vliashko
2 июн 2022 в 09:25

.NET 6 и провайдеры баз данных

18 мин 13K

.NET*, C#*, SQL*

Из песочницы

Все материалы, которые будут показываться в ходе данной статьи будут доступны по данной ссылке. Вполне возможно, что со временем данный репозиторий будет обновляться, или, некоторые захотят сами принять участие в его развитии.

Можно ли сегодня представить разработку, будь то десктопы или веб, без использования баз?

Ну, чисто в теории можно, есть еще старенькие проекты, использующие файловую систему, идею которых можно еще увидеть в университетских лабораторных по сей день.

В чем же так плоха файловая система? Ну на самом деле, говоря на своем опыте, можно выделить следующие пункты:

1. Блокировка файла, в который идет запись
2. Отсутствие специализированных программ для работы с файлами (аналог СУБД)

Да, в какой-то мере можно выделить еще минусы, или попытаться закрыть уже названные мной. Но в целом главная идея базы данных – это удобство для чтения данных, а также наличие огромного числа инструментов для работы с данными (возможность быстрого поиска по полям таблицы, соединение таблиц, группировка записей, индексирование и т.д.)

Будем считать, что я смог в какой-то мере убедить, или хотя бы заинтриговать тем, что базы – это крутой механизм, который надо знать и уметь использовать.

Говоря о базах, я упомянул понятие запросов. В общих чертах запрос – это команда, которую ты говоришь выполнить базе. Запросы пишутся на языке SQL, состоят из предложений, и вот основные из них:

- SELECT
- FROM
- JOIN
- WHERE
- GROUP BY
- HAVING
- ORDER BY

Для данной статьи будем использовать базу данных состоящую из 3 таблиц:

dbo.Student

- *Id*
- *Name*
- *Course*
- *BirthDate*

dbo.Department

- *Id*
- *Name*

dbo.Coursework

- *Id*
- *StudentId*
- *DepartmentId*
- *DeliveryDate*

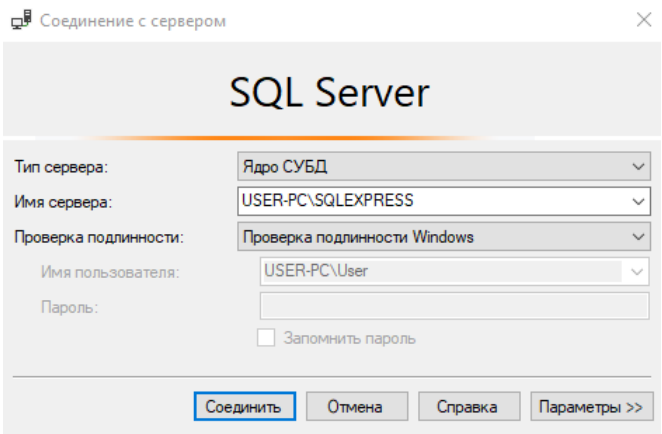
```
-- SELECT указывает на то, какие поля мы хотим выбрать.
-- Если указать *, то это означает выбор всех полей.
SELECT Student.Name, Student.BirthDate, COUNT(*) AS [Количество курсовых]
-- FROM указывает на то, из какой таблицы мы хотим вытащить данные
FROM Student
-- JOIN предназначен для объединения таблиц по какому либо условию.
-- В данном случае мы делаем связь по айдишникам студентов.
JOIN Coursework ON Student.Id = Coursework.StudentId
-- WHERE позволяет фильтровать выбору по какому либо условию .
-- Так, в этом случае я ищу тех студентов, которые имеют в имени начало Vladzimir.
WHERE Student.Name LIKE 'Vladzimir%'
-- GROUP BY нужен для группировки выборки, в данном случае группируем по студентам,
-- чтобы найти сколько у каждого студента сдано курсовых.
GROUP BY Student.Name, Student.BirthDate
-- HAVING представляет собой вторичную фильтрацию, и используется после GROUP BY.
-- В данном случае нам интересны те студенты, у которой больше 1 курсовой работы.
HAVING COUNT(*) > 1
-- ORDER BY служит для сортировки. Так, мы сортируем по убыванию
-- по полю день рождения.
-- Для сортировки по возрастанию надо убрать ключевое слово DESC.
ORDER BY BirthDate DESC
```

Отлично! Итого мы получаем всех студентов и количество их курсовых, если их сдано больше 1. Также нас интересуют только те студенты, у которых имя начинается с Vladzimir. При этом для удобства отсортировали по убыванию даты рождения.

