

**top**

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# Технология доступа к базам данных ADO.NET

# Урок №6

## Знакомство с Entity Framework

### Содержание

<b>Понятие ORM системы.....</b>	<b>4</b>
Новые термины:.....	9
<b>Архитектура Entity Framework .....</b>	<b>10</b>
Entity Data Model .....	10
Слой Служб объектов (Object Services Layer).....	11
Слой Клиентского провайдера данных (Entity Client data provider) .....	12
Новые термины:.....	13
<b>Способы создания Entity Data Model .....</b>	<b>14</b>
База данных сначала (Database first) .....	14
Модель сначала (Model first) .....	14
Код сначала (Code First) .....	15
Новые термины:.....	15

<b>Класс DbContext .....</b>	<b>16</b>
Новые термины: .....	21
<b>База данных сначала (Database first) .....</b>	<b>22</b>
Создание БД в LocalDB. ....	25
Создание EDM для Database first .....	29
<b>LINQ to Entities (введение) .....</b>	<b>38</b>
First() и FirstOrDefault() .....	38
Single() и SingleOrDefault() .....	39
ToList() .....	41
OrderBy() .....	42
Find() .....	43
<b>Заполнение БД .....</b>	<b>45</b>
<b>Свойства навигации .....</b>	<b>48</b>
Новые термины: .....	51
<b>Домашнее задание .....</b>	<b>52</b>

# Понятие ORM системы

---

На сегодняшний день хранение информации в реляционных БД является стандартным приемом в большинстве случаев. БД доказали свою эффективность и безопасность. Вы помните, что в реляционных БД информация хранится в связанных таблицах. Представьте, что вы пишете приложение, которое должно работать с базой данных. Вы уже знакомы с технологией ADO.NET, поэтому можете оценить те задачи, которые вам надо при этом решить.

Приложению необходимо получить данные из таблиц БД и разместить у себя в объектах каких-либо классов. В технологии ADO.NET такими классами могут быть *DataTable* или *SqlDataReader*. Вы конечно же помните, что каждая СУБД работает со своими типами данных, часто не имеющими точного соответствия в среде разработки приложения. Как вы думаете, почему класс *SqlDataReader* содержит методы типа *GetInt16()*, *GetInt32*, *GetInt64()*, *GetFloat()*, *GetDouble()*, *GetDecimal()* и еще несколько десятков подобных? Именно для того, чтобы как можно точнее прочитать данные из БД. Но даже при таком наборе инструментов, точное соответствие получить можно далеко не всегда. Во многих случаях приходится прибегать к последнему аргументу — типу *Object*. Вы помните, что *Object* является базовым для всех остальных типов, и поэтому, мы можем любой тип данных преобразовать к *Object*. К чему это приводит? К тому, что для чтения данных из

каждой таблицы БД приходится писать «персональный» код, настраивать его на структуру таблицы, следить за соответствием типов данных. А таблиц в реальной БД могут быть десятки, и скажу вам по секрету, даже сотни. Конечно же, проявив достаточную настойчивость и трудолюбие вы этот код напишете и отладите! А когда принесете его сдавать заказчику, вдруг узнаете, что несколько таблиц в БД изменились. И вам придется изменять свой код, в котором вы работаете с этими таблицами. А этот код, конечно же связан с другими обработками в вашем приложении — придется вносить изменения и там. Конечно же, вашего труда и затраченного вами времени на создание этой обработки данных никому не жалко ☺.

Но ведь и приложению, выполняя этот код, придется тратить кучу ресурсов. Вот это уже не допустимо.

Можно ли оптимизировать этот процесс? Давайте рассмотрим такой подход. В базе данных есть таблица Book, с которой вам надо работать. Вы создаете в приложении класс и называете его тоже Book (хотя 'то не обязательно). В этом классе вы создаете свойства, соответствующие полям таблицы, и методы для считывания данных из таблицы в объекты своего класса. Также вы создаете метод, позволяющий записывать объекты вашего класса в таблицу БД. Я думаю, вы помните, что такое инкапсуляция? Здесь она проявляется в чистом виде — вы инкапсулируете в этом классе все необходимое взаимодействие с таблицей Book. Вы создаете соответствие «таблица БД — класс приложения». При этом каждая строка, прочитанная из таблицы Book, будет превращаться в отдельный объект

класса Book. Здесь же надо реализовать необходимое преобразование типов данных между полями таблицы и свойствами вашего класса. Аналогичную работу проделать для всех таблиц БД, с которыми необходимо работать приложению. Если собрать всю эту обработку в одном месте, то это здорово уменьшит зависимость остального кода приложения от структуры БД.

Какие плюсы такого подхода? Инкапсуляция работы с БД в одном месте. Если изменится структура какой-либо таблицы, а по закону Ньютона Мерфи она обязательно изменится, вам надо будет внести изменения только в классе, соответствующем изменившейся таблице. Какие минусы такого подхода? Вопросы соответствия типов на уровне каждого класса и каждой таблицы надо решать все равно. Надо создавать много новых классов в приложении и продумывать их взаимодействие с другими частями приложения. Надо создавать контейнеры для хранения и обработки объектов этих классов. Другими словами, такой подход, решает одни проблемы, но создает другие. Кроме того, мы предположили, что одной таблице БД соответствует один класс в приложении, а это предположение далеко от реальной жизни. Давайте рассуждать дальше.

Введем несколько терминов, которые будут необходимы нам в дальнейшем. Вы должны помнить, что в БД информация об одном объекте может располагаться не в одной таблице, а в нескольких связанных таблицах. Например, для описания книги полезно хранить информацию об издательстве, выпустившем книгу. Однако правила

нормализации таблиц требуют вынести информацию об издательстве в отдельную таблицу, например, *Publisher*, а в таблицу *Book* вставить внешний ключ, ссылающийся на описание издательства. При таком подходе информация об одной книге будет храниться уже в двух таблицах — *Book* и *Publisher*. Аналогично в отдельную таблицу можно вынести и информацию об авторе книги, и, возможно, что-то еще. При переносе в приложение данные о каждой книге уже надо будет хранить в нескольких классах, связанных между собой.

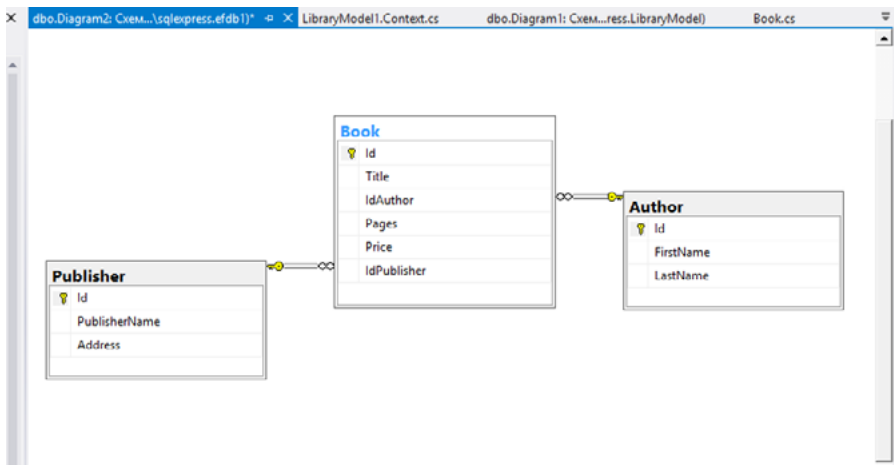


Рис. 1. Структура БД

Запомните первый термин — «объектная модель». Объектная модель — это группа классов приложения, связанных между собой и использующихся для хранения, обработки и отображения данных из БД. И теперь вместо соответствия «таблица БД — класс приложения» правильнее говорить о соответствии «БД — объектная модель».

