
# Технология доступа к базам данных

# ADO.NET

# Урок №9

# Код сначала (Code First)

## Содержание

# Код сначала (Code First)

Технология «код сначала» появилась в версии Entity Framework 4.1. При использовании этого подхода разработчик концентрируется на создании классов и связей между ними. Причем работа эта выполняется в коде. Затем на основании этих классов Entity Framework создает новую БД или же отображает созданные классы на существующую БД. Давайте разработаем еще одно приложение, в котором рассмотрим работу с Entity Framework по технологии «код сначала».

### Создание классов приложения

Создайте в Visual Studio новое консольное приложение с именем LibraryCodeFirst. Затем добавьте в состав приложения следующие новые классы.

```
public class Author
{
    public Author()
    {
    }

    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public partial class Publisher
{
```

```
    public Publisher()
    {
    }
    public int Id { get; set; }
    public string PublisherName { get; set; }
    public string Address { get; set; }
}

public class Book
    {
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public int Pages { get; set; }
    public int Price { get; set; }
    public int PublisherId { get; set; }
}
```

Вы, конечно же, узнали классы нашей библиотеки. Обратите внимание на то, что в этих классах не созданы свойства навигации. Это наши «пользовательские классы», необходимые для реализации логики разрабатываемого приложения. Кроме этих классов мы также должны создать класс, содержащий в себе функциональность контекста БД. Этот класс может выглядеть так:

```
public class LibraryContext : DbContext
{
    public LibraryContext()
           : base()
    {

    }
   public DbSet<Author> Authors { get; set; }
```

```
    public DbSet<Publisher> Publishers { get; set; }
    public DbSet<Book> Books { get; set; }
}
```

Для создания этого класса необходимо добавить в проекте ссылку на пространство имен System.Data.Entity, в котором описаны классы DbContext и DbSet<>. А в файле, где описан класс, надо добавить инструкцию:

```
using System.Data.Entity;
```

Выполните перестройку приложения, чтобы добавленные классы были скомпилированы и включены в состав приложения.

Кроме этого, в приложение должен быть интегрирован Entity Framework. Вы уже видели, как добавляется Entity Framework с помощью NuGet. Существует еще один способ работы с NuGet — С помощью консоли. Давайте познакомимся с этим новым способом.

> *Это можно сделать так: выбираем меню Сервис (Tools) — Диспетчер пакетов NuGet (NuGet package manager) — Консоль диспетчера пакетов (Package manager console). Затем в нижнем окне консоли, после подсказки PM>, можно вводить и выполнять необходимые команды. Эта консоль скоро понадобится на в работе с этим приложением.*

Теперь давайте посмотрим, что мы получили в результате проделанной работы. Добавьте в метод Main() уже привычный для вас код, который позволит увидеть результаты проделанной работы:

```
Author author = new Author { FirstName = "Isaac",
  LastName = "Azimov" };
using (LibraryContext db = new LibraryContext())
{
    db.Authors.Add(author);
    db.SaveChanges();
    var ac = db.Authors.ToList();

    foreach (var a in ac)
    {
        Console.WriteLine(a.FirstName +" "+a.
          LastName);
    }
}
```

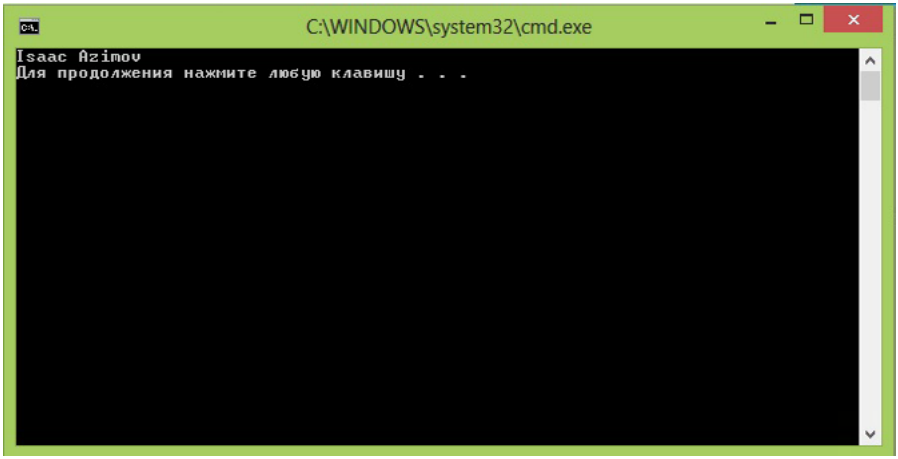


Рис. 1. Результат тестирования приложения

## Создание базы данных

После того как вы получили такое консольное окно, нам есть о чем поговорить. Вы убедились в том, что наши классы работают, мы можем обращаться к БД, через контекст LibraryContext и выполнять в базе дан-

ных запросы. Но ведь мы с вами не создавали никакой базы данных! Тем не менее, БД существует, и она была создана Entity Framework. Давайте разберем, каким образом фреймворк создал БД, и как мы можем управлять процессом создания БД.

Entity Framework создает базу данных, основываясь на параметре, передаваемом конструктору базового класса контекста БД, в нашем случае — конструктору родителя класса LibraryContext. Посмотрите на определение этого класса. В нашем случае мы не передаем конструктору родительского класса никаких параметров. Что происходит при этом?

