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

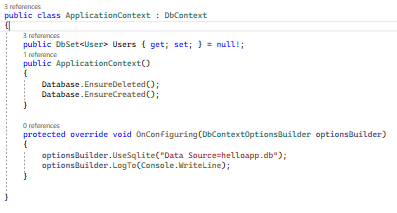


Результат

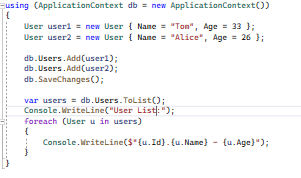


# Log To

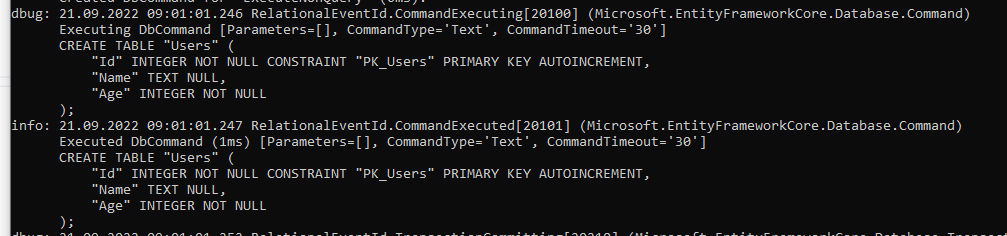
* Для логгирования информации можно использовать метод LogTo(). Он применяется при конфигурации класса контекста данных.

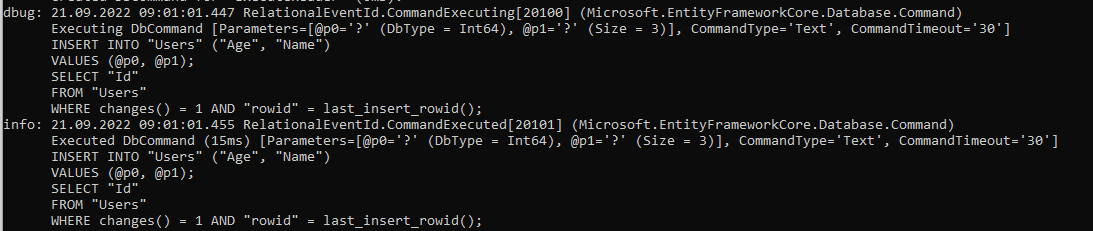


В методе OnConfiguring() у передаваемого в качестве параметра объекта DbContextOptionsBuilder вызывается метод LogTo(), в который передается делегат Action<string> - то есть некоторое действие, которое принимает один параметр типа string и и ничего не возвращает. Именно такое действие представляет традиционный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