Для реализации такого соответствия предназначаются ORM системы. ORM значит Object Relational Mapping, что можно перевести как «отображение объектов в связанные таблицы». Проблема, которую мы с вами обсуждаем, встала перед разработчиками уже очень давно. Первые коммерчески успешные ORM системы появились еще в 1995 году. Это был фреймворк TOPLink, созданный на языке Smalltalk. Этот фреймворк был портирован для ранних версий Java. Чуть позже в операционной системе NeXT был представлен продукт DbKit, решающий эту же задачу. На сегодняшний день существует очень много ORM систем для разных платформ и разных языков программирования. Чтобы убедиться в этом, посмотрите этот обзор: [https://en.wikipedia.org/wiki/List\\_of\\_object-relational\\_mapping\\_software](https://en.wikipedia.org/wiki/List_of_object-relational_mapping_software).

Предмет нашего изучения, Entity Framework, является ORM системой, разработанной фирмой Microsoft. Давайте вспомним, что обозначает слово фреймворк, стоящее в названии этого продукта. Хотя понятие фреймворк интуитивно понятно, я бы еще раз попытался объяснить его. Например, таким образом. Это — программное обеспечение или платформа, предоставляющие стандартизированный способ решения задачи проектирования и разработки какой-либо системы (или подсистемы).

Впервые система Entity Framework появился в .NET Framework 3.5 Service Pack 1 в 2008 году. Первая версия продукта вызвала ряд критических замечаний, но разработчики очень основательно усовершенствовали Entity Framework и сегодня он является одним из лидеров в дан-



ном сегменте инструментов. Уже очень скоро вы увидите, что использование ADO.NET — не единственный способ работать с БД. Но это совсем не значит, что технология ADO.NET больше не нужна. Она по-прежнему является важным и востребованным инструментом. Но теперь вы получаете новый инструмент, который позволяет выполнить работу более эффективно.

Итак, задача Entity Framework — реализовывать соответствие между объектами, используемыми в приложении, и связанными таблицами в БД. При этом Entity Framework автоматически решает многие вопросы, возникающие при создании такого соответствия. Это все мы будем рассматривать в наших уроках.

### **Новые термины:**

- *Объектная модель* — это группа связанных между собой классов приложения, использующихся для хранения данных из БД;
- *ORM* — Object Relationl Mapping или «отображение объектов в связанные таблицы».

# Архитектура Entity Framework

---

## Entity Data Model

К настоящему моменту мы выяснили, что Entity Framework это фреймворк для ADO.NET, который предоставляет разработчику улучшенные способы для хранения и обработки информации в БД и для отображения информации из БД в приложение. Давайте рассмотрим внутреннее устройство Entity Framework. Для этого надо познакомиться с моделью данных (*Entity Data Model*, сокращенно *EDM*) этого фреймворка. EDM описывает взаимосвязь между классами в приложении и таблицами в БД. В EDM выделяют три составляющие:

1. **Концептуальная модель** (*Conceptual model*), в которой описываются классы приложения и взаимоотношения между ними.
2. **Отображение** (*Mapping*), содержащее схему соответствия между *Conceptual model* и *Storage model*, т.е. между классами приложения и таблицами БД. Вся эта информация о структуре БД (*Storage model*), о модели данных (*Conceptual model*) и об их взаимном отображении содержится в XML в файле с расширением *.edmx*. Однако кроме EDM Entity Framework содержит еще ряд важных компонентов, называемых слоями.

3. **Модель хранилища** (*Storage model*), в которой описываются связанные таблицы, расположенные в БД.

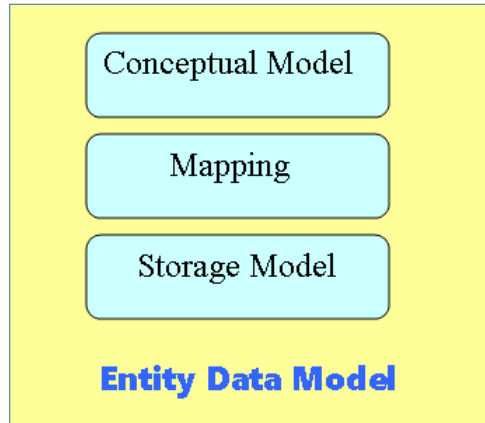


Рис. 2. Архитектура Entity Framework

### Слой Служб объектов (Object Services Layer)

Согласно пятому правилу Кодда, «Система управления реляционными базами данных должна поддерживать хотя бы один реляционный язык». Для доступа к СУБД Entity Framework предлагает два способа: LINQ to Entities и Entity SQL.

**LINQ to Entities** — это расширение LINQ для создания запросов к Conceptual model (т.е. к объектам классов приложения) на языках C# или VB. Далее рассмотрим использование LINQ to Entities подробно.

**Entity SQL**, согласно MSDN, «представляет собой независимый от хранилища язык запросов, аналогичный языку SQL. Entity SQL позволяет выполнять запросы к данным

сущности, представленным либо в виде объектов, либо в табличной форме». Этот язык несколько сложнее LINQ to Entities и требует специального рассмотрения.

Так вот, **слой служб объектов** (*Object Services Layer*) — это важнейший компонент Entity Framework, который позволяет пользователю использовать язык программирования (*LINQ to Entities* или *Entity SQL*) для создания запросов к БД. Этот слой работает с объектами классов приложения, синхронизируя их с данными в таблицах БД. В этом слое выполняются такие действия, как фиксирование текущего состояния объектов и преобразование данных, полученных из таблиц БД в результате выполнения запроса, в объекты классов приложения.

### Слой Клиентского провайдера данных (Entity Client data provider)

Получив запрос на LINQ to Entities или на Entity SQL, этот слой преобразовывает его в SQL и передает в Слой провайдера данных.

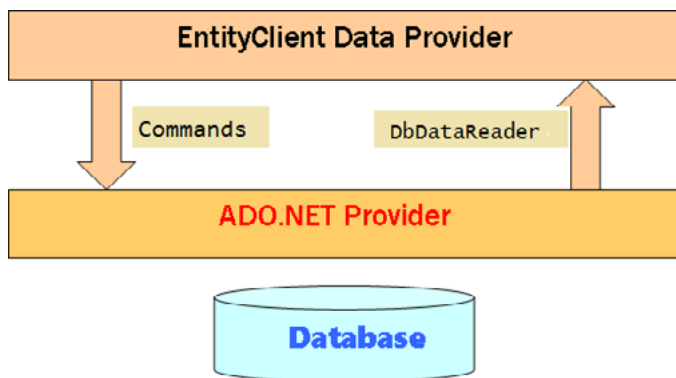


Рис. 3. Взаимодействие слоев

## Слой провайдера данных ADO.NET (ADO.NET data provider)

Этот слой предназначен для непосредственного обращения к СУБД с использованием технологии ADO.NET. К этому моменту запросы, созданные на LINQ to Entities или на Entity SQL, должны быть конвертированы в SQL запросы.

### Новые термины:

- **EDM** (*Entity Data Model*) — это модель, описывающая взаимоотношение объектов классов в приложении, с одной стороны, и связанных таблиц в БД, с другой.