Немного практики с SQL и он уже не кажется таким страшным и сложным. Хотя практиковаться с ним надо много, так как существует запросы куда больше и сложнее.

Теперь хорошая возможность перейти к нашей разработке.. Как же нам подружить наше приложение с базой данных?

Первое с чем надо определиться - это сервер и непосредственно наша база. Если мы пользуемся СУБД, то узнать данные параметры очень легко:



В данном случае, так как я пользовался MS SQL Server, то использовать Microsoft SQL Server Managment Studio (SSMS) является лучшим решением. При использовании, допустим, PostgreSQL можно использовать PG Admin в качестве СУБД.

При установке SQL сервера, в зависимости от версии (Express или расширенная) будет доступно несколько серверов:

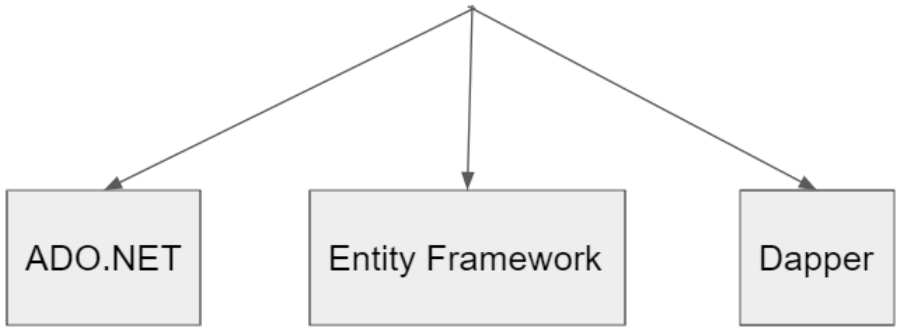
{Имя компьютера}\SQLEXPRESS,

{Имя компьютера}.

Также отдельно можно поставить (localdb)\MSSQLLocalDB.

Теперь, определившись с сервером и базой данных (название можно придумать какое угодно, но для примеров ниже база будет называться University), можно приступать к изучению платформы .NET.

На самом деле на платформе .NET есть три основных решения для данной ситуации (в реальности их будет и больше, скорее всего, но реально поддерживаемых, стабильных и проверенных только три).



Поговорим немного о теории.

У нас есть 3 основных понятия, от которых и будет опираться:

1. ORM - это расшифровывается как Object/Relational Mapping, предоставляет большой перечень работы с базой, и базовые удобства, например как, CRUD операции из под коробки, также поддержка Change Tracker, Unit of Work и т.д.
2. Micro-ORM - представляет собой возможность сопоставления данных из таблиц с классами C#. На этом всё.
3. Провайдер базы данных - предоставляет возможность установить соединение с базой данных и отправить туда запрос. Всё остальное лежит на плечах программиста.

Поэтому, когда говорим об ADO.NET - это провайдер, EntityFramework (EF) - ORM, а Dapper - micro-ORM.

ADO.NET

Необходимо наличие следующих пакетов:

- System.Data.SqlClient

Первым, и главным столпом является ADO.NET. Данный посредник между приложением и базой данных является самым старым, и предоставляет наибольшую свободу при работе с данными. Что предоставляет нам данная технология? На самом деле не так много, он позволяет открыть соединение с базой, и возможность отправки запроса в базу, а дальше.. Ну делайте что хотите в общем, его это уже не касается. В связи с этим большая необходимость в том, чтобы самим

отлавливать все исключения, и самим закрыть соединение с базой после выполнения запроса (да, для особо ленивых придумана конструкция using еще).

