Темы урока

Взаимодействие с БД через EF Core

Настройка логирования работы EF Core для отображения SQL-инструкций

Устанавливаем NuGet пакеты

- Microsoft.Extensions.Logging
- Microsoft.Extensions.Logging.Console

Затем в класс OnlineStoreContext добавляем логирование команд в консоль:

```
public class OnlineStoreContext : DbContext
{
	public static readonly LoggerFactory MyConsoleLoggerFactory =
		new LoggerFactory(new[]
		{
			new ConsoleLoggerProvider(
				(category, level) =>
					category == DbLoggerCategory.Database.Command.Name
					&& level == LogLevel.Information,
				true)
		});
	//...
}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
	optionsBuilder
		.UseLoggerFactory(MyConsoleLoggerFactory)
		.UseSqlServer(_connectionString);
```

```
}
```

Disclaimer *При это мы увидим сообщение о том, что данных подход к добавлению логирования считается устаревшим, однако, на текущий момент это всё ещё остаётся самым простым способом работы с логированием в консольных приложениях. Поскольку для нас это код является не целевым, а вспомогательным, я буду использовать его, так как он наиболее простым образом решает задачу визуализации SQL-запросов, отправляемых серверу со стороны EF Core.*

Контекст

На сегодняшнем уроке мы будем продолжать работать с базой данных `OnlineStoreEF`, созданной с помощью EF Core на прошлом уроке. Чтобы быть в контексте, можно скачать классную работу с прошлого урока — код, на котором мы остановились находится в проекте **L34_C02_working_with_ef_core_final**.

Добавление простых объектов

Давайте начнём с добавления в базу простых объектов, например, создадим нового клиента.

Взаимодействие с БД будет производиться через созданный нами контекст `OnlineStoreContext`: Помните, мы создавали там свойства типа `DbSet<SomeEntity>`?

```
public class OnlineStoreContext : DbContext
{
      public DbSet<Product> Products { get; set; }
      public DbSet<Customer> Customers { get; set; }
      public DbSet<OrderItem> OrderItems { get; set; }
      public DbSet<Order> Orders { get; set; }
}
```

Вставка значений может осуществляться через эти свойства, а может и напрямую.

Давайте посмотрим как это происходит. Чтобы не раздувать огромный метод `Main` в классе `Program`, создадим ещё метод `InsertCustomers`, в котором будем играть с добавлением клиентов нашего магазина:

```
static void Main(string[] args)
{
      //Console.WriteLine("Hello World!");
      InsertCustomers();
}

private static void InsertCustomers()
{
}
```

Теперь создадим нашего нового клиента. Обратите внимание, что согласно дефолтному поведению за установку значения идентификатора нашего объекта отвечает сам EF, поэтому задавать его при

создании не нужно. Что касается самого контекста, он реализует интерфейс `IDisposable`, поэтому основной паттерн его использования — внутри конструкции `using`.

```
private static void InsertCustomers()
{
	var customer = new Customer { Name = "Andrei Golyakov" };
	using (var context = new OnlineStoreContext())
	{
		context.Customers.Add(customer);
	}
}
```

И если мы выполним такой код… (*драматическая пауза* :) ...ничего не произойдёт :)))

Потому что, чтобы изменения локального контекста передать БД, необходимо вызвать метод `SaveChanges`:

```
private static void InsertCustomers()
{
	var customer = new Customer { Name = "Andrei Golyakov" };
	using (var context = new OnlineStoreContext())
	{
		context.Customers.Add(customer);
		context.SaveChanges();
	}
}
```