Если конструктору родительского класса контекста БД не передавать параметры, Entity Framework создаст базу данных с именем ПространствоИменПроекта.ИмяКлассаКонтекста. В нашем случае имя созданной БД будет LibraryCodeFirst.LibraryContext. Эта база данных будет по умолчанию создана на LocalDB и в ней будут созданы таблицы, соответствующие нашим классам Author, Publisher и Book.

Вы уже поняли, что создание БД зависит от определения конструктора класса контекста БД. Давайте разберем, какие еще варианты этого конструктора мы можем использовать, и как это влияет на создаваемую БД.

Если конструктору базового класса передать строковый параметр, то этот параметр будет интерпретирован как имя создаваемой БД. Например, если конструктор класса LibraryContext будет выглядеть таким образом:

```
public LibraryContext()
        : base("LibraryDb")
{
}
```

то имя созданной БД будет LibraryDb.

Но существует еще один способ интерпретации строкового параметра конструктора базового класса. Если строковое значение, переданное этому конструктору, совпадет с именем строки подключения в файле конфигурации приложения App.config (или Web.config для веб-приложения), то будет создана база данных соответственно значениям, указанным в строке подключения. Например, если конструктор класса LibraryContext будет выглядеть таким образом:

```
public LibraryContext()
        : base("MyConnString")
{
}
```

а в файле App.config будет указана такая строка подключения:

```
<connectionStrings>
    <add name="MyConnString"
    connectionString="Data Source=(LocalDb)\
    v11.0;Integrated Security=SSPI;
    Initial Catalog=MyLibraryDb;"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

а имя созданной БД будет, конечно же, MyLibraryDb, а создана эта БД будет на сервере на LocalDB, потому что так

указано в строке подключения (Data Source = (LocalDb)\v11.0). Сам файл с БД, имя которого будет в этом случае MyLibraryDb.mdf (потому, что такое значение задано в Initial Catalog), будет располагаться в корневой папке вашего профиля на компьютере. Вы прекрасно понимаете, что в строке подключения можете указывать любые допустимые значения. Например, для такой строки подключения:

```
<configuration>
    <connectionStrings>
        <add name="MyConnString"
        connectionString="Data Source=PC-IDEA\
         SQLEXPRESS;
        Initial Catalog=LibraryCodeFirst; Integrated
         Security=true"
        providerName="System.Data.SqlClient"/>
    </connectionStrings>
</configuration>
```

имя созданной БД будет, конечно же LibraryCodeFirst, потому, что такое значение указано в параметре Initial Catalog=LibraryCodeFirst. А создана эта БД будет на сервере SQLEXPRESS, потому, что так указано в строке подключения в параметре Data Source=PC-IDEA\SQLEXPRESS, где PC-IDEA — имя компьютера.

### Инициализация базы данных в технологии «код сначала»

В нашем приложении база данных была создана фреймворком при первом запуске. Возникает закономерный вопрос: что произойдет с БД при втором, третьем запуске приложения? И как можно изменить уже созданную БД,

при необходимости изменения созданных классов? Давайте разберем эти и другие вопросы. Процессом создания БД управляют специальные классы инициализаторы.

Существует четыре способа создания БД при подходе «код сначала». Первое правило по умолчанию работает таким образом. Описывается это правило классом **CreateDatabaseIfNotExists**. Если база данных, заданная в конструкторе базового класса контекста БД, еще не существует, она будет создана в соответствии с правилами, описанными выше. Однако вы должны учитывать такой момент. Если в дальнейшем вы измените модель (т.е. определение своих классов в приложении) и запустите приложение при уже существующей БД, то получите исключительную ситуацию.

Второе правило описывается классом **DropCreate DatabaseIfModelChanges** и работает так. Если БД еще не существует, она создается при запуске приложения. А если в дальнейшем вы измените созданные классы, из которых состоит модель, и запустите приложение, то существующая БД будет удалена, а вместо нее будет создана новая БД, соответствующая измененной модели. Понятно, что существующие в БД данные в этом случае будут потеряны.

Третий способ инициализации описывается классом **DropCreateDatabaseAlways** и работает таким образом, что новая БД создается при всяком запуске приложения, независимо от того, существует уже БД или нет. Это бывает полезным в тех случаях, когда вам надо иметь свежую БД при всяком запуске приложения.

Четвертое правило создания БД является альтернативой предыдущим трем. Если ни одно из трех указанных правил инициализации БД вас не устраивает, вы можете создать свое собственное правило инициализации.

Рассмотрим, каким образом можно выбрать в приложении одно из четырех указанных правил инициализации. Для этого необходимо использовать свойство Database в конструкторе контекста БД. В нашем приложении конструктор контекста БД оставлен пустым. Но если вы хотите использовать ту или иную стратегию инициализации, вы должны выполнить это именно в конструкторе контекста БД.

Хотите выбрать первую стратегию инициализации — модифицируйте определение контекста БД таким образом:

```
public class LibraryContext : DbContext
{
    public LibraryContext()
            : base("LibraryDb")
    {
         Database.SetInitializer<LibraryContext>(new
         CreateDatabaseIfNotExists<LibraryContext>());
    }
    public DbSet<Author> Authors { get; set; }
    //public DbSet<Publisher> Publishers { get; set; }
}
```

Как вы видите, необходимо вызвать от имени свойства Database generic метод SetInitializer<>() и передать ему в качестве параметра объект типа, соответствующий выбранной стратегии инициализации. Используя такой подход, вы можете применить в методе SetInitializer() лю-

бой из трех классов инициализаторов и соответственно, получить любой из трех способов создания БД. Здесь все просто. Теперь давайте рассмотрим, как создать свой собственный инициализатор.

Для этого надо создать свой класс инициализатор, производный от какого-либо класса инициализатора: CreateDatabaseIfNotExists, DropCreateDatabaseIfModelChanges или DropCreateDatabaseAlways. Все классы инициализаторы, в свою очередь, являются производными от интерфейса IDatabaseInitializer<>. В этом интерфейсе объявлен метод Seed(), который и надо реализовать в собственном классе инициализаторе. Такой класс может выглядеть, например, так. Ту логику инициализации БД, которую вы хотите реализовать, надо описать в методе Seed().