Основные классы, используемые для работы с ADO.NET:

- SqlConnection
- SqlCommand
- SqlDataReader
- SqlParameter

Так, напишем консольное приложение, для того, чтобы попробовать работу с ADO.NET:

Первое, что мы делаем - создаем объект класса SqlConnection, в который передаем connectionString, эта переменная в которой содержится строка подключения к нашей базе

В моем случае она будет такая:

```
var connectionString =
  "Data Source=.\\SQLEXPRESS;Initial Catalog=University;Integrated Security=True";
```

```
// Объявлем соединение с определенной строкой подключения.
var sqlConnection = new SqlConnection(connectionString);
```

В дальнейшем можем приступать в работе с базой:

```
try
{
    sqlConnection.Open();
    Console.WriteLine("SQL соединение открыто.");

    // Добавление (аналогичный код для обновления / удаления).
    var sqlCommand = sqlConnection.CreateCommand();
    sqlCommand.CommandText = "INSERT INTO Student VALUES ('TestUser', 1, '20220101')";
    var affectedRows = sqlCommand.ExecuteNonQuery();
    Console.WriteLine($"Число затронутых строк: {affectedRows}");

    // Чтение.
    var sqlCommandForRead = sqlConnection.CreateCommand();
    sqlCommandForRead.CommandText = "SELECT * FROM Student";
    SqlDataReader reader = sqlCommandForRead.ExecuteReader();

    if (reader.HasRows)
    {
        while (reader.Read())
        {
            // При использовании reader[""] - мы получаем object,
            // если хотим конкретный тип,
            // то используем reader.GetString() / reader.GetInt() и т.д.
            Console.WriteLine($"Студент с Id: {reader["Id"]}, " +
                $"с курсом: {reader["Course"]}, " +
                $"с именем: {reader["Name"]}, " +
                $"с датой рождения: {reader["BirthDate"]}");
        }
    }

    reader.Close();

    // Получение результата агрегатной функции
```

```
    var sqlCommandForCount = sqlConnection.CreateCommand();
    sqlCommandForCount.CommandText = "SELECT COUNT(*) FROM Student";
    var count = sqlCommandForCount.ExecuteScalar();
    Console.WriteLine($"Полное число студентов: {count}");

    // Есть два варианта параметризации запросов.
    var name = "Some Student Name";

    // Плохой вариант, так как позволяет получать и изменять данные
    //при помощи механизма SQL-инъекций.
    var sqlString = $"INSERT INTO Student VALUES ('{name}', 1, '20220101')";

    var sqlCommandForInsertBadPractice = new SqlCommand(sqlString)
    {
        Connection = sqlConnection
    };

    affectedRows = sqlCommandForInsertBadPractice.ExecuteNonQuery();

    // Хороший вариант, добавление SQL параметров.
    sqlString = $"INSERT INTO Student VALUES (@name, 1, '20220101')";
    var sqlParamForName = new SqlParameter("@name", name);
    var sqlCommandForInsertGoodPractice = new SqlCommand(sqlString);

    // Добавление параметра.
    sqlCommandForInsertGoodPractice.Parameters.Add(sqlParamForName);
    affectedRows = sqlCommandForInsertBadPractice.ExecuteNonQuery();
}
catch (Exception ex)
{
    Console.WriteLine($"Ошибка: {ex.Message}");
    throw;
}
finally
{
    sqlConnection.Close();
    Console.WriteLine("SQL соединение закрыто.");
}
```

В целом тут собрано несколько запросов, но надо отметить что основная идея тут одна:

1. Пишем конструкцию try catch finally
2. В try открываем соединение, в finally его закрываем
3. Затем происходит выбор: что нам нужно - чтение или изменение?

Так, в случае изменения данных создается экземпляр класса SqlCommand, затем указывается SQL запрос и выполняется метод ExecuteNonQuery(), который возвращает нам число затронутых строк:

```
var sqlCommand = sqlConnection.CreateCommand();
sqlCommand.CommandText = "INSERT INTO Student VALUES ('TestUser', 1, '20220101')";
var affectedRows = sqlCommand.ExecuteNonQuery();
Console.WriteLine($"Число затронутых строк: {affectedRows}");
```

В данном случае мы делаем вставку 1 записи, а значит на консоли увидим, что число затронутых строк также ровно 1.