Давайте поставим breakpoint на строчку `context.SaveChanges();`, запустим наш код в отладке и, после остановки, нажмём F10, чтобы выполнить инструкцию `SaveChanges` и снова остановиться. Если мы посмотрим в консоль, то увидим там, что же EF Core отправил за запрос в БД. Там будет что-то наподобие:

```
Executed DbCommand (133ms) [Parameters=[@p0='?' (Size = 50) (DbType = AnsiString)],
CommandType='Text', CommandTimeout='30']
	SET NOCOUNT ON;
	INSERT INTO [Customers] ([Name])
	VALUES (@p0);
	SELECT [Id]
	FROM [Customers]
	WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

Давайте первый раз подробно разберём, что мы тут видим?

Собственно, верхняя строка — это описание того, что у нас выполняется обычная команда (`CommandType='Text'`) Т.е. это вызов никакой не хранимой процедуры(!), EF в данном случае оперирует обычными текстовыми командами. Зато, видно, что передаётся параметр и именем `@p0`, значение которого от нас скрыто (`@p0='?'`). Видно также, что тип данных параметра — строка длиной 50 символов: `(Size = 50) (DbType = AnsiString)`.

А дальше, просто текст SQL-запроса, очень схожего с запросом на вставку новой записи в таблицу, который мы с вами писали самостоятельно.

Вставляется новая запись:

```
INSERT INTO [Customers] ([Name])
VALUES (@p0);
```

Затем извлекается только что добавленный идентификатор:

```
SELECT [Id]
FROM [Customers]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
```

Этот идентификатор EF Core читает для того, чтобы заполнить поле `Id` нашего объекта `Customer`. Пока мы всё ещё находимся в точке остановки, наведите курсор на объект `customer` и раскройте его свойства, чтобы убедиться, что теперь его поле `Id` заполнено, **и это EF Core сделал не "рассчитав" идентификатор в отрыве от БД, а получил честный Id только что вставленной записи штатными средствами MS SQL Server**.

Можно пойти в SQL Management Studio и убедиться, что запись добавлена в базу :)

К слову сказать, не обязательно обращаться к целевому `DbSet`-у для вставки, можно добавлять сущности напрямую через метод `Add` самого контекста:

```
using (var context = new OnlineStoreContext())
{
	context.Add(customer);
	context.SaveChanges();
}
```

Это удобно, когда нужно вставить сразу несколько сущностей разного типа, однако, на мой взгляд, это несколько ухудшает читабельность

Также можно показать, что нескольких сущностей возможна с помощью метода `AddRange`.

Рассказываем по слайду немного теории про работу метода SaveChanges.

Открываем SQL Server Profiler: в окне SQL Management Studio пункт меню *Tools* > *SQL Server Profiler* (в двух словах объясняя, зачем он нужен — для логирования обращений к SQL Server). Включаем профилирование наших запросов к локальной базе данных. Настройки по умолчанию не показывают транзакционные обёртки, поэтому при создании нового трейса необходимо:

1. Перейти на таб Events Selection
2. Кликнуть флажок Show all events (справа внизу)
3. Раскрыть узел Transactions
4. Кликнуть флажки слева от
    a. TM: Begin Tran starting
    b. TM: Commit Tran completed

После этого выполнить код по вставке ещё раз (Лучше сразу с методами Add и AddRange) и показать, что каждый вызов SaveChanges предваряется открытием транзакции, и завершается её подтверждением:

```
var customer = new Customer { Name = "Alex" };
using (var context = new OnlineStoreContext())
{
      context.Add(customer);
      context.SaveChanges();
}