```
public class MyInitializer :
    DropCreateDatabaseAlways< LibraryContext>
{
    protected override void Seed(LibraryContext context)
    {
        base.Seed(context);
        //здесь надо реализовать необходимые действия
        //по инициализации БД
    }
}
```

Давайте реализуем какой-либо конкретный вариант метода Seed(). Например, мы хотим, чтобы всякий раз при создании наша база данных уже содержала в своих таблицах определенную информацию. Назовем это предварительным заполнением БД. Помните, в первом

приложении по технологии «база данных сначала» у нас был метод Init(), заполняющий БД? Что-то подобное мы реализуем сейчас в методе Seed().

```
public class MyInitializer :
    DropCreateDatabaseAlways<LibraryContext>
{
    protected override void Seed(LibraryContext context)
    {
        //здесь надо реализовать необходимые действия
        //по инициализации БД
        IList<Author> authors = new List<Author>();
        authors.Add(new Author { FirstName= "Clifford",
          LastName= "Simak" });
        authors.Add(new Author { FirstName= "Ray",
          LastName= "Bradbury" });
        authors.Add(new Author { FirstName= "Harry",
          LastName= "Harrison" });
        context.Authors.AddRange(authors);
        base.Seed(context);
    }
}
```

Если вы создадите в своем приложении такой класс, то всякий раз при создании БД в таблицу Author будут заноситься приведенные три строки, и вы будет получать уже частично заполненную таблицу. Конечно же, это только иллюстрация. Но теперь вы можете реализовать и более сложные варианты инициализации БД.

Иногда возникает необходимость отключить инициализацию БД в приложении — например, чтобы не терять уже существующие данные в таблицах. Отключить инициализацию можно с помощью метода SetInitilizer(), вызвав этот метод таким образом:

```
Database.SetInitializer<LibraryContext>(null);
```

Еще один способ отключения инициализации состоит в использовании файла App.config, в котором надо добавить такой элемент:

```
<appSettings>
    <add key="DatabaseInitializerForType
      LibraryCodeFirst.LibraryContext,
      LibraryCodeFirst value="Disabled" />
</appSettings>
```

### Миграция в технологии «код сначала»

Каждый из трех рассмотренных подходов инициализации БД приводит к созданию новой БД. При таком подходе имеющаяся в БД информация удаляется. А как быть в том случае, если мы изменили модель и хотим изменить БД, чтобы она соответствовала измененной модели? А еще при этом мы хотим сохранить в БД имеющуюся информацию? В этом случае нам надо воспользоваться технологией, называемой «миграция». Конечно же, кроме миграции также можно использовать для достижения этих целей собственный класс инициализатор, как это было описано выше. В этом классе надо самостоятельно реализовать все необходимые действия по изменению БД и переносу данных. Но миграция позволяет сделать все это более эффективно.

Рассмотрим миграцию подробно на примере нового проекта. Создадим новый консольный проект с именем

LibraryCodeFirstMigration. Установим в этом проекте с помощью NuGet последнюю версию Entity Framework. Добавим ссылку на пространство имен System.Data.Entity. Теперь добавим в проект следующие классы:

```
public class Author
{
    public Author()
    {
    }
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class LibraryContext : DbContext
{
    public LibraryContext()
            : base("DbMigration")
    {
        //Database.SetInitializer<LibraryContext>(new
        //DropCreateDatabaseAlways<LibraryContext>());
     }
     public DbSet<Author> Authors { get; set; }
}
```

Мы намеренно не добавляем изначально класс Publisher, чтобы показать его добавление с помощью миграции. Обратите внимание на закомментированную строку в конструкторе контекста БД. Чуть позже мы посмотрим, как наличие или отсутствие этой строки влияет на поведение нашего приложения. Чтобы видеть результат выполнения приложения, приведем метод Main() к такому виду:

```
static void Main(string[] args)
{
    Author author = new Author { FirstName = "Isaac",
     LastName = "Azimov" };
    using (LibraryContext db = new LibraryContext())
    {
        db.Authors.Add(author);
        db.SaveChanges();
        var ac = db.Authors.ToList();
        foreach (var a in ac)
        {
            Console.WriteLine(a.FirstName + " " +
              a.LastName);
        }
    }
}
```

Запустите наше приложение в первый раз. Вы увидите в консольном окне строку «Isaac Azimov», как и следовало ожидать. Запустите наше приложение во второй раз. Теперь вы увидите в консольном окне эту строку, повторенную дважды. Наши методы не проверяют данные на совпадение при добавлении в БД и поэтому допускают дублирование. Что мы и увидели. Эти действия подтверждают, что БД создана. При втором запуске приложения мы имеем дело с той же БД. Если вы раскомментируете строку кода

