

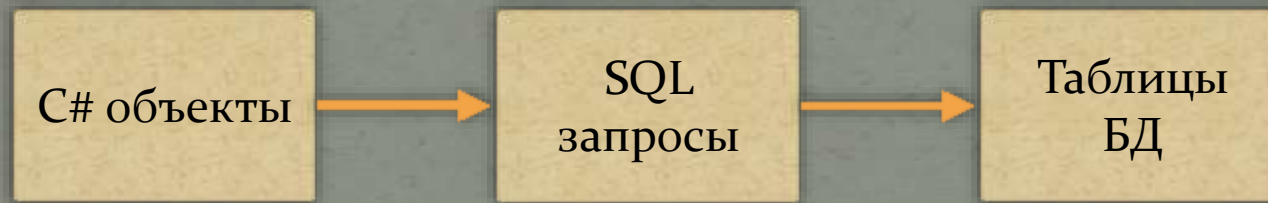
Разработка .NET приложений

Лекция 17

Entity Framework Core

Что такое Entity Framework Core

- ORM (Object-Relational Mapper) для .NET.
 - Позволяет работать с БД через объекты С#.
 - Позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными как с объектами независимо от типа хранилища



- Поддерживает множество СУБД (SQL Server, PostgreSQL, SQLite, MySQL, Azure Cosmos DB, MongoDB, Oracle и др.)
- Кроссплатформенный (Windows, Linux, macOS).
- Open Source - <https://github.com/dotnet/efcore>
- Избавляет разработчика от написания SQL-запросов вручную.

EF Core vs EF6

Критерий	EF Core	EF6
Платформы	.NET Core / .NET	.NET Framework
Поддержка СУБД	Широкая (через провайдеры)	В основном SQL Server
Производительность	Выше	Ниже
LINQ	Полная поддержка	Ограниченная
Миграции	Улучшенные	Базовые

Современная, лёгкая и расширяемая версия.
Последняя версия EF Core 10
Новая версия выходит с новой версией .NET

Установка и настройка EF Core

- NuGet пакеты:
 - Microsoft.EntityFrameworkCore (основной пакет)
 - Microsoft.EntityFrameworkCore.SqlServer (для подключения к SQL Server)
 - Npgsql.EntityFrameworkCore.PostgreSQL (для подключения к PostgreSQL)
 - Microsoft.EntityFrameworkCore.Sqlite (для подключения к Sqlite)
 - Microsoft.EntityFrameworkCore.Cosmos (для подключения к Azure Cosmos)
 - MySql.EntityFrameworkCore или Pomelo.EntityFrameworkCore.MySql (для подключения к MySql)
 - Microsoft.EntityFrameworkCore.InMemory (функциональность провайдера базы данных в памяти)
 - Microsoft.EntityFrameworkCore.Tools (для миграций и генерации классов по готовой БД)
- Остальные пакеты подтянутся по зависимостям

2 подхода при разработке

- Code First
 - Разработчик начинает описывать классы в C#
 - Описывает наследник от DbContext, настройки особенности структуры DB
 - База генерируется автоматически по описанным классам и DbContext
- DB First
 - Есть готовая БД
 - По таблицам БД генерируются классы на C# и класс наследник от DbContext

Code First

Code First

1. Устанавливаем нужный nuget пакет. Например для MS SQL Server
 - `dotnet add package Microsoft.EntityFrameworkCore.SqlServer`
2. Описываем сущности в C# (Entity)

```
class Product
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string? Description { get; set; }
    public double Weight { get; set; }
    public decimal Cost { get; set; }
}
```

Code First

3. Создаем класс наследник от DbContext
 - Настраиваем модель БД
 - Через свойства DbSet<TEntity> настраиваем таблицы в БД
 - Настраиваем строку подключения к БД

```
class ShopContext : DbContext
{
    public DbSet<Product> Products { get; set; } // Таблица в БД

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            @"Data Source=(LocalDB)\MSSQLLocalDB;Database=Shop;Integrated Security=True;");
    }
}
```

Чтение данных из БД

4. Для использования:
 - Создаем контекст
 - Обращаемся к коллекциям

```
using (ShopContext context = new ShopContext())
{
    var products = context.Products.ToList();
    foreach (var item in products)
    {
        Console.WriteLine(item.Title);
    }
}
```

- Не нужно самостоятельно писать запросы к БД

Демонстрация

Code First

Использование LINQ

```
using (ShopContext context = new ShopContext())
{
    var bears = await context.Products
        .Where(p => p.Title == "Bear")
        .OrderBy(p => p.Cost)
        .Take(3)
        .Select(p => new { p.Id, p.Title, p.Description, p.Cost })
        .ToListAsync();

    foreach (var bear in bears)
    {
        Console.WriteLine(bear);
    }
}
```

EF Core автоматически напишет запросы в БД
DECLARE @p int = 3;

```
SELECT TOP(@p) [p].[Id], [p].[Title], [p].[Description], [p].[Cost]
FROM [Products] AS [p]
WHERE [p].[Title] = N'Bear'
ORDER BY [p].[Cost]
```

Добавление данных

- Создаем сущности
- Добавляем в коллекции в DbContext с помощью методов
 - Add(), AddRange()
 - (нет смысла использовать асинхронные версии AddAsync(), AddRangeAsync(), поскольку нет обращения к БД при их вызове)
 - Реальное сохранение в БД произойдет только при вызове SaveChanges()
- Вызываем SaveChanges() или SaveChangesAsync()

```
using (ShopContext context = new ShopContext())
{
    var bear = new Product { Title = "Bear", Description = "Cool Bear", Cost = 10 };
    context.Products.Add(p); // Добавление как самой сущности, так и ее зависимостей
    await context.SaveChangesAsync();
}
```

- Особенность: DbContext не поддерживает одновременное выполнение несколько асинхронных операций

Изменение данных

- Получаем сущности
- Изменяем их
- Вызываем `SaveChanges()` или `SaveChangesAsync()`

```
using (ShopContext context = new ShopContext())
{
    var bear = await context.Products.FirstAsync(p => p.Id == 1);
    bear.Title = "Big Bear";
    bear.Cost *= 1.1;
    await context.SaveChangesAsync();
}
```

Изменение данных

- Если сущность была получена из другого контекста, то необходимо подключить ее к текущему контексту вызвав метод `Update()`

```
Product bear;  
// Получили в одном контексте  
using (ShopContext context = new ShopContext())  
{  
    bear = await context.Products.FirstAsync(p => p.Id == 1);  
} // контекст закрыт, трекинга изменений сущности больше нет  
  
bear.Title = "Big Bear";  
bear.Cost *= 1.1;  
  
// Меняем в другом контексте  
using (ShopContext context = new ShopContext())  
{  
    context.Products.Update(bear); // или context.Update(bear);  
    await context.SaveChangesAsync();  
}
```

Удаление данных

- Получаем сущности
- Вызываем на коллекции Remove()
- Вызываем SaveChanges() или SaveChangesAsync()

```
using (ShopContext context = new ShopContext())  
{  
    var bear = await context.Products.FirstAsync(p => p.Id == 1);  
    context.Products.Remove(bear);  
    await context.SaveChangesAsync();  
}
```

- Все изменения в БД применяются только при вызове SaveChanges()

Демонстрация

Манипуляции с данными

Настройка строки подключения

- Строки подключения к БД выносятся в файл конфигурации (json или xml)

- Добавляем файл конфигурации (например, appsettings.json)

```
{  
  "ConnectionStrings": {  
    "ShopConnection": "Data Source=(LocalDB)\\MSSQLLocalDB;Database=Shop;Integrated Security=True;"  
  }  
}
```

- Устанавливаем Copy to Output (любой вариант) в свойствах файла в проекте
- Подключаем nuget пакет Microsoft.Extensions.Configuration.Json
- В коде получаем строку подключения из конфигурационного файла

```
IConfiguration config = new ConfigurationBuilder()  
    .SetBasePath(Directory.GetCurrentDirectory())  
    .AddJsonFile("appsettings.json")  
    .Build();  
  
string shopConnectionString = config.GetConnectionString("ShopConnection");
```

Используем строку подключения

- Полученную строку подключения можно использовать при настройке DbContext
- Например так:

```
class ShopContext : DbContext
{
    private readonly string connectionString;

    public ShopContext(string connectionString)
    {
        this.connectionString = connectionString;
    }

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

Демонстрация

Connection Strings в файле конфигурации

Создание базы данных

- Метод `Database.EnsureCreated()` проверяет, есть ли база данных, и если ее нет создает ее вместе со всеми необходимыми объектами (таблицами, индексами и т.д.)