var customers = new[]
{
      new Customer { Name = "Pavel" },
      new Customer { Name = "Eugeni" }
};
using (var context = new OnlineStoreContext())
{
      context.AddRange(customers);
      context.SaveChanges();
}
```

Вот как будет выглядеть trace-лог для такого кода:

EventClass	TextData	ApplicationName
TM: Begin Tran starting	`BEGIN TRANSACTION`	Core .Net SqlClient Data Provider

RPC:Completed	`exec sp_executesql N'SET NOCOUNT ON;`
	`INSERT INTO [Customers] ([Name])`
	`VALUES (@p0);`
	`SELECT [Id]`
	`FROM [Customers]`
	`WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();`
	
	`',N'@p0 varchar(50)',@p0='Alex'`
		Core .Net SqlClient Data Provider

TM: Commit Tran completed	`COMMIT TRANSACTION`	Core .Net SqlClient Data Provider
...	`...`	...
TM: Begin Tran starting	`BEGIN TRANSACTION`	Core .Net SqlClient Data Provider

RPC:Completed	`exec sp_executesql N'SET NOCOUNT ON;`
	`INSERT INTO [Customers] ([Name])`
	`VALUES (@p0);`
	`SELECT [Id]`
	`FROM [Customers]`
	`WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();`
	
	`',N'@p0 varchar(50)',@p0='Pavel'`
		Core .Net SqlClient Data Provider

RPC:Completed	exec sp_executesql N'SET NOCOUNT ON; INSERT INTO [Customers] ([Name]) VALUES (@p0); SELECT [Id] FROM [Customers] WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity(); ',N'@p0 varchar(50)',@p0='Eugeni'	Core .Net SqlClient Data Provider
TM: Commit Tran completed	COMMIT TRANSACTION	Core .Net SqlClient Data Provider

Можно заметить в trace-логе мы видим имена параметров, в то время, как в логе нашего консольного приложения мы их не видим. По умолчанию значения скрыты из соображений безопасности, однако, для наших нужд это можно изменить добавив в `optionsBuilder` строчку `.EnableSensitiveDataLogging(true)`:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
	optionsBuilder
		.UseLoggerFactory(MyConsoleLoggerFactory)
		.EnableSensitiveDataLogging(true)
		.UseSqlServer(_connectionString);
}
```

Самостоятельная работа

Создаём несколько новых сущностей `Product` используя методы `Add` и `AddRange` соответствующего DbSet-а.

Упражняемся в специально отведённом для этого месте — методе `InsertProducts`.

Должно получиться что-то вроде:

```
private static void InsertProducts()
{
	var product = new Product { Name = "Fenix 5 Plus Sapphire", Price = 73990 };
	using (var context = new OnlineStoreContext())
	{
		context.Products.Add(product);
		context.SaveChanges();
	}

	var products = new[]
	{
		new Product { Name = "Forerunner 645 Music", Price = 42200 },
		new Product { Name = "MARQ Aviator", Price = 208400 }
	};
	using (var context = new OnlineStoreContext())
	{
```

```
            context.AddRange(products);
            context.SaveChanges();
      }
}
```

Выборка простых объектов

Итак, пока мы упражнялись с различными видами вставок и изучением логов, в нашей базе данных появилось некоторое количество клиентов.

Давайте извлечём вставленные данные из нашей БД с помощью EF Core. По аналогии со вставкой создадим метод для чтения данных `SelectCustomers`:

```
static void Main(string[] args)
{
      //Console.WriteLine("Hello World!");
      InsertCustomers();
      SelectCustomers();
}

private static void SelectCustomers()
{
}
```

Чтобы выбрать все сущности определённого типа, можно просто вызвать LINQ-метод `ToList` у соответствующего DbSet-а:

```
using (var context = new OnlineStoreContext())
{
      var allCustomers = context.Customers.ToList();
}
```

Дальше рассказать и показать по слайдам про синтаксис LINQ-запросов:

```
private static void SelectCustomers()
{
      using (var context = new OnlineStoreContext())
      {
            var allCustomers = context.Customers.ToList();

            var allCustomersLinqSyntax = (
                  from c
                  in context.Customers
                  select c
            ).ToList();
      }
}
```

Лично я предпочитаю методы. Субъективно, они удобнее :)

Иногда возникает необходимость пройтись в цикле по запрошенным данным и выполнить над каждым прочитанным элементом некоторые действия. Например:

```
var customers = context.Customers.ToList();
foreach (var customer in customers)
{
      Console.WriteLine(customer.Name);
}
```

При этом сначала выполнится запрос на извлечение данных, он разместится в памяти в виде списка и `foreach` будет пробегать уже по этому списку.