```
//Database.SetInitializer<LibraryContext>(new
DropCreateDatabaseAlways<LibraryContext>());
```

в конструкторе класса LibraryContext и запустите приложение еще несколько раз, то увидите, что всякий раз в консоль-

ном окне будет одна единственная строка «Isaac Azimov». Вы, конечно же, понимаете — это происходит потому, что выбранный инициализатор DropCreateDatabaseAlways<> создает новую БД при каждом запуске. Не забудьте снова закомментировать строку инициализации, чтобы новая БД не создавалась при каждом запуске нашего приложения. Запустите приложение несколько раз, чтобы в БД в таблице Author было создано несколько строк.

Теперь займемся миграцией. Миграция появилась в Entity Framework начиная с версии 4.3. Главная задача миграции — изменять БД при изменении модели без потери данных. Миграция не требует от программиста написания какого-либо специфического кода. Вместо этого необходимо выполнять некоторые команды в Package Manger Console (консоли управления пакетами). Существует несколько важных команд, с которыми надо ознакомиться. Прежде всего, это команды Enable-Migrations, Add-Migration *MigrationName* и Update-Database.

Давайте активируем в нашем приложении механизм миграции. Для этого надо выполнить в консольном окне менеджера пакетов команду Enable-Migrations.

Чтобы активировать консольное окно менеджера пакетов, надо в Visual Studio выполнить следующие опции меню: Tools → NuGet Package Manager → Package Manager Console (Сервис — Диспетчер пакетов NuGet – Консоль диспетчера пакетов). По умолчанию консольное окно менеджера пакетов появится под рабочей областью Visual Studio, подсказка в этом окне имеет вид PM>.

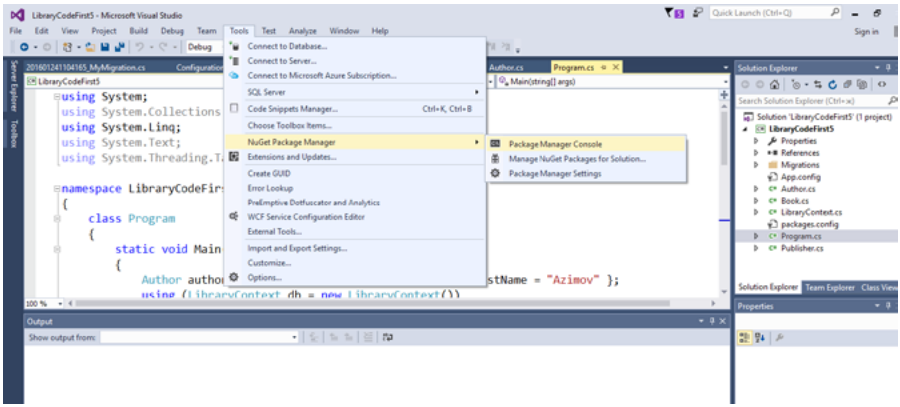


Рис. 2. Вызов консоли менеджера пакетов

Сначала надо ввести в это окно команду EnableMigrations, нажать Enter и подождать, пока команда выполнится. Окно будет выглядеть таким образом:

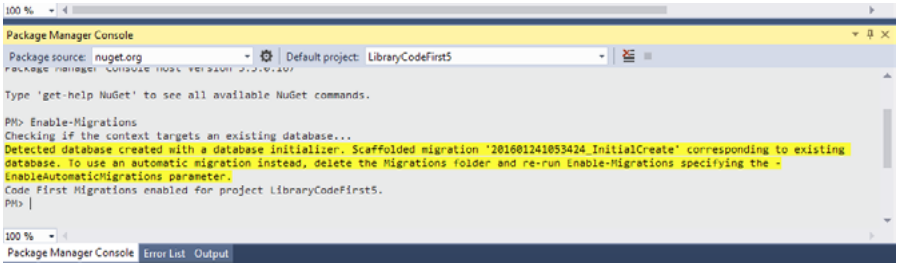


Рис. 3. Выполнение команды Enable-Migrations в консоли менеджера пакетов

Давайте разберем, к чему привело выполнение этой команды.

***Во-первых***, в нашем проекте была создана папка Migrations, в которой создан файл Configuration.cs, содержащий класс Configuration. В этом классе размещены настройки миграции.

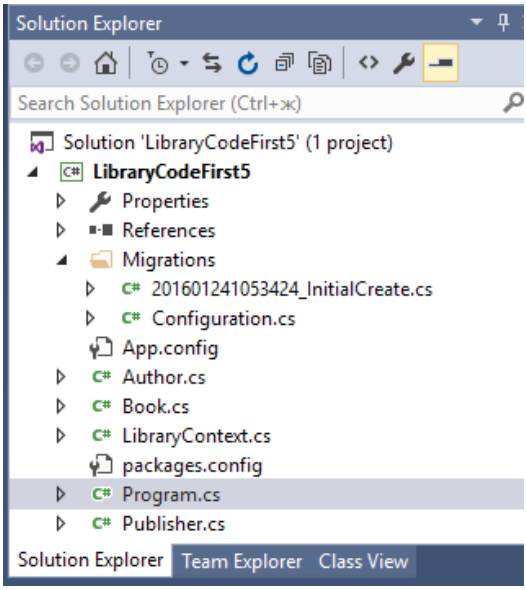


Рис. 4. Папка Migrations в составе проекта

В моем проекте этот класс выглядит таким образом:

```
internal sealed class Configuration :
  DbMigrationsConfiguration
  <LibraryCodeFirstMigration.LibraryContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }
    protected override void Seed(
        LibraryCodeFirstMigration.LibraryContext context)
        {
        //  This method will be called after
        //  migrating to the latest version.
        //  You can use the DbSet<T>.AddOrUpdate()
        //  helper extension method
        //  to avoid creating duplicate seed data. E.g.
```

