# Практическая работа №6.
# Работа с Entity Framework

**Цель работы**: научиться подключаться к базе данных с использованием Entity Framework.

## Теоретический материал

Entity Framework – это библиотека для языка программирования C#, которая позволяет подключаться к базе данных, используя её объектную модель. Каждая таблица базы данных при этом представляется как класс, колонки этой таблицы – как свойства этого класса, строки таблицы – как объекты этого класса. При этом, как правило, эти классы не пишутся вручную, а генерируются по базе данных с использованием специализированных инструментов (INFORMATION_SCHEMA).

Свойства – это особая возможность языка программирования C#, прежде всего, предназначенная для разработки пользовательского интерфейса. Свойство – это пара методов с семантикой получения и присвоения значения. Первый метод называется **`get`**, второй – **`set`**. Метод **`get`** не имеет параметров и возвращает значение некоторого типа. Метод **`set`** не имеет возвращаемого значения (**`void`**) и имеет единственный параметр **`value`** того же типа, что и тип возвращаемого значения метода **`get`**. Фактически, для кода, работающего с экземпляром класса, свойство выглядит как поле. Но поскольку свойство представляет собой пару методов, там может быть дополнительная логика, которая не сводится к тому, чтобы получить значения поля и изменить его. При разработке пользовательского интерфейса, как правило, эта логика состоит в том, чтобы инициировать событие **`PropertyChanged`** после изменения значения свойства, если был вызван метод **`set`**, и значение параметра value отличается от значения поля, которое возвращает метод **`get`**. Для того чтобы использовать это событие, в классе нужно реализовать интерфейс **`INotifyPropertyChanged`**.

Чаще всего в начале разработки класса используются автореализуемые свойства. При объявлении таких свойств просто пишется **`{ get; set; }`**. Компилятор сам создаст поле для хранения значения такого свойства, метод **`get`** будет возвращает значение этого поля, а метод **`set`** будет присваивать ему значение параметра **`value`**. Автореализуемое свойство создать почти так же просто, как поле, но это позволяет заложить совместимость с будущими версиями программы на случай, если какая-нибудь дополнительная логика потребуется в методах **`get`** и **`set`**. Дело в том, что в метаданных программы свойства и поля выглядят по-разному. Поэтому если программа использует рефлексию для работы с метаданными, замена поля свойством может привести к ошибкам в работе программы: код, ориентированный на работу с метаданными полей, не будет работать с метаданными свойств.

Классы, генерируемые для каждой таблицы в Entity Framework, как правило, достаточно просты. Они содержат только свойства и, если требуется интеграция с графическим интерфейсом, реализацию интерфейса **INotifyPropertyChanged**. Более того, как правило, интеграция с пользовательским интерфейсом на уровне классов, моделирующих таблицы в базе данных, используется только в очень простых приложениях; в более сложных приложениях это может привести к запутыванию кода и поэтому не используется. Но в любом случае, все эти классы не содержат кода для подключения к базе данных – для этого используется отдельный класс, который называется **DbContext**. Для подключения к конкретной базе данных необходимо создать наследника этого класса, свойства которого имеют тип **DbSet<MyTable>**, где **MyTable** – это класс, моделирующий определённую таблицу. Как правило, этот класс тоже не пишется вручную и автоматически генерируется по базе данных.

Выполнение действий CRUD делается следующим образом. Для создания новых записей в классе **DbSet** есть метод **Add**, в качестве параметра принимающий объект, тип которого указывается через параметр обобщённого типа. Для чтения данных нужно обойти содержимое объекта типа **DbSet** (свойства контекста) с помощью цикла **foreach** либо используя LINQ. Для обновления данных необходимо получить объект на чтение и изменить значения некоторых его свойств. Для удаления объекта необходимо вызвать метод **Remove** и передать ему объект, тип которого указывается через параметр обобщённого типа. После внесения изменений в контекст через этим методы необходимо синхронизировать его с базой данных, вызвав у контекста метод **SaveChanges**.

Если объект, моделирующий строку конкретной таблицы, не был получен путём обхода содержимого свойства контекста, но требуется его обновить или удалить, для присоединения его к контексту можно использовать метод **DbSet.Attach**. Вызов этого метода приравнивает переданный ему объект к тем, которые были сформированы путём запроса к базе данных. Для поиска по ключу предусмотрен метод **DbSet.Find**, позволяющий передать значения колонок ключа в качестве параметра. При этом следует помнить, что эти значения не контролируются компилятором C#, и использовать его нужно с осторожностью.