```
using (ShopContext dc = new ShopContext(shopConnectionString))
{
    dc.Database.EnsureCreated();
}
```

- Не рекомендуется использовать в продакшене, поскольку несовместима с миграциями
- Можно также пересоздавать базу данных с нуля

```
using (ShopContext dc = new ShopContext(shopConnectionString))
{
    await dc.Database.EnsureDeletedAsync(); // Удаляет БД
    await dc.Database.EnsureCreatedAsync(); // Создает БД заново
}
```

Настройка модели данных

- 3 способа настройки
 - Принятые соглашения
 - С помощью атрибутов у сущностей
 - Настройка модели в `OnModelCreating()` у наследника `DbContext`

Сущности в модели ↔ Таблицы в БД

- Все типы сущностей DbSet и DbSet<TEntity> в DbContext включаются в модель данных и сопоставляются с таблицами в базе данных
 - Свойство DbSet ↔ Таблица в БД
 - DbSet<Product> Products { get; set; } - в базе будет таблица dbo.Products

- Правило именования таблиц можно изменить с помощью атрибута на сущности (атрибут из пространства имен System.ComponentModel.DataAnnotations.Schema)

```
[Table("CustomerMaster")] // или проигнорировать сущность [NotMapped]
public class Customer {...}
```

- Можно добавить или проигнорировать сущность и в методе OnModelCreating()

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<City>();
        modelBuilder.Ignore<Country>();
    }
}
```

Сущности в модели \leftrightarrow Таблицы в БД

- В БД также добавляются все типы на которые ссылается сущность, даже если они не добавлены явно как DbSet свойство в DbContext

```
class ShopContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
class Manufacturer
{
    public int Id { get; set; }
    public string Name { get; set; }
}
class Product
{
    public int Id { get; set; }
    public Manufacturer Manufacturer { get; set; }
}
```

В БД будут 2 таблицы:

- **dbo.Products**
- **dbo.Manufacturer**

Свойства сущности ↔ столбцы в БД

- Соглашения:
 - в модель включает все публичные свойства сущности, доступные для записи и чтения
 - Свойства Id или ИмяСущностиId (например, а примере ProductId) будут считаться первичным ключом в БД
- Свойства могут быть проигнорированы для БД с помощью атрибута **[NotMapped]** или в методе OnModelCreating() в DbContext

```
class Product
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    [NotMapped]
    public decimal Cost { get; set; }
}
```

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().Ignore(p=>p. Description);
    }
}
```

Поля сущности \leftrightarrow столбцы в БД

- EF Core умеет вызывать параметризованный конструктор при создании сущности и задавать через него ряд свойств. Свойства, отсутствующие в конструкторе, будут задаваться напрямую через свойства. Имена параметров должны совпадать с именем свойства (отличия в case допускаются).
- Название столбцов можно настроить с помощью атрибута `[Column("Имя Столбца В БД")]`
`[Column("product_id")]`
`public int Id { get; set; }`
- Можно использовать поля для маппинга в столбцы БД

```
class ShopContext : DbContext {  
    protected override void OnModelCreating(ModelBuilder modelBuilder) {  
        modelBuilder.Entity< Product >().Property("Id").HasColumnName("product_id");  
    }  
}
```

```
class Product  
{  
    private int id;  
    public int Id => id;  
    private string title;  
    public Product(string title)  
    {  
        this.title = title;  
    }  
}
```

```
class ShopContext : DbContext  
{  
    protected override void OnModelCreating(ModelBuilder modelBuilder)  
    {  
        modelBuilder.Entity< Product >().Property("Id").HasField("id");  
        modelBuilder.Entity< Product >().Property("title");  
    }  
}
```

Обязательные свойства

- По умолчанию
 - Не обязательные свойства
 - свойство, допускающее значение null -> NULL столбец
 - Обязательные свойства
 - свойство, не допускающее значение null -> NOT NULL столбец
- С помощью атрибута **[Required]** можно пометить обязательные свойства

