

## Темы урока

<b>Entity Framework Core</b>	<b>1</b>
Первое знакомство: EF Core	1
Миграции	2
Первый проект с EF Core (Code First)	2
Созаём приложение и доменные классы	2
Рефакторинг классов	3
Добавляем EF Core с помощью NuGet Package Manager	4
EF Core Data Model	4
EF Core Migrations	5
Изучаем файлы созданной Миграции	7
Применяем Миграцию	7
Изучаем созданную БД	7
Тюнинг схемы БД	8
Data Annotations	8
Fluent API	9
Scaffolding	10
<b>Домашнее задание</b>	<b>11</b>

## Entity Framework Core

### Первое знакомство: EF Core

Рассказываем по слайду что EF Core это кроссплатформенная ORM. Рассказываем про ORM.

Платформа Entity Framework представляет собой набор технологий ADO.NET, обеспечивающих разработку приложений, связанных с обработкой данных.

Entity Framework (EF) Core — это кроссплатформенная и расширяемая ORM с открытым исходным кодом.

ORM (Object-Relational Mapping) — объектно-реляционное отображение, или преобразование) — технология, позволяющая связывать базы данных с концепциями объектно-ориентированных языков программирования.

ORM помогает работать с данными, как с объектами, т.е. на более высоком уровне, нежели подключения и SQL-запросы.

Хорошо бы добавить какие есть плюшки (например, возможность работы с большим количеством популярных СУБД - Oracle, MySQL, PostgreSQL, и конечно же MS SQL Server), но это на усмотрение лектора.

## Миграци

### Первый проект с EF Core (Code First)

#### Созадём приложение и доменные классы

Мы напишем приложение на базе EF Core на тему недавнего примера небольшого интернет-магазина.

Вспоминаем схему БД OnlineStore. У нас было 4 таблицы:

- Customer: Id, Name
- Product: Id, Name, Price
- Order: Id, CustomerId, OrderDate, Discount
- OrderItem: OrderId, ProductId, NumberOfItems

Вообще доменные классы необходимо держать в отдельной доменной сборке, однако для простоты демонстрации мы будем делать всё на базе консольного приложения.

Создадим простое консольное приложение (Core), сделаем внутри него папку Domain и создадим классы этих сущностей в папке. (Создаём сущности один-в-один трансформируя поля соответствующих таблиц в свойства).

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public DateTimeOffset OrderDate { get; set; }
    public decimal Discount { get; set; }
}

public class OrderItem
{
    public int OrderId { get; set; }
    public int ProductId { get; set; }
```

```
        public int NumberOfItems { get; set; }  
    }
```

## Рефакторинг классов

Теперь откинемся на спинку кресла и посмотрим на это немного со стороны... До того, как мы начали говорить о БД, нормализации и внешних ключах, вы бы не стали проектировать наши классы таким образом. Вы стали думать как DB-разработчики, но у них под рукой есть инструкции SELECT и JOIN и они могут быстро объединить данные создав сущность состоящую не из идентификаторов, а только важных для бизнес-задачи данных.

Давайте попробуем вернуть объектный облик нашим классам.

Один пример я разберу тут подробно, дальше по аналогии.

Разберёмся с сущностью `OrderItem`. Она содержит две связи (два внешних ключа) — на сущность продукт (`ProductId`) и на сущность заказа (`OrderId`).

Заменяем `ProductId` на просто `Product`.

А вот ссылка на заказ — `OrderId` — здесь вообще инвертирована для возможности организовать связь наиболее естественным для реляционной базы образом. Однако, с точки зрения бизнес-логики, это лишь организация хранения информации о том, что в заказе содержится список из нескольких элементов заказа.

Таким образом, из этого класса `OrderItem` вообще убирается, а вот в классе `Order` появляется список элементов `OrderItem`.

Итого, получаем вот такие классы (`Customer` и `Product` без изменений):