Иногда возникает соблазн сделать вот так:

```
foreach (var customer in context.Customers)
{
      Console.WriteLine(customer.Name);
}
```

Это плохой пример! При выполнении такого кода до завершения последней итерации мы держим соединение с SQL Server открытым. Конечно в случае вывода в консоль это незначительное время, однако, надо понимать, что если мы пытаемся там выполнить какие-то проверки данных или запросить дополнительные данные по этому клиенту, соединение всё это время будет держаться открытым!

Фильтрация запрашиваемых данных

Пара слов о производительности.

Вместо того, чтобы каждый раз создавать экземпляр класса контекста можно создать один экземпляр в рамках использующего контекст класса:

```
class Program
{
      private static OnlineStoreContext _context = new OnlineStoreContext();

      static void Main(string[] args)
      {
            ...
      }
}
```

Создадим новый метод `MoreQueries()`, в котором отфильтруем всех клиентов с именем Andrei:

```
private static void MoreQueries()
{
      var customers = _context.Customers.Where(c => c.Name == "Andrei").ToList();
}
```

Для фильтрации мы используем LINQ-метод Where, которому в качестве параметра передаётся лямбда-выражение для фильтрации.

Если запустить такой метод и посмотреть на лог в консоли, мы увидим там вот такой запрос:

```
SELECT [c].[Id], [c].[Name]
FROM [Customers] AS [c]
WHERE [c].[Name] = 'Andrei'
```

Обратите внимание, что тут нет никаких параметров. Имя **захардкожено** прямо в команде! Это особенность, которой EF обладает с первой версии. Подробно всё расписано на слайде, но если коротко:

- Если искомее значение передаются в виде параметра, то в SQL-команду добавляется параметр;
- Если искомое значение вставляется константой прямо в лямбда-выражение, то в SQL-команду параметр НЕ добавляется.

Можно это продемонстрировать перенеся строку в переменную:

```
var name = "Andrei";
customers = _context.Customers.Where(c => c.Name == name).ToList();
```

Затем по слайду показываем методы LINQ to Entities.

Обращаем внимание, что на пустой коллекции методы `First`, `Single` и `Last` вызовут исключение! Чтобы вместо этого вернуть null, лучше воспользоваться методами `FirstOrDefault`, `SingleOrDefault` или `LastOrDefault`.

Также на слайде обращено внимание на необходимость для `Last`/`LastOrDefault` использовать `OrderBy`.

Для поиска по частичному совпадению можно использовать один из двух вариантов на выбор:

Во-первых, метод Contains транслируется в LIKE-подобную SQL-инструкцию:

```
var customersAndrei = _context.Customers
      .Where(c => c.Name.Contains("Andrei"))
      .ToList();
```

превращается в следующий SQL:

```
SELECT [c].[Id], [c].[Name]
FROM [Customers] AS [c]
WHERE CHARINDEX(N'Andrei', [c].[Name]) > 0
```

Для большей точности можно воспользоваться специальной функцией из набора EF Core (необходимо добавить (`using Microsoft.EntityFrameworkCore;`):

```
var customersAndrei = _context.Customers
      .Where(c => EF.Functions.Like(c.Name, "Andrei%"))
      .ToList();
```

превращается в следующий SQL:

```
SELECT [c].[Id], [c].[Name]
FROM [Customers] AS [c]
WHERE [c].[Name] LIKE 'Andrei%'
```

Причём здесь мы имеем большую гибкость, так как сами управляем шаблоном и можем делать его сколь угодно сложным. В данном случае мы ищем все имена *начинающиеся* со слова “Andrei”.