В случае чтения данных необходимо создать также создать создать команду, однако вместо того, чтобы вызвать ExecuteNonQuery(), надо будет вызвать ExecuteReader(), который вернет нам экземпляр SqlDataReader.

```
var sqlCommandForRead = sqlConnection.CreateCommand();
sqlCommandForRead.CommandText = "SELECT * FROM Student";
SqlDataReader reader = sqlCommandForRead.ExecuteReader();
```

После получения данного экземпляра, проверяем, вернул ли он какие-то либо строки, и если вернул, то тогда начинаем их читать

```
if (reader.HasRows)
{
  while (reader.Read())
  {
    // При использовании reader[""] - мы получаем object,
    // если хотим конкретный тип,
    // то используем reader.GetString() / reader.GetInt() и т.д.
    Console.WriteLine($"Студент с Id: {reader["Id"]}, " +
                      $"с курсом: {reader["Course"]}, " +
                      $"с именем: {reader["Name"]}, " +
                      $"с датой рождения: {reader["BirthDate"]}");
  }
}
```

После того, как мы выйдем из цикла while - обязательно закрываем reader.

```
reader.Close();
```

В случае необходимости получения данных путем вычисления агрегатной функции (COUNT, MIN, MAX, AVG, SUM) - применяют метод ExecuteScalar(), который возвращает первый столбец первой строки (чтоб в целом нам и нужно).

```
var sqlCommandForCount = sqlConnection.CreateCommand();
sqlCommandForCount.CommandText = "SELECT COUNT(*) FROM Student";
var count = sqlCommandForCount.ExecuteScalar();
Console.WriteLine($"Полное число студентов: {count}");
```

Теперь переходим наверное к самому интересному, а именно параметризация запросов.

В общем случае это можно сделать двумя способами: конкатенация строк и SQL-параметры, поговорим про каждый из этих методов по отдельности.

Пусть у нас будет переменная name, содержащая некоторую строку:

```
var name = "Some Student Name";
```

Интерполяция строк

К плюсам этого способа можно выделить более простой способ написания, который просто встраивает переменные в строку при помощи интерполяции строк

```
var sqlString = $"INSERT INTO Student VALUES ('{name}', 1, '20220101')";

var sqlCommandForInsertBadPractice = new SqlCommand(sqlString)
{
```

```
    Connection = sqlConnection
};

affectedRows = sqlCommandForInsertBadPractice.ExecuteNonQuery();
```

В чем минус этого метода? В тот, что входная строка никак не валидируется, а значит если внешний код никак об этом не позаботиться, то имеет место быть всякие SQL-инъекции, лишние добавления записей и т.д.

SQL-параметры

Тут ситуация гораздо лучше и не пропускает невалидные ситуации, которые могут быть в ситуации выше, однако приходится написать больше кода:

```
sqlString = $"INSERT INTO Student VALUES (@name, 1, '20220101')";
var sqlParamForName = new SqlParameter("@name", name);
var sqlCommandForInsertGoodPractice = new SqlCommand(sqlString);

sqlCommandForInsertGoodPractice.Parameters.Add(sqlParamForName);
affectedRows = sqlCommandForInsertBadPractice.ExecuteNonQuery();
```

На этом основные возможности ADO.NET заканчиваются. В целом основная идея - следит за ошибками со стороны провайдера, и писать SQL код.

Dapper

Необходимо наличие следующих пакетов:

- Dapper
- System.Data.SqlClient

Много лишней теории тут говорить не буду. В целом Dapper - это посредник, которому всё еще нужен SqlConnection, однако открытие и закрытие уже будет автоматическим и в общем случае будет использоваться оператор using. Также один из важных плюсов Dapper - это сопоставление результатов запроса с классами C#, а значит не придется не придется делать страшные манипуляции с reader, как это было в случае с ADO.NET

Классы, используемые при работы с Dapper

- SqlConnection

И используемые от него методы: .Query<T>() и .Execute().