```
public class Customer  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
}  
  
public class Product  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public decimal Price { get; set; }  
}  
  
public class OrderItem  
{  
    public int ProductId { get; set; }  
    public int NumberOfItems { get; set; }  
}
```

```
public class Order
{
    public Order()
    {
        OrderItems = new List<OrderItem>();
    }

    public int Id { get; set; }
    public Customer Customer { get; set; }
    public DateTimeOffset OrderDate { get; set; }
    public decimal Discount { get; set; }
    public List<OrderItem> OrderItems { get; set; }
}
```

## Добавляем EF Core с помощью NuGet Package Manager

Существует пакет базовой логики, который называется `Microsoft.EntityFrameworkCore`. Актуальная версия 2.2.4, однако важно здесь, что в зависимостях у него находится `.NETStandard v.2.0`. Т.е. мы могли бы писать доменную логику в библиотеках в отрыве от конкретной имплементации стандартных классов .NET.

Однако, вернёмся к нашей библиотеке. Она содержит базовые классы EF, однако в ней нет методов для работы с конкретной базой данных. Провайдеры различных баз данных лежат в отдельных NuGet-пакетах и также требуются к установке.

Есть более простой метод, чем набирать все необходимые пакеты по одному. Нужно поставить пакет конкретного поставщика БД и все необходимые для начала работы пакеты подтянутся автоматически, так как они находятся в зависимостях у целевого пакета. Давайте найдём пакет `Microsoft.EntityFrameworkCore.SqlServer`. Установим его.

После установки можно пройтись в Solution Explorer по его зависимостям и зависимостям его зависимостей и посмотреть, что он “тянет” за собой.

## EF Core Data Model

Создаём класс, который будет определять модель нашей базы данных `OnlineStoreContext`. Эта часть как раз и будет играть роль Data Access Layer’a, так что мы располагаем её в папке Data нашего проекта. Наследуемся от `DbContext`.

Добавляем DbSet’ы.

Затем мы должны указать строку подключения, в нашем текущем примере строка подключения будет захардкожена, в дальнейшем мы увидим как можно передавать её динамически.

```
public class OnlineStoreContext : DbContext
{
    private readonly string _connectionString;

    public DbSet<Product> Products { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<OrderItem> OrderItems { get; set; }
    public DbSet<Order> Orders { get; set; }

    public OnlineStoreContext()
    {
        _connectionString =
            @"Data Source=localhost\SQLEXPRESS;" +
            "Initial Catalog=OnlineStoreEF;" +
            "Integrated Security=true;";
    }

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

## EF Core Migrations

Теория про миграции сводится к тому, что при изменениях в классах модели данных создаётся migration сущность, которая применяется в БД и меняет её схему.

Поскольку создание таких сущностей это нетипичная для обычной работы приложения вещь (речь идёт об изменениях в коде приложения), то их создание также вынесено в отдельный NuGet-пакет — `Microsoft.EntityFrameworkCore.Tools`. Добавляем его к нашему приложению.

Идём в окно Package Management Console.

► **Важно!** В Package Management Console в выпадающем списке Default Project должно быть выбрано запускаемое приложение (Console/Web), причём то же самое, что выставлено как “Set as StartUp Project” в Solution Explorer.

Пишем команду:

```
PM> get-help entityframeworkcore
```

возможно придётся обновить файлы справки, соглашаемся.

В итоге получим следующую информацию:

The following Entity Framework Core commands are available.

Cmdlet	Description
Add-Migration	Adds a new migration.
Drop-Database	Drops the database.
Get-DbContext	Gets information about a DbContext type.
Remove-Migration	Removes the last migration.
Scaffold-DbContext	Scaffolds a DbContext and entity types for a database.
Script-Migration	Generates a SQL script from migrations.
Update-Database	Updates the database to a specified migration.

Команда Add-Migration создаёт “миграцию” — SQL-скрипт на изменение схемы данных БД согласно модели данных (нашему контексту БД).