```
public class Product
{
    public int Id { get; set; }
    [Required]
    public string? Title { get; set; }
}
```

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().Property(p => p. Title).IsRequired(); }
}
```

Настройка ключей

- По умолчанию
 - В качестве первичного ключа используется свойство, которое называется Id или [имя_класса]Id.
 - В БД: CONSTRAINT "PK_Products" PRIMARY KEY("Id")
- С помощью атрибута **[Key]** можно пометить свойство, которое будет являться первичным ключом

```
public class Product
{
    [Key]
    public int Number { get; set; }
    public string? Title { get; set; }
    public string Code { get; set; }
}
```

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().HasKey(p => p. Number);
        // modelBuilder.Entity<Product>().HasKey(p => new { p. Number, p.Title});
        modelBuilder.Entity<Product>().HasAlternateKey(p => p. Code);
    }
}
```

- CONSTRAINT "PK_Products" PRIMARY KEY("Number")
- -- CONSTRAINT "PK_Products" PRIMARY KEY("Number", "Title")
- CONSTRAINT "AK_Products_Code" UNIQUE("Code"),

Настройка индексов

- По умолчанию
 - индекс создается для каждого свойства, которое используется в качестве внешнего ключа
- С помощью атрибута **[Index]** можно указать свойства, по которым будет строиться индекс

```
public class Product
```

```
{
```

```
    [Index("Code", IsUnique = true)]
```

```
    [Index("Code", "Title")]
```

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

```
    public string? Title { get; set; }
```

```
    public string Code { get; set; }
```

```
}
```

```
class ShopContext : DbContext
```

```
{
```

```
    protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
    {
```

```
        modelBuilder.Entity<Product>().HasIndex(p => p. Code)
```

```
            .IsUnique();
```

```
        modelBuilder.Entity<Product>().HasIndex(p => new { p. Code, p.Title});
```

```
        modelBuilder.Entity<Product>().HasIndex(p => p. Title)
```

```
            .HasFilter("[Title] IS NOT NULL");
```

```
    }
```

```
}
```

Автогенерация значений

- Соглашение:
 - Если при добавлении / обновлении объекта у него уже установлено значение для свойства, то это значение используется при вставке или обновлении в таблицу.
 - Если для свойства явным образом не установлено значение, то для свойства устанавливается значение по умолчанию (null / 0, Guid.Empty и т.д.).
 - Генерация значений может быть на стороне клиента или на стороне базы в зависимости от провайдера БД
 - Если значение генерируется БД, то при добавлении объекта назначается временное значение, которое заменяется значением, сгенерированным БД при вызове метода SaveChanges().
 - Для всех первичных ключей типов int и GUID автоматически генерируется значение при вставке в БД. Задавать их значение в коде при добавлении нового объекта нельзя.
- С помощью атрибута **[DatabaseGenerated]** можно изменить поведение автогенерации

```
public class Product
{
    [DatabaseGenerated(
        DatabaseGeneratedOption.None)]
    public int Id { get; set; }
}
```

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().Property(p => p. Id)
            .ValueGeneratedNever();
    }
}
```

Значения по умолчанию

- Для свойств, где
 - дефолтное значение C# не устраивает (null / 0 / ...)
 - нет автоинкрементов
- **HasDefaultValue** – устанавливает дефолтное значение для вставки в БД
- **HasDefaultValueSql** – устанавливает дефолтное значение для вставки в БД, генерируемого БД
- **HasComputedColumnSql** – задает вычисляемые столбцы

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().Property(p => p.Discount)
            .HasDefaultValue(10);
        modelBuilder.Entity<Order>().Property(o => o.Date)
            .HasDefaultValueSql("DATETIME('now')");
        modelBuilder.Entity<Order>().Property(o => o.Address)
            .HasComputedColumnSql("Country || ' - ' || City");
    }
}
```

Constraints

- **HasCheckConstraint** – устанавливает CHECK CONSTRAINT в БД
- **HasMaxLength** – устанавливает максимальную длину значения (string, byte[])
 - Можно задать также с помощью атрибута на свойстве [MaxLength(50)]

