Тогда мы могли бы использовать класс контекста так

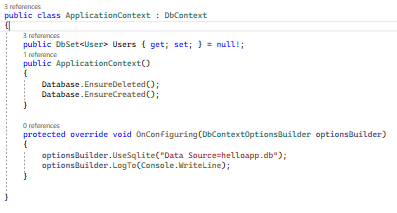


Результат

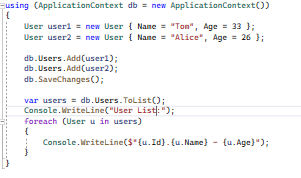


# Log To

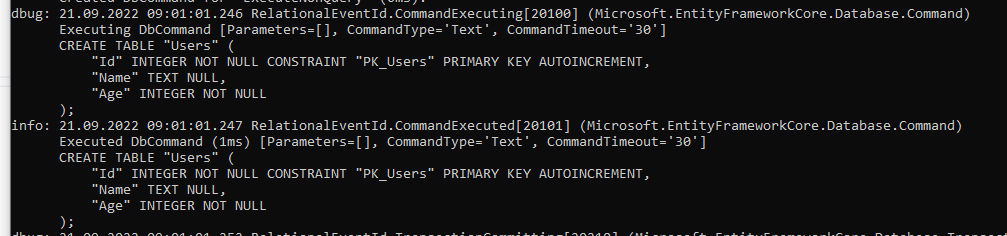
* Для логгирования информации можно использовать метод LogTo(). Он применяется при конфигурации класса контекста данных.

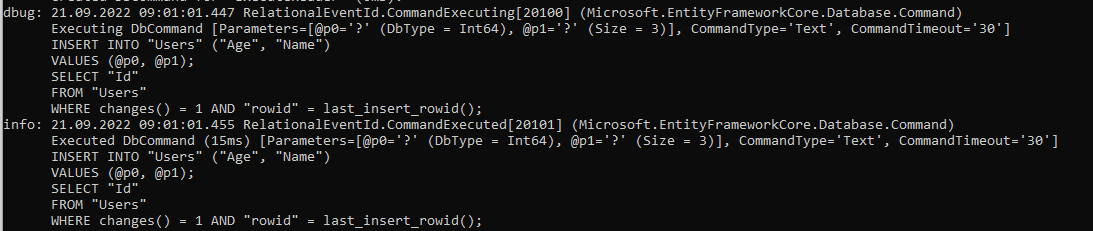


В методе OnConfiguring() у передаваемого в качестве параметра объекта DbContextOptionsBuilder вызывается метод LogTo(), в который передается делегат Action<string> - то есть некоторое действие, которое принимает один параметр типа string и и ничего не возвращает. Именно такое действие представляет традиционный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

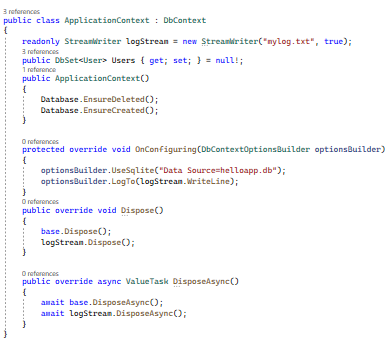


Результат





Другим распространенным способом логгирования является вывод в файл



Результат

<https://drive.google.com/file/d/1zYcc_VigSCDPADO38ScggeN3NatlRoxS/view?usp=sharing>