```
            //    context.People.AddOrUpdate(
            //     p => p.FullName,
            //     new Person { FullName = "Andrew Peters" },
            //     new Person { FullName = "Brice Lambson" },
            //     new Person { FullName = "Rowan Miller" }
            //     );
        }
    }
```

***Во-вторых***, в папке Migrations создан еще один файл с именем, состоящим из уникальной строки (из значений даты и времени) и суффикса _InitialCreate. В моем приложении имя этого файла такое — 201601241053424_InitialCreate.cs. В этом файле описаны сущности, по которым создана БД:

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Authors",
            c => new
                {
                Id = c.Int(nullable: false,
                  identity: true),
                FirstName = c.String(),
                LastName = c.String(),
                })
            .PrimaryKey(t => t.Id);
    }

    public override void Down()
    {
        DropTable("dbo.Authors");
    }
}
```

***В-третьих***, в базе данных нашего приложения была добавлена таблица с именем _MigrationHistory. В этой таблице будут фиксироваться изменения БД при каждом выполнении миграции. Сейчас в этой таблице есть только одна строка, фиксирующая изначальное состояние БД.

Итак, к этому моменту мы выполнили в нашем приложении подготовительную работу, которая позволит нам в дальнейшем использовать миграции. Напомню еще раз, миграция — это способ изменения БД при изменении EDM (модели), при котором в БД сохраняются все данные.

Теперь давайте изменим нашу модель и посмотрим, как работает миграция. Добавим в состав модели еще один класс Publisher:

```
public class Publisher
{
    public Publisher()
    {
    }

    public int Id { get; set; }
    public string PublisherName { get; set; }
    public string Address { get; set; }
}
```

Кроме этого, надо убрать комментарий у строки свойства

```
//public DbSet<Publisher> Publishers { get; set; }
```

в классе LibraryContext. Вы понимаете, что модель изменилась, и наша БД уже не соответствует ей. Синхрони-

зируем существующую БД нашего проекта с измененной моделью. И сделаем это с помощью миграции.

Надо снова перейти в консольное окно менеджера пакетов, ввести и выполнить следующую команду — Add-Migration. После указания самой команды надо через пробел ввести имя добавляемой миграции. Давайте используем имя MyMigration. Полностью наша команда должна выглядеть так:

```
Add-Migration MyMigration
```

После успешного выполнения этой команды окно будет выглядеть таким образом:

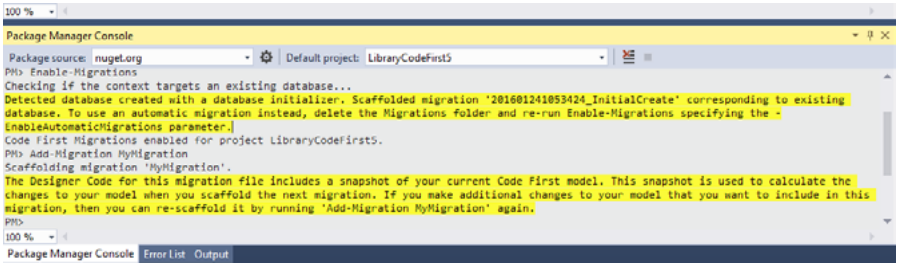


Рис. 5. Выполнение команды Add Migration
в консоли менеджера пакетов

3. В папке Migrations создан еще один файл с именем, состоящим из уникальной строки и суффикса, совпадающего с придуманным вами именем миграции — 201601241104165_MyMigration.cs. В этом файле перечислены сущности, которые будут добавлены при изменении БД. В моем проекте этот файл выглядит таким образом:

```
public partial class MyMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
              "dbo.Publishers",
              c => new
                  {
                  Id = c.Int(nullable: false,
                    identity: true),
                  PublisherName = c.String(),
                  Address = c.String(),
              })
        .PrimaryKey(t => t.Id);

    }

    public override void Down()
    {
        DropTable("dbo.Publishers");
    }
}
```

Следует отметить, что после выполнения команды Add-Migration наша БД еще не изменилась. Просто был подготовлен сценарий изменения в файле 201408161938363_MyMigration.cs. Чтобы изменить БД согласно этому сценарию, надо выполнить третью команду Update-Database (рис. 6).

И после выполнения команды Update-Database в нашей БД появится таблица Publisher. В таблице _MigrationHistory появится строка, в которой будет содержаться информация о выполненном изменении БД. А если вы заглянете в таблицу Author, то увидите, что данные, занесенные

туда во время предыдущих выполнений приложения, сохранились.

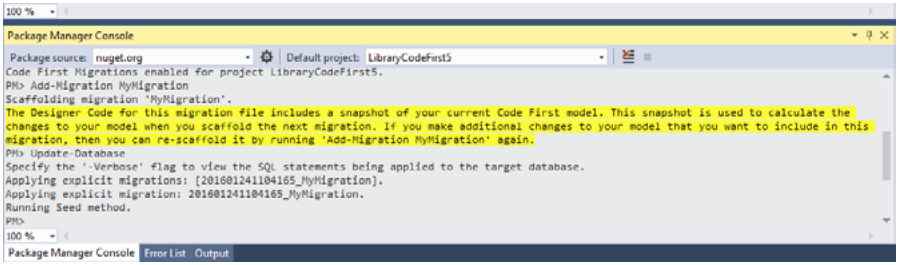


Рис. 6. Выполнение команды Update-Database в консоли менеджера пакетов

Помните, после первых запусков нашего приложения в БД должно было сохраниться несколько строк «Isaac Azimov»? Если вы не забыли убрать режим создания новой БД при каждом запуске приложения, то запустив приложение сейчас, вы увидите существовавшие ранее в БД данные плюс новые строки. Это значит, что миграция была успешно выполнена. Теперь при каждом последующем изменении модели вам надо будет выполнять только две миграционные команды: Add-Migration MigrationsName и Update-Database.