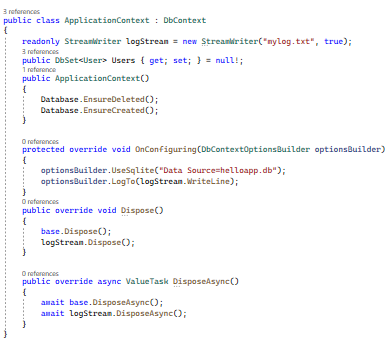


Результат





Другим распространенным способом логгирования является вывод в файл



Результат

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

**Настройка логгирования**

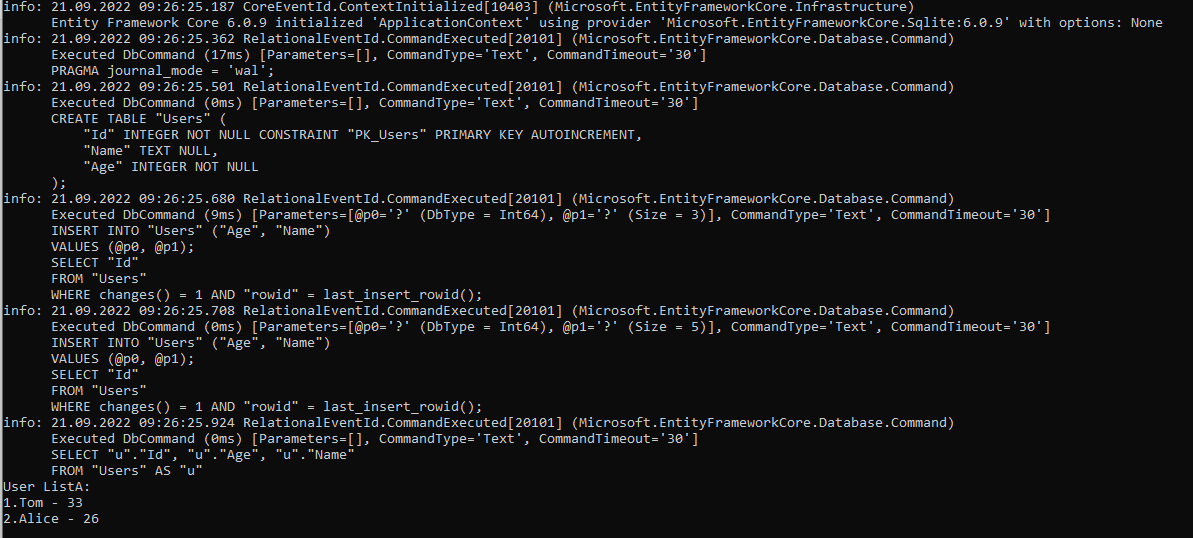
**Уровень логгирования**

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

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

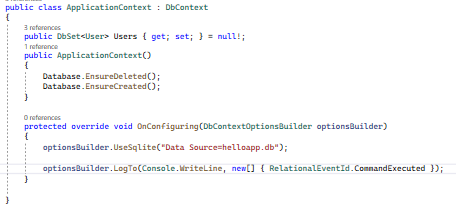
Результат



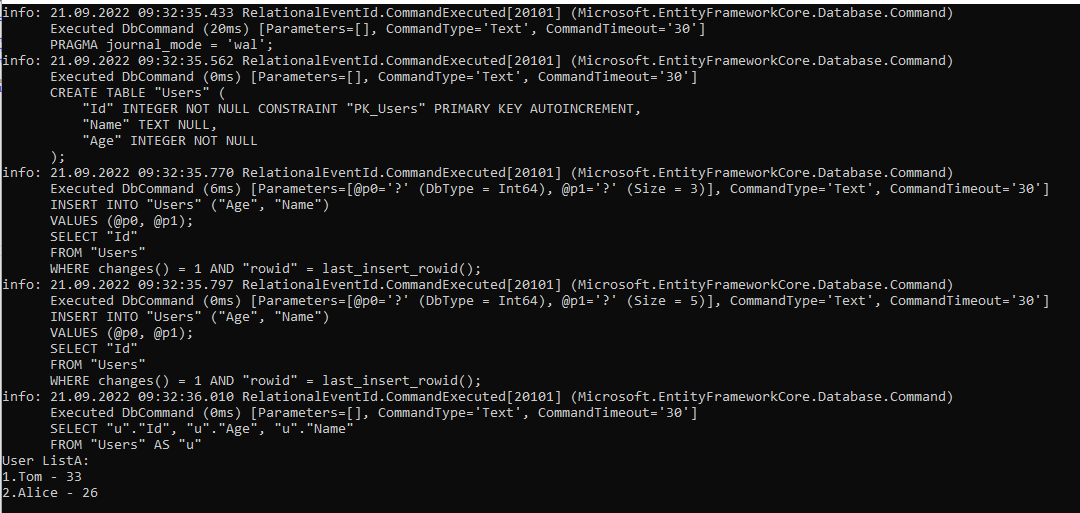
### Конкретизация сообщений

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

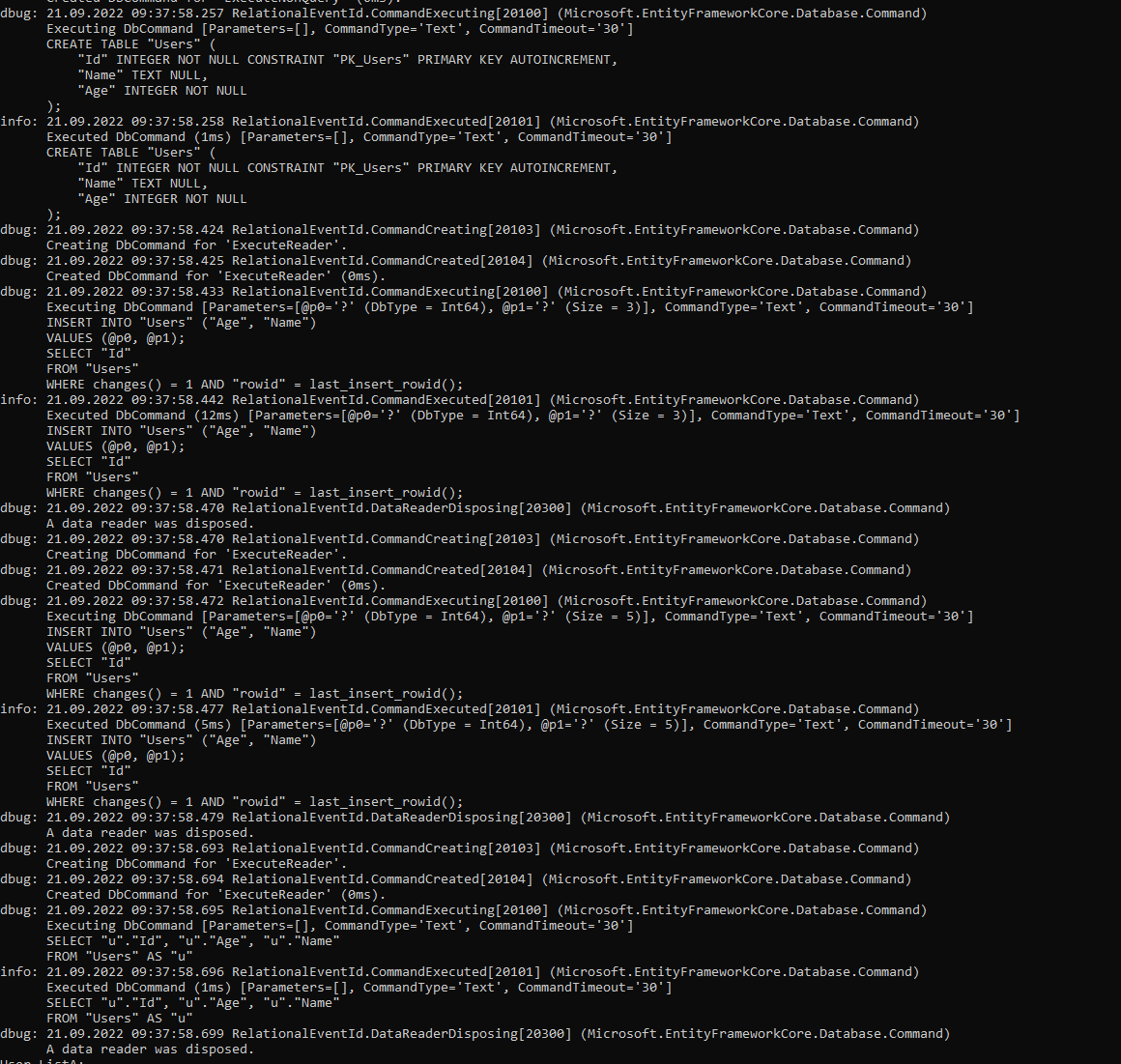
Другим способом фильтрации сообщений представляет использование категорий, которые представлены классом DbLoggerCategory и который позволяет задать нужные категории логгирования:

* 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

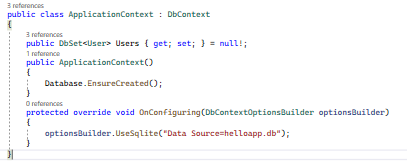
Например, выведем в лог информацию только об исполняемых командах:

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

Результат



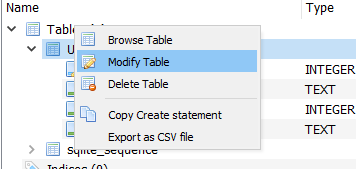
А для работы с базой данных использовался следующий контекст данных:

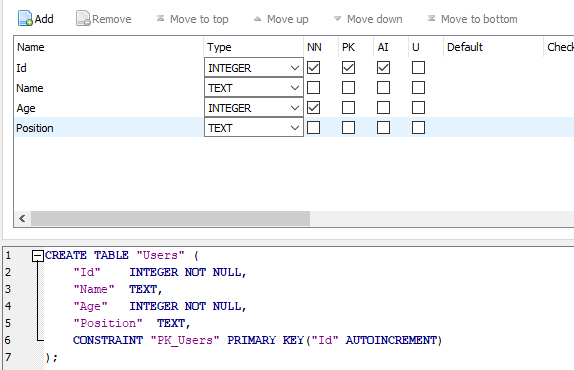


в User

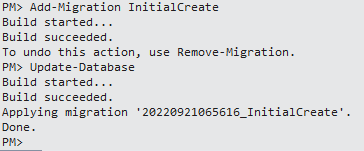
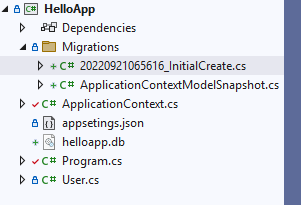
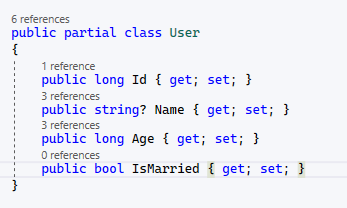
public string? Position { get; set; }   // Новое свойство - должность пользователя

в DB Browser добавляем поле Position





Pakcage manager console



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