**Настройка логгирования**

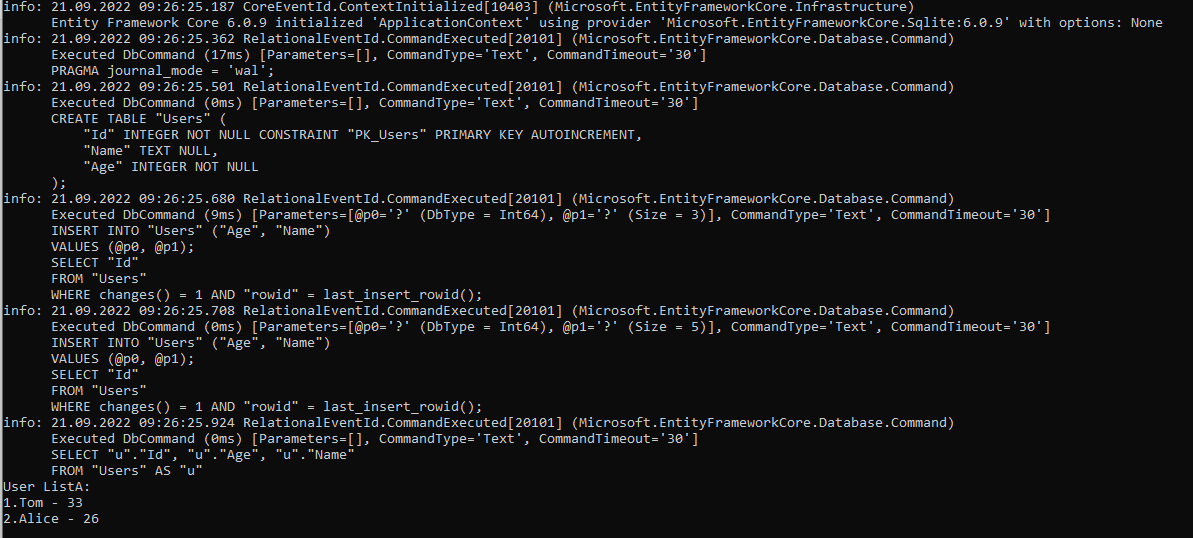
**Уровень логгирования**

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

|  |  |
| --- | --- |
|  |  |

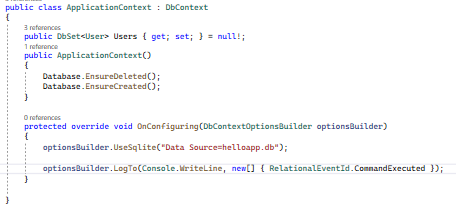
Результат



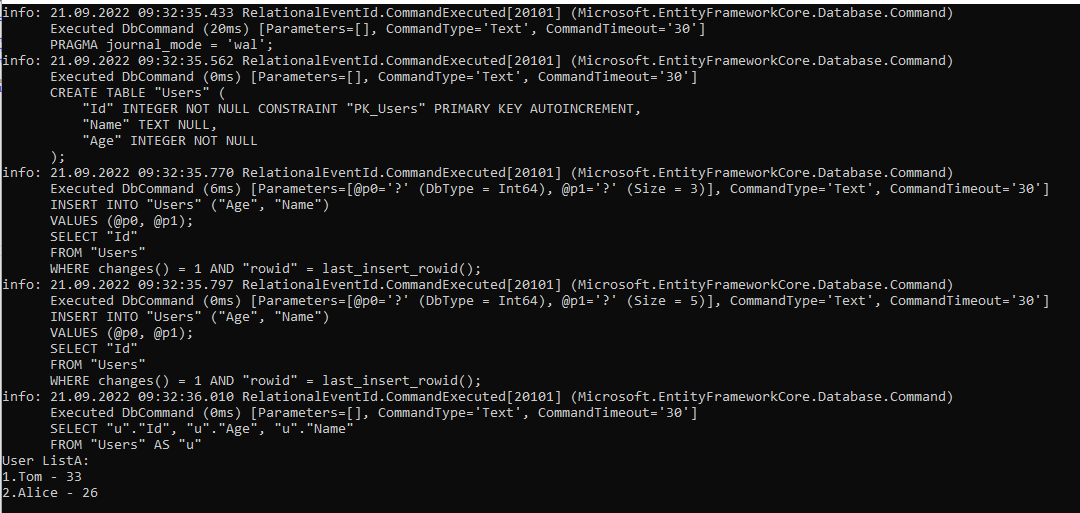
### Конкретизация сообщений

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



Результат



#### Категории сообщений

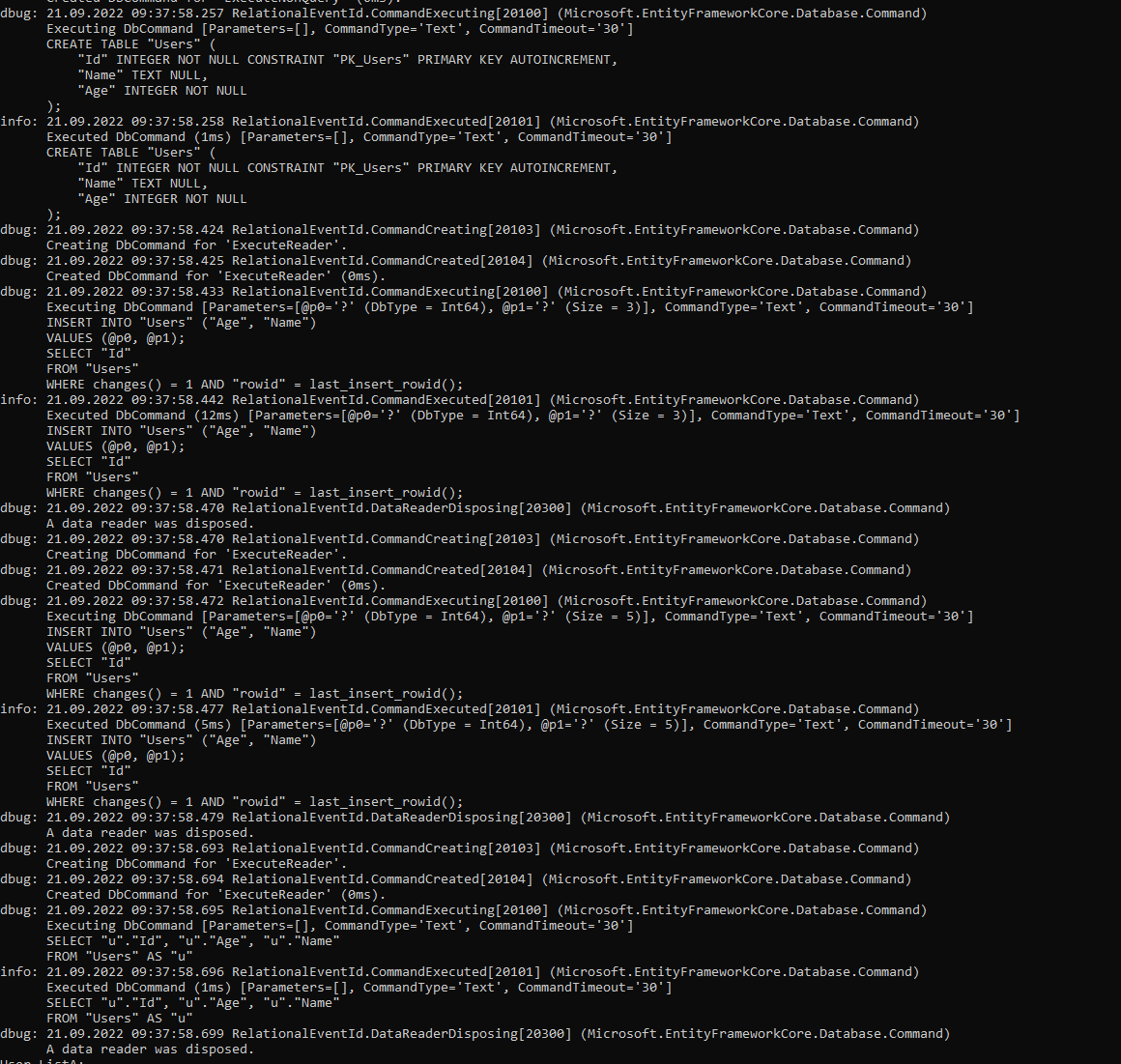
Другим способом фильтрации сообщений представляет использование категорий, которые представлены классом DbLoggerCategory и который позволяет задать нужные категории логгирования:

* 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

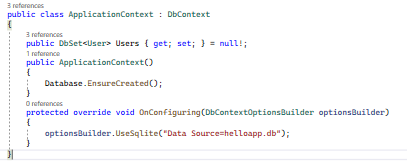
Например, выведем в лог информацию только об исполняемых командах:

optionsBuilder.LogTo(Console.WriteLine, new[] { DbLoggerCategory.Database.Command.Name });

Результат



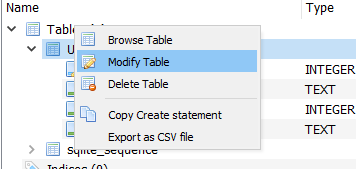
А для работы с базой данных использовался следующий контекст данных:

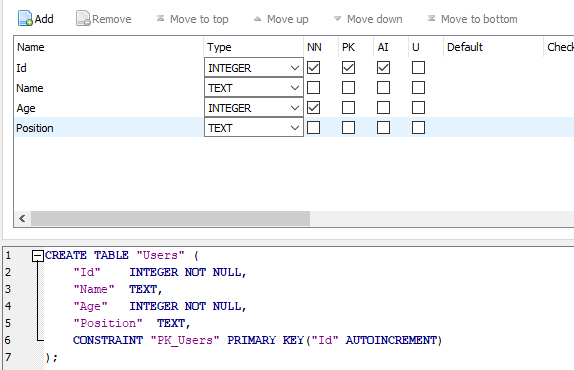


в User

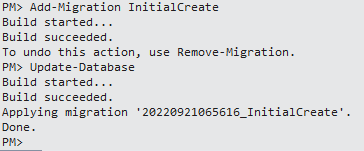
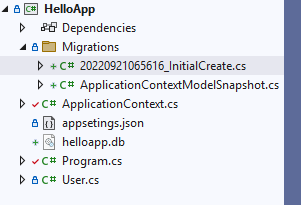
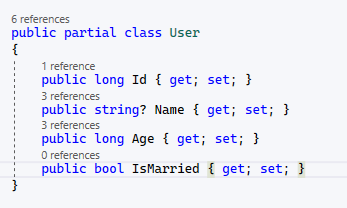
public string? Position { get; set; }   // Новое свойство - должность пользователя

в DB Browser добавляем поле Position



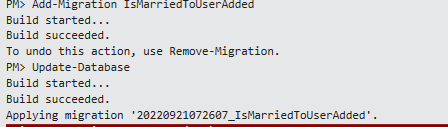


Pakcage manager console

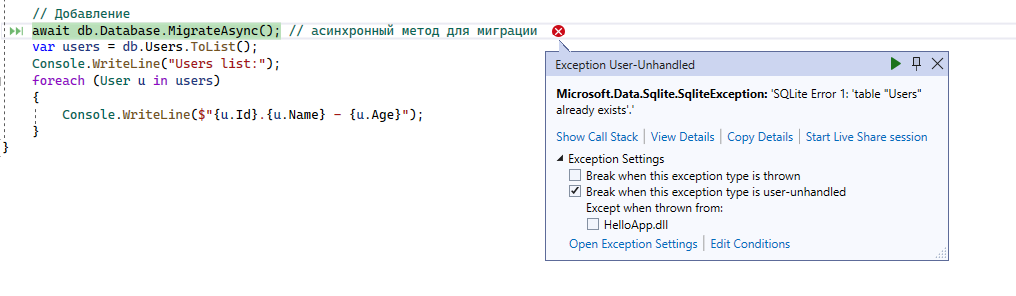


база данных также будет содержать дополнительную таблицу \_EFMigrationsHystory, которая будет хранить информацию о миграциях.

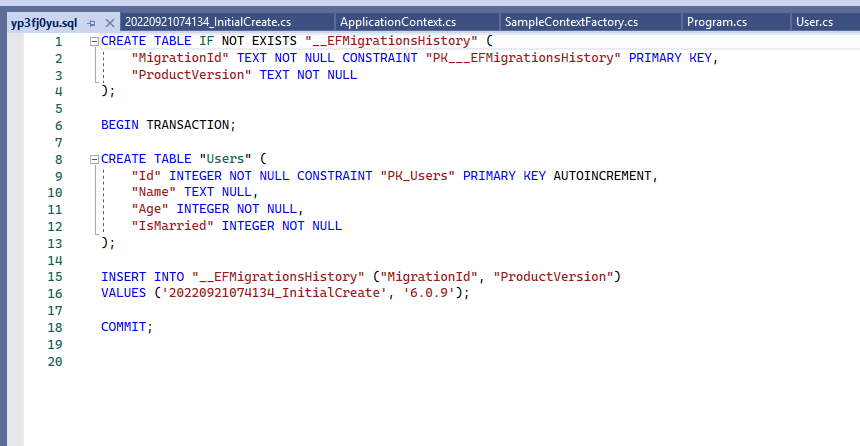




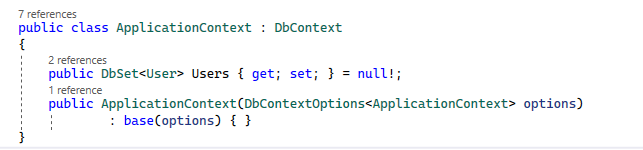


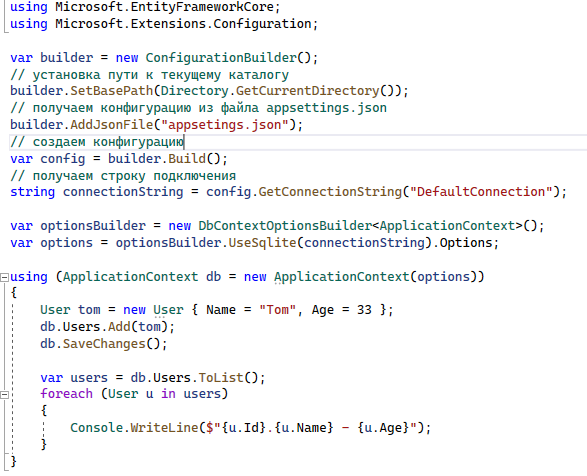


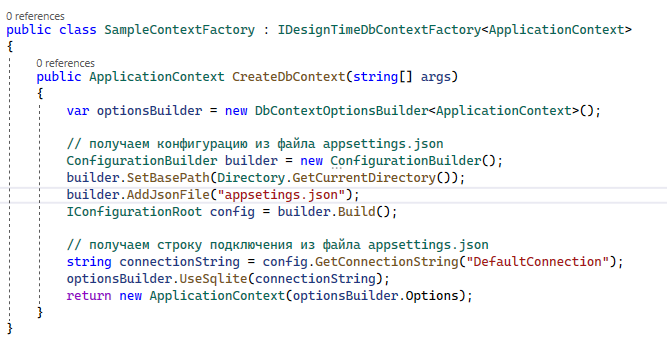




#### Миграция, если конструктор контекста принимает параметр DbContextOptions







**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:

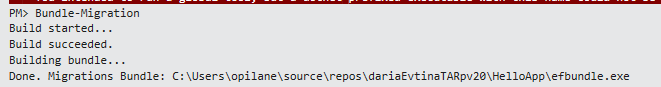
dotnet ef database update

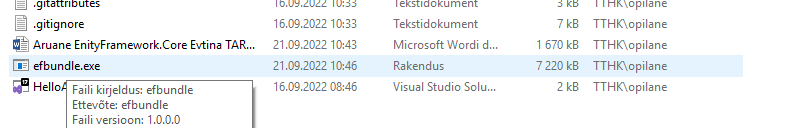
Для создания скрипта sql по миграции применяется следующая команда:

dotnet ef migrations script

С передачей названия миграции:

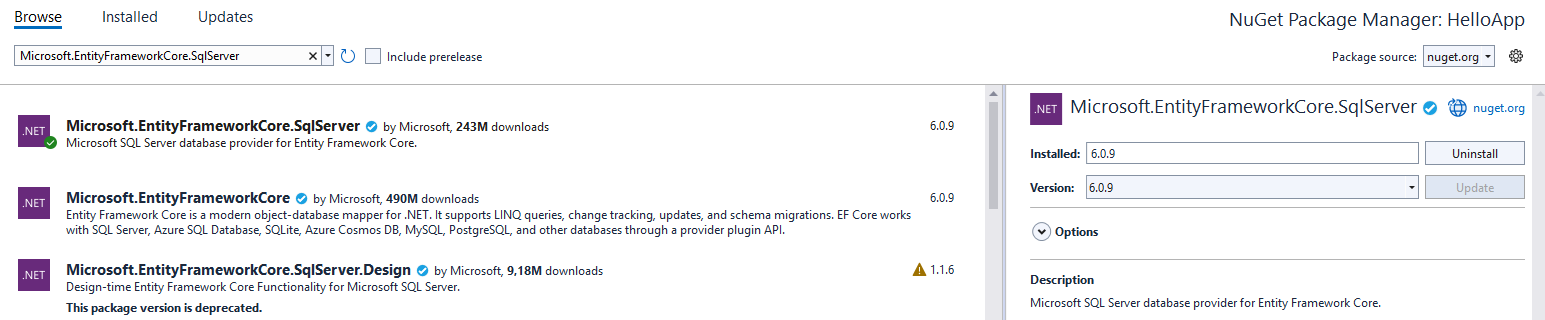
dotnet ef migrations script InitialCreate

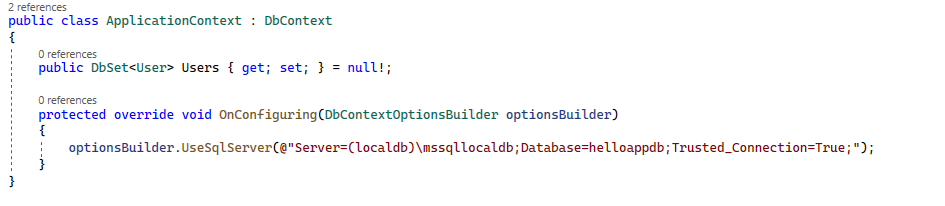




# MySQL SERVER

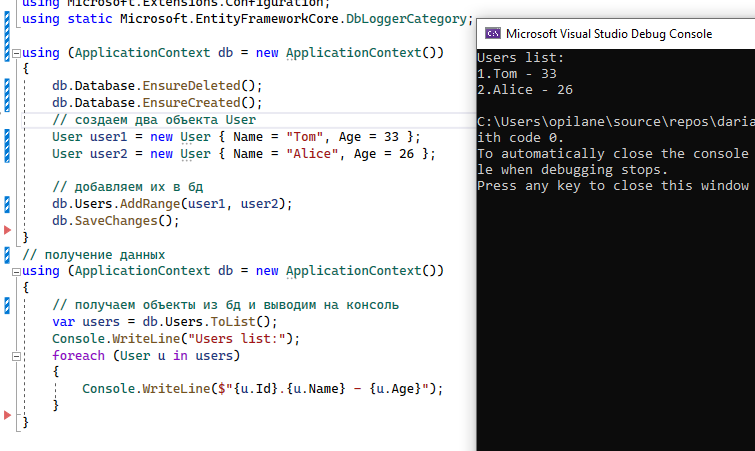
в проект необходимо добавить Nuget-пакет Microsoft.EntityFrameworkCore.SqlServer



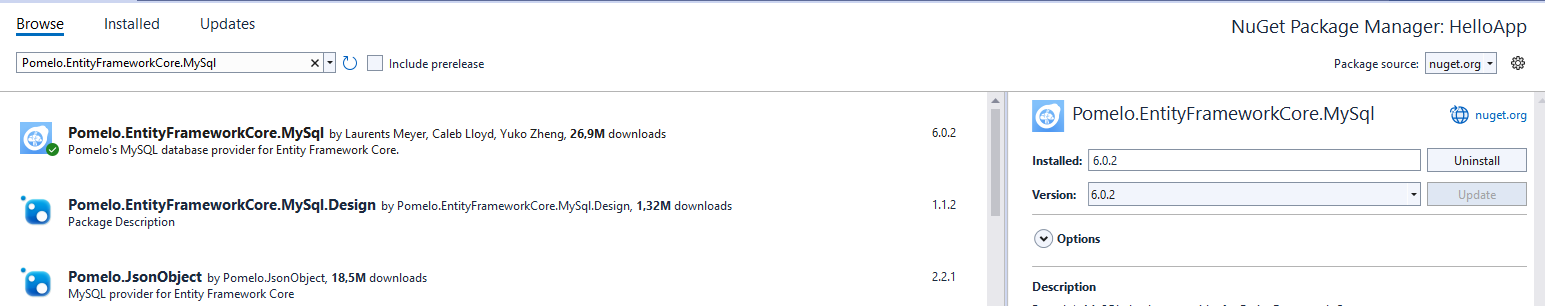


Для подключения к SQL Server у класса DbContextOptionsBuilder определен метод расширения UseSqlServer, в который передается строка подключения для соединения с MS SQL Server. Строка подключения разбивается на несколько частей:

* Server: название сервера. В данном случае используется специальный движок MS SQL Server - localdb, который предназначен специально для нужд разработки. Для MS SQL Server Express этот параметр, как правило, имеет значение .\SQLEXPRESS
* Database: название базы данных
* Trusted\_Connection: устанавливает проверку подлинности