Команда Update-Database применяет указанную миграцию (запускает подготовленный SQL-скрипт).

Создаём миграцию. Для этого выполняем команду Add-Migration с указанием имени миграции:

```
PM> Add-Migration InitialCreate
```

Видим ошибку:

```
Your startup project 'L33_C01_working_with_ef_core' doesn't reference
Microsoft.EntityFrameworkCore.Design. This package is required for the Entity Framework
Core Tools to work. Ensure your startup project is correct, install the package, and try
again.
```

Добавляем в зависимости NuGet-пакет .

**Запускаем ещё раз.**

Ошибка:

```
The entity type 'OrderItem' requires a primary key to be defined.
```

Добавляем поле в Id в класс OrderItem:

```
public class OrderItem
{
    public int Id { get; set; }
    public Product Product { get; set; }
    public int NumberOfItems { get; set; }
}
```

## Запускаем ещё раз.

### Изучаем файлы созданной Миграции

Файлы располагаются в папке Migrations. Они достаточно читабельны, не смотря на то, что это ещё не SQL-код, а лишь промежуточные инструкции к написанию SQL-кода.

Чтобы увидеть непосредственно SQL-код, можно запустить в Package Management Console выполнить инструкцию: Script-Migration.

### Применяем Миграцию

Для среды разработки рекомендуется накатывать миграции используя PS-команду Update-Database.

Для production-среды рекомендуется использовать SQL-скрипт.

Обратите внимание, что в первый раз базы данных не существует. Когда миграция накатывается через PS-команду update-database, БД будет создана автоматически.

Если же мы накатываем SQL-скрипт вручную, необходимо предварительно создать пустую базу данных!

Запускаем PS-команду

```
PM> Update-Database -verbose
```

Параметр verbose позволит увидеть все шаги обновления нашей БД.

### Изучаем созданную БД

Обращаем внимание, что сгенерированная база получилась очень похожа на нашу собственную. В ней определены первичные и внешние ключи, индексы для всех полей внешних ключей, и даже отношение между таблицами Orders и OrderItems реализовано в точности, как мы задумали, когда проектировали БД сами.

## Тюнинг схемы БД

### Data Annotations

Однако не всё так гладко, как кажется с первого взгляда! Посмотрите внимательно, например, на поле `Name` таблицы `Customers`. Оно допускает пустые значения и имеет тип данных `NVARCHAR(MAX)`.

Чтобы это исправить, необходимо воспользоваться Data Annotations атрибутами. Изменяем класс `Customer` следующим образом (потребуется добавить `using System.ComponentModel.DataAnnotations;`):

```
public class Customer
{
    public int Id { get; set; }

    [Required]
    [MaxLength(50)]
    public string Name { get; set; }
}
```

Теперь создадим новую миграцию. В окне `Package Management Console` запускаем команду:

```
PM> Add-Migration UpdateCustomerName
```

Смотрим на предупреждение, смотрим, какой получится скрипт (посмотрим только разницу, принесённую последней миграцией):

```
PM> Script-Migration -From InitialCreate -To UpdateCustomerName
```

Применяем изменения:

```
PM> Update-Database -verbose
```

Мы разобрались с пустыми значениями и максимальной длиной, однако нам также хотелось бы увидеть здесь `VARCHAR` вместо `NVARCHAR`.

Для этого добавим атрибут `Column` к полю `Name` со следующими параметрами (потребуется добавить `using System.ComponentModel.DataAnnotations.Schema;`):

```
[Column("Name", TypeName = "VARCHAR(50)")]
public string Name { get; set; }
```

Как видно, мы также можем задать имя поля отличное от значения по умолчанию.