Для подключения к PostgreSQL необходимо создать объект класса **DbContextOptionsBuilder<MyDbContext>**, где **MyDbContext** – это класс контекста для подключения к конкретной базе данных и вызвать у него метод `UseNpgsql`, передав в качестве параметра строку подключения. Она состоит из набора пар «ключ=значение», разделённых точкой с запятой. Ключи для подключения к базе данных PostgreSQL следующие: **Host**, **Port**, **Data base**, **Username**, **Password**. Значения должны быть те же самые, что использовались для подключения к базе данных в pgAdmin.

# Практическая часть

С помощью Visual Studio Installer установите необходимые компоненты.

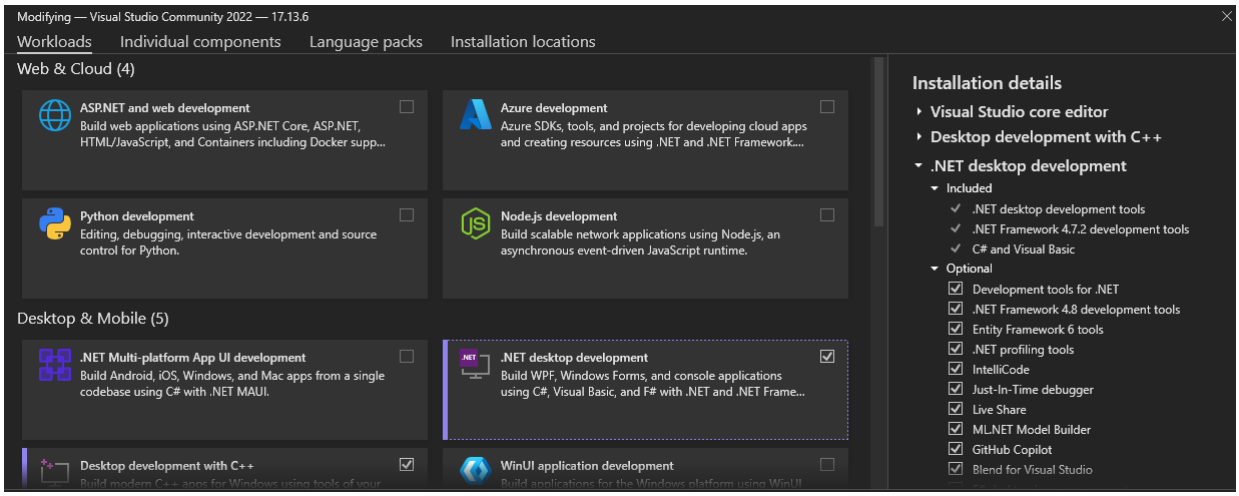


Рисунок 1. Visual Studio Installer

Заходим и строке разработчика кликаем на Tools → Get Tools and Features/ Ищем C# и оставляем обязательно. Необходима рабочая нагрузка .NET desktop development.

Создаём проект консольного приложения на C#. Для поддержки новых версий Entity Framework нужно использовать вариант Microsoft, а не .NET Framework.

В файле кодогенератора замените значения переменных **`namespace_name`** и **`schema_name`** на имя, заданное при создании проекта (по умолчанию это будет что-то наподобие ConsoleApplication1, но желательно при создании проекта задать более осмысленное название), и имя схемы, в которой находятся объекты вашей базы данных. После этого выполните скрипт кодогенратора. В результате получится результат выполнения запроса, напоминающий рисунок 2.