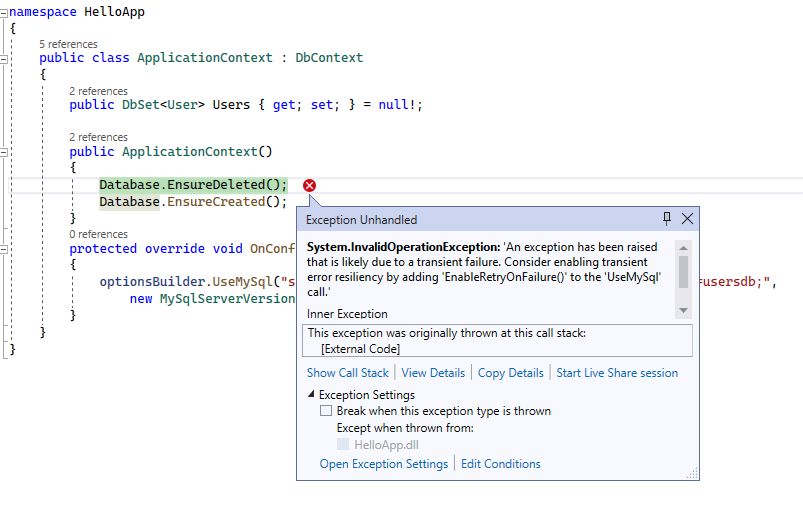


Для подключения к MySQL добавим через Nuget пакет Pomelo.EntityFrameworkCore.MySql





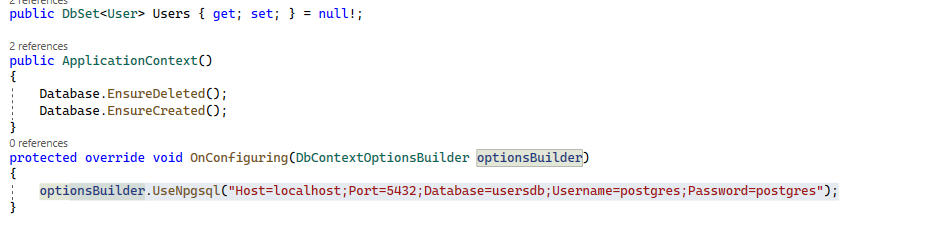
Для работы с MySQL вызывается метод UseMySql(), в который передается строка подключения. В строке подключения указываются адрес сервера (параметр server), имя пользователя в субд (User), его пароль (Password) и имя базы данных (Database).



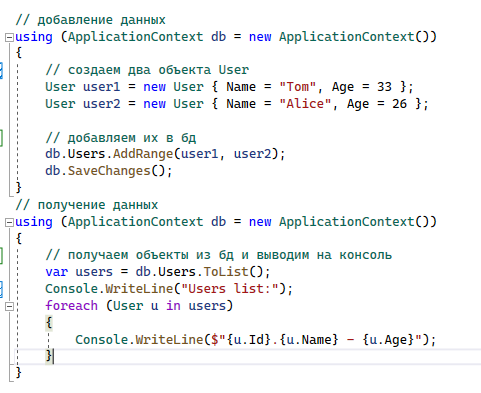
* не смогла подключиться к базе данных т.к такого пользователя нет и такой базы данных

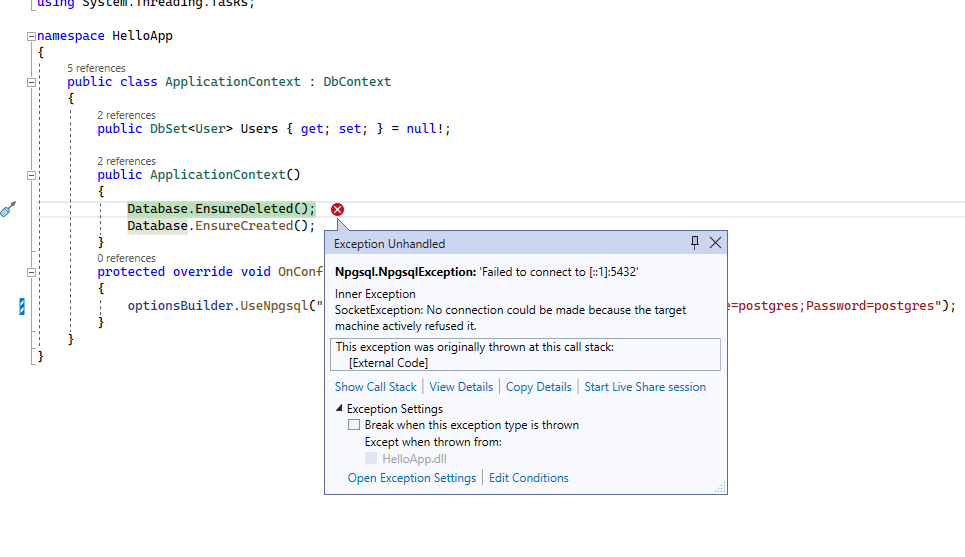
Для работы с базой данных PostgreSQL в проект необходимо добавить через Nuget пакет Npgsql.EntityFrameworkCore.PostgreSQL





Programm.cs

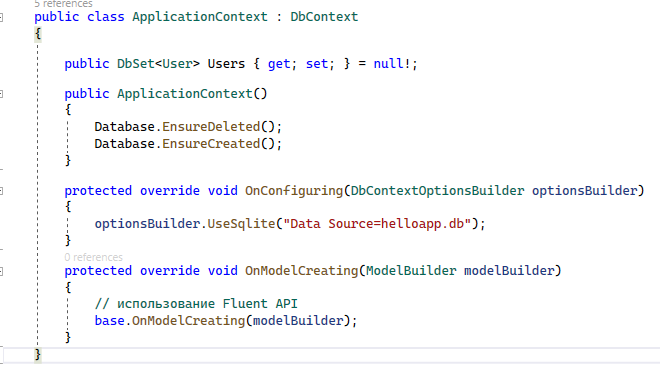




* не смогла подключиться к базе данных т.к такого пользователя нет и такой базы данных

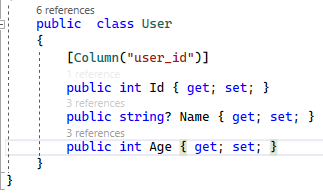
### Fluent API

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.



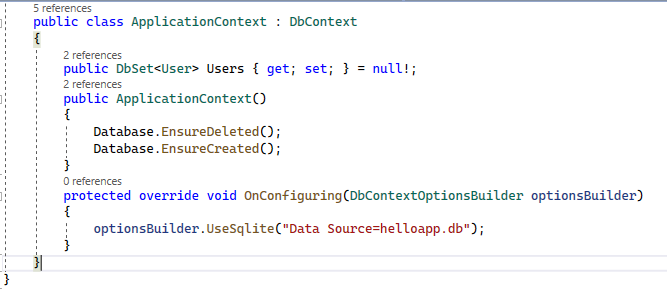
В данном случае атрибут Column представляет аннотацию, которая указывает, что свойство Id будет сопоставляться со столбцом "user\_id" (а не Id, как бы было по умолчанию).

