EF

https://www.entityframeworktutorial.net/

https://docs.microsoft.com/ru-ru/ef/

Гаан Виктор Викторович

ORM(Object-Relational Mapping)

- Когда необходимо обеспечить работу с данными в терминах классов, а не таблиц данных.
- Избавляет программиста от написания большого количества кода.

Entity Framework

EF - это современный модуль сопоставления "объект — база данных" для .NET.

ADO.NET обязывает помнить о физической структуре серверной базы данных.

EF сокращает разрыв между базой данных и объектноориентированным программированием.

ADO .NET

Введение в ADO .NET

- Набор классов для взаимодействия с реляционными базами данных
- Унифицированный интерфейс
- MS SQL Server, MySQL, Oracle, Access, DB2
- Гибкий выбор поставщика данных

Способы работы с ADO .NET

Способы работы с ADO .NET

- Подключенный уровень
  - объект подключения, объект команды, объект чтения данных
- Автономный уровень
  - DataSet, DataTable
  - Entity Framework (EF)

Способы работы с ADO .NET

Основные объекты для работы

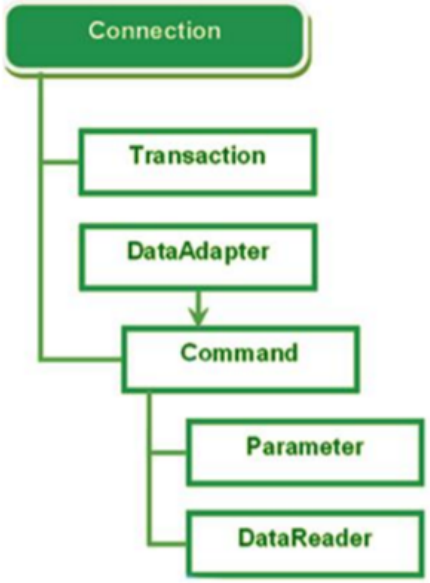
- Connection
- Command
- DataReader
- DataSet
- DataAdapter

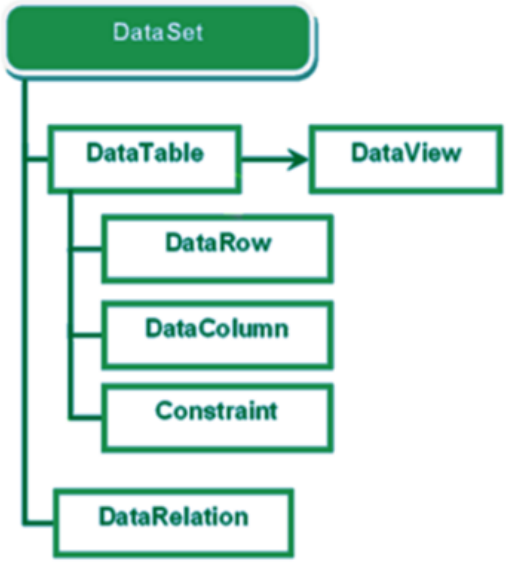
Способы работы с ADO .NET

Основные объекты для работы

- Connection
- Command
- DataReader
- DataSet
- DataAdapter

Объектная модель ADO.NET





Поставщики данных

Поставщики данных

- Нет единого набора типов
- Есть различные поставщики данных
- Основные объекты поставщиков данных:
  - Connection
  - Command
  - Parameter
  - DataReader
  - DataAdapter
  - Transaction

Подключенный уровень

Шаги для работы с БД:

1. Создать объект подключения
2. Создать объект команды
3. Вызвать метод ExecuteReader() у объекта команды
4. Обработать записи с помощью метода Read()

Подключенный уровень

```
using (SqlConnection cn = new SqlConnection())
{
    cn.ConnectionString = connectionString;

    cn.Open();
}

SqlCommand command = new SqlCommand();
command.CommandText = sqlExpression;
command.Connection = connection;
SqlDataReader reader = command.ExecuteReader();
while (reader.Read()) // построчно считываем данные
{
    object id = reader.GetValue(0);
    object name = reader.GetValue(1);
    object age = reader.GetValue(2);
}
```

```
// Создать и открыть подключение
using (SqlConnection cn = new SqlConnection())
{
    cn.ConnectionString =
    @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
    "Initial Catalog=TestDB";
    cn.Open();

    // Создать объект команды SQL
    string strSQL = "Select * From Department";
    SqlCommand myCommand = new SqlCommand(strSQL, cn);

    // Получить объект чтения данных с помощью ExecuteReader()
    using (SqlDataReader myDataReader = myCommand.ExecuteReader())
    {
        // Организовать цикл по результатам
        while (myDataReader.Read())
        {
            Console.WriteLine(
                $"Office name: {myDataReader["Name"]}, Location: {myDataReader["Location"]}");
        }
    }
}
```

Подключенный уровень

SqlCommand

Чтобы выполнить команду, необходимо применить один из методов:

- ExecuteNonQuery
  - INSERT, UPDATE, DELETE
- ExecuteReader
  - SELECT
- ExecuteScalar
  - Min, Max, Sum, Count

Параметризированные объекты команд

Параметризированные объекты команд

Использование SQL параметров в виде объектов

- Сокращает количество опечаток
- Обычно выполняются быстрее
- Защищают от SQL injection атак