Чтобы долго не тянуть - перейдем сразу к написанию консольного приложения по работе с Dapper:

```
using (var sqlConnection = new SqlConnection(connectionString))
{
    // Добавление (аналогичный код для обновления / удаления).
    sqlConnection.Execute(
      "INSERT INTO Student VALUES ('TestUserDapper', 1, '20220101')"
    );

    // Чтение данных.
    var students = sqlConnection.Query<Student>("SELECT * FROM Student").ToList();

    foreach (var student in students)
    {
        Console.WriteLine($"Студент с Id: {student.Id}, " +
```

```
            $"с курсом: {student.Course}, " +
            $"с именем: {student.Name}, " +
            $"с датой рождения: {student.BirthDate}");
    }

    // Получение результата агрегатной функции
    // В данном случае необходимо использование .FirstOrDefault(), так как
    // .Query<T> возвращает IEnumerable<T>, что является коллекцией.
    // И так как мы знаем что результатом будет 1 запись, то без зазрений совести
    // можем применить .FirstOrDefault(), чтобы получить число записей.
    var count = sqlConnection.Query<int>("SELECT COUNT(*) FROM Student")
                                                .FirstOrDefault();

    Console.WriteLine($"Общее число записей в таблице студентов: {count}");

    // Использование параметров.
    sqlConnection.Execute("INSERT INTO Student VALUES (@Name, @Course, @BirthDate)",
        new Student
        {
            Name = "SomeParamName",
            Course = 2,
            BirthDate = new DateTime(2022, 04, 04)
        });

    // Анонимные объекты new { }.
    sqlConnection.Execute("DELETE FROM Student WHERE Name = @name",
        new { name = "TestUserDapper" });
}
```

Даже сравнивания по объему кода уже видно, насколько Dapper проще в использовании.

В целом использование у Dapper следующее:

1. В конструкции using создать экземпляр SqlConnection с переданной в него строкой подключения
2. В зависимости от того, хотим ли получить данные, или их изменить - написать .Query<T> или .Execute

При чтении данных мы используем .Query<T>, где T - класс, в который будут мапиться результатами из базы. Так, вся работа которую мы делали руками, получая каждое значение каждой строки руками - Dapper делает за нас, и на выходе мы получаем IEnumerable<T>.

```
var students = sqlConnection.Query<Student>("SELECT * FROM Student").ToList();
// В случае единственного перечисления по коллекции students приведение к ToList()
// является избыточным и сделано только в учебных целях.
```

Для получения результата агрегатной функции в общем случае используется также Query<T>, где в T передается тип данных (int, double, float и т.д), а затем берется первая запись из полученной коллекции, так как такая выборка на стороне базы возвращает 1 строку с 1 столбцом.

```
var count = sqlConnection.Query<int>("SELECT COUNT(*) FROM Student")
                                        .FirstOrDefault();
```

В случае, когда мы работаем с параметрами, мы можем передавать напрямую экземпляр класса, или анонимный объект.

Так, например, при добавлении записи мы можем написать следующую запись:

```
sqlConnection.Execute("INSERT INTO Student VALUES (@Name, @Course, @BirthDate)",
        new Student
        {
            Name = "SomeParamName",
            Course = 2,
            BirthDate = new DateTime(2022, 04, 04)
        });
```

Dapper сам произведет необходимый маппинг по имени. Если типы не соответствует - будет выброшено исключение.

И пример использования анонимного объекта, если не хотим создавать какой-то класс:

```
sqlConnection.Execute("DELETE FROM Student WHERE Name = @name",
        new { name = "TestUserDapper" });
```

На этом основные возможности Dapper заканчиваются. Он прост в использовании, и как посмотрим далее, достаточно производителен.

EntityFramework

Необходимо наличие следующих пакетов:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools (Не обязательный)

Я даже не знаю с чего начать, данный фреймворк является самый настоящим монстром, и обладает огромным количеством возможностей. О некоторых из них мы поговорим в этой статье, однако если будет необходимость - то EF можно обсудить более детально в отдельной статье.

