Введение
Entity Framework

ADO.NET

ORM Entity Framework

Entity Framework (EF) – относится к ORM технологиям

ORM (Object-Relational Mapping) – технология выполняющее связывание реляционной базы данных с концепциями объектно-ориентированных языков программирования.

В результате, ORM технология предоставляет разработчику работать с объектами языка программирования, что является более высоким уровнем абстракции, по сравнению с понятиями реляционной БД.

Сущности (Entity)

Сущность (Entity) представляет набор данных, ассоциированных с определенным объектом.

Сущность, может обладать рядом свойств. Свойства могут представлять как простые данные (например, типа int), так сложные структуры данных.

И 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**.

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.

В EF для выборки данных из БД используется язык LINQ. С его помощью можно не только извлекать определенные строки из БД, но и получать объекты, связанные различными ассоциативными связями.

Entity Data Model

Модель данных сопоставляет классы сущностей с реальными таблицами в БД и состоит из трех уровней:

- Концептуального,
- уровень хранилища,
- уровень сопоставления.

На концептуальном уровне происходит 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 определет сопоставление между свойствами класса сущности и столбцами таблиц.

Таким образом, мы можем через классы, определенные в приложении, взаимодействовать с таблицами из базы данных.

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. Модель:

```
// Модель
   public class Car
   {
       public int CarId { get; set; }

       public string Model { get; set; }

       public int Year { get; set; }

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

Контекст для работы с БД

```
public class Parking : DbContext
    {
        public DbSet<Car> Cars { get; set; }

        public Parking()
            : base()
        {

        }

        // Конструктор для использования на уже открытом соединении
        public Parking(DbConnection existingConnection, bool contextOwnsConnection)
            : base(existingConnection, contextOwnsConnection)
        {

        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<Car>().MapToStoredProcedures();
        }
    }
```

```
public static void ExecuteExample()
{
    string connectionString = "server=localhost;port=3305;database=parking;uid=root;";
    using (MySqlConnection connection = new MySqlConnection(connectionString)) {
        using (Parking contextDB = new Parking(connection, false)) {
            contextDB.Database.CreateIfNotExists();
        }  connection.Open();
        MySqlTransaction transaction = connection.BeginTransaction();
        try {  // DbConnection that is already opened
            using (Parking context = new Parking(connection, false)) {
                // Interception/SQL logging
                context.Database.Log = (string message) => { Console.WriteLine(message); };
                // Passing an existing transaction to the context
                context.Database.UseTransaction(transaction);
                // DbSet.AddRange
                List<Car> cars = new List<Car>();

                cars.Add(new Car { Manufacturer = "Nissan", Model = "370Z", Year = 2012 });
                cars.Add(new Car { Manufacturer = "Ford", Model = "Mustang",Year = 2013 });
                cars.Add(new Car { Manufacturer ="Chevrolet",Model ="Camaro",Year = 2012});
                cars.Add(new Car { Manufacturer = "Dodge", Model ="Charger",Year = 2013 });

                context.Cars.AddRange(cars);
                context.SaveChanges();
            }
            transaction.Commit();
        }
        catch {
            transaction.Rollback();
            throw;
        }
    }
}
```

Основные операции с данными

Общая информация

Большинство операций с данными представляют собой CRUD-операции (Create, Read, Update, Delete), то есть получение данных, создание, обновление и удаление. Entity Framework позволяет легко производить данные операции.

Для примера используется следующие модель и контекст данных:

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

public class PhoneContext : DbContext
{
    public DbSet<Phone> Phones { get; set; }
    ...
}
```

Добавление

```
using (PhoneContext db = new PhoneContext()) {
    Phone p1 = new Phone { Name = "Samsung Galaxy S7", Price = 20000 };
    Phone p2 = new Phone { Name = "iPhone 7", Price = 28000 };

    // добавление
    db.Phones.Add(p1);
    db.Phones.Add(p2);
    db.SaveChanges();   // сохранение изменений

    var phones = db.Phones.ToList();
    foreach (var p in phones)
        Console.WriteLine("{0} - {1} - {2}", p.Id, p.Name, p.Price);
}
```