Параметризированные объекты команд

```
private void InsertColor(int id, string name, SqlConnection connection)
{
    string sql = string.Format("Insert Into Colors" +
    " (ColorID, Name) Values" +
    "(@ColorID, @Name)");

    using (SqlCommand cmd = new SqlCommand(sql, connection))
    {
        SqlParameter param = new SqlParameter();
        param.ParameterName = "@ColorID";
        param.Value = id;
        param.SqlDbType = SqlDbType.Int;
        cmd.Parameters.Add(param);

        param = new SqlParameter();
        param.ParameterName = "@Name";
        param.Value = name;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);

        cmd.ExecuteNonQuery();
    }
}
```

Автономный уровень

Автономный уровень

Автономный уровень в ADO .NET

Можно использовать объектную модель ADO .NET автономно.

Моделирование реляционных данных с помощью модели объектов, находящихся в памяти: таблицы, отношения между таблицами, ограничения, первичные ключи и др.

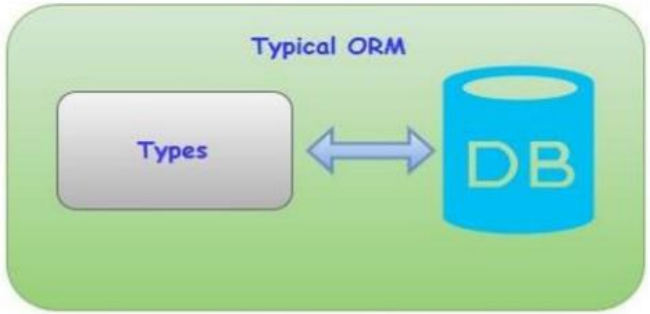
Адаптер данных

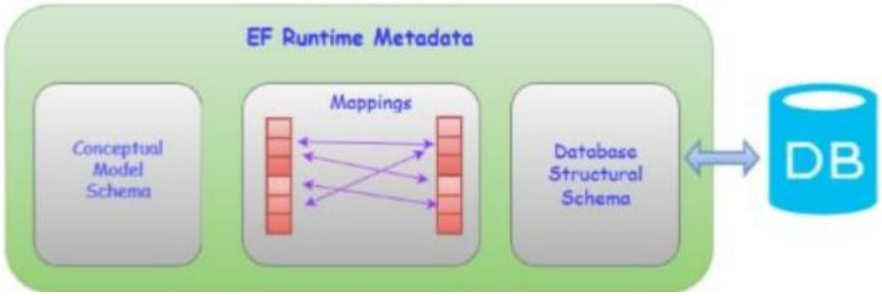
Выбирает и обновляет данные с помощью объектов DataSet

DataSet → DataTable → DataRow, DataColumn

Адаптеры данных удерживают подключение открытым в течение минимально возможного времени