## Конфигурирование классов модели (DataAnnotation)

Мы с вами уже говорили о соглашениях Entity Framework. Соглашения позволяют фреймворку понять, что, например, свойство с именем Id является первичным ключом, а свойство с именем ИмяСвязаннойСущности. Id — внешним ключом. Эта информация критически

важна для построения модели. Но вполне реальна ситуация, когда в нашем распоряжении есть модель, в которой подобные договоренности не соблюдены. В конце концов, соглашения не являются обязательными, и мы можем называть свои классы и их свойства совершенно произвольным образом.

Как же, в таком случае, построить БД на основе кода, не соблюдающего соглашения Entity Framework? Только не предлагайте вариант переименовывания соответствующих сущностей ☺

Переименовывать ничего не надо, поскольку у нас есть механизм, позволяющий решить эту задачу. Этот механизм называется DataAnnotation (аннотация данных) и базируется на применении атрибутов, определенных в пространстве имен System.ComponentModel.DataAnnotations. Давайте рассмотрим отвлеченный пример применения основных из этих атрибутов и охарактеризуем каждый из них:

```
[Table("EmployeeInfo")]
public class Employee
{
    public Employee() { }
    [Key]
    public int EmpId { get; set; }
    [Required(ErrorMessage="Student Name is Required" )]
    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string EmployeeName { get; set; }
    [NotMapped]
    public int? Age { get; set; }
```

```
        [NotMapped]
        public DateTime Birthday { get; set; }

        public int DeptId { get; set; }
        [ForeignKey("DeptId")]
        public virtual Department Department { get; set; }
    }
```

[**Table**(**"EmployeeInfo"**)] — для класса, помеченного этим атрибутом, в БД должна быть создана таблица с именем EmployeeInfo;

[**Key**] — помечает свойство, которое будет являться первичным ключом;

[**Required**(**ErrorMessage="Student Name is Required"**)] — валидационный атрибут, требующий, чтобы данное поле было обязательным для заполнения;

[**Column**(**"Name", TypeName="ntext"**)] — помечает свойство, которое будет в создаваемой таблице полем с именем Name и типом NText;

[**MaxLength**(**20**)] — валидационный атрибут, ограничивающий длину текстового поля;

[**NotMapped**] — помечает свойство, которое НЕ БУДЕТ отображаться в поле таблицы;

[**ForeignKey**(**"DeptId"**)] — указывает внешний ключ для свойства навигации.

О некоторых других атрибутах из пространства имен System.ComponentModel.DataAnnotations, применяющихся для аннотации данных, вы можете прочесть на MSDN.

Я думаю, вы согласитесь с тем, что в применении этих атрибутов все достаточно наглядно.

## Конфигурирование связей с помощью атрибутов

Атрибуты могут использоваться не только для определения свойств таблиц. С помощью атрибутов в технологии «код сначала» можно конфигурировать и связи между таблицами. Вы уже знаете, что свойства навигации зависят от типа связи между сущностями. Поэтому очень важно правильно описывать требуемые связи. Давайте посмотрим, как можно задавать с помощью атрибутов связи разных типов.

Рассмотрим связь «один к одному». В нашей БД таких связей нет, но, для примера, давайте проведем мысленный эксперимент. Предположим, что у нас появился класс Address:

```
public class Address
{
    public Address()
    {
    }

    public int Id { get; set; }
    public string City { get; set; }
    public string PostAddress { get; set; }
}
```

Мы хотим хранить в этой таблице адреса издательств, причем у каждого издательства может быть один-единственный адрес. При связи «один к одному» первичный ключ одной таблицы является первичным и одновременно внешним ключом связанной таблицы. Давайте свяжем

классы (а значит и таблицы) Publisher и Address связью «один к одному». Для этого надо модифицировать эти классы таким образом:

```
public class Address
{
    public Address()
    {
    }
    [Key, ForeignKey("Publisher")]
    public int Id { get; set; }
    public string City { get; set; }
    public string PostAddress { get; set; }
}

public class Publisher
{
    public Publisher()
    {
    }

    public int Id { get; set; }
    public string PublisherName { get; set; }
    public string Address { get; set; }
    [Required]
    public virtual Address StudentAddress { get; set; }
}
```

В классе Address мы описали свойство как первичный ключ (атрибут [Key]) и одновременно как внешний ключ (атрибут [ForeignKey]), указав, что этот внешний ключ связан с таблицей Publisher.

Рассмотрим связь «один ко многим». В нашем приложении из первого урока такая связь присутствовала между таблицами Author и Book. Одному автору может

соответствовать несколько книг. Давайте рассмотрим, как можно описать такой вид связи.

```
public class Author
{
    public Author()
    {
        Books = new List<Book>();
    }

    public int Id { get; set; }
    [Required]
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual ICollection<Book>
      Books { get; set; }
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int PublisherId { get; set; }

    public virtual Author Author { get; set; }
}
```

Такое описание этих классов приведет к созданию между ними связи «один ко многим».