Изменение

```
using (PhoneContext db = new PhoneContext()) {
    // получаем первый объект
    Phone p1 = db.Phones.FirstOrDefault();

    p1.Price = 30000;
    db.SaveChanges();   // сохраняем изменения
}
```

Однако, если объект получен в одном контексте, а изменения следует сохранить во втором, тогда необходимо явным образом установить для его состояния значение EntityState.Modified:

```
using (PhoneContext db = new PhoneContext()) {
    if (p1 != null) {
        p1.Price = 60000;
        db.Entry(p1).State = EntityState.Modified;
        db.SaveChanges();
    }
}
```

Удаление

```
using (PhoneContext db = new PhoneContext()) {
    Phone p1 = db.Phones.FirstOrDefault();
    if (p1 != null) {
        db.Phones.Remove(p1);
        db.SaveChanges();
    }
}
```

Если объект получен в одном контексте, а удаление выполняется во втором, тогда необходимо явным образом установить для его состояния значение EntityState.Deleted:

```
using (PhoneContext db = new PhoneContext()) {
    if (p1 != null) {
        db.Entry(p1).State = EntityState.Deleted;
        db.SaveChanges();
    }
}
```

Привязка объекта к контексту

В случае изменения или удаления объекта, когда он получен в другом контексте, в место изменения его состояния можно выполнить привязку к контексту с помощью метода Attach:

```
Phone p1;
using (PhoneContext db = new PhoneContext()) {
    p1 = db.Phones.FirstOrDefault();
}
// редактирование
using (PhoneContext db = new PhoneContext()) {
    if (p1 != null) {
        db.Phones.Attach(p1);
        p1.Price = 999;
        db.SaveChanges();
    }
}
// удаление
using (PhoneContext db = new PhoneContext()) {
    if (p1 != null) {
        db.Phones.Attach(p1);
        db.Remove(p1);
        db.SaveChanges();
    }
}
```

Привязка данных к DataGridView

```
PhoneContext db;

// При уст. начальных значений, например, в конструкторе формы:
db = new PhoneContext();
db.Phones.Load();

dataGridView1.DataSource = db.Phones.Local.ToBindingList();
```

Навигационные свойства

Навигационным свойством называется свойство, связывающее сущность с другой сущностью, или набором сущностей БД.

Пример: футболисты и команды футболистов:

```
class Player
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Position { get; set; }
    public int Age { get; set; }

    public int? TeamId { get; set; } // = Нав. Свойство + Имя ключа из связ.таб
    public Team Team { get; set; } // навигационной свойство
}

class Team
{
    public int Id { get; set; }
    public string Name { get; set; } // название команды
    public string Coach { get; set; } // тренер

    public ICollection<Player> Players { get; set; } // навигационной свойство
}
```

Внешний ключ позволяет получать связанные данные. Например, после генерации базы данных с помощью Code First таблица Players будет иметь следующее определение:

```
CREATE TABLE [dbo].[Players] (
    [Id]       INT            IDENTITY (1, 1) NOT NULL,
    [Name]     NVARCHAR (MAX) NULL,
    [Position] NVARCHAR (MAX) NULL,
    [Age]      INT            NOT NULL,
    [TeamId]   INT            NULL,
    CONSTRAINT [PK_dbo.Players] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.Players_dbo.Teams_TeamId] FOREIGN KEY ([TeamId])
        REFERENCES [dbo].[Teams] ([Id])
);
```

Способы загрузки и получения связанных данных

Жадная загрузка:

```
db.Players.Include(p => p.Team)
db.Teams.Include(t => t.Players)
```

Явная загрузка:

```
var p = db.Players.FirstOrDefault();
// Reference - для подгрузки одиночного объекта
db.Entry(p).Reference("Team").Load();

var t = db.Teams.FirstOrDefault();
// Collection - для подгрузки коллекции
db.Entry(t).Collection("Players").Load();
```

Ленивая загрузка:

- Данные подгружаются при первом обращении к навигационному свойству.
- Классы, использующие ленивую загрузку должны быть публичными, а их навигационные свойства должны иметь модификаторы public и virtual.
- Ни каких специальных методов загрузки вызывать не надо.