Entity Framework

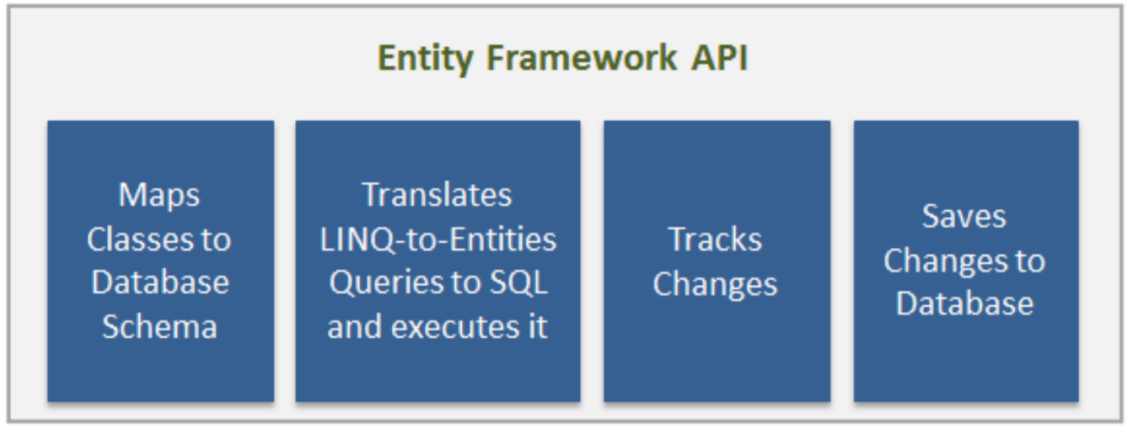




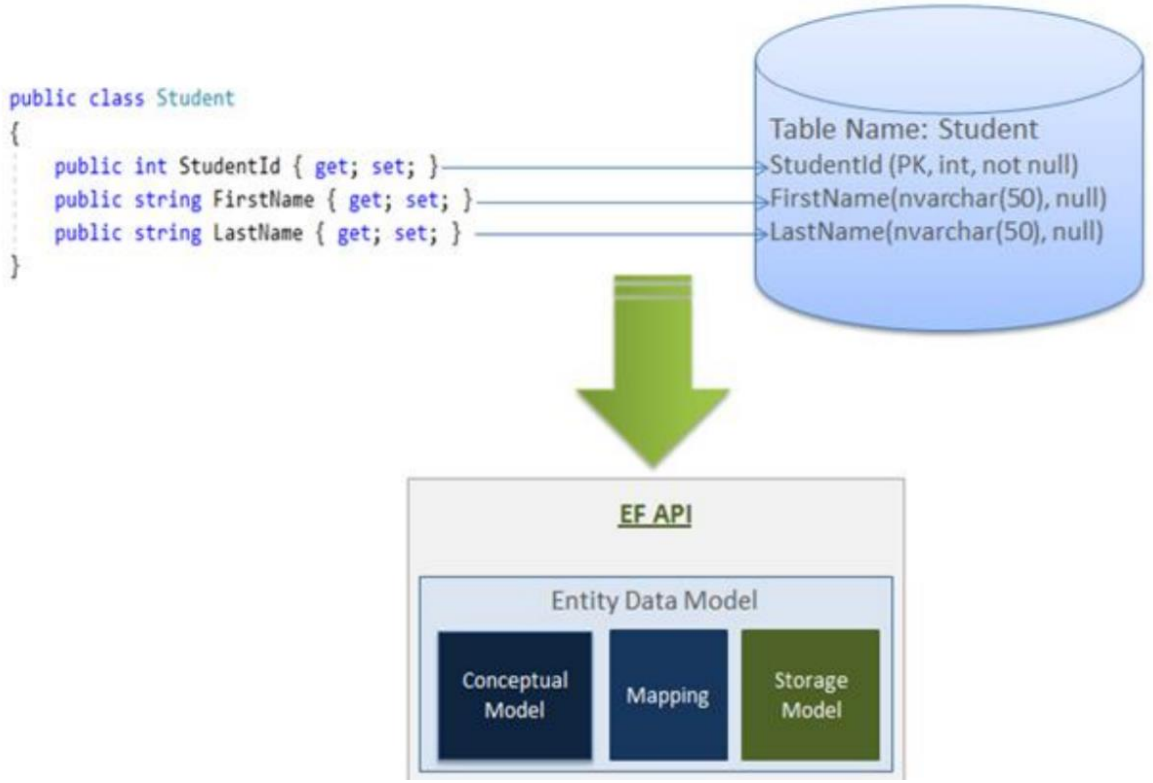
Entity Framework (Свойства)

- Кросс-платформенность;
- Моделирование. EDM (Entity Data Model);
- Запросы. LINQ;
- Отслеживание изменений в объектах;
- Сохранение изменений. INSERT, UPDATE и DELETE;
- Optimistic concurrency;
- Транзакции;
- Кеширование;
- Конфигурации. Data annotations attributes, Fluent API;
- Миграции;

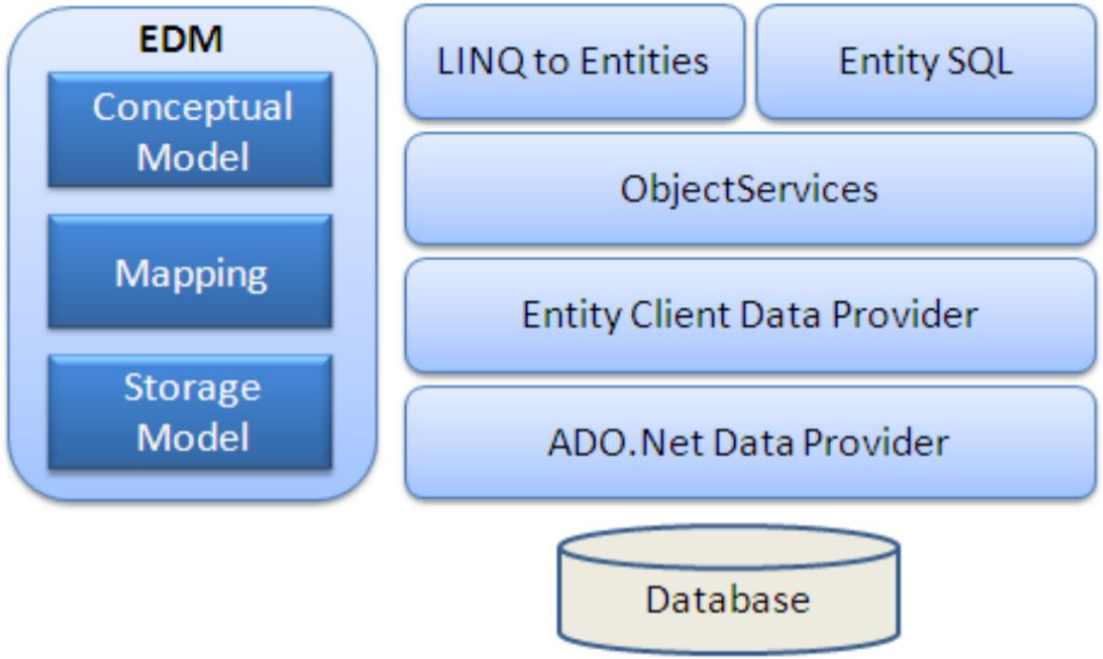
Entity Framework (Как работает)