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

* Условности (conventions)
* Fluent API
* Аннотации данных

### 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



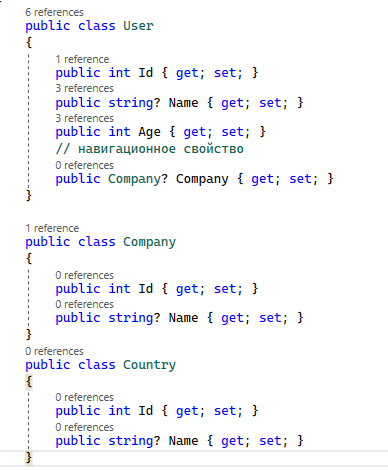
### Ссылочные nullable-типы и DbSet

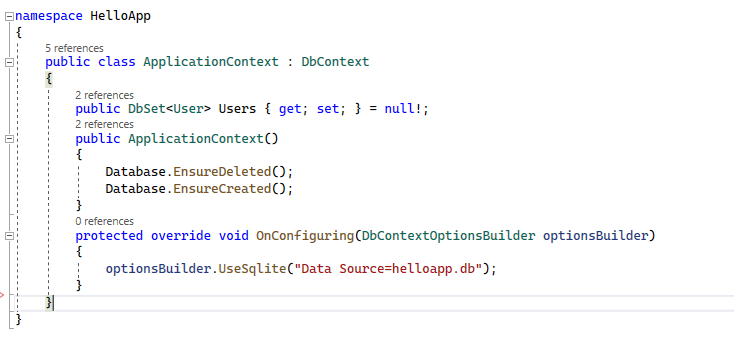
Класс DbSet, как и другие типы, является ссылочным. А, начиная с C# 10 и .NET 6 автоматически применяется функциональность ссылочных nullable-типов. И переменные/свойства тех типов, которые не являются nullable, следует инициализировать некотором значением перед их использованием



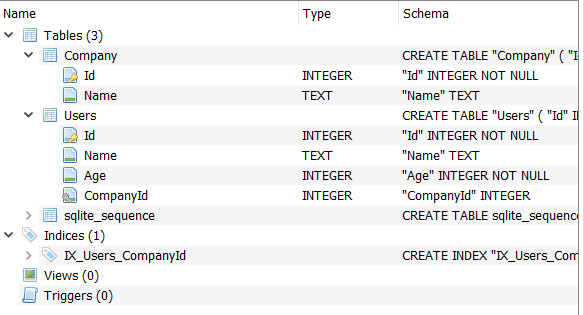
### Включение сущностей в модель без DbSet

Но кроме того, в модель также включаются типы, на которые есть ссылки в сущностях, которые уже включены в модель, например, через свойства DbSet.

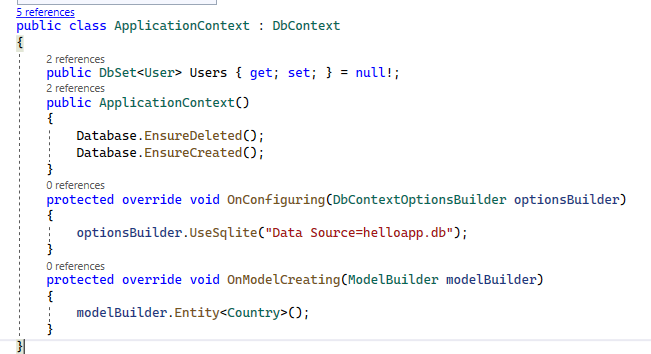




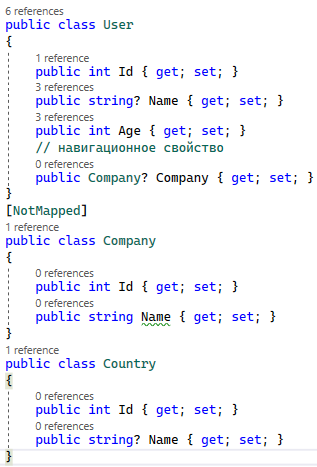
После создания базы данных в ней будут созданы две таблицы: Users и Company. А третий класс - Country никак не используется в сущностях User и Company, для Country нет свойства DbSet в классе контекста, поэтому она не будет включена в контекст и для нее не будет создаваться таблица в бд.



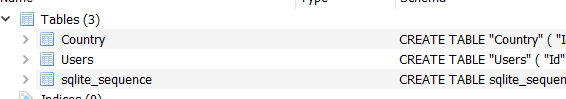
Еще один способ включения сущности в модель представляет вызов Entity() объекта ModelBuilder в методе OnModelCreating():

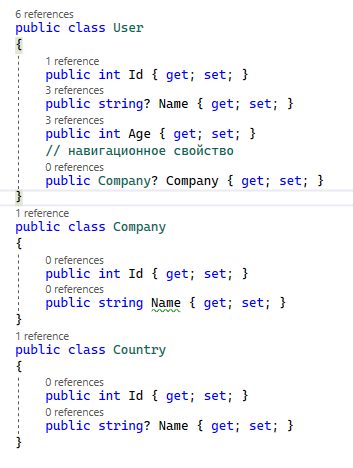


Аннотации данных предполагают установку над классом атрибута [NotMapped]:



При исключении сущности Company в базе данных будет только одна таблица Users, причем она не будет содержать столбца, который бы сопоставлялся со свойством Company класса User:

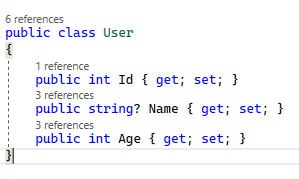






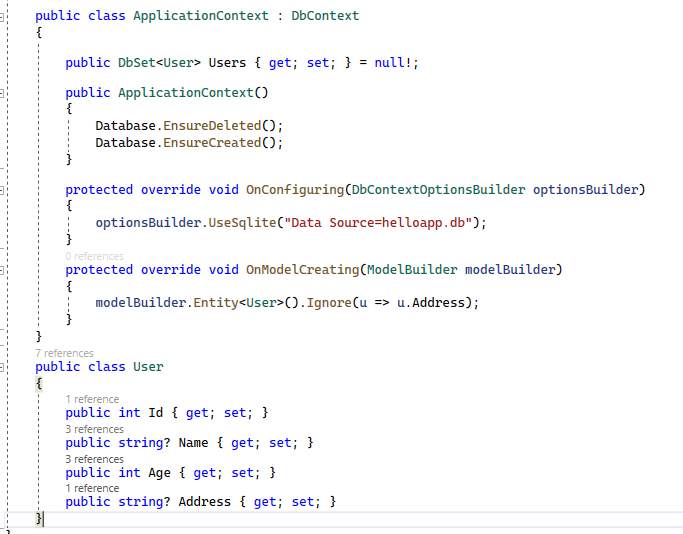
### Свойства сущности

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():

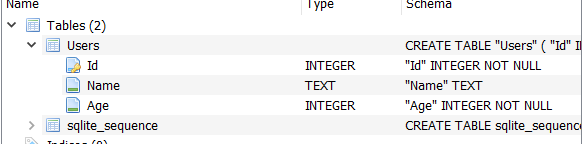


Здесь свойство Address исключается из модели, и для него не будет создаваться столбец в таблице Users.

Исключение с помощью аннотаций данных:

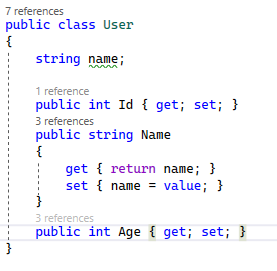
### 

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

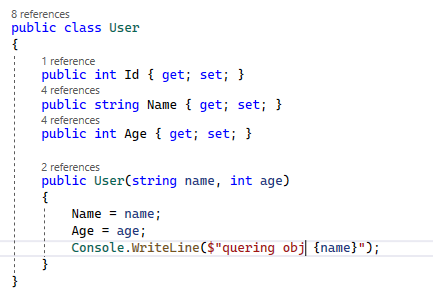
В примере выше применялись автосвойства, которые представляют сокращенную версию свойств без полноценных блоков get и set. Однако свойства не обязательно должны представлять именно автосвойства. Для хранения значений они могут использовать поля класса и иметь полноценные блоки get и set. Например:



### Конструкторы сущностей

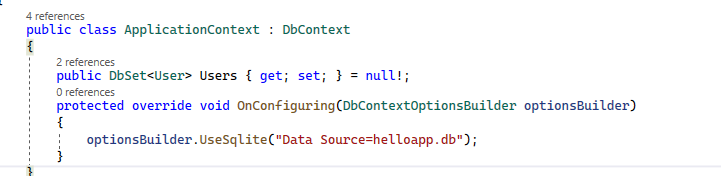
Когда EF Core создает объект сущности, например, при после получения данных из БД, он вначале вызывае конструктор по умолчанию, который не имеет параметров, и затем передает каждому свойству полученные из бд значения.

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:

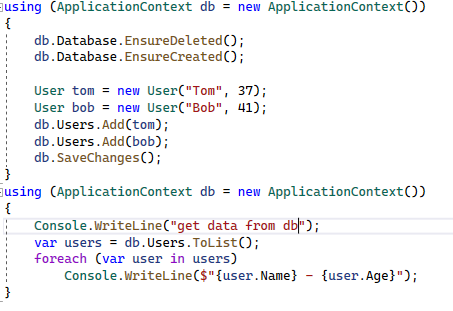


Класс User имеет три свойства и через конструктор устанавливает два из них.

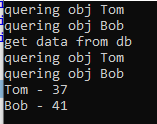
Пусть у нас будет стандартный контекст данных:



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

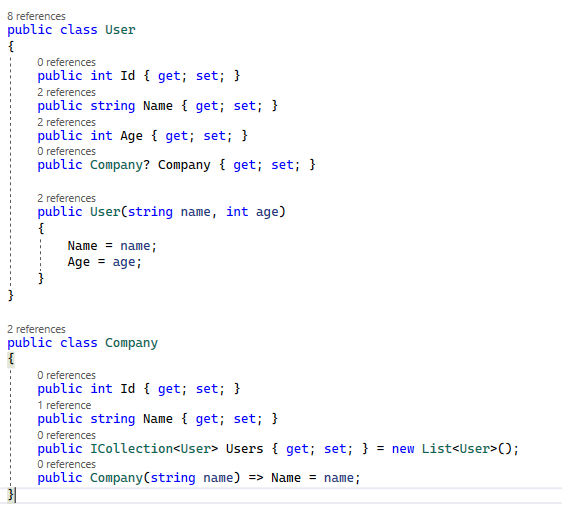


консольный вывод:

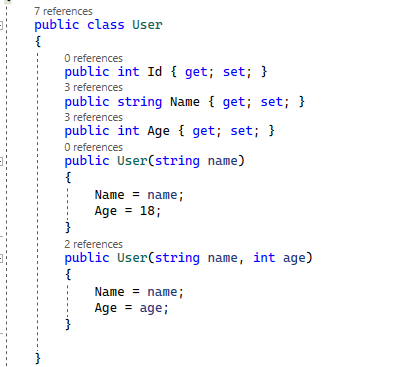


Здесь надо учитывать несколько моментов:

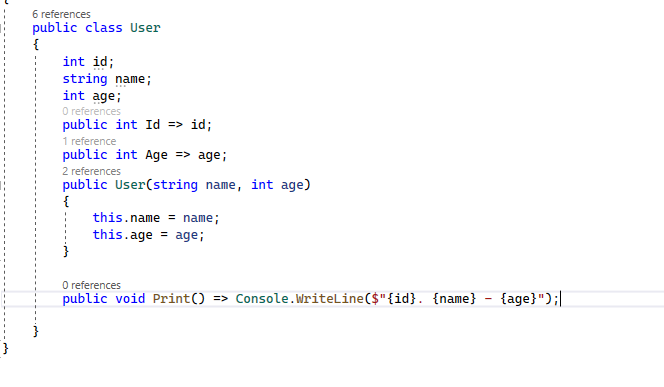
* Необязательно для всех свойств определять в конструкторе свои параметры. Например, свойство Id не устанавливается в конструкторе. Те свойства, для которых в конструкторе не определено параметров, устанавливаются напрямую, как в общем случае.
* Параметры и свойства должны соответствовать по имени и 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:



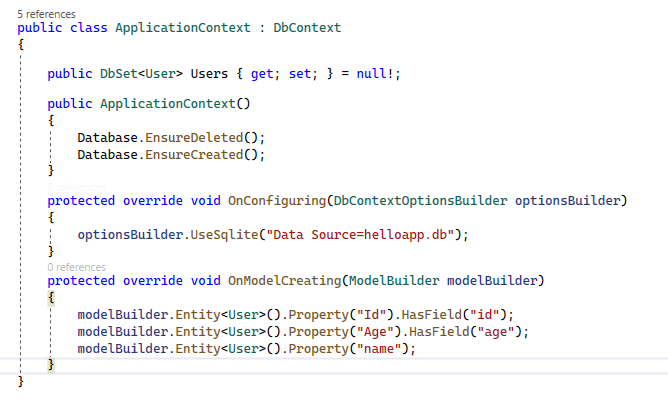
При этом класс может определять несколько конструкторов с разным количеством параметров:

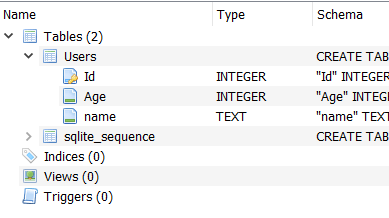


Кроме свойств Entity Framework также может использовать поля класса (в том числе приватные) для сопоставления со столбцами. Например, возьмем следующую сущность:

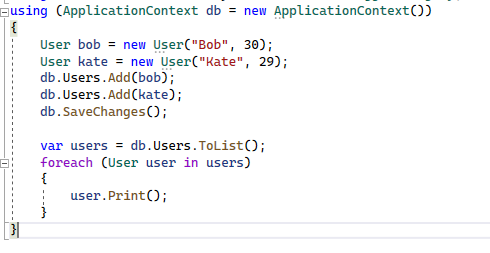
Здесь в классе User определено три поля. Все они приватные, недоступные извне. Кроме того, есть два свойства для чтения, которые возвращают значения полей. Два поля - name и age устанавливаются только через конструктор. Третье поле - id, как мы ожидаем, будет устанавливаться при добавлении объекта сущности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

Настроим класс контекста для применения этих полей:





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



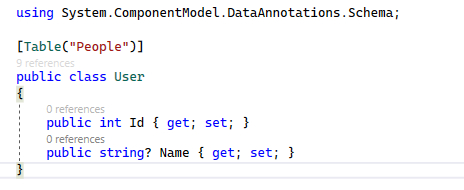


### Сопоставление таблиц

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

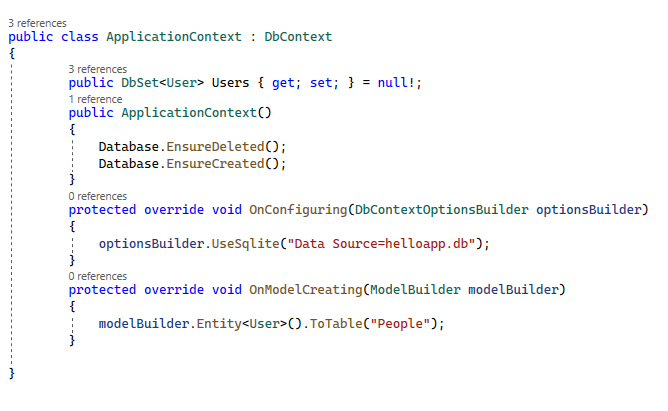
Атрибут Table позволяет переопределить сопоставление с таблицей по имени:



Теперь сущность User будет сопоставляться с таблицей "People".

#### Метод ToTable

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

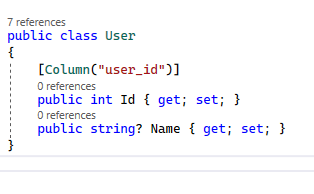


### Сопоставление столбцов

По умолчанию каждое свойство сопоставляется с одноименным столбцом.

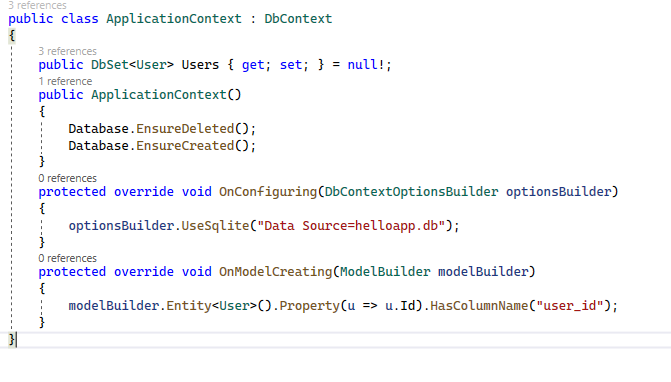
#### Атрибут Column

Атрибут Column переопределяет сопоставление:



#### Метод HasColumnName

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

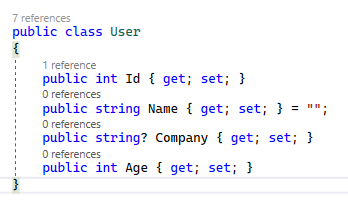


### Обязательные свойства

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.

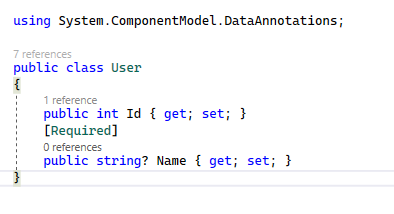
Свойство является обязательным, если оно не допускает значение null.

Например, возьмем следующую модель:



### 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:

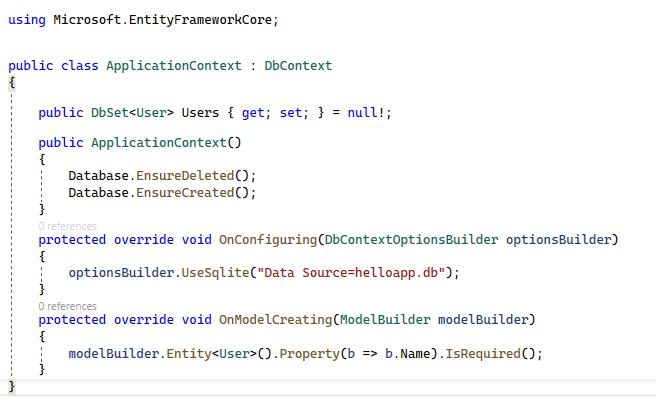


А столбец Name в базе данных будет определен как NOT NULL.

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():



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]:

