# Entity Framework Core

# Entity Framework

**Entity Framework** представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища.

# Entity Framework

Центральной концепцией *Entity Framework* является понятие **сущности** или **entity**. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами.

# Entity Framework

У каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют **ключами**.

# Entity Framework

Сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

# Entity Framework

Другим ключевым понятием является **Entity Data Model**. Эта модель сопоставляет классы сущностей с реальными таблицами в БД.

# Entity Framework

Entity Data Model состоит из трех уровней:

- **Концептуальный уровень** - определение классов сущностей, используемых в приложении.
- **Уровень хранилища** - определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.
- **Уровень сопоставления (маппинга)** - служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц.

# Способы взаимодействия с БД

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

- **Database first:** Entity Framework создает набор классов, которые отражают модель конкретной базы данных.
- **Model first:** сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
- **Code first:** разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы.

# Пример использования EF Core

# Пример использования EF Core

Для работы с  БД нам необходимо вначале добавить в проект пакет EntityFramework Core. В данном случае мы будем использовать MS SQL Server, поэтому нам надо искать пакет: Microsoft.EntityFrameworkCore.SqlServer (представляет функциональность Entity Framework для работы с MS SQL Server)

# Пример использования EF Core

Далее нам надо определить модель, которая будет описывать данные. Пусть наше приложение будет посвящено работе с пользователями. Поэтому добавим в проект новый класс User:

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

# Пример использования EF Core

Надо отметить, что Entity Framework требует определения ключа элемента для создания первичного ключа в таблице в бд. По умолчанию при генерации бд EF в качестве первичных ключей будет рассматривать свойства с именами Id или [Имя_класса]Id (то есть UserId).

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

# Пример использования EF Core

Взаимодействие с базой данных в Entity Framework Core происходит посредством специального класса - контекста данных. Поэтому добавим в наш проект новый класс, который назовем ApplicationContext .

# Пример использования EF Core

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

Основу функциональности *Entity Framework Core* для работы с *MS SQL Server* составляют классы, которые располагаются в пространстве имен **Microsoft.EntityFrameworkCore.**

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

**DbContext:** определяет контекст данных, используемый для взаимодействия с базой данных:

```
using Microsoft.EntityFrameworkCore;


public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

**DbSet/DbSet<TEntity>:** представляет набор объектов, которые хранятся в базе данных:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

DbContextOptionsBuilder: устанавливает параметры подключения:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

DbContextOptionsBuilder: устанавливает параметры строки подключения:

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users => Set<User>();
    public ApplicationContext() => Database.EnsureCreated();

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Пример использования EF Core

Строка подключения разбивается на несколько частей:

- **Server:** название сервера. В данном случае используется специальный движок MS SQL Server - localdb, который предназначен специально для нужд разработки;
- **Database:** название базы данных;
- **Trusted_Connection:** устанавливает проверку подлинности;

# Пример использования EF Core

```
using (ApplicationContext db = new ApplicationContext())
{
    // создаем два объекта User
    User tom = new User { Name = "Tom", Age = 33 };
    User alice = new User { Name = "Alice", Age = 26 };

    // добавляем их в бд
    db.Users.Add(tom);
    db.Users.Add(alice);
    db.SaveChanges();
    Console.WriteLine("Объекты успешно сохранены");

    // получаем объекты из бд и выводим на консоль
    var users = db.Users.ToList();
    Console.WriteLine("Список объектов:");
    foreach (User u in users)
    {
        Console.WriteLine($"{u.Id}.{u.Name} - {u.Age}");
    }
}
```

# Пример использования EF Core

В результате после запуска программа выведет на консоль:

```
Объекты успешно сохранены
Список объектов:
1.Tom - 33
2.Alice - 26
```

# Управление схемой БД и миграции

# Управление схемой БД и миграции

Если мы меняем модели в *Entity Framework*, которые входят в контекст данных, например, добавляем в нее какие-то новые свойства или удаляем некоторые свойства, то необходимо, чтобы база данных также применяла эти изменения.

# Управление схемой БД и миграции

Например, мы хотим добавить в класс User новое свойство:

```
public class User
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public int Age { get; set; }
    public string? Position { get; set; }   // Новое свойство - должность пользователя
}
```

# Управление схемой БД и миграции

Если нам не важны данные в БД и мы хотим ее просто пересоздать для соответствия новой структуре классов, то через контекст данных можно вызывать метод **Database.EnsureDeleted** для удаления и затем метод **Database.EnsureCreated** для создания бд.

# Управление схемой БД и миграции