```
class ShopContext : DbContext
{
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Product>().ToTable(
            t => t.HasCheckConstraint("ValidCost", "Cost > 0");
            // CONSTRAINT "ValidCost" CHECK(" Cost " > 0)

        modelBuilder.Entity<Order>().Property(o => o.Title)
            .HasMaxLength(50);
    }
}
```

Отношения между сущностями

- Навигационные свойства (по соглашению):

```
public class Product
{
    public int Id { get; set; }
    public Manufacturer Manufacturer { get; set; }
}
```

- Будут сгенерирован столбец <Имя сущности><Ключ сущности>
 - В текущем примере **ManufacturerId**
- Будет создан внешний ключ

```
CREATE TABLE [Products] (
    [Id] int NOT NULL IDENTITY,
    [ManufacturerId] int NOT NULL,
    CONSTRAINT [PK_Products] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Products_Manufacturers_ManufacturerId]
        FOREIGN KEY ([ManufacturerId]) REFERENCES [Manufacturers] ([Id])
        ON DELETE CASCADE);
);
```

- Достаточно определить в одной из сущностей

Отношения между сущностями

- Внешние ключи (по соглашению):

```
public class Product
{
    public int Id { get; set; }
    public int ManufacturerId { get; set; }
    public Manufacturer Manufacturer { get; set; }
}
```

- Аналогично будет сгенерирован столбец <Имя сущности><Ключ сущности>
 - В текущем примере ManufacturerId
- Будет создан внешний ключ

```
CREATE TABLE [Products] (
    [Id] int NOT NULL IDENTITY,
    [ManufacturerId] int NOT NULL,
    CONSTRAINT [PK_Products] PRIMARY KEY ([Id]),
    CONSTRAINT [FK_Products_Manufacturers_ManufacturerId]
        FOREIGN KEY ([ManufacturerId]) REFERENCES [Manufacturers] ([Id])
    ON DELETE CASCADE);
```

);

Отношения между сущностями

- Можно также настраивать внешние ключи через:

- Атрибут [ForeignKey]:

```
public class Product
{
    public int Id { get; set; }
    public int ManufacturerKey { get; set; }
    [ForeignKey(" ManufacturerKey ")] // название столбца с внешним ключом
    public Manufacturer Manufacturer { get; set; }
}
```

- OnModelCreating() в DbContext:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>()
        .HasOne(p => p. Manufacturer)
        .WithMany(m => m. Manufacturers)
        .HasForeignKey(p => p.ManufacturerKey);
}
```

Отношение многие ко многим

- Указываем коллекции сущностей в обеих сущностях

```
public class Product
{
    public int Id { get; set; }
    public List<Manufacturer> Manufacturers { get; set; }
}
public class Manufacturer
{
    public int Id { get; set; }
    public List<Product> Products { get; set; } = new List<Product>();
}
```

- Развязочная таблица будет создана автоматически

```
CREATE TABLE [ManufacturerProduct] (
    [ManufacturersId] int NOT NULL,
    [ProductsId] int NOT NULL,
    CONSTRAINT [PK_ManufacturerProduct] PRIMARY KEY ([ManufacturersId], [ProductsId]),
    CONSTRAINT [FK_ManufacturerProduct_Manufacturers_ManufacturersId]
        FOREIGN KEY ([ManufacturersId]) REFERENCES [Manufacturers] ([Id]) ON DELETE CASCADE,
    CONSTRAINT [FK_ManufacturerProduct_Products_ProductsId]
        FOREIGN KEY ([ProductsId]) REFERENCES [Products] ([Id]) ON DELETE CASCADE );
```

Отношение многие ко многим

- Можно также настроить в `OnModelCreating()` в `DbContext`
- Это пригодится, если нужно задать какое-то особенное имя для развязочной таблицы