# Способы создания Entity Data Model

---

Мы установили, что Entity Data Model является основой фреймворка Entity Framework. У разработчика есть несколько способов создания Entity Data Model. Каждый из этих способов фактически приводит к одному результату, но имеет свои отличительные характеристики. Рассмотрим эти способы. Мы будем использовать Visual Studio 2015 и Entity Framework 6.1, подходящие для написания приложений для версий .NET 4.0 and .NET 4.5. Но прежде чем переходить к созданию примеров, перечислим и кратко охарактеризуем все эти способы.

## **База данных сначала (Database first)**

Как понятно из названия, при этом подходе сначала проектируется и разрабатывается БД. Затем на основе созданной БД Entity Framework создает EDM. Дальше уже EDM создает в приложении классы, соответствующие таблицам БД, т.е. Conceptual model. Классы, соответствующие таблицам БД и создаваемые фреймворком в приложении, называются сущностями. При таком подходе БД является как бы фундаментом приложения.

## **Модель сначала (Model first)**

В этом случае разработчик сначала должен создать в Visual Studio EDM. Для этого Visual Studio предоставляет специальный режим работы в дизайнера. Мы рас-

смотрим, как это делается. Затем на основе созданной Модели данных создается БД. Этот подход рекомендуется использовать в том случае, когда вы четко представляете себе, а еще лучше — знаете структуру БД.

### **Код сначала (Code First)**

Этот способ работы с Entity Framework появился, начиная с версии 4.1. Он особенно близок программистам, так как требует от них выполнения традиционных действий — написания кода. При этом подходе программист создает в коде необходимые классы. Затем на основе этих классов Entity Framework создает EDM и БД.

### **Новые термины:**

- ***Database first, Model first, Code First*** — это разные способы создания EDM при работе с Entity Framework.

# Класс DbContext

Для изучения Entity Framework нам потребуется БД. Конечно же, для простоты, чтобы не отвлекать внимание от знакомства с Entity Framework, эта база данных будет несколько условной. Создадим на MS SQL Server новую базу данных, например Library, а в ней создадим таблицы Book, Author и Publisher.

```
CREATE TABLE Author
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    FirstName VARCHAR(100) NOT NULL,
    LastName VARCHAR(100) NOT NULL
)

CREATE TABLE Publisher
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    PublisherName VARCHAR(100) NOT NULL,
    Address VARCHAR(100) NOT NULL
)

CREATE TABLE Book
(
    Id INT NOT NULL IDENTITY(1,1) PRIMARY KEY,
    Title VARCHAR(100) NOT NULL,
    IdAuthor INT NOT NULL FOREIGN KEY REFERENCES
    Author(Id),
    Pages INT,
    Price INT,
    IdPublisher INT NOT NULL FOREIGN KEY
    REFERENCES Publisher(Id)
)
```



Базы данных вы создавать умеете, поэтому на этом этапе работы останавливаться не будем. Скрипты для создания таблиц приведены лишь для того, чтобы имена таблиц и полей в них совпадали в приложениях каждого из вас. Если у вас есть установленный MS SQL Server, вы можете запустить Management Studio и создать БД на этом сервере. В этом случае запомните адрес сервера, где вы создали свою БД. Вам скоро понадобится этот адрес. Свою БД вы также можете создать в LocalDB, как предлагает по умолчанию Visual Studio 2015.