Entity Data Model



Архитектура Entity Framework



Entity Framework

Context Class

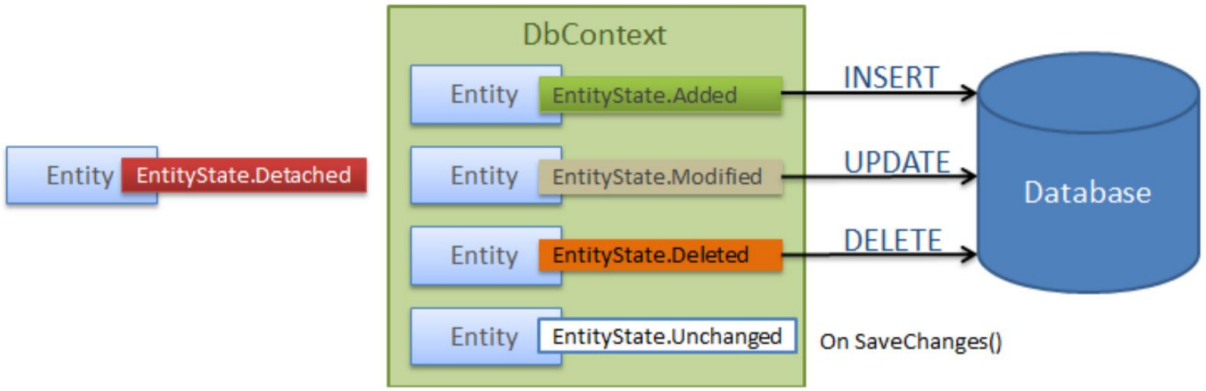
Entity

- POCO Entities (Plain Old CLR Object)
- Dynamic Proxy Entities (POCO Proxy)

Properties

- Scalar property
- Navigation property: Reference Navigation и Collection Navigation

Entity State



Entity State

Класс ChangeTracker в Entity Framework Core начинает отслеживать все сущности, как только они извлекаются с помощью DbContext, пока они не выйдут за пределы его области. EF отслеживает все изменения, примененные ко всем сущностям и их свойствам.

Unchanged

Все объекты, полученные с помощью прямого запроса SQL или запросов LINQ-to-Entities, а также объекты, после сохранения будут иметь состояние Unchanged.

```
using (EFContext context = new EFContext())
{
    User user = context.Users.First();
    User userFromSql = context.Users.FromSqlRaw("SELECT * FROM Users WHERE Id=2").First();
    Display(context.ChangeTracker.Entries());
}



static void Display(IEnumerable<EntityEntry> entities)
{
    entities.ToList().ForEach(x => Console.WriteLine($"Тип сущности: {x.Entity.GetType().Name}," +
        $"Состояние: {x.State}"));
}
```

Added

Объекты добавленные с помощью, метода Add(), будут помечены как Added.

```
using (EFContext context = new EFContext())
{
    User user = new User()
    {
        Name="Bill",
        RoleId=1,
    };
    context.Add(user);
    Display(context.ChangeTracker.Entries());
}



static void Display(IEnumerable<EntityEntry> entities)
{
    entities.ToList().ForEach(x => Console.WriteLine($"Тип сущности: {x.Entity.GetType().Name}," +
        $"Состояние: {x.State}"));
}
```

Modified

Объекты обновленные с помощью, метода Update(), будут помечены как Modified.

```
using (EFContext context = new EFContext())
{
    User user = context.Users.First();
    user.Name = "Bill";
    context.Update(user);
    Display(context.ChangeTracker.Entries());
}



static void Display(IEnumerable<EntityEntry> entities)
{
    entities.ToList().ForEach(x => Console.WriteLine($"Тип сущности: {x.Entity.GetType().Name}," +
        $"Состояние: {x.State}"));
}
```

Deleted

Объекты определенные на удаление с помощью, метода Remove(), будут помечены как Deleted.

```
using (EFContext context = new EFContext())
{
    User user = context.Users.First();
    context.Remove(user);
    Display(context.ChangeTracker.Entries());
}



static void Display(IEnumerable<EntityEntry> entities)
{
    entities.ToList().ForEach(x => Console.WriteLine($"Тип сущности: {x.Entity.GetType().Name}," +
        $"Состояние: {x.State}"));
}
```

Detached

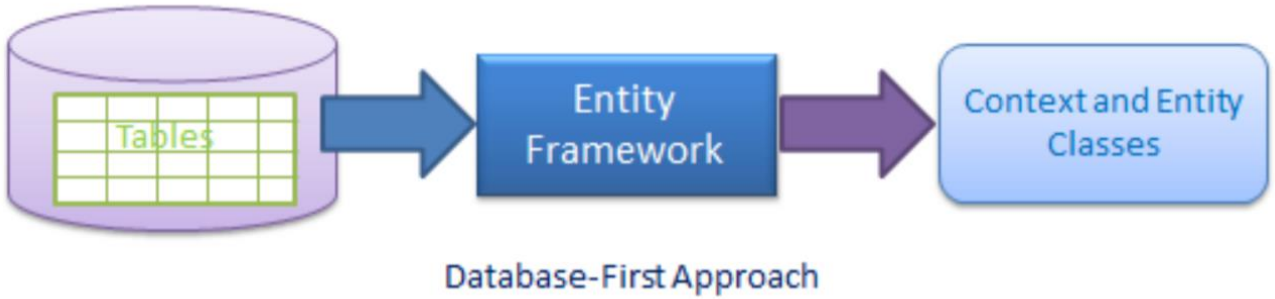
Объекты, которые не отслеживаются и не используют экземпляр DbContext, их называют отключенные объекты.

```
using (EFContext context = new EFContext())
{
    User user = new User() { Name="Jack",RoleId=1};
    Console.WriteLine(context.Entry(user).State);
}
```