То же можно сделать и для таблицы, используя атрибут `Table`:

```
[Table("Customer", Schema = "dbo")]
public class Customer
{
    ...
}
```



```
}
```

```
PM> Add-Migration UpdateCustomerName2
PM> Script-Migration -From UpdateCustomerName -To UpdateCustomerName2
PM> Update-Database
```

## Fluent API

Надо упомянуть, что вообще понятие fluent-интерфейс - это часто используемый в C# подход. Например при подготовке WebHost в Web API и в более обыденных местах, например в StringBuilder. Поэтому лучше говорить об этом не опуская контекста — EF Core Fluent API.

Можно переопределить метод `OnModelCreating` метод в производном контексте и использовать `ModelBuilder` API для настройки модели.

Это наиболее эффективный метод настройки, который позволяет задать конфигурацию без изменения классов ваших сущностей. Конфигурация `Fluent API` имеет самый высокий приоритет и переопределяет и соглашения и `Data Annotations`.

Для начала давайте определим ограничение на уникальность `Name` всё той же таблицы `Customer`:

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

```
PM> Add-Migration UpdateCustomerName3
PM> Script-Migration -From UpdateCustomerName2 -To UpdateCustomerName3
PM> Update-Database
```

Теперь давайте посмотрим на нашу сущность `OrderItem`. Нам пришлось добавить туда идентификатор, чтобы первый раз не спорить с EF Core, однако, мы знаем, что для нас первичным ключом будет сочетание полей `OrderId` и `ProductId`. Нам придётся немного поменять сам класс сущности (заменить `Id` на `OrderId`):

Ну и раз уж мы его меняем, давайте попробуем изменить поле `NumberOfItems` — укажем необходимость иметь значение с ограничением от 1 до 100:

```
public class OrderItem
{
    public int OrderId { get; set; }
    public Product Product { get; set; }
    [Range(1, 100)]
    public int NumberOfItems { get; set; }
}
```

```
}
```

и дописать следующую конструкцию в modelBuilder:

```
public class OnlineStoreContext : DbContext
{
    ...
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        ...
        modelBuilder
            .Entity<OrderItem>()
                .HasKey("OrderId", "ProductId")
                .HasName("PK_OrderItems");
    }
}
```

```
PM> Add-Migration UpdateOrderItems
```

```
PM> Script-Migration -From UpdateCustomerName3 -To UpdateOrderItems
```

```
PM> Update-Database
```

К сожалению, как мы видим, не все изменения, которые мы запросили у EF Core, воплотились в жизнь так, как нам хотелось бы. Очередность столбцов и проверка на значения осталась за кадром. Первичный ключ, также создан не-кластерным.

Чтобы сделать его таковым необходимо добавить специфичный для SQL Server метод:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    ...
    modelBuilder
        .Entity<OrderItem>()
            .HasKey("OrderId", "ProductId")
            .HasName("PK_OrderItems");
            .ForSqlServerIsClustered(true);
}
```

```
PM> Add-Migration UpdateOrderItems2
```

```
PM> Script-Migration -From UpdateOrderItems -To UpdateOrderItems2
```

```
PM> Update-Database
```

## Scaffolding

Просто показать пример, если будет время.

Запустить скрипт на существующую базу в новом проекте (не забыть добавить необходимые NuGet-пакеты):

```
PM> Scaffold-DbContext -provider Microsoft.EntityFrameworkCore.SqlServer
    -connection "Data Source=localhost\SQLEXPRESS;Initial
    Catalog=Correspondence;Integrated Security=true;"
```

Показать сгенерированные классы.

## Домашнее задание

Если не успели на уроке (что вряд ли) привести схему БД `CorrespondenceEF` к тому, что мы проектировали на 27 уроке, то заданием будет доделать эту работу до максимально близкого состояния.

С помощью `Data Annotation` атрибутов и/или `Fluent API` добиться максимального соответствия схемы `CorrespondenceEF`, создаваемой `EF Core`, оригинальной схеме БД `Correspondence`, разработанной на уроке (скрипты прилагаются).

SQL-файл с желаемой схемой — **Lesson\_34\_ForHomeWork.zip** — приложить к домашке (чтобы было на что ровняться).