Мы уже выяснили, что при создании EDM по технологии «база данных сначала», Entity Framework добавляет в состав текущего проекта классы, соответствующие таблицам БД. Еще раз повторим, что в терминологии Entity Framework эти классы называются сущностями. Для нашей БД мы получим в приложении классы Author, Publisher и Book. Однако оказывается, что Entity Framework добавляет не только эти классы. Кроме классов, соответствующих таблицам БД, Entity Framework создает еще один класс, который называется контекстом БД. Это один из центральных классов Entity Framework. Давайте познакомимся с ним ближе. Например, для нашей БД этот класс может выглядеть таким образом:

```
public partial class LibraryEntities: DbContext
{
    public LibraryEntities()
        :base("name=LibraryEntities")
    {
    }
}
```

```
protected override void
OnModelCreating(DbModelBuilder modelBuilder)
{
    throw new UnintentionalCodeFirstException();
}

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

Как видно, этот класс является производным от класса `System.Data.Entity.DbContext`. Основные задачи, которые выполняет `LibraryEntities`, следующие:

- установка связи с БД;
- выполнение наших запросов к БД путем конвертации значений из таблиц БД в объекты классов приложения (сущности) и наоборот;
- Отслеживание и сохранение изменений в состоянии объектов классов приложения (сущностей) после выполнения запросов.

В более ранних версиях Entity Framework для этих целей использовался класс `ObjectContext`. Однако работать с ним было несколько сложнее. Класс `DbContext` является оберткой вокруг `ObjectContext` и предоставляет разработчику более удобный механизм работы с БД. Кстати, `ObjectContext` и `DbContext` не являются абстрактными классами, хотя и используются в качестве базовых классов. Наследование необходимо для того, чтобы добавить в контекст текущей БД (в нашем случае в класс `LibraryEntities`)

свойства, отвечающие за доступ к данным в этой БД. При использовании класса DbContext эти свойства имеют тип `DbSet<T>`, где параметр `T` представляет имена сущностей модели, т.е. имена классов, соответствующих таблицам БД.

Обратите внимание на конструктор этого класса. Поскольку `LibraryEntities` должен установить связь с БД, мы должны каким-либо образом указать ему, какая БД должна быть подключена. Для этих целей используется перегруженный конструктор класса DbContext. Посмотрите в MSDN описание конструкторов класса DbContext по этой ссылке: [https://msdn.microsoft.com/ru-ru/library/system.data.entity.dbcontext\(v=vs.113\).aspx](https://msdn.microsoft.com/ru-ru/library/system.data.entity.dbcontext(v=vs.113).aspx)

Я считаю полезным привести эти описания с MSDN здесь.

<code>DbContext()</code>	Создает новый экземпляр контекста с использованием соглашений для создания имени базы данных, с которой будет установлено соединение. Имя по соглашению представляет собой полное имя (пространство имен + имя класса) производного класса контекста. Как это используется при создании соединения, см. в примечаниях к классу.
<code>DbContext(String)</code>	Создает новый экземпляр контекста с использованием соглашений для создания имени или строки подключения базы данных, с которой будет установлено соединение. Как это используется при создании соединения, см. в примечаниях к классу.
<code>DbContext (DbContextCompiledModel)</code>	Создает новый экземпляр контекста с использованием соглашений для создания имени

	базы данных, с которой будет установлено соединение, и инициализирует его из заданной модели. Имя по соглашению представляет собой полное имя (пространство имен + имя класса) производного класса контекста. Как это используется при создании соединения, см. в примечаниях к классу.
<code>DbContext</code> ( <code>DbConnection</code> , <code>Boolean</code> )	Создает новый экземпляр контекста с использованием существующего соединения с базой данных. Соединение не будет освобождено при освобождении контекста, если <code>contextOwnsConnection</code> является <code>false</code> .
<code>DbContext</code> ( <code>String</code> , <code>DbCompiledModel</code> )	Создает новый экземпляр контекста с использованием указанной строки в качестве имени или строки подключения с базой данных, с которой будет установлено соединение, и инициализирует его из заданной модели. Как это используется при создании соединения, см. в примечаниях к классу.
<code>DbContext</code> ( <code>ObjectContext</code> , <code>Boolean</code> )	Создает новый экземпляр контекста на основе существующего объекта <code>ObjectContext</code> .
<code>DbContext</code> ( <code>DbConnection</code> , <code>DbCompiledModel</code> , <code>Boolean</code> )	Создает новый экземпляр контекста с использованием существующего соединения с базой данных и инициализирует его из заданной модели. Соединение не будет освобождено при освобождении контекста, если <code>contextOwnsConnection</code> является <code>false</code> .

В приведенном примере используется второй вариант — конструктор, которому передается имя строки подключения к требуемой БД.

Теперь, когда мы рассмотрели механизм работы Entity Framework, можно перейти к конкретным примерам.

### Новые термины:

- **Контекст БД** — это специальный класс, производный от системного класса DbContext и предназначенный для установления связи с БД и для выполнения запросов к БД .
- **Сущность** — это класс, соответствующий таблице БД, автоматически создаваемый Entity Framework. Свойства этого класса соответствуют полям таблицы.

# База данных сначала (Database first)

---