Давайте пофантазируем еще раз и рассмотрим описание связи «многие ко многим». Предположим, что мы решили модифицировать нашу БД таким образом, чтобы иметь возможность для каждой книги указывать

несколько авторов. Я думаю, каждый из вас может назвать несколько успешных авторских коллективов. Помните «банду четырех» (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс), написавших знаменитую Design Patterns: Elements of Reusable Object-OrientedSoftware? Или И. Ильфа и Е. Петрова? Или же Л. Ландау и Е. Лифшица, написавших выдающийся многотомный «Курс теоретической физики»? Чтобы иметь возможность хранить в нашей БД такие книги, мы должны создать между таблицами Author и Book связь «многие ко многим». Это можно сделать таким образом:

```
public class Author
{
    public Author()
    {
        Books = new List<Book>();
    }

    public int Id { get; set; }

    [Required]
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual ICollection<Book>
      Books { get; set; }
}

public class Book
{
    public Book()
    {
        Authors = new HashSet<Author>();
    }
```

```
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int PublisherId { get; set; }

    public virtual ICollection<Author>
      Authors { get; set; }
}
```

Как видите, в этом случае навигационные свойства в связанных сущностях должны быть коллекциями. При обработке этого кода Entity Framework создаст в БД третью таблицу с именем AuthorBook, которая будет содержать в себе первичные ключи обоих таблиц, т.е. AuthorId и BookId. В данном случае будет полезно не называть первичные ключи в обеих таблицах совпадающими именами Id. Имена, указанные выше, соответствуют договоренностям Entity Framework, поэтому их можно не помечать атрибутами [Key].

## Fluent API

Помимо использования атрибутов существует еще один механизм, позволяющий разработчику управлять созданием БД при использовании подхода Code First. Этот механизм называется Fluent API и представляет собой набор методов, которые надо использовать при создании БД по существующим моделям.

Давайте создадим еще один проект, в котором рассмотрим создание БД при подходе Code First с помощью

Fluent API. Создадим опять консольное приложение с именем LibraryCodeFirstFluent.

Добавим в состав приложения три модели и класс контекста БД.

```
public class Author
{
    public Author()
    {
        Books = new List<Book>();
    }

    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Publisher
{
    public Publisher()
    {
    }

    public int Id { get; set; }
    public string PublisherName { get; set; }
    public string Address { get; set; }
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int PublisherId { get; set; }
}
```

```
public class LibraryContext : DbContext
{
    public LibraryContext()
           : base("LibraryDb")
    {
    }

    public DbSet<Author> Authors { get; set; }
    public DbSet<Publisher> Publishers { get; set; }
    public DbSet<Book> Books { get; set; }
}
```

Снова добавим в метод Main() уже знакомый вам код:

```
Author author = new Author { FirstName = "Isaac",
LastName = "Azimov" };
using (LibraryContext db = new LibraryContext())
{
    db.Authors.Add(author);
    db.SaveChanges();

    var ac = db.Authors.ToList();
    foreach (var a in ac)
    {
        Console.WriteLine(a.FirstName +" "+a.LastName);
    }
}
```

И запустим приложение. Пока что все выглядит, как раньше. Но дальнейшие наши действия будут новыми. В классе DbContext определен виртуальный метод с именем OnModelCreating(). Давайте откроем определение нашего класса контекста и добавим в него переопределение этого метода:

```
protected override void
   OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
}
```

Как понятно по имени этого метода, он вызывается всякий раз при создании моделей. В этом методе мы можем явно указать, каким образом должны создаваться таблицы в БД на основе наших моделей. Параметр этого метода типа DbModelBuilder управляет отображением классов моделей в таблицы БД. Давайте рассмотрим основные методы этого класса и назначение этих методов.

## Краткий обзор Fluent API

Чтобы указать, что вы работаете с сущностью Author и создаете в БД таблицу, соответствующую этой сущности, надо от имени параметра этого метода modelBuilder вызвать generic метод Entity<Author>(). Все дальнейшие методы, управляющие созданием таблицы Author, можно вызывать в chain режиме, хотя можно и прерывать цепочку вызовов.

```
modelBuilder.Entity<Author>().HasKey(t => t.AuthorId);
```

метод HasKey указывае, какое свойство модели Author является первичным ключом создаваемой таблицы.

В приведенном ниже примере методы HasKey() и HasDatabaseGeneratedOption() указывают, что первичным ключом является поле Id, и что значения этого

поля задаются самой БД автоматически. Т.е. это аналог SQL функции identity(1,1):

```
modelBuilder.Entity<Author>().HasKey(a => a.Id);
modelBuilder.Entity<Author>().Property(a=>a.Id).
HasDatabaseGeneratedOption(DatabaseGeneratedOption.
Identity);
```

Обратите внимание на метод Property(), который в данном случае уточняет, для какого поля таблицы вызывается метод HasDatabaseGeneratedOption().

Если вы хотите ограничить длину какого-либо поля, можете сделать это, например, таким вызовом:

```
modelBuilder.Entity<Author>().Property(t =>
t.FirstName).HasMaxLength(50);
```

И снова обратите внимание на использование метода Property().

Если вы хотите указать, что какое-либо поле таблицы должно быть NOT NULL, можете это сделать так:

```
modelBuilder.Entity<Author>().Property(t =>
t.FirstName).IsRequired();
```

Если вы хотите указать, что какое-либо свойство модели не должно отображаться в поле таблицы, можете это сделать так:

```
modelBuilder.Entity<Author>().Ignore(t => t.
LastName);
```

Если вы хотите указать, что какое-либо свойство модели должно отображаться в поле таблицы, но при этом иметь другое имя, можете это сделать так:

```
modelBuilder.Entity<Author>().Property(t => t.
LastName).
HasColumnName("AuthorName");
```

В этом случае в созданной таблице БД поле, соответствующее свойству модели LastName, будет называться AuthorName.