Вот несколько сильных сторон от EF:

1. Поддержка разных способов синхронизации (Code First, Database First)
2. Миграции
3. LINQ To Entities (и расширения напрямую из пакета EntityFrameworkCore)
4. AsNoTracking
5. CRUD операции

Начнем с самого начала - EF является достаточно большой системой, которая требует много подготовительной работы, но сполна награждает за неё. Так, например, создадим все классы для нашей базы данных (Student, Department, Coursework):

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

  public int Course { get; set; }

  [StringLength(90)]
  public string Name { get; set; }

  public DateTime BirthDate { get; set; }
}
```

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

  [StringLength(90)]
  public string Name { get; set; }
}

public class Coursework
{
  public int Id { get; set; }

  public int StudentId { get; set; }

  public int DepartmentId { get; set; }

  public DateTime DeliveryDate { get; set; }
}
```

После создания данных классов (что в целом достаточно легко), нам необходимо создать её один класс - который обычно называется - НазваниеБазыContext, так в данном случае это будет UniversityContext.

```
public class UniversityContext : DbContext
{
    public UniversityContext() { }
    public UniversityContext(DbContextOptions options) : base(options) { }

    public DbSet<Student> Student { get; set; }

    public DbSet<Department> Department { get; set; }

    public DbSet<Coursework> Coursework { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
  {
    var dbConfig = new DbConfiguration();
        optionsBuilder.UseSqlServer(dbConfig.GetConnectionString("connString"));
  }
}
```

Тут уже немного сложнее, поэтому пройдемся более детально по этому классу.

Наличие DbSet<T>. В целом, для простого понимания - DbSet представляет собой коллекцию (но не в памяти, а удаленную), которая представляет каждую отдельную таблицу. Так, например, DbSet<Student> Student - говорит о том, что у нас "есть" таблица со столбцами такими как поля в классе Student, и название у такой таблицы Student.

Также есть переопределение метода OnConfiguring(DbContextOptionsBuilder optionsBuilder). Это нужно для того, чтобы определить, с какой базой данный будет связан наш контекст. В целом там гораздо больше различных настроек, но на данный момент это основная.

Немного про конструкторы: в данном случае их 2, хотя для наших целей достаточно и одного. В общем случае при разработке веб-приложений и использования механизма DI нам будет достаточно второго конструктора, который принимает параметры. Однако в данном кейсе мы делаем все настройки в методе OnConfiguring, поэтому нам достаточно просто создавать контекст с пустым конструктором. Но не всё так просто.

Есть одна необходимость - это миграции. Чуть ниже мы обсудим что это такое, но для того, чтобы это механизм работал - нам нужно сделать одно из двух условий:

- Иметь конструктор без параметров
- Иметь класс, который реализует интерфейс IDesignTimeDbContextFactory<T>, где T - наш контекст. Вот пример реализации этого интерфейса:

```
public class UniversityContextFactory :
    IDesignTimeDbContextFactory<UniversityContext>
{
    public UniversityContext CreateDbContext(string[] args)
    {
        var dbConfig = new DbConfiguration();
        var optionsBuilder = new DbContextOptionsBuilder<UniversityContext>();
            optionsBuilder.UseSqlServer(dbConfig.GetConnectionString("connString"));

        return new UniversityContext(optionsBuilder.Options);
    }
}
```

Миграции

Миграции - магическое слово EF и одна из самых сильных его сторон. Что такое миграции?

Это механизм, который позволяет нам создать некоторое подобие гита, только для базы данных. Фактически, внося изменения в какую-нибудь из моделей или контекст, вы “фиксируете” эти изменения и создаете миграцию. Она имеет два метода: Up и Down. Соответственно при помощи данных методов вы можете двигаться “вверх” или “вниз”. Немало важный плюс миграций - это то, что они не удаляют данные, когда накатываются на базу. Работа с миграциями всегда будет идти по циклу: внесли изменения в C# классы, создали миграцию, применили миграцию.