Подходы к разработке

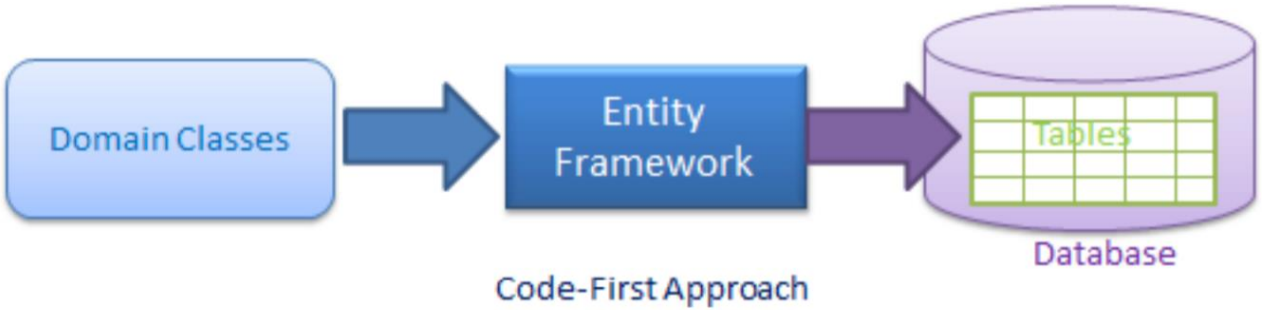
- Database-First (Только EF6)
- Code-First (EF 6 и EF Core)
- Model-First (Только EF6)

Database-First

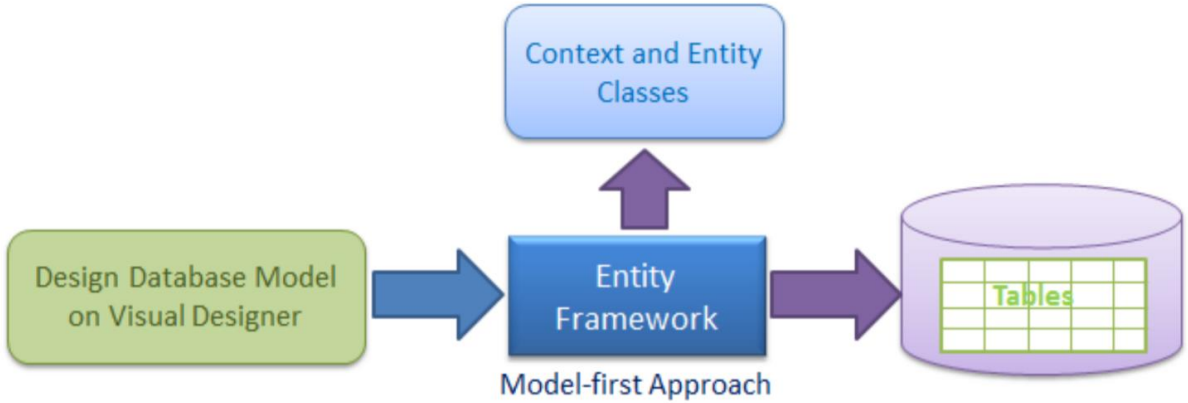


Database-First Approach

Code-First


Domain Classes
Entity Framework
Tables
Database
Code-First Approach

Model-First


Context and Entity Classes
Design Database Model on Visual Designer
Entity Framework
Tables
Model-first Approach

Стратегии инициализации БД

- CreateDatabaseIfNotExists
- DropCreateDatabaseIfModelChanges
- DropCreateDatabaseAlways
- Custom DB Initializer

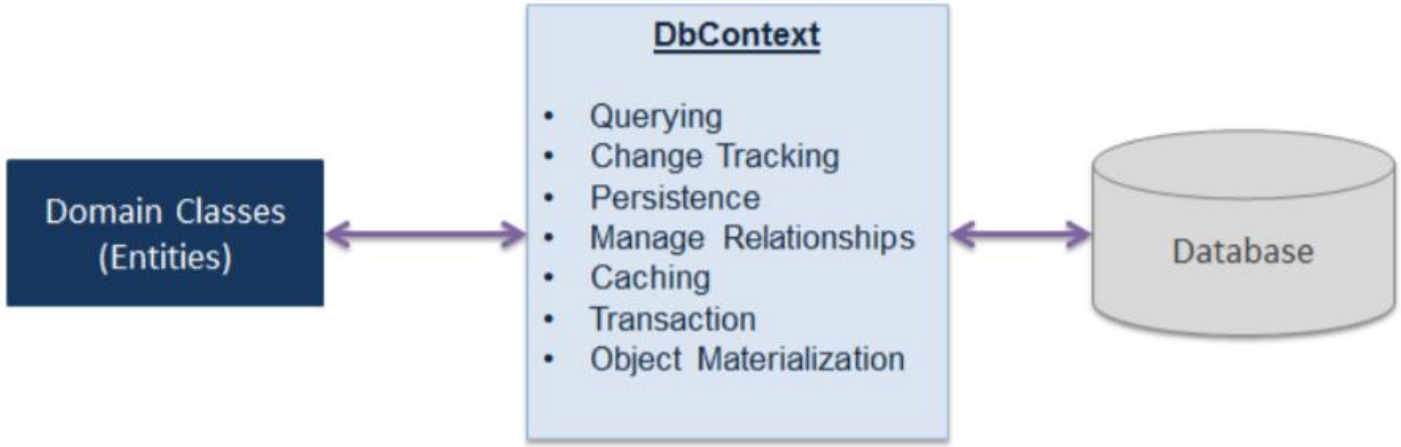
Конфигурирование классов

- Data Annotation Attributes
- Fluent API

Data Annotations

- Table
- Column
- Index
- ForeignKey
- NotMapped
- DatabaseGenerated
- Key
- Timestamp
- Required
- MinLength / MaxLength
- И другие