Обновление данных в БД

Попробуем выполнить простое переименование клиента (создадим отдельный метод `UpdateCustomers`):

```
private static void UpdateCustomers()
{
	var customer = _context.Customers.First();
	customer.Name = "Mr. " + customer.Name;
	_context.SaveChanges();
}
```

У нас вызовется исключение, так как мы указали EF Core, что поле Name является альтернативным ключом. Мы это делали для того, чтобы достичь UNIQUE CONSTRAINT, однако, в EF Core альтернативный ключ используется для внешних связей с другими сущностями. В связи с этим нам необходимо сделать небольшие изменения в нашем методе `OnModelCreating` и создать и применить новую миграцию. Заменяем код по созданию альтернативного ключа:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
	modelBuilder
		.Entity<Customer>()
			.HasAlternateKey(p => p.Name)
			.HasName("UQ_Customers_Name");
	…
}
```

на код по созданию индекса с уникальными значениями:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
	modelBuilder
		.Entity<Customer>()
			.HasIndex(p => p.Name)
			.IsUnique();
	…
}
```

Создаём и применяем миграцию:

```
PM> Add-Migration UpdateCustomerNameIndex
PM> Update-Database
```

Возвращаемся к апдейту имени клиента. Запускаем наш метод `UpdateCustomer`. Это трансформируется в следующие две команды SQL:

```
-- первая
SELECT TOP(1) [c].[Id], [c].[Name]
FROM [Customers] AS [c]

-- вторая
-- Executed DbCommand (145ms) [Parameters=[@p1='7', @p0='Mr. Alex' (Nullable = false)
(Size = 50) (DbType = AnsiString)], CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
UPDATE [Customers] SET [Name] = @p0
WHERE [Id] = @p1;
SELECT @@ROWCOUNT;
```

Это лёгкая задача, так как мы читаем и обновляем данные в одном контексте.

```
private static void UpdateCustomers()
{
      var customer = _context.Customers.First();
      customer.Name = "Mr. " + customer.Name;
      _context.SaveChanges();
}
```

Это лёгкая задача для EF Core. Поскольку контекст сначала читает объект, а потом вносит в него же изменения, он может создать очень эффективную команду на обновление.

Обновление “отсоединённых” объектов (вне контекста)

То, что мы рассматривали в предыдущем примере — было обновление объектов в контексте, т.е. мы обновляли объекты, которые сами только что же и прочитали.

Однако, такой сценарий возможен не всегда. Например, в случае Web API, чтобы не держать соединение контекста мы будем пользоваться схемой где в каждом методе для получения или обновления данных мы будем создавать контекст, пользоваться им и освобождать, используя конструкцию `using` в рамках метода API.

А это означает, что контекст не сможет постоянно быть в курсе того, что происходит в реальной базе данных с объектами. Например, к моменту обновления данных может случится так, что объект уже поменялся.

Рассмотрим пример обновления объекта, о котором изначально EF Core ничего не знает. Пишем отдельный метод `QueryAndUpdateProductDisconnected`. Для наглядности я должен взять объект посложнее, чем Customer, так как у клиента всего два поля — идентификатор и имя. Мне же нужно показать, что происходит с полями, которые и не являются идентификатором и не обновляются по значению. Поэтому я проведу свой пример на объекте типа Product:

```
private static void QueryAndUpdateProductDisconnected()
{
      var product = _context.Products.First();
      product.Price *= 0.1M;

      using (var newContextInstance = new OnlineStoreContext())
      {
            newContextInstance.Products.Update(product);
            newContextInstance.SaveChanges();
      }
}
```

Этот пример отличается от предыдущих тем, что мы читаем объект в одном контексте, а обновляем в другом. Здесь мы вызываем метод `Update` и подсказываем EF Core, что такой объект уже есть, и его данные нужно обновить. Сначала контекст вычитает его текущее состояние (начнёт отслеживание объекта), а потом уже обновит его.

Давайте поставим точку остановки, чтобы посмотреть лог и запустим этот код.

Мы увидим 2 запроса:

```
-- Executed DbCommand (160ms)
SELECT TOP(1) [p].[Id], [p].[Name], [p].[Price]
FROM [Products] AS [p]

-- Executed DbCommand (72ms)
-- [Parameters=[@p2='1', @p0='Fenix 5 Plus Sapphire' (Size = 4000), @p1='7398.999']]
SET NOCOUNT ON;
UPDATE [Products] SET [Name] = @p0, [Price] = @p1
WHERE [Id] = @p2;
SELECT @@ROWCOUNT;
```

Первый из них вычитывает объект в первый контекст.