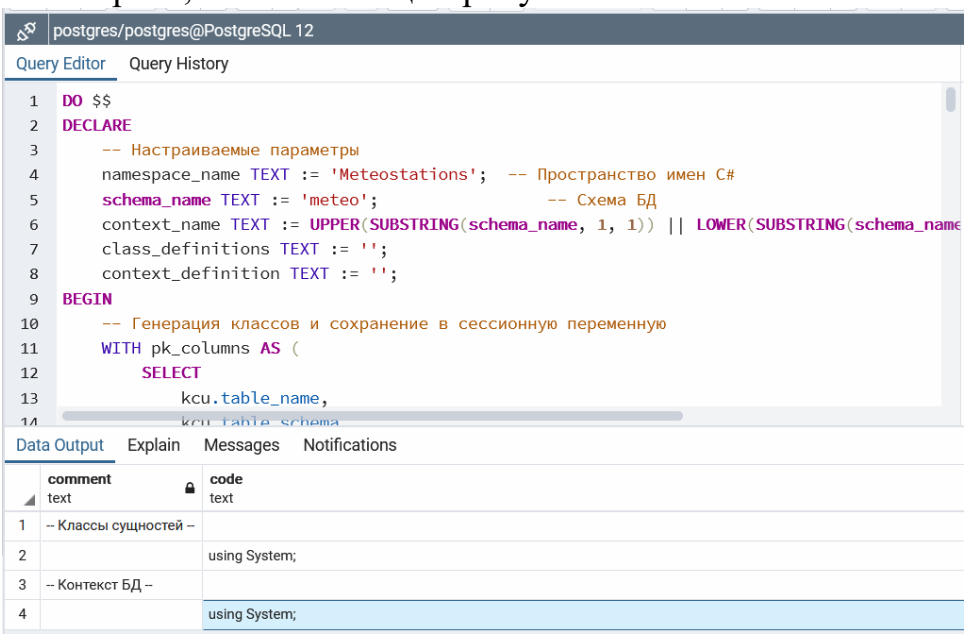


Рисунок 2. Кодогенератор.

В результате выполнения запроса в колонке code будут классы с объектной моделью таблиц базы данных и класс контекста. В Visual Studio откройте Обозреватель решений и в созданном проекте создайте два файла на C#: для контекста и для объектной модели таблиц. Скопируйте туда код, сгенерированный кодогенератором, из pgAdmin.

В классе Program в методе Main (или, если файл Program.cs пустой – просто в этом файле) создайте объект **DbContextOptionsBuilder**, настройте подключение к базе данных например, так:

```
var builder
    = new DbContextOptionsBuilder<PublicDbContext>();

builder.UseNpgsql("Host=localhost;Port=5432;" +
"Database=test_db;Username=postgres;Password=admin");
```

В блоке using создайте контекст:

```
using (var context = new MyDbContext(builder.Options))
{
}
```

Внутри фигурных скобок разместите обращения к таблицам:

```
context.MyTable.Find(myId); // Поиск по ключу
context.SaveChanges();  // Сохранение данных
```

Для того чтобы проект собрался, необходимо установить нужные библиотеки. Это делается с помощью NuGet, которую можно вызвать через контекстное меню на ссылках проекта в Обозревателе решений:

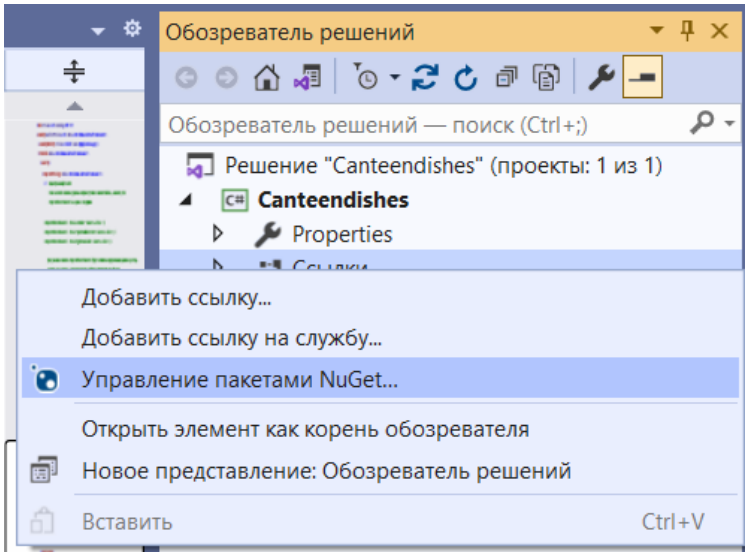


Рисунок 3. Вызов NuGet

После этого в NuGet необходимо добавить в проект EntityFramework и провайдер NpgSql для Entity Framework (рисунок 4).

