

# **Работа с данными в Entity Framework Core**

**Проектирование и разработка веб-сервисов**

# Вводные понятия

- Анемичные и толстые модели
- Модели представлений
- Привязка модели
- Состояние модели
- Источники привязки модели

# Доп. материал по Entity Framework

1. <https://metanit.com/sharp/entityframeworkcore/>
2. <https://metanit.com/sharp/aspnet5/12.1.php>

# Понятие Entity Framework Core

Entity Framework представляет прекрасное ORM-решение, которое позволяет автоматически связать обычные классы языка C# с таблицами в базе данных. Entity Framework Core нацелен в первую очередь на работу с СУБД MS SQL Server, однако поддерживает также и ряд других СУБД. В данном случае мы будем работать с базами данных в MS SQL Server.

По умолчанию в проекте библиотеки Entity Framework отсутствуют, и их надо добавить. Это можно сделать разными способами:

- через Nuget,
- через Package Manager Console
- вписав нужную зависимость в project.json.

# Понятие Entity Framework Core

Для взаимодействия с MS SQL Server через Entity Framework необходим пакет **Microsoft.EntityFrameworkCore.SqlServer**.

Перед работой с базой данных нам предварительно надо создать эту базу данных в соответствии с вышеопределенными моделями (Add-Migration, Update-Database). И для этого потребуется пакет **Microsoft.EntityFrameworkCore.Tools**.

В проекте для версии ASP.NET Core 2.0 и выше эти пакеты уже присутствуют по умолчанию.

# 1. Создание сущностей

Далее добавим в проект новую папку, которую назовем Models. И в этой папке определим новый класс Phone:

```
public class Phone
{
    public int Id { get; set; }
    public string Name { get; set; } // название смартфона
    public string Company { get; set; } // компания
    public int Price { get; set; } // цена
}
```

## 2. Добавление контекста

Чтобы взаимодействовать с базой данных через Entity Framework нам нужен контекст данных - класс, унаследованный от класса **Microsoft.EntityFrameworkCore.DbContext**. Поэтому добавим в папку Models новый класс, который назовем **MobileContext**:

```
using Microsoft.EntityFrameworkCore;

namespace EFDataApp.Models
{
    public class MobileContext : DbContext
    {
        public DbSet<Phone> Phones { get; set; }
        public MobileContext(DbContextOptions<MobileContext>
options)
            : base(options)
        {
        }
    }
}
```

## 2. Добавление контекста

Свойство **DbSet** представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. При этом по умолчанию название свойства должно соответствовать множественному числу названию модели в соответствии с правилами английского языка. То есть `phone` - название класса модели представляет единственное число, а `phones` - множественное число.

Через параметр `options` в конструктор контекста данных будут передаваться настройки контекста.



### 3. Настройка подключения к БД

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Для этого изменим файл appsettings.json, добавив в него определение строки подключения:

```
{  
  "ConnectionStrings": {  
    "DefaultConnection":  
    "Server=(Localdb)\\mssqlLocaldb;Database=mobiledsdb;Trusted_Conn  
    ection=True;"  
  },  
  // остальное содержимое файла  
}
```

### 3. Настройка подключения к БД

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Для этого изменим файл appsettings.json, добавив в него определение строки подключения:

```
{  
  "ConnectionStrings": {  
    "DefaultConnection":  
    "Server=(Localdb)\\mssqlLocaldb;Database=mobiledsdb;Trusted_Conn  
    ection=True;"  
  },  
  // остальное содержимое файла  
}
```

## 4. Настройка сервиса

Последним шагом в настройке проекта является изменение файла Startup.cs. В нем нам надо изменить метод ConfigureServices():

```
public void ConfigureServices(IServiceCollection services)
{
    // получаем строку подключения из файла конфигурации
    string connection =
Configuration.GetConnectionString("DefaultConnection");
    // добавляем контекст MobileContext в качестве сервиса
    services.AddDbContext<MobileContext>(options =>
        options.UseSqlServer(connection));

    services.AddMvc();
}
```

## **4. Настройка сервиса**

Добавление контекста данных в виде сервиса позволит затем получать его в конструкторе контроллера через механизм внедрения зависимостей.