Запускаем Visual Studio 2015 и создаем новый проект. Пусть сейчас это будет консольное приложение C# с именем LibraryDbFirst. Понятно, что приложение не обязательно должно быть именно консольным. Просто это самый простой вид приложения, и мы не будем отвлекаться от вопросов использования Entity Framework.

В этом проекте мы создадим Entity Data Model для уже существующей БД, т.е. по технологии «база данных сначала». Прежде чем двигаться дальше, полезно убедиться, что в Visual Studio установлена требуемая версия Entity Framework. Здесь будет полезно сделать небольшое отступление, чтобы поговорить о пакете NuGet. Если вы знаете, что такое NuGet, просто пропустите три следующих абзаца.

**NuGet** — это бесплатное расширение Visual Studio, предназначенное для добавления сторонних библиотек в ваше приложение. Также NuGet позволяет обновлять и удалять эти библиотеки. Добавляемая библиотека разворачивается в проекте в виде пакета. NuGet-пакет является набором файлов, упакованных в один файл с расширением .nupkg в формате Open Packaging Conventions (OPC). В свою очередь, OPC — это просто zip-файл с некоторыми метаданными. Скорее всего NuGet уже установлен в вашей Visual Studio. Это можно проверить в меню Project (Проект).

Посмотрите, есть ли там команда Manage NuGet packages (Управление пакетами NuGet). Если есть — активируй-

те ее и можете работать с NuGet. Если такой команды нет, надо перейти в меню Tools — NuGet packageManager (Сервис — Диспетчер пакетов NuGet). Активировав этот пункт меню, вы увидите команду Manage NuGet Packages for Solution . Активируйте эту команду и вы таким образом получите возможность работать с NuGet. Этот пакет будет полезен вам не только при работе с Entity Framework, но и в других ситуациях.

Работа с этим пакетом очень проста и удобна. Вспомните, как вы обычно устанавливали различные расширения и плагины. Сначала надо найти ссылку на нужный дистрибутив, убедиться, что дистрибутив работоспособный, скачать его, выполнить установку. Для NuGet надо указать только имя требуемого вам расширения. NuGet сам найдет дистрибутив, и сам выполнит установку. Конечно же, при этом вы должны быть подключены к Интернету. После активации NuGet вы увидите такое окно:

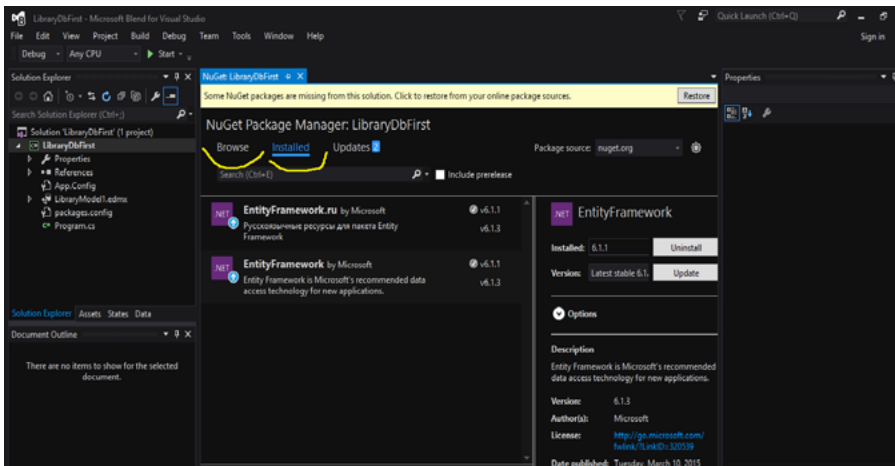


Рис. 4. Работа с NuGet

В появившемся окне перейдите по ссылке `Installed` и проверьте, есть ли среди установленных пакетов `Entity Framework`. Как видно на рисунке в моем случае `Entity Framework` уже установлен. На момент написания этого урока установлена версия `EntityFramework 6.1.1`. Скорее всего, у вас такая же ситуация. Если же в вашем проекте `EntityFramework` не установлен, перейдите левее на ссылку `Browse`, в окно поиска введите `EntityFramework`, а когда пакет будет найден, нажмите справа кнопку `Install`. Установка занимает совсем немного времени.

Вот теперь мы готовы приступить к работе с `Entity Framework`. Это действие надо будет повторять в каждом проекте, в котором вы будете работать с `Entity Framework`. Да, это увеличивает размер вашего проекта, но вы скоро убедитесь в том, что это, пожалуй, единственный недостаток использования `Entity Framework`. Преимуществ от использования `Entity Framework` гораздо больше.

Теперь вернемся к созданию `Entity Data Model` по технологии «база данных сначала». Такой подход подразумевает, что БД уже создана. Возможно, вы уже создали себе нашу БД из трех таблиц на каком-нибудь своем сервере. Но я предлагаю вам рассмотреть вопрос создания новой БД в новом продукте компании Microsoft для работы с БД, в так называемом `LocalDB`. Опыт работы с `LocalDB` будет полезен в любом случае, поэтому, даже если вы создали базу данных на выделенном сервере, я советую вам создать ее на `LocalDB`. Для этого следуйте дальнейшим инструкциям.



## Создание БД в LocalDB

После создания нового приложения, например, с именем LibraryDbFirst, надо создать новую БД. По умолчанию Visual Studio 2015 создает базу данных в LocalDB. LocalDB — это облегченный вариант SQL Server Express Edition. Он поддерживает всю функциональность SQL Server Express Edition (за исключением FileStream), но устанавливается быстрее и выполняется в пользовательском режиме, а не как служба. LocalDB не предназначен для сценариев с удаленным подключением, но, поскольку сама БД хранится в файле (с расширением .mdf), позволяет легко переносить приложение с компьютера на компьютер. Другими словами, LocalDB — это идеальное средство для отладки (и не только), и научиться работать с ним будет полезно каждому. Более подробную информацию о LocalDB можете поискать в сети — например, здесь:

<http://geekswithblogs.net/krislankford/archive/2012/06/19/sql-server-2012-express-localdb-how-to-get-started.aspx>.

А мы перейдем к созданию нашей БД. В созданном проекте активируйте пункты меню Вид-Обозреватель Серверов (View-Server Explorer) (Рис. 5).

В правой части вашего экрана появится окно Обозреватель Серверов (Server Explorer), в нем вы увидите раздел Подключения данных (Data Connections). Наведите курсор на этот раздел, вызовите контекстное меню и выберите команду Добавить подключение (Connect to Database) (Рис. 6).

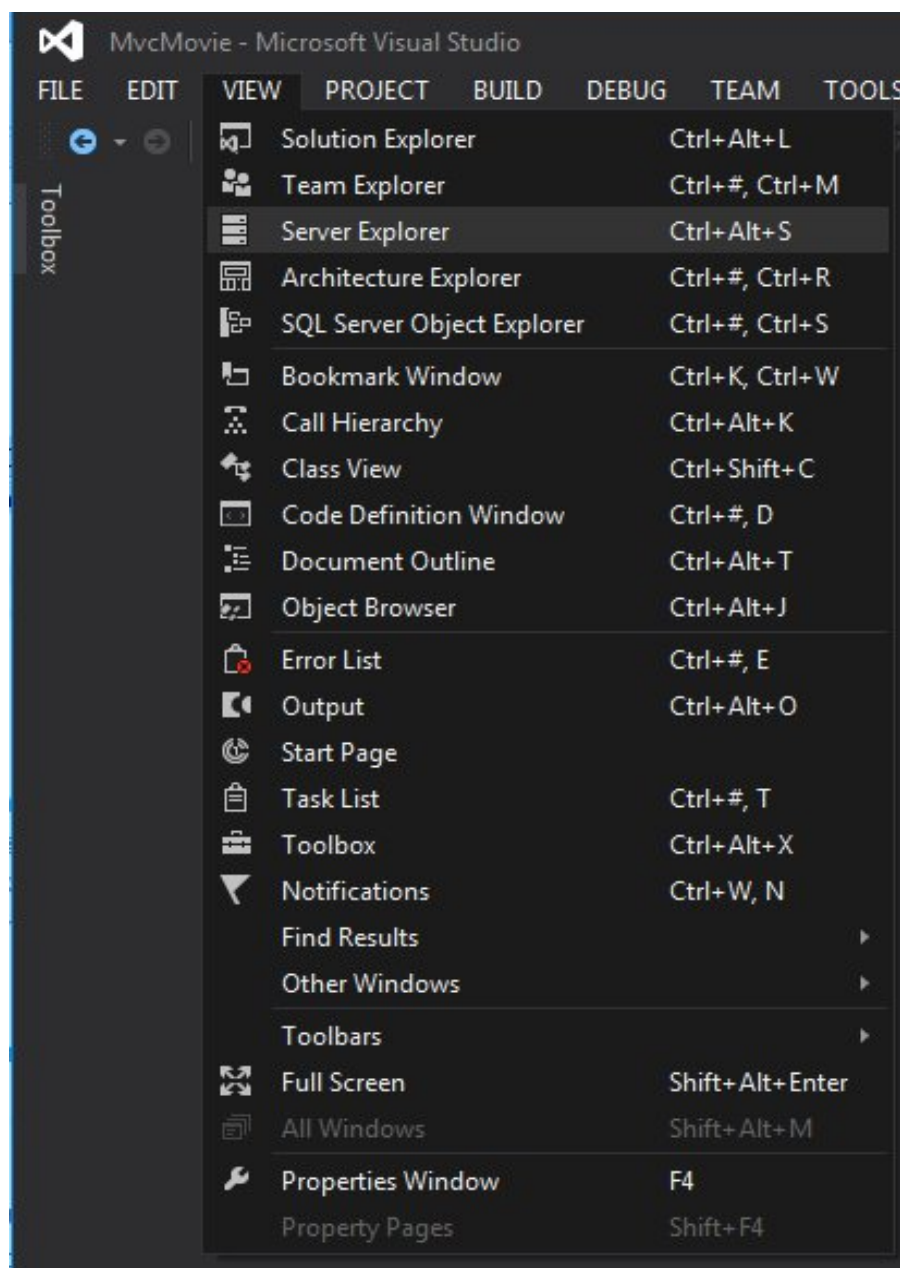


Рис. 5. Выбор Обзорера серверов

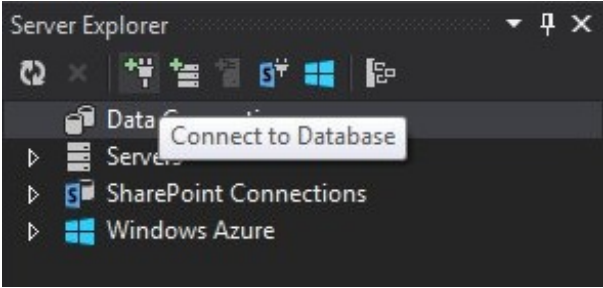


Рис. 6. Создание подключения к БД

В появившемся окне в поле «Источник данных» в качестве источника данных надо указать Microsoft SQL Server (SqlClient). Если у вас указано другое значение, нажмите кнопку Изменить (Change) и выберите это значение. В поле «Имя сервера» необходимо занести значение такое значение — (localdb)\v11.0. Это и есть зарегистрированное имя сервера LocalDB. Его надо указать с точностью до каждого символа, как указано здесь. Правда, регистр символов в значении localdb не важен (Рис. 7).