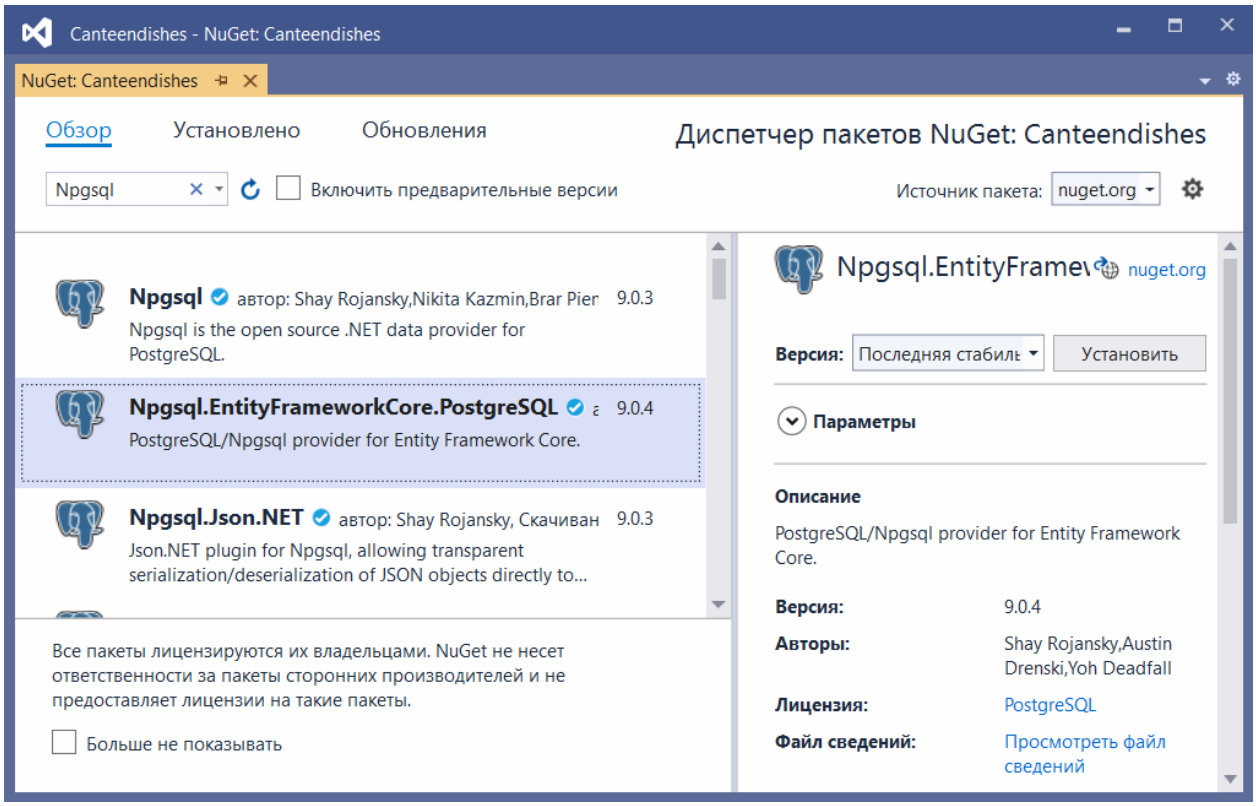


Рисунок 4. Установка провайдера Npgsql для EntityFramework.

Убедитесь, что проект собирается, запускается и подключается к базе данных.

## Задание

1. Настройте проект Visual Studio для работы консольного приложения с PostgreSQL через Entity Framework
2. С помощью кодогенератора сгенерируйте код объектной модели базы данных и контекста для базы данных по своему курсовому проекту по курсу «Управление данными».
3. Выберите таблицу с автоинкрементом, вставьте в неё строку с использованием метода **`Add`**, выведите на консоль сгенерированный идентификатор (**`Console.WriteLine`**).
4. Выведите всё содержимое таблицы, в которую была вставлена строка. Желательно для перебора колонок таблицы использовать рефлексию.
5. Получить какую-нибудь другую запись по ключу с помощью метода **`Find`** и изменить её поля, затем сохраните изменения.
6. Удалите запись из базы данных с помощью метода **`Remove`**. Это должна быть не та запись, которая была изменена.
7. Выведите изменённое содержимое таблицы.

## Оформление отчёта

Отчёт должен иметь титульный лист с указанием ФИО и группы студента, а так же темы практической работы. После этого должен быть скриншот NuGet с установленными библиотеками для работы с PostgreSQL и Entity Framework в консольном приложении. После этого необходимо привести код сгенерированных классов. Если для их генерации использовался Scaffold, необходимо привести команды, которые использовались для генерации кода. Дальше необходимо указать, какая таблица была выбрана для экспериментов. Должен быть приведён код на C#, который вставляет строку в эту таблицу, сохраняет данные, а потом выводит на консоль сгенерированный идентификатор. После кода нужно показать фрагмент скриншота, где видно, что это корректно отработало. Затем необходимо привести код вывода всего содержимого таблицы: сначала заголовки колонок (получить все свойства через рефлексию), затем все строки (значения каждого свойства получать через рефлексию). После кода нужно показать фрагмент скриншота, где видно, что это корректно отработало. Затем необходимо привести фрагмент кода, который получает запись по ключу, меняет значения некоторых её свойств и сохраняет изменения. После этого нужен скриншот, выводящий изменённое содержимое таблицы. Затем необходимо привести код на C#, который удаляет запись из таблицы и сохраняет данные. После этого снова нужен скриншот, выводящий изменённое содержимое таблицы.