DbContext


DbContext
Querying
Change Tracking
Persistence
Manage Relationships
Caching
Transaction
Object Materialization
Domain Classes
(Entities)
Database

DbContext

Экземпляр DbContext представляет сеанс работы с базой данных и может использоваться для запроса и сохранения экземпляров сущностей. DbContext — это сочетание шаблонов единиц работы и репозитория.

Основные методы в DbContext

- Entry
- SaveChanges / SaveChangesAsync
- Set
- OnModelCreating

Запросы

Три типа запросов к данным:

- LINQ-to-Entities (EF6 и EF Core)
- Entity SQL (Полный функционал EF6)
- Native SQL (Полный функционал EF6)

Типы загрузок навигационных свойств

Три типа загрузок:

- Ранняя загрузка / Eager Loading
- Ленивая загрузка / Lazy Loading
- Явная загрузка / Explicit Loading

https://docs.microsoft.com/ru-ru/ef/core/querying/related-data/

Ранняя загрузка

```
using (EFContext context = new EFContext())
{
    var query = context.Orders.Include(x=>x.User).Include(x=>x.Positions).ToList();
}
```

Ранняя загрузка реализуется с помощью метода расширения Include, где указывается навигационное свойство, которое надо подгрузить к нашей основной коллекции.
Для работы с Include, не забудьте добавить зависимость — using Microsoft.EntityFrameworkCore;

Ленивая загрузка

```
using (EFContext context = new EFContext())
{
    var query = context.Orders.ToList();

    var order = query[0];

    var user = order.User;
}
```

Для работы ленивой загрузки надо, через менеджер пакетов NuGet поставить Microsoft.EntityFrameworkCore.Proxies.
А также в вашем контексте, в методе конфигурации, добавить, метод (optionsBuilder.UseLazyLoadingProxies()) который разрешает использование ленивой загрузки.

Явная загрузка

```
using (EFContext context = new EFContext())
{
    var order = context.Orders.FirstOrDefault();

    context.Entry(order).Reference(x => x.User).Load();
    context.Entry(order).Collection(x => x.Positions).Load();
}
```

Reference — для загрузки навигационных объектов.
Collection — для загрузки навигационных коллекций.

Entity Sql (EF 6)

```
using (var ctx = new SchoolDBEntities())
{
    //Querying with Object Services and Entity SQL
    string sqlString = "SELECT VALUE st FROM Students "
                       + "AS st WHERE st.StudentName == 'Bill'";
    var objctx = (ctx as IObjectContextAdapter).ObjectContext;
    ObjectQuery<Student> student = objctx.CreateQuery<Student>(sqlString);
    Student newStudent = student.First<Student>();
}
```

```
using (EFContext context = new EFContext())
{
    var query = context.Database.ExecuteSqlRaw("DELETE FROM Users WHERE Id=1");
}
```

Этот вариант выполнения, не только DELETE, UPDATE, INSERT, но также и процедур, представлений и т.д.

Native SQL

```
using (EFContext context = new EFContext())
{
    var users = context.Users.FromSqlRaw("SELECT * FROM Users WHERE FirstName = N'Иван'").ToList();
}
```

Этот метод, используется для всевозможных выборок из таблицы.

Наследование

EF может сопоставлять иерархию типов .NET с базой данных. Это позволяет создавать сущности .NET в коде обычным образом, используя базовые и производные типы и позволяя EF легко создавать соответствующую схему базы данных, выдавать запросы и т. д. Фактические сведения о сопоставлении иерархии типов являются зависимыми от поставщика; на этой странице описывается поддержка наследования в контексте реляционной базы данных.

Наследование

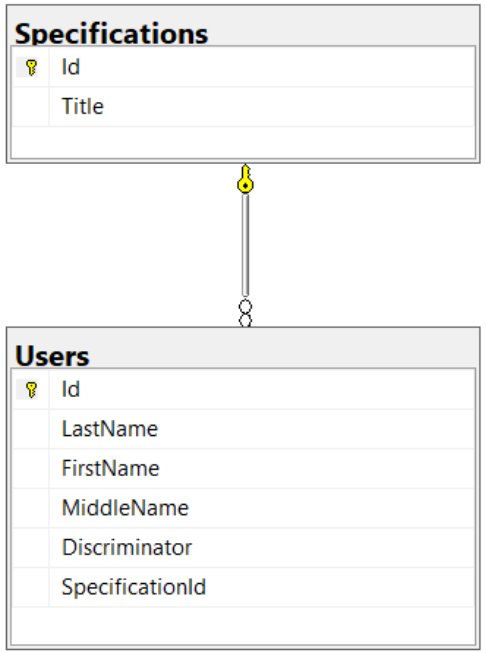
По умолчанию EF сопоставляет наследование, используя шаблон «одна таблица на иерархию ». Функция «подтаблица» использует одну таблицу для хранения данных всех типов в иерархии, а столбец дискриминатора используется для указания типа, представляемого каждой строкой.