Для создания миграции нам нужно открыть Package Manager Console в Visual Studio и написать следующую команду:

```
Add-Migration <Название Миграции>
```

В результате выполнения этой команды появится 2 класса (только при первой миграции, потом будет 1 класс).

Один из классов - ModelSnapshot, которую является некоторым сборщиком миграций, и знает, в каком порядке они должны применяться.

Второй класс - это непосредственно наша миграция и имеет данный класс следующий вид:

```
public partial class Initial : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Coursework",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    StudentId = table.Column<int>(type: "int", nullable: false),
                    DepartmentId = table.Column<int>(type: "int", nullable: false),
                    DeliveryDate = table.Column<DateTime>(type: "datetime2",
                        nullable: false)
                },
                constraints: table =>
                {
```

```
                table.PrimaryKey("PK_Coursework", x => x.Id);
            });

        migrationBuilder.CreateTable(
            name: "Department",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(type: "nvarchar(90)",
                    maxLength: 90, nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Department", x => x.Id);
            });

        migrationBuilder.CreateTable(
            name: "Student",
            columns: table => new
            {
                Id = table.Column<int>(type: "int", nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Course = table.Column<int>(type: "int", nullable: false),
                Name = table.Column<string>(type: "nvarchar(90)",
                    maxLength: 90, nullable: false),
                BirthDate = table.Column<DateTime>(type: "datetime2",
                    nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Student", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Coursework");

        migrationBuilder.DropTable(
            name: "Department");

        migrationBuilder.DropTable(
            name: "Student");
    }
}
```

Как и говорилось выше - миграция имеет два метода, один применяется, если мы применяем нашу миграцию (то есть идем "вверх"), а второй применяется, когда мы отменяем миграцию (то есть идем "вниз"). Так, в нашем примере мы создали миграцию с именем Initial, и при применении миграции у нас появятся три таблицы, а при отмене миграции - удалятся 3 таблицы.

Можете попробовать удалить все таблицы из базы (даже саму базу), и ввести в Package Manager Console следующую команду:

```
Update-Database
```

После этого можете удивляться результату.

На самом деле, Миграции являются частью принципа Code First, в котором мы пишем код, а потом говорим, что EF применил этот код для базы. Однако существует и второй принцип - Database First,

который, по названию, означает, что сначала мы создаем базу, а потом только C# код. На самом деле реализуется это достаточно простым механизмом, что называется одной командой, которая имеет следующий вид, и вводится в всё тот же Package Manager Console:

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=University;Trusted_Connection=True;'
```

И так, как и я сказал, это один из самых важных и популярных механизмов, который предоставляет EF, но не единственный.

LINQ To Entities

Еще одна классная возможность EF - поддержка построения запросов без знания SQL, так как EF сам делает все преобразования C# кода в SQL.

Например, если вернёмся к тому SQL скрипту, который был в самом начале:

```
SELECT Student.Name, Student.BirthDate, COUNT(*) AS [Количество курсовых]
FROM Student
JOIN Coursework ON Student.Id = Coursework.StudentId
WHERE Student.Name LIKE 'Vladzimir%'
GROUP BY Student.Name, Student.BirthDate
HAVING COUNT(*) > 1
ORDER BY BirthDate DESC
```

На языке LINQ это будет выглядеть так:

```
var sqlQueryToLinq = dbContext.Student
  .Where(student => student.Name.Contains("Vladzimir"))
  .Join(dbContext.Coursework,
        student => student.Id,
        coursework => coursework.StudentId,
        (student, coursework) => student)
  .GroupBy(student => new { student.Name, student.BirthDate })
  .Where(grouped => grouped.Count() > 1)
  .OrderByDescending(grouped => grouped.Key.BirthDate)
  .Select(grouped =>
          new { grouped.Key.Name, grouped.Key.BirthDate, Count = grouped.Count() })
  .ToList();
```

Выглядит сложно для первого понимания, но в целом любая часть соответствует предложениям из SQL. А трактовка значка '=>' несколько сложная, если не понять что такое делегаты. Но если пояснение, то его можно посмотреть тут:

- Пояснение по поводу LINQ запроса

AsNoTracking

Еще одна возможность EntityFramework, о которой немного подробнее надо рассказать:

Внутри EntityFramework содержится много разных способ отслеживания изменений, кеширования и т.д.

Так вот один из явных инструментов является Change Tracker, который создает некоторую связь между данными из таблиц и объектами C#. Также, если нам это механизм кажется излишним, то

мы его может отключить, при помощи метода AsNoTracking().

Рассмотрим такой пример для большей наглядности этого механизма:

```
var studentVladzimir = dbContext.Student
        .Where(student => student.Name.Contains("Vladzimir"))
        .FirstOrDefault();

studentVladzimir.Course = 999;
dbContext.SaveChanges();

var studentVladzimirWithoutTracking = dbContext.Student
      .Where(student => student.Name.Contains("Vladzimir"))
      .AsNoTracking()
      .FirstOrDefault();

studentVladzimirWithoutTracking.Course = 777;
dbContext.SaveChanges();
```

Что мы ожидаем увидеть в таблице, если найдем такую запись? Результат будет 999.

Почему так? Потому что когда мы получаем данные при помощи нашего контекста, то он не создаем полностью независимый объект, а ставит между ним и записью из базы связь, которая будет обновлять запись в базе при применении SaveChanges().

В случае применения AsNoTracking() - контекст не будет устанавливать связь, а просто создать новый объект, как будто сделали просто new Student().

CRUD-операции

Напоследок, небольшой бонус, который дает EF - готовые реализации для добавления, удаления и изменения записи.

Так, например, для добавления используется следующий код:

```
dbContext.Student.Add(
  new Student()
    {
    Name = "SomeStudent For EF Test",
    Course = 3,
    BirthDate = new DateTime(2022, 4, 7)
    });
```

Аналогичный код используется и для обновления (Update), и удаления (Remove).

Также, один из важных плюсов, поддержка - AddRange, UpdateRange, RemoveRange, они являются возможность делать bulk-операции.

Немного тестов и результаты

В целом, мы познакомились с основными провайдерами баз данных. На самом деле на проектах можно встретить каждый из этих провайдеров, а иногда и несколько сразу.

Я решил написать парочку тестов, в частности для получения данных, так как в целом любое изменение данных во всех провайдерах примерно одинаково.

Так, например, вот такие результаты показали провайдеры на 20000 данных.

Method	Time	Allocated Memory
GetAll_EF_WithTracking	51.79 ms	22 MB
GetAll_EF_WithNoTracking	16.52 ms	6 MB
GetAll_ADO	7.351 ms	2 MB
GetAll_Dapper	13.07 ms	4 MB

Как видим, EF показываем не самые лучшие результаты, и по большей части это связано с тем, что мы всячески блокируем в тестах попытки кеша каких либо результатов. В реальных условиях ситуация будет такая, что разница между Dapper и EF может быть до 5%. Однако тяжеловесность EF показывает то, как много памяти он кушает.

В большинстве своем - получение такого числа данных - очень редкий кейс, и в среднем надо вытягивать от 1 до 100 записей за раз, и на таких данных разница во времени будет минимальна между ними.

Вот некоторые выводы к которым можно прийти, прочитав эту статью:

1. .NET предоставляет различные механизмы работы, и выбор достаточно внушительный, каждый из представленных механизмов отличается от двух других.
2. Знание SQL необходимо, но не обязательно. Влияние EntityFramework с годами увеличивается, как и его производительность.
3. Мы всегда смотрим не только на производительность, но и на то насколько быстро мы можем написать наш код. В таких случаях зачастую выбор остается между Dapper и EntityFramework.

На этом наверное всё, огромное спасибо за прочтение этой статьи!

Как и говорилось в начале, вы можете попробовать сделать всё сами, при помощи github-репозитория, который будет в открытом доступе.

Теги: ado.net, entity, dapper, benchmark

Хабы: .NET, C#, SQL

5 Карма 0 Рейтинг

Vladzimir Liashko @vliashko

.NET Software Engineer