```
using Microsoft.EntityFrameworkCore;

public class ApplicationContext : DbContext
{
    public DbSet<User> Users { get; set; };
    public ApplicationContext()
    {
        Database.EnsureDeleted();   // удаляем бд со старой схемой
        Database.EnsureCreated();   // создаем бд с новой схемой
    }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
                                      Database=helloappdb;
                                      Trusted_Connection=True;")
    }
}
```

# Управление схемой БД и миграции

В то же время при удалении происходит полное удаление данных, что в ряде случаев может быть нежелательным. И в этом случае лучше использовать миграции.

# Миграция

Миграция по сути представляет план перехода базы данных от старой схемы к новой.

# Миграция

Для создания миграции в используется команда:

```
Add-Migration название_миграции
```

Название миграции представляет произвольное название, главное чтобы все миграции в проекте имели разные названия.

# Миграция

После создания миграции ее надо выполнить с помощью команды:

```
Update-Database
```

# Основные операции с данными. CRUD

# Основные операции с данными. CRUD

Большинство операций с данными так или иначе представляют собой CRUD операции (Create, Read, Update, Delete), то есть создание, получение, обновление и удаление. Entity Framework Core позволяет легко выполнять все эти действия.

# Добавление

Для добавления объекта используется метод Add, определенный у класса DbSet, в который передается добавляемый объект:

```
using (ApplicationContext db = new ApplicationContext())
{
    // создаем два объекта User
    User tom = new User { Name = "Tom", Age = 33 };
    User alice = new User { Name = "Alice", Age = 26 };

    // добавляем их в бд
    db.Users.Add(tom);
    db.Users.Add(alice);
    db.SaveChanges();
    Console.WriteLine("Объекты успешно сохранены");
}
```

# Добавление

Метод Add устанавливает значение Added в качестве состояния нового объекта. Поэтому метод db.SaveChanges() сгенерирует выражение INSERT для вставки модели в таблицу.

# Добавление

Если нам надо добавить сразу несколько объектов, то мы можем воспользоваться методом AddRange():

```
User tom = new User { Name = "Tom", Age = 33 };
User alice = new User { Name = "Alice", Age = 26 };
db.Users.AddRange(tom, alice);
```

# Удаление

Удаление производится с помощью метода Remove:

```
db.Users.Remove(user);
db.SaveChanges();
```

Данный метод установит статус объекта в Deleted, благодаря чему Entity Framework при выполнении метода db.SaveChanges() сгенерирует SQL-выражение DELETE.

# Удаление

Если необходимо удалить сразу несколько объектов, то можно использовать метод RemoveRange():

```
User? firstUser = db.Users.FirstOrDefault();
User? secondUser = db.Users.FirstOrDefault(u=>u.Id==2);
if (firstUser != null && secondUser != null)
{
    db.Users.RemoveRange(firstUser, secondUser);
    db.SaveChanges();
}
```

# Редактирование

При изменении объекта Entity Framework сам отслеживает все изменения, и когда вызывается метод SaveChanges(), будет сформировано SQL-выражение UPDATE для данного объекта, которое обновит объект в базе данных.

```
// получаем первый объект
User? user = db.Users.FirstOrDefault();
if (user != null)
{
    //обновляем объект
    user.Name = "Bob";
    user.Age = 44;
    db.SaveChanges();
}
```

# Редактирование

При необходимости обновить одновременно несколько объектов, применяется метод UpdateRange():

```
db.Users.UpdateRange(tom, alice);
```

# Асинхронный API

Вместо метода SaveChanges() для асинхронного выполнения зароса к бд можно использовать его асинхронный двойник - SaveChangesAsync(). Также, для добавления данных определены асинхронные методы AddAsync и AddRangeAsync. Пример применения асинхронного API:

```
using (ApplicationContext db = new ApplicationContext())
{
    User tom = new User { Name = "Tom", Age = 33 };
    User alice = new User { Name = "Alice", Age = 26 };

    // Добавление
    await db.Users.AddRangeAsync(tom, alice);
    await db.SaveChangesAsync();
}
```

# Отношения между сущностями

# Внешние ключи и навигационные свойства

Для связей между сущностями в Entity Framework Core применяются внешние ключи и навигационные свойства.

# Внешние ключи и навигационные свойства