### Определение связей в Fluent API

Давайте рассмотрим, как с помощью Fluent API можно задавать связи между создаваемыми таблицами. Рассмотрим связь между таблицами Author и Book. Мы хотим указать, что одной записи в таблице Author может соответствовать много записей в таблице Book. Для этого нам надо внести изменения в модели, добавив туда свойства навигации. Это, в свою очередь, приведет к тому, что созданная БД перестанет соответствовать моделям, что и вызовет исключительную ситуацию. Здесь нам на помощь придет миграция.

Добавляем в наши модели свойства навигации, необходимые для создания наших связей:

Модели при этом будут такими:

```
public class Author
{
    public Author()
    {
        Books = new List<Book>();
    }
```

```
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual ICollection<Book> Books { get; set; }
}

public class Publisher
{
    public Publisher()
    {

    }

    public int Id { get; set; }
    public string PublisherName { get; set; }
    public string Address { get; set; }
    public virtual ICollection<Book> Books { get; set; }
}

public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int PublisherId { get; set; }

    public virtual Author Author { get; set; }
    public virtual Publisher Publisher { get; set; }
}
```

В Fluent API существует ряд методов, позволяющих описывать связи для создаваемых таблиц. Например, связь «один ко многим» будет описываться таким образом:

```
modelBuilder.Entity<Book>(). //таблица Book
HasRequired<Author>(a => a.Author). //связывается
                              //с таблицей Author
WithMany(s => s.Books).       //у которой есть
                              //свойство навигации Books
HasForeignKey(s => s.AuthorId). //а внешним ключом
                              //будет поле AuthorId
WillCascadeOnDelete(false); //при этом каскадное
                            //удаление не использовать
```

Теперь вносим изменения в метод OnModelCreating(), чтобы он выглядел таким образом:

```
protected override void
  OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    //указываем первичный ключ для таблицы Author
    modelBuilder.Entity<Author>().HasKey(a => a.Id);
    modelBuilder.Entity<Author>().Property(a => a.Id).
        HasDatabaseGeneratedOption
          (DatabaseGeneratedOption.Identity);

    //  указываем, что поле FirstName не может быть
    //NULL и задаем ему максимальную длину
    modelBuilder.Entity<Author>().
      Property(t => t.FirstName).
    IsRequired().HasMaxLength(128);

    //указываем первичный ключ для таблицы Publisher
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Publisher>().HasKey(a => a.Id);
    modelBuilder.Entity<Publisher>().Property(a => a.Id).
    HasDatabaseGeneratedOption
      (DatabaseGeneratedOption.Identity);
```

```
    //указываем первичный ключ для таблицы Book
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Book>().HasKey(a => a.Id);
    modelBuilder.Entity<Book>().Property(a => a.Id).
    HasDatabaseGeneratedOption
     (DatabaseGeneratedOption.Identity);

    //задаем связь между таблицами Book и Author
    modelBuilder.Entity<Book>().
        <Author>(a => a.Author).
        WithMany(s => s.Books).
        HasForeignKey(s => s.AuthorId).
        WillCascadeOnDelete(false);

    //задаем связь между таблицами Book и Publisher
    modelBuilder.Entity<Book>().
        HasRequired<Publisher>(a => a.Publisher).
        WithMany(s => s.Books).
        HasForeignKey(s => s.PublisherId).
        WillCascadeOnDelete(false);
}
```

Выполняем миграцию. Выбираем меню Сервис (Serice) — Диспетчер пакетов библиотек (Library package manager) — Консоль диспетчера пакетов (Package manager console). Затем в нижнем окне консоли, после подсказки PM>, вводим и выполняем такую команду:

```
Enable-Migrations,
```

Затем вводим и выполняем команду:

```
Add-Migration migration1
```

И еще одну команду:

```
Update-Database
```

После выполнения этих команд БД в нашем приложении будет пересоздана, в ней появятся связи, которые мы определили в методе OnModelCreating() с помощью Fluen API. Вы должны также помнить, что при использовании техники миграций мы добавили в наше приложение класс Configuration в папке Migrations. В этом классе есть метод Seed(), предназначенный для инициализации БД. Поэтому, если вы хотите заполнять какие-либо таблицы в своей БД, делать это надо в этом методе.

Мы с вами рассмотрели технологию Code First и те способы, которые есть у программиста для управления созданием БД по имеющимся моделям. Теперь вы знаете, что управлять созданием БД при использовании Code First можно с помощью класса инициализатора, с помощью атрибутов, а также с помощью Fluent API.

Мы разобрали последний урок о замечательном фреймворке Entity Framework и сейчас вам надо понять и запомнить еще одну важную деталь. Использование фреймворков становится уже не просто нормой современной разработки программных систем, а скорее даже требованием. Это обусловлено теми преимуществами, которые предоставляют такие системы: автоматизация рутинных действий, ускорение разработки, читабельность и масштабируемость созданного кода. Но вы также понимаете, что владение инструментом приходит с опытом, поэтому я хочу вам пожелать находить в своих будущих проектах место для Entity Framework и применять его как можно чаще. И если среди тех технологий, которые вы перечисляете в своих резюме, будут красоваться буквы Entity Framework, то это повысит вашу привлекательность, как специалистов.