# Обновление базы данных

После определения всех настроек используем миграции для создания базы данных. Для этого построим проект и откроем окно Package Manager Console. Его можно найти в меню Tools -> Nuget Package Manager -> Package Manager Console.

Последовательно введем в это окно две команды. Сначала выполним команду:

***Add-Migration Initial***

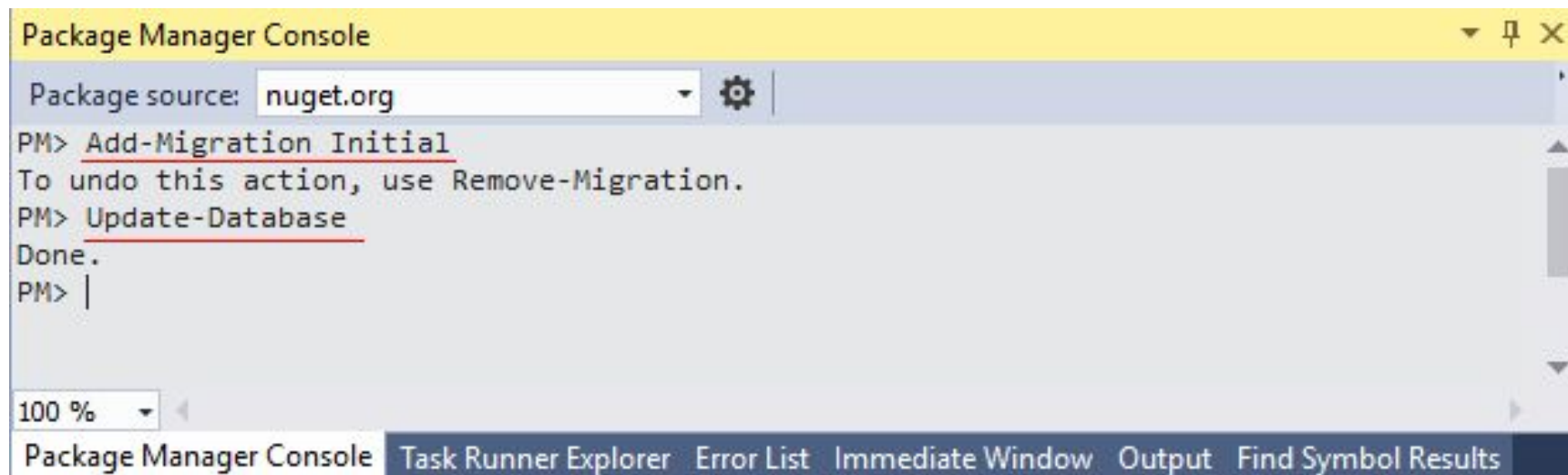
Эта команда добавит в проект новую миграцию.

Затем выполним команду:

***Update-Database***

Эта команда по миграции Initial собственно сгенерирует базу данных.

# Обновление базы данных



The screenshot shows the Package Manager Console window in Visual Studio. The title bar is yellow and says "Package Manager Console". Below the title bar is a light blue header area with a dropdown menu set to "nuget.org" and a gear icon. The main area is a light gray text box containing the following text:

```
PM> Add-Migration Initial  
To undo this action, use Remove-Migration.  
PM> Update-Database  
Done.  
PM> |
```

At the bottom of the console, there is a zoom level dropdown set to "100 %". Below the console is a dark blue tab bar with the following tabs: "Package Manager Console", "Task Runner Explorer", "Error List", "Immediate Window", "Output", and "Find Symbol Results".