На этом этапе можете нажать кнопку Проверить подключение (Test connection), чтобы проверить доступность сервера. Затем придумайте имя для создаваемой БД и занесите его в поле «Выберите или введите имя базы данных». Нажмите кнопку ОК.

После выполнения этих действий в разделе Подключения данных (Data Connections) окна Обозреватель Серверов (Server Explorer) появится подключение к созданной вами БД. Разверните этот узел и выделите внутри узел Таблицы (Tables). Пока этот узел пустой, т.к. в БД никаких таблиц нет. Давайте создадим нужные нам таблицы. Это можно сделать двумя способами: в графическом режиме в верхней

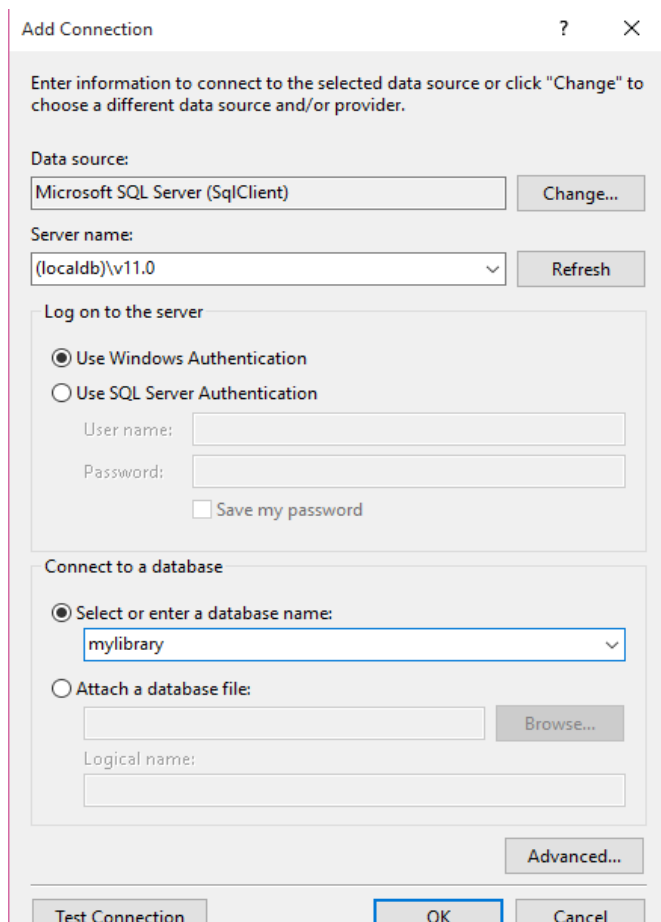


Рис. 7. Выбор сервера и создание БД

части окна или с помощью запроса в нижней части окна. Поскольку у нас уже есть запросы, давайте используем их. Скопируйте запрос для создания таблицы Author и вставьте его поверх заготовки запроса в нижней части окна. Затем нажмите вверху кнопку Обновить, а в следующем окне нажмите кнопку Обновить базу данных. Аналогичным образом создайте две другие таблицы.

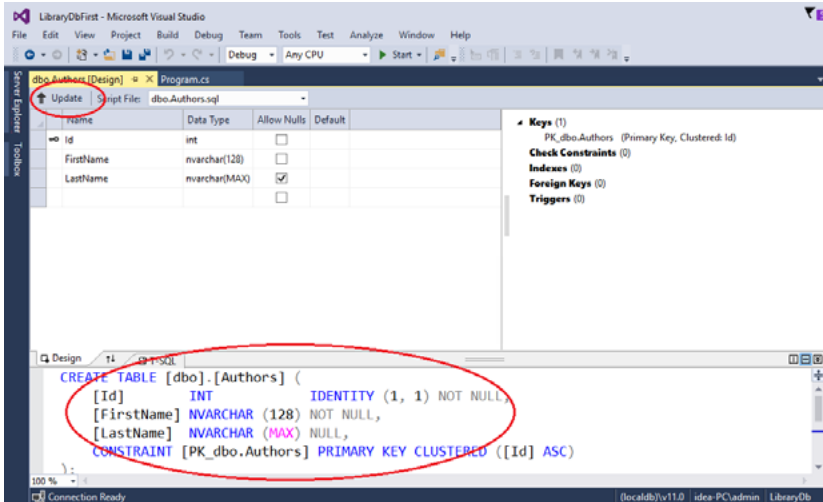


Рис. 8. Создание таблиц БД

Теперь БД готова, и мы можем вернуться к главной теме нашего разговора — Entity Framework.

## Создание EDM для Database first

Сейчас нам надо рассмотреть процесс использования Entity Framework для случая Database first. Что это значит? Это значит, что мы хотим подготовить созданное пустое приложение для работы с уже существующей БД с помощью Entity Framework. Другими словами, мы должны создать в нашем приложении сущности — классы, соответствующие таблицам нашей БД, и класс контекста базы данных для связи с БД и выполнения запросов к БД. Чтобы сделать это, надо выполнить следующие действия:

1. Перейти в обозреватель решения созданного проекта, выбрать там проект и активировать контекстное меню (нажав правую кнопку мыши);

2. Выбрать команду Add-New item (Добавить-Создать элемент);
3. В появившемся окне найти и выбрать элемент ADO.NET EDM Model (Модель ADO.NET EDM) с именем по умолчанию Model1.edmx;
4. Заменить имя на LibraryModel1.edmx и нажать кнопку Add (Добавить);
5. В появившемся мастере создания моделей EDM выбрать опцию Create from database (Создать из базы данных) и нажать кнопку Next (Далее);

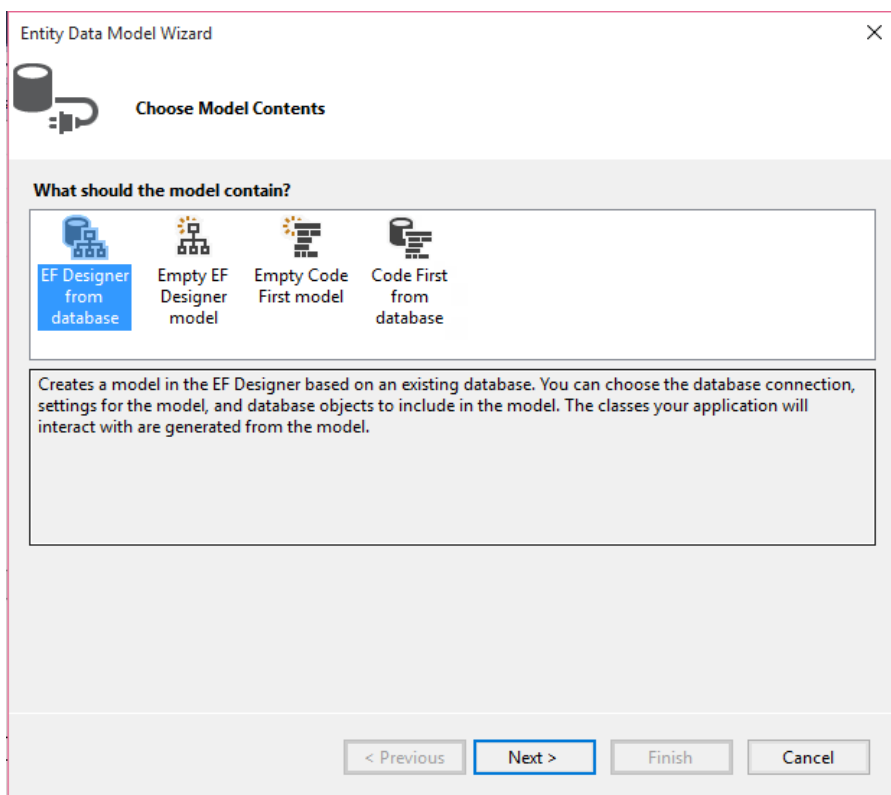
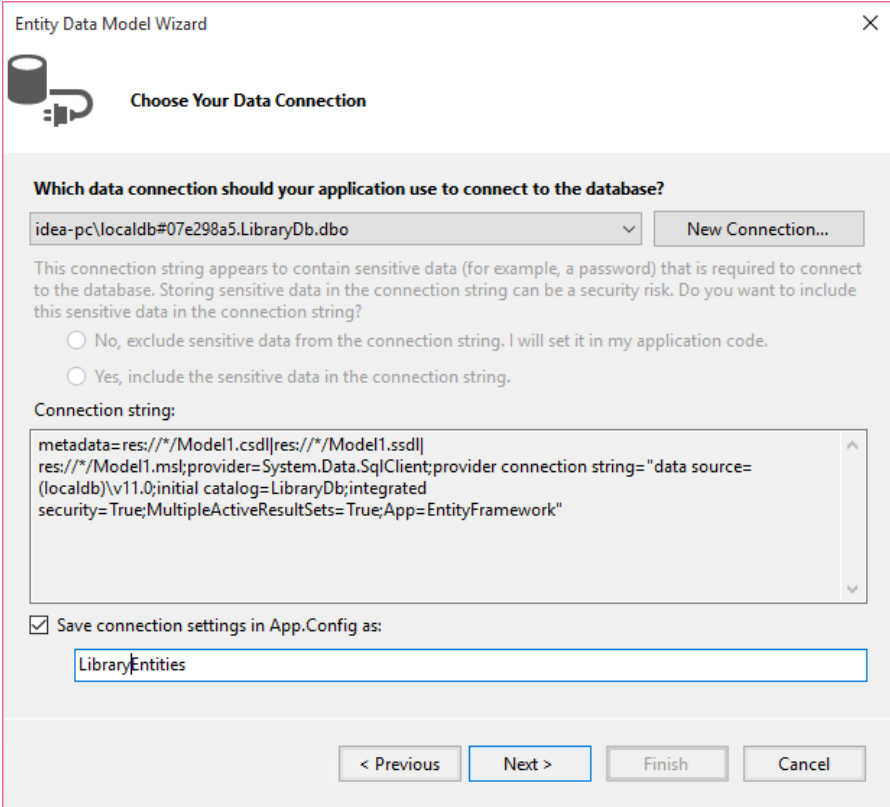


Рис. 9. Создание EDM

6. На этом этапе надо либо выбрать существующее подключение к базе данных, либо создать новое подключение. Мы выбираем существующее подключение к базе данных и указываем в нижнем окне имя LibraryEntities;



The image shows the 'Entity Data Model Wizard' dialog box, specifically the 'Choose Your Data Connection' step. The title bar reads 'Entity Data Model Wizard'. Below the title bar is a toolbar with a database cylinder icon and a 'Choose Your Data Connection' label. The main area contains the question 'Which data connection should your application use to connect to the database?'. A dropdown menu shows 'idea-pc\localdb#07e298a5.LibraryDb.dbo'. To the right is a 'New Connection...' button. Below this, a text block explains that the connection string might contain sensitive data and asks if it should be included. Two radio buttons are present: 'No, exclude sensitive data from the connection string. I will set it in my application code.' (selected) and 'Yes, include the sensitive data in the connection string.' Below the radio buttons is a 'Connection string:' label and a text area containing the connection string: 'metadata=res://\*/Model1.csdl|res://\*/Model1.ssdl|res://\*/Model1.msl;provider=System.Data.SqlClient;provider connection string="data source=(localdb)\v11.0;initial catalog=LibraryDb;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"'. At the bottom, there is a checkbox 'Save connection settings in App.Config as:' which is checked, followed by a text box containing 'LibraryEntities'. At the very bottom are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Рис.10. Создание подключения

7. Жмем Next и получаем доступ к окну, отображенному на рис. 3;
8. В следующем окне вы увидите в древовидном списке структуру своей БД, здесь надо распахнуть узел таблиц,

отметить чекбоксы возле каждой из наших трех таблиц и нажать кнопку Finish;

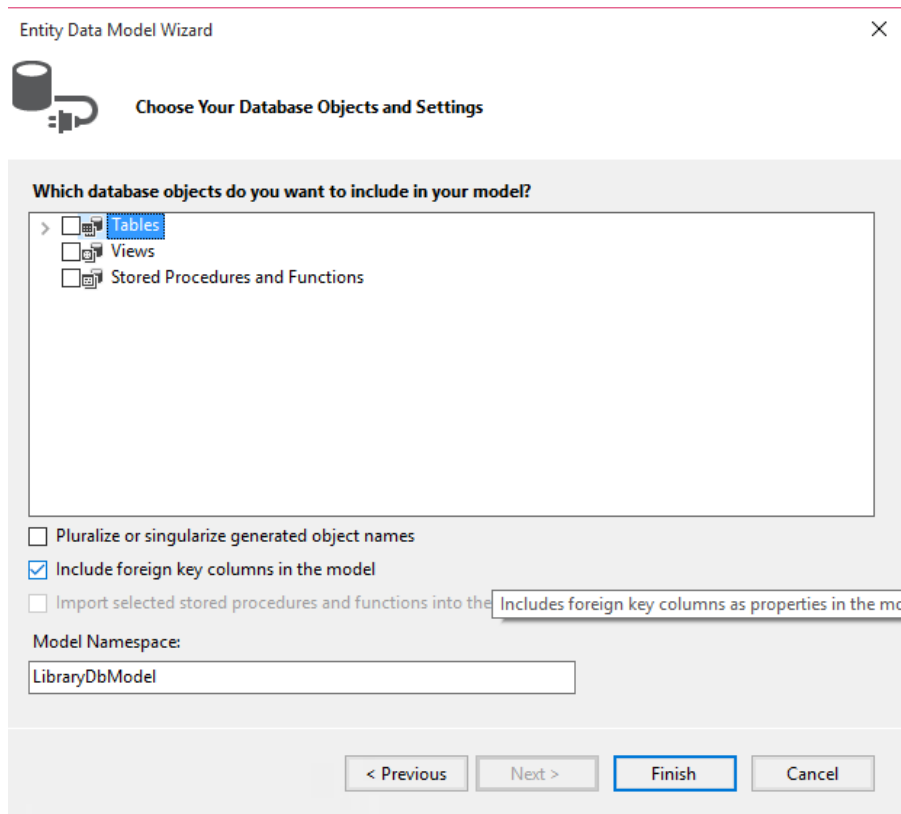


Рис.11. Выбор компонент БД

Теперь Visual Studio отобразит диаграмму вашей БД. У меня это выглядит таким образом (Рис. 12).

После выполнения всех этих действий вы увидите в обозревателе серверов созданное подключение. Вы можете развернуть узел для созданного подключения и увидеть структуру подключенной БД. В обозревателе решений



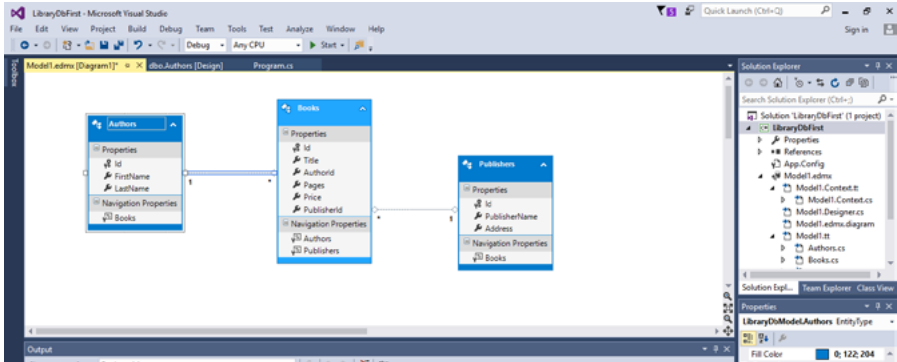


Рис. 12. Диаграмма EDM

появится объект `LibraryModel1.edmx`. Его устройство мы рассмотрим позже.

Итак, мы создали базу данных и консольный проект, в котором установили Entity Framework и создали Entity Data Model для нашей базы данных. Что нам это дает?

Давайте вспомним, что такое Conceptual Model и Storage Model в архитектуре Entity Data Model. Если вы помните, что это такое, то вы должны понимать, что после создания Entity Data Model в составе нашего проекта появились три класса с именами `Author`, `Publisher` и `Book`. Эти классы называются сущностями, и созданы они на основе таблиц БД. Для каждой таблицы БД в составе приложения был создан класс. Поля таблицы превратились в классах в свойства — с такими же именами, как имена полей в таблице, и с соответствующими типами данных. Давайте сейчас выполним в проекте некоторые действия, демонстрирующие возможности Entity Framework, а затем обсудим полученные результаты и поговорим о внутреннем механизме.

Добавьте рядом с методом Main() два следующих метода:

```
static void AddAuthor(Author author)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        db.Author.Add(author);
        db.SaveChanges();

        Console.WriteLine("New author added:" +
            author.LastName);
    }
}

static void GetAllAuthors()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        var au = db.Author.ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+" "+
                a.LastName);
        }
    }
}
```

А в самом методе Main() реализуйте вызов этих методов, например, так:

```
static void Main(string[] args)
{
    Author author = new Author{ FirstName="Isaac",
                                   LastName="Azimov"};

    AddAuthor(author);
    GetAllAuthors();
}
```

Запустите приложение, и вы увидите, что в таблице Author появилась запись «Isaac Azimov». Давайте внимательно рассмотрим созданные методы. Я думаю, что в добавленных методах вас прежде всего заинтересовала строка кода

```
LibraryEntities db = new LibraryEntities()
```

Разверните в обозревателе решений узел LibraryModel1.edmx, внутри него разверните узел LibraryModel1.Context.tt, и, наконец, выделите внутри последнего узла файл LibraryModel1.Context.cs, в котором и определен класс LibraryEntities. Мы с вами поговорили о роли этого класса в Entity Framework, и сейчас вы видите его в составе своего приложения. У меня этот класс выглядит таким образом:

```
public partial class LibraryEntities: DbContext
{
    public LibraryEntities()
        : base("name=LibraryEntities")
    {
    }

    protected override void
    OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

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

Важно отметить, что DbContext наследует интерфейсу IDisposable, поэтому удобно использовать объект контекста в блоке using. Кроме этого, полезно знать, что определение этого класса можно изменять вручную.