Наследование (Пример)

```
public class User
{
    Ссылок: 0
    public int Id { get; set; }
    ссылка: 1
    public string LastName { get; set; }
    ссылка: 1
    public string FirstName { get; set; }
    ссылка: 1
    public string MiddleName { get; set; }
}
```

```
public class Employee:User
{
    Ссылок: 0
    public int SpecificationId { get; set; }
    Ссылок: 0
    public Specification Specification { get; set; }
}
```

```
public class Specification
{
    Ссылок: 0
    public int Id { get; set; }
    Ссылок: 0
    public string Title { get; set; }
    Ссылок: 0
    public ICollection<Employee> Employees { get; set; }
}
```



Пример реализации Code First (Data Annotations)

```
public class User
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    Ссылок: 0
    public int Id { get; set; }
    [Required]
    [MaxLength(255)]
    ссылка: 1
    public string? LastName { get; set; }
    [Required]
    [MaxLength(255)]
    ссылка: 1
    public string? FirstName { get; set; }
    [MaxLength(255)]
    ссылка: 1
    public string? MiddleName { get; set; }
    [Required]
    [MaxLength(255)]
    Ссылок: 0
    public string? Login { get; set; }
    [Required]
    [MaxLength(255)]
    Ссылок: 0
    public string? Password { get; set; }

    [NotMapped]
    Ссылок: 0
    public string? FullName => $"{LastName} {FirstName} {MiddleName}";
    [InverseProperty("User")]
    Ссылок: 0
    public ICollection<Order>? Orders { get; set; }
}
```

Пример реализации Code First (Data Annotations)

```
[Table("Orders")]
Ссылок: 3
public class Order
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    Ссылок: 0
    public int Id { get; set; }
    Ссылок: 0
    public DateTime CreationDate { get; set; }
    [ForeignKey(nameof(User))]
    Ссылок: 0
    public int UserId { get; set; }
    [InverseProperty("Orders")]
    ссылка: 1
    public User? User { get; set; }
    [InverseProperty("Order")]
    Ссылок: 0
    public ICollection<Position>? Positions { get; set; }
}
```

Пример реализации Code First (Data Annotations)

```
[Table("Dishes")]
Ссылок: 2
public class Dish
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    Ссылок: 0
    public int Id { get; set; }
    [Required]
    [MaxLength(255)]
    Ссылок: 0
    public string? Name { get; set; }
    [Required]
    [Column(TypeName = "decimal(10,2)")]
    Ссылок: 0
    public decimal Price { get; set; }

    [InverseProperty("Dish")]
    Ссылок: 0
    public ICollection<Position>? Positions { get; set; }
}
```

Пример реализации Code First (Data Annotations)

```
[Table("Positions")]
Ссылок: 4
public class Position
{
    ссылка: 1
    public int OrderId { get; set; }
    ссылка: 1
    public int DishId { get; set; }
    Ссылок: 0
    public int Count { get; set; }
    [InverseProperty("Positions")]
    Ссылок: 0
    public Dish? Dish { get; set; }
    [InverseProperty("Positions")]
    Ссылок: 0
    public Order? Order { get; set; }
}
```

Пример реализации Code First (Data Annotations)

```
public class EFContext : DbContext
{
    Ссылок: 0
    public DbSet<User>? Users { get; set; }
    Ссылок: 0
    public DbSet<Dish>? Dishes { get; set; }
    Ссылок: 0
    public DbSet<Order>? Orders { get; set; }
    Ссылок: 0
    public DbSet<Position>? Positions { get; set; }
    ссылка: 1
    public EFContext()
    {

    }
    Ссылок: 0
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Position>().HasKey(x => new {x.OrderId,x.DishId});
    }
    Ссылок: 0
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Data Source=DESKTOP-OR7MOG3;Database=EFDatabase;Trusted_Connection=True;");
    }
}
```

Пример реализации Code First (Data Annotations)

[Key] — Определение первичного ключа. Также можно определять составные ключи, но только в EF6.
[DatabaseGenerated(DatabaseGeneratedOption.Identity)] — Определение генерации значения первичного ключа.
[Required] — Обозначает атрибут, как Not Null.
[MaxLength(255)] —Обозначает кол-во длину строки или массива. (NVARCHAR(255),VARCHAR(255),CHAR(255),VARBINARY(255) и т.д).
[NotMapped] — Обозначаются свойства, который в последствии маппинга, будут игнорироваться.
[InverseProperty("User")] — Определяем связь(one-to-many) между коллекцией(many) и навигационным свойством (one).
[ForeignKey(nameof(User))] — Определяет внешний ключ.

Пример реализации Code First (Fluent API)

Атрибуты определяются в методе контекста OnModelCreating. Классы точно такие же, только без атрибутов(Data Annotations).