# Операции с моделями

Вначале изменим имеющийся по умолчанию контроллер HomeController:

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using EFDataApp.Models; // пространство имен моделей

namespace EFDataApp.Controllers
{
    public class HomeController : Controller
    {
        private MobileContext db;
        public HomeController(MobileContext context)
        {
            db = context;
        }
    }
}
```

# Создание и вывод

Методы чтения и вставки данных:

```
public IActionResult Index()
{
    return View(db.Phones.ToList());
}
public IActionResult Create()
{
    return View();
}
[HttpPost]
public IActionResult Create(PHONE phone)
{
    db.Phones.Add(phone);
    db.SaveChanges();

    return RedirectToAction("Index");
}
```



# Редактирование и удаление

Для редактирования и удаления служат методы Update и Remove, принимающие объект, с которым производится операция.

# Сортировка и фильтрация

Для сортировки применяются методы LINQ – **OrderBy** и **OrderByDescending**

При необходимости упорядочить данные сразу по нескольким критериям можно использовать методы `ThenBy()`(для сортировки по возрастанию) и `ThenByDescending()`. Например, отсортируем по двум значениям:

```
var phones = db.Phones.OrderBy(p => p.Price).ThenBy(p=>p.Company.Name);
```

Для фильтрации применяется метод LINQ – **WHERE**

# Постраничная навигация

Для постраничной навигации используется:

*// page – текущая страницы*

*// pageSize – количество элементов на странице*

*var users = \_context.Users.Skip((page - 1) \* pageSize).Take(pageSize).ToList();*

# Выражения LIKE и IN

Использование выражения **LIKE**:

```
var users = _context. Users.Where(x => x.name.StartsWith("value"));
```

```
var users = _context. Users.Where(x => x.name.Contains("value"));
```

```
var users = _context. Users.Where(x => x.name.EndsWith("value"));
```

Использование **IN**:

```
List<string> values = new List<string> { "a", "b" };
```

```
var users = _context.Users.Where(x => values.Contains(x.name));
```

# Выражение LIKE

Начиная с версии 2.0 в Entity Framework Core можно использовать метод `EF.Functions.Like()`. Он позволяет транслировать условие в выражение с оператором LIKE на стороне MS SQL Server. Метод принимает два параметра - оцениваемое выражение и шаблон, с которым сравнивается его значение. Например, найдем все телефоны, в названии которых есть слово "Galaxy":

```
using (ApplicationContext db = new ApplicationContext())
{
    var phones = db.Phones.Where(p => EF.Functions.Like(p.Name, "%Galaxy%"));
    foreach (Phone phone in phones)
        Console.WriteLine($"{phone.Name} ({phone.Price})");
}
```

# Выражение LIKE

Для определения шаблона могут применяться ряд специальных символов подстановки:

1. %: соответствует любой подстроке, которая может иметь любое количество символов, при этом подстрока может и не содержать ни одного символа
2. \_: соответствует любому одиночному символу
3. [ ]: соответствует одному символу, который указан в квадратных скобках
4. [ - ]: соответствует одному символу из определенного диапазона
5. [ ^ ]: соответствует одному символу, который не указан после символа ^

# Агрегатные операции

## 1. Количество элементов в выборке:

```
int number1 = db.Phones.Count();  
// найдем кол-во моделей, которые в названии содержат Samsung  
int number2 = db.Phones.Count(p => p.Name.Contains("Samsung"));
```

## 2. Минимальное, максимальное и среднее значения

```
int minPrice = db.Phones.Min(p => p.Price);  
// максимальная цена  
int maxPrice = db.Phones.Max(p => p.Price);  
// средняя цена на телефоны фирмы Samsung  
double avgPrice = db.Phones.Average(p => p.Price);
```

## 3. Сумма значений (Sum).

# Загрузка связанных данных

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public int CompanyId { get; set; }
    public Company Company { get; set; }
}

public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<User> Users { get; set; }
    public Company()
    {
        Users = new List<User>();
    }
}
```



# Загрузка связанных данных

Для загрузки связанных данных используется выражение **Include**:

```
IQueryable<User> source = _context.Users.Include(x =>  
x.Company);
```

```
var count = source.Count();  
var items = source.ToList();
```

# Объединение Join

В результате объединения получается новый объект:

```
List<TripleObject> Data = _context.t_data_indicators.Where(x =>
x.CalculationId ==
_calculationId).Join(_context.t_sprav_indicators,
    d => d.indicator_id, // t_data_indicators
    s => s.indicator_id, // t_sprav_indicators
    (d, s) => new TripleObject { param1 =
d.pech_id, param2 = s.symbol, param3 = d.value }
    ).ToList();
```