Откройте в обозревателе решений конфигурационный файл App.config и найдите в нем строку подключения к БД. И сам конфигурационный файл и строка подключения были созданы Entity Framework. Как вы видите, имя строки подключения соответствует тому, что используется в конструкторе LibraryEntities и передается конструктору базового класса DbContext.

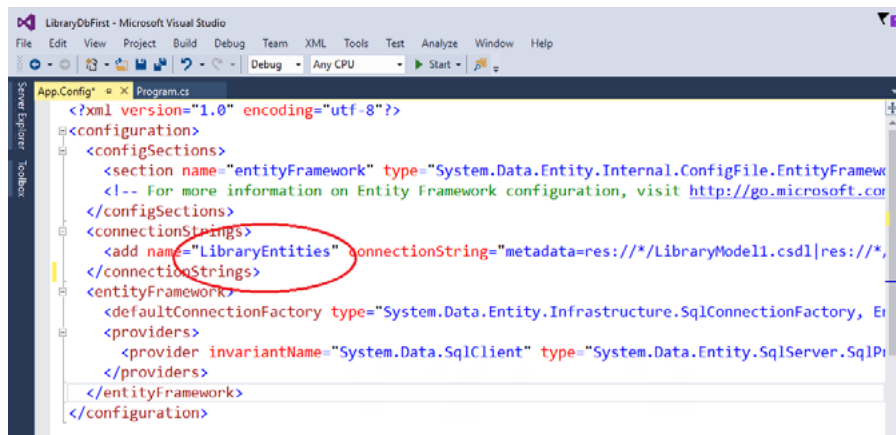


Рис. 13. Файл конфигурации приложения

Как вы понимаете, наши методы AddAuthor() и GetAllAuthors() являются просто иллюстрациями, доказывающими работоспособность Entity Framework. Конечно же, надо немного изменить эти методы, чтобы работа с ними была более комфортной. Например, было бы лучше, чтобы метод AddAuthor() перед добавлением

в БД информации о новом авторе, проверял, нет ли там уже такой информации. Кроме того, было бы полезно иметь в своем распоряжении метод, возвращающий не весь список авторов, а выбранного автора, например по Id. Любое из таких изменений требует от нас написания определенных запросов к БД. Поэтому давайте подробнее рассмотрим Linq to Entities, чтобы научиться с его помощью эффективно выполнять требуемые запросы.

# LINQ to Entities (введение)

## First() и FirstOrDefault()

Все примеры, приведенные ниже, для простоты будем рассматривать применительно к таблице Author в БД и к сущности Author в приложении. Предположим, вы хотите извлечь из таблицы информацию об авторе с именем «Charles», при этом вас устроит первая обнаруженная запись, удовлетворяющая вашему критерию поиска. В этом случае вам следует воспользоваться методами First() или FirstOrDefault(). Вы должны помнить, что, используя Linq, вы можете писать код как в формате запросов, так и в формате методов. Если вы используете формат методов, то все параметры методов необходимо писать с использованием лямбда-выражений. Давайте рассмотрим и обсудим два варианта метода, позволяющего получить из таблицы БД первую запись об авторе с именем, совпадающим со значением параметра fname.

Первый вариант написан в формате запросов:

```
static Author GetAuthorByName(string fname)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var author = (from s in db.Author
                      where s.FirstName == fname
                      select s).FirstOrDefault<Author>();
        return author;
    }
}
```

Второй вариант написан в формате методов:

```
static Author GetAuthorByName1(string fname)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var author = db.Author.Where( (x) =>
            x.FirstName == fname).FirstOrDefault();
        return author;
    }
}
```

Оба метода должны вернуть объект типа Author. Вас не должен вводить в заблуждение тот факт, что тип объекта author мы описали спецификатором var. К расширениям языка C# вы уже должны привыкнуть. Давайте поговорим о методах First() и FirstOrDefault() и об отличие между ними. Оба метода предназначены для выбора из таблицы БД одной единственной (первой встреченной) записи, удовлетворяющей критерию отбора. В нашем случае — s.FirstName == fname. Отличие между этими двумя методами проявляется тогда, когда такой записи в таблице нет. В этом случае First() выбрасывает исключение, а FirstOrDefault() возвращает null.

### Single() и SingleOrDefault()

Предположим, теперь вы хотите извлечь из таблицы информацию об авторе со значением Id равным 2, при этом вы понимаете, что такая запись либо присутствует в таблице в единственном числе, либо отсутствует совсем. Для решения этой задачи можно воспользоваться парой методов Single() или SingleOrDefault(). Эти методы работают

таким образом, чтобы вернуть вам запись, отвечающую заданному критерию отбора, и при этом выполнить проверку того, что такая запись действительно единственная. Давайте предположим, что мы выполняем поиск в таблице с миллионом строк записи с `Id=2`. Предположим, что эта строка в таблице расположена второй от начала таблицы. Если бы мы выполняли поиск методом `First()`, то получили бы результат после обработки второй строки таблицы. Если же использовать метод `Single()`, то он тоже найдет искомую строку после обработки второй строки таблицы. Но этот метод НЕ ПРЕКРАТИТ РАБОТУ после нахождения искомой строки. Он продолжит перебор всего миллиона строк, чтобы убедиться, что найденная запись — единственная, отвечающая критерию отбора. А если он обнаружит еще строку, отвечающую критерию отбора, то выбросит исключение. Отличия между `Single()` и `SingleOrDefault()` заключаются в разном поведении при отсутствии элемента, отвечающего критерию отбора: первый выбрасывает исключение, второй — возвращает `null`. Соответствующие методы в двух формах написания могут выглядеть таким образом:

В формате запросов:

```
static Author GetAuthorById(int id)
{
    using (LibraryEntities db = new
LibraryEntities())
    {
        var author = (from s in db.Author
            where s.Id == id
            select s).Single();
        return author;
    }
}
```



В формате методов:

```
static Author GetAuthorById1(int id)
{
    using (LibraryEntities db = new
        LibraryEntities())
    {
        var author = db.Author.Where( (x) =>
            x.Id == id).SingleOrDefault();
        return author;
    }
}
```

## ToList()

Если вы хотите извлечь из таблицы все строки, отвечающие критерию отбора, можете использовать метод ToList(). Например, мы хотим вывести список всех авторов, фамилия которых начинается с буквы «А».

В формате методов:

```
static void GetAllAuthors()
{
    using (LibraryEntities
        db = new LibraryEntities ())