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    //Ключи
    modelBuilder.Entity<User>().HasKey(x => x.Id);
    modelBuilder.Entity<Dish>().HasKey(x => x.Id);
    modelBuilder.Entity<Order>().HasKey(x => x.Id);
    //Определение составных ключей
    modelBuilder.Entity<Position>().HasKey(x => new { x.OrderId, x.DishId });

    //Свойства
    modelBuilder.Entity<User>().Ignore(x => x.FullName);
    modelBuilder.Entity<User>().Property(x => x.FirstName)
        .IsRequired()
        .HasMaxLength(255);
    modelBuilder.Entity<User>().Property(x => x.LastName)
        .IsRequired()
        .HasMaxLength(255);
    modelBuilder.Entity<User>().Property(x => x.MiddleName)
        .HasMaxLength(255);

    //Связи
    modelBuilder.Entity<Order>()
        .HasOne(x => x.User)
        .WithMany(x => x.Orders)
        .HasForeignKey(x => x.UserId)
        .OnDelete(DeleteBehavior.NoAction);
    modelBuilder.Entity<Position>()
        .HasOne(x => x.Order)
        .WithMany(x => x.Positions)
        .HasForeignKey(x => x.OrderId)
        .OnDelete(DeleteBehavior.NoAction);
    modelBuilder.Entity<Position>()
       .HasOne(x => x.Dish)
       .WithMany(x => x.Positions)
       .HasForeignKey(x => x.DishId)
       .OnDelete(DeleteBehavior.NoAction);
}
```

Управление схемами баз данных

- Миграции
- API создания и удаления
- Реконструирование

Миграции

Функция миграции в EF Core позволяет последовательно применять изменения схемы к базе данных, чтобы синхронизировать ее с моделью данных в приложении без потери существующих данных.
Для реализации миграции, у вас должно быть предустановлены следующие пакеты:
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.Design
Microsoft.EntityFrameworkCore.SqlServer (Или другой выбранный вами провайдер)
https://www.entityframeworktutorial.net/efcore/pmc-commands-for-ef-core-migration.aspx

Миграции

- Реализовать миграции, можно посредством консоли менеджера пакетов, который находится Средства→Диспетчер пакетов NuGet→Консоль диспетчера пакетов.
- Или посредством вызова команд в CLI .NET Core.

Миграции

```
PM> Add-Migration -Name "FirstMigration" -Context "EFContext"
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
```

Add-Migration, позволяет создавать миграцию.
-Name, это аргумент функции, определяющий наименование миграции, произвольное имя.
-Context, это второй агрумент, определяющий наименование вашего контекста.
По итогу у нас сформируется папка Migrations, с файлами миграции:
1. FirstMigration, с конфигурацией нашей сущностей.
2. Snapshot, нашего контекста для возможности откатить изменения.

Миграции

```
PM> Update-Database -Context "EFContext"
Build started...
Build succeeded.
Applying migration '20220324042858_FirstMigration'.
Done.
```

В контексте строка должна быть, такого формата:
Data source=DESKTOP-OR7MOG3;Database=MyDatabase;Integrated Security=true;

Более подробно про синтаксис, строки подключения: https://docs.microsoft.com/ru-ru/dotnet/framework/data/adonet/connection-string-syntax#sqlclient-connection-strings

API создания и удаления

EnsureCreated() Методы и EnsureDeleted() предоставляют упрощенную альтернативу EnsureCreated() для управления схемой базы данных. Эти методы полезны в сценариях, когда данные являются временными и могут быть удалены при изменении схемы.
Переход от EnsureCreated в миграцию не является эффективным процессом. Самый простой способ сделать это — удалить базу данных и создать ее повторно с помощью миграции. Если предполагается использование миграций в будущем, лучше всего начать с миграции, а не использовать EnsureCreated .

EnsureDeleted

EnsureDeleted Метод приведет к удалению базы данных, если она существует. Если у вас нет соответствующих разрешений, возникает исключение.

EnsureCreated

EnsureCreated создаст базу данных, если она не существует, и инициализировать схему базы данных. Если существуют какие-либо таблицы (включая таблицы для другого DbContext класса), схема не будет инициализирована.

Реконструирование

```
PM> Scaffold-DbContext 'Data source=DESKTOP-OR7MOG3;Database=Agents;Integrated Security=true;' Microsoft.EntityFrameworkCore.SqlServer
Build started...
Build succeeded.
To protect potentially sensitive information in your connection string, you should move it out of source code. You can avoid scaffolding
the connection string by using the Name= syntax to read it from configuration - see https://go.microsoft.com/fwlink/?linkid=2131148. For
more guidance on storing connection strings, see http://go.microsoft.com/fwlink/?LinkId=723263.
```

Реконструирование — это процесс формирования шаблонов классов типов сущностей и класса DbContext на основе схемы базы данных. его можно выполнить с помощью Scaffold-DbContext команды в средствах EF Core диспетчер пакетов Console (PMC) или dotnet ef dbcontext scaffold команды интерфейса командной строки (CLI) .net.
Пример выполнен с помощью консоли диспетчера пакетов, для реализации нужны следующие пакеты:
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.Tools
Microsoft.EntityFrameworkCore.Design
Microsoft.EntityFrameworkCore.SqlServer (Или другой выбранный вами провайдер)
https://docs.microsoft.com/ru-ru/ef/core/managing-schemas/scaffolding?tabs=vs