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Product>()
        .HasMany(p => p.Manufacturers)
        .WithMany(m => m.Products)
        .UsingEntity(j => j.ToTable("ProductCreators"));
}
```

- Есть возможность добавлять дополнительные столбы в развязочную таблицу

Демонстрация

Отношения между сущностями (FK)

Загрузка связанных данных

- Загрузка связанных данных через навигационные свойства
 - Eager loading (жадная загрузка)
 - Через методы `Include()` и `ThenInclude()`

```
var products = await context.Products.Include(p => p.Manufacturer).ToListAsync();
foreach (var product in products)
{
    Console.WriteLine($"{product.Title} - {product.Manufacturer.Name}");
}
```
 - Подгрузка данных в контекст для дальнейшего использования
 - Приводит к LEFT JOIN запросу в БД
 - Explicit loading (явная загрузка)
 - Явная загрузка данных с помощью метода `Load()`. Фактически нужно для кэширование данных в контексте
 - `db.Products.Where(p => p.ManufacturerId == 10).Load();`
 - `foreach (var item in products.Manufacturers)`
 - Lazy loading (ленивая загрузка)
 - Автоматическая подгрузка связанных сущностей

Наследование

- EF Core поддерживает 3 варианта наследования
 - Table Per Hierarchy (TPH)
 - Одна таблица на иерархию классов
 - Создается дополнительный столбец – дискриминатор. Будет содержать название класса данных в строке таблицы
 - Используется по умолчанию
 - Table Per Type (TPT)
 - Отдельная таблица для каждого класса из иерархии (с внешним ключем на таблицу с базовым классом)
 - Каждая таблица содержит только данные своего наследника и внешний ключ на таблицу с базовым классом
 - Table Per Class (TPC)
 - Таблица на каждый отдельный тип
 - Столбцы в каждой таблице создаются по всем свойствам, в том числе и унаследованным
 - Нужен способ генерации уникального Id по всем таблицам в иерархии (например, использование Guid)
- Стратегию можно задать в OnModelCreating() в DbContext
 - `modelBuilder.Entity<Product>().UseTpcMappingStrategy();` // или другие варианты стратегии

Настройка логирования запросов

- При настройке конфигурации можно настроить и логирование запросов EF Core

```
class ShopContext : DbContext
{
    ...
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
        optionsBuilder.LogTo(Console.WriteLine, LogLevel.Information);
    }
}
```

- Можно детально настроить логирование, например; уровень логирования, категории логируемых событий и др.
- Отлично интегрируется с `Microsoft.Extensions.Logging`

Демонстрация

Логирование запросов

Миграции

- EF Core поддерживает поэтапное обновление/изменение ДБ с помощью механизма миграций
- Устанавливаем nuget пакеты
 - Microsoft.EntityFrameworkCore.Tools
 - Microsoft.EntityFrameworkCore.Design
- В Package Manager Console
 - Создание миграции
 - Add-Migration название_миграции
 - Add-Migration InitialCreate
 - Применение миграции
 - Update-Database
- Применение миграции при старте приложения (несовместима с EnsureCreated)

```
using (ShopContext context = new ShopContext())  
{  
    await context.Database.MigrateAsync();  
}
```

Демонстрация

Миграции

DB First



DB First

- Используется, когда база данных уже существует
- Генерирует необходимые сущности и DbContext с настройками для подключения к указанной базе
- Устанавливаем nuget пакеты:
 - Microsoft.EntityFrameworkCore.SqlServer (или нужный провайдер)
 - Microsoft.EntityFrameworkCore.Tools
 - Microsoft.EntityFrameworkCore.Design
- Вызываем генерацию классов по БД
 - Либо через Package Manager Console в Visual Studio
 - **Scaffold-DbContext** "строка подключения" провайдер_бд
 - **Scaffold-DbContext** "Data Source=(LocalDB)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=True;" Microsoft.EntityFrameworkCore.SqlServer -outputdir Entities
 - Либо через .NET CLI
 - Сначала устанавливаем инструменты для работы с EF Core
 - dotnet tool install --global dotnet-ef
 - Запускаем генерацию
 - **dotnet ef dbcontext scaffold** "строка подключения" провайдер_бд
 - **dotnet ef dbcontext scaffold** "Data Source=(LocalDB)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=True;" Microsoft.EntityFrameworkCore.SqlServer --output-dir Entities

Демонстрация

DB First