Второй обновляет объект используя новый, только что созданных контекст #2. И поскольку он ничего не знает о предыдущем состоянии объекта он не может создать максимально эффективный запрос. **Обратите внимание ещё раз вот на что:**

1. Мы обновили в коде только цену объекта!
2. В SQL-запросе на апдейт фигурируют все поля объекта, а не только цена!

Т.е. когда мы использовали метод Update у DbSet-а продуктов мы сказали EF Core - “в этом объекте **что-то** поменялось“ и EF Core на всякий случай приводит все поля объекта к текущим новым значениям.

Удаление данных из БД

Удаление объекта может показаться несколько необычным, если вы ещё не работали с EF Core ранее. Если очень коротко объяснить в чём здесь несоответствие между ожиданием и реальностью,

то это будет звучать так: “**для удаления необходимо иметь объект целиком, просто идентификатора недостаточно**”!

Давайте посмотрим на удаление отслеживаемых объектов. Напишем метод `DeleteWhileTracked`.

```
private static void DeleteWhileTracked()
{
	var customer = _context.Customers.FirstOrDefault(c => c.Name == "Andrei");
	if (customer != null)
	{
		_context.Customers.Remove(customer);
		_context.SaveChanges();
	}
}
```

Это приведёт к исполнению вот такого SQL-кода:

```
-- var customer = _context.Customers.FirstOrDefault(c => c.Name == "Andrei");
SELECT TOP(1) [c].[Id], [c].[Name]
FROM [Customers] AS [c]
WHERE [c].[Name] = 'Andrei'

-- _context.Customers.Remove(customer);
--Executed DbCommand (54ms) [Parameters=[@p0='4'...]
SET NOCOUNT ON;
DELETE FROM [Customers]
WHERE [Id] = @p0;
SELECT @@ROWCOUNT;
```

Т.е. в сам запрос, как видите, передаётся только `Id`.

Аналогичная ситуация с неотслеживаемыми объектами. Напишем метод `DeleteWhileNotTracked`.

```
private static void DeleteWhileNotTracked()
{
	var customer = _context.Customers.FirstOrDefault(c => c.Name == "Eugeni");
	if (customer != null)
	{
		using (var newContextInstance = new OnlineStoreContext())
		{
			newContextInstance.Customers.Remove(customer);
			newContextInstance.SaveChanges();
		}
	}
}
```

```
-- Executed DbCommand (84ms) [Parameters=[], ...]
SELECT TOP(1) [c].[Id], [c].[Name]
FROM [Customers] AS [c]
WHERE [c].[Name] = 'Eugeni'

-- Executed DbCommand (42ms) [Parameters=[@p0='9'], ...]
SET NOCOUNT ON;
```

```
DELETE FROM [Customers]
WHERE [Id] = @p0;
SELECT @@ROWCOUNT;
```

Но, что если у вас нет целого объекта, а есть только его идентификатор? К сожалению, это означает, что сначала вам придётся вычитать объект из базы и потом уже его удалить.

Альтернативой будет удаление по Id через вызов прямого SQL-запроса на удаление или вызов соответствующей хранимой процедуры:

```
_context.Database.ExecuteSqlCommand(
      "EXEC DELETE FROM [dbo].[Customers] WHERE Id = {0}", 1);
```

Домашнее задание

- Попробовать самостоятельно реализовать библиотеку слоя данных для чат-бота (по сути интерфейс `IReminderStorage`) на базе SQL Server не через примитивы ADO.NET Core, а через Entity Framework Core. **Reminder.Storage.SqlServer.EF**
- Для самопроверки рекомендуется также написать библиотеку с тестами **Reminder.Storage.SqlServer.EF.Tests**.