```
public class Company
{
    public int Id { get; set; }
    public string? Name { get; set; } // название компании
}

public class User
{
    public int Id { get; set; }
    public string? Name { get; set; }

    public int CompanyId { get; set; }       // внешний ключ
    public Company? Company { get; set; }    // навигационное свойство
}
```

# Внешние ключи и навигационные свойства

Свойство CompanyId в классе User является внешним ключом, а свойство Company - навигационным свойством. По умолчанию название внешнего ключа должно принимать одно из следующих вариантов имени:

- *Имя_навигационного_свойства+Имя ключа из связанной сущности*
- *Имя_класса_связанной_сущности+Имя ключа из связанной сущности*

# Внешние ключи и навигационные свойства

При использовании классов нам достаточно установить либо одно навигационное свойство, либо свойство-внешний ключ.

# Внешние ключи и навигационные свойства

Установка главной сущности по навигационному свойству:

```
using (ApplicationContext db = new ApplicationContext())
{
    Company company1 = new Company { Name = "Google" };
    Company company2 = new Company { Name = "Microsoft" };
    User user1 = new User { Name = "Tom", Company = company1 };
    User user2 = new User { Name = "Bob", Company = company2 };
    User user3 = new User { Name = "Sam", Company = company2 };

    db.Companies.AddRange(company1, company2);  // добавление компаний
    db.Users.AddRange(user1, user2, user3);     // добавление пользователей
    db.SaveChanges();

    foreach (var user in db.Users.ToList())
    {
        Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");
    }
}
```

# Внешние ключи и навигационные свойства

Консольный вывод программы:

```
Tom работает в Google
Bob работает в Microsoft
Sam работает в Microsoft
```

# Внешние ключи и навигационные свойства

Установка главной сущности по свойству-внешнему ключу :

```
using (ApplicationContext db = new ApplicationContext())
{
    Company company1 = new Company { Name = "Google" };
    Company company2 = new Company { Name = "Microsoft" };
    db.Companies.AddRange(company1, company2);  // добавление компаний
    db.SaveChanges();

    User user1 = new User { Name = "Tom", CompanyId = company1.Id };
    User user2 = new User { Name = "Bob", CompanyId = company1.Id };
    User user3 = new User { Name = "Sam", CompanyId = company2.Id };

    db.Users.AddRange(user1, user2, user3);     // добавление пользователей
    db.SaveChanges();
    foreach (var user in db.Users.ToList())
    {
        Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");
    }
}
```

# Внешние ключи и навигационные свойства

Консольный вывод программы:

```
Tom работает в Google
Bob работает в Google
Sam работает в Microsoft
```

# Внешние ключи и навигационные свойства

Здесь необходимо отметить один момент: для устновки свойства внешнего ключа CompanyId нам необходимо знать его значение. Однако поскольку оно связано со свойством Id класса Company, значение которого генерируется при добавление объекта в БД, соответственно в данном случае необходимо сначала добавить объект Company в базу данных.

# Загрузка связанных данных.

# Загрузка связанных данных.

Через навигационные свойства мы можем загружать связанные данные. И здесь у нас три стратегии загрузки:

- Eager loading (жадная загрузка);
- Explicit loading (явная загрузка);
- Lazy loading (ленивая загрузка);

# Загрузка связанных данных.

Рассмотрим, что представляет собой eager loading или жадная загрузка. Она позволяет загружать связанные данные с помощью метода Include(), в который передается навигационное свойство.

```
var users = db.Users
            .Include(u => u.Company)  // подгружаем данные по компаниям
            .ToList();

foreach (var user in users)
    Console.WriteLine($"{user.Name} - {user.Company?.Name}");
```

# Загрузка связанных данных.

На уровне базы данных это выражение будет транслироваться в следующий SQL-запрос:

```
SELECT "u"."Id", "u"."CompanyId", "u"."Name", "c"."Id", "c"."Name"
FROM "Users" AS "u"
LEFT JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"
```

# Загрузка связанных данных.

То есть на уровне базы данных это будет означать использование выражения LEFT JOIN, который присоединяет данные из другой таблицы.

```
SELECT "u"."Id", "u"."CompanyId", "u"."Name", "c"."Id", "c"."Name"
FROM "Users" AS "u"
LEFT JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"
```

# Загрузка связанных данных.

Стоит отметить, что если данные уже ранее были загружены в контекст данных или просто ранее были в него добавлены, то можно не использовать метод Include для их получения, так как они уже в контексте. Например, возьмем выше приведенный пример

```
using (ApplicationContext db = new ApplicationContext())
{
    Company company1 = new Company { Name = "Google" };
    Company company2 = new Company { Name = "Microsoft" };
    User user1 = new User { Name = "Tom", Company = company1 };
    User user2 = new User { Name = "Bob", Company = company2 };
    User user3 = new User { Name = "Sam", Company = company2 };

    db.Companies.AddRange(company1, company2);  // добавление компаний
    db.Users.AddRange(user1, user2, user3);     // добавление пользователей
    db.SaveChanges();

    foreach (var user in db.Users.ToList())
    {
        Console.WriteLine($"{user.Name} работает в {user.Company?.Name}");
    }
}
```

# Загрузка сущностей со сложной многоуровневой структурой

В примере выше структура моделей довольна простая - главная сущность связана с другой простой сущностью. Рассмотрим более сложную структуру моделей. Допустим, у каждой компании есть связанная сущность - страна, где находится компания

# Загрузка сущностей со сложной многоуровневой структурой

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

# Загрузка сущностей со сложной многоуровневой структурой

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

    public int CountryId { get; set; }
    public Country? Country { get; set; }
}
```

# Загрузка сущностей со сложной многоуровневой структурой

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

# Загрузка сущностей со сложной многоуровневой структурой

Допустим, вместе с пользователями мы хотим загрузить и страны, в которых базируются компании пользователей. То есть получается, что нам нужно спуститься еще на уровень ниже: User - Company - Country. Для этого нам надо применить метод ThenInclude(), который работает похожим образом, что и Include.

# Загрузка сущностей со сложной многоуровневой структурой

```
using (ApplicationContext db = new ApplicationContext())
{
    // получаем пользователей
    var users = db.Users
        .Include(u => u.Company)  // подгружаем данные по компаниям
            .ThenInclude(c => c!.Country)    // к компаниям подгружаем данные по странам
        .ToList();
    foreach (var user in users)
        Console.WriteLine($"{user.Name} - {user.Company?.Name} - {user.Company?.Country?.Name}");
}
```

В этом случае мы можем использовать оператор ! (null-forgiving оператор), чтобы указать, что значение null в данной ситуации невозможно.

# Загрузка сущностей со сложной многоуровневой структурой

В итоге на уровне базы данных это выльется в следующий код SQL:

```
SELECT "u"."Id", "u"."CompanyId", "u"."Name", "c"."Id", "c"."CountryId", "c"."Name", "c0"."Id", "c0"."Name"
FROM "Users" AS "u"
LEFT JOIN "Companies" AS "c" ON "u"."CompanyId" = "c"."Id"
LEFT JOIN "Countries" AS "c0" ON "c"."CountryId" = "c0"."Id"
```

# Загрузка сущностей со сложной многоуровневой структурой

Также мы можем использовать тот же метод Include для загрузки данных далее по цепочке:

```
using (ApplicationContext db = new ApplicationContext())
{
    var users = db.Users
        .Include(u => u.Company!.Country)
        .ToList();
    foreach (var user in users)
        Console.WriteLine($"{user.Name} - {user.Company?.Name} - {user.Company?.Country!.Name}");
}
```

# LINQ to Entities

# LINQ to Entities

Для извлечения данных из базы данных Entity Framework Core использует технологию LINQ to Entities. В основе данной технологии лежит язык интегрированных запросов LINQ (Language Integrated Query). LINQ предлагает простой и интуитивно понятный подход для получения данных с помощью выражений, которые по форме близки выражениям языка SQL.

# LINQ to Entities

Хотя при работе с базой данных мы оперируем запросами LINQ, но, реляционные базы данных как MS SQL Server или SQLite, понимают только запросы на языке SQL. Поэтому Entity Framework Core, используя выражения LINQ to Entities, транслирует их в определенные запросы, понятные для используемого источника данных.

# LINQ to Entities

Создавать запросы мы можем двумя способами: через операторы LINQ и через методы расширения.

```
var users = (from user in db.Users.Include(p=>p.Company)
             where user.CompanyId == 1
             select user).ToList();

//////////////////////////////////////////////////////

using(ApplicationContext db = new ApplicationContext())
{
    var users = db.Users.Include(p=>p.Company).Where(p=> p.CompanyId == 1);
}
```

# LINQ to Entities

Для большинства методов определены асинхронные версии, при необходимости получать данные в асинхронном режиме, мы можем их задействовать:

```
using Microsoft.EntityFrameworkCore;

using (ApplicationContext db = new ApplicationContext())
{
    var users = await db.Users
                        .Include(p => p.Company)
                        .Where(p => p.CompanyId == 1)
                        .ToListAsync();     // асинхронное получение данных

    foreach (var user in users)
        Console.WriteLine($"{user.Name} ({user.Age}) - {user.Company?.Name}");
}
```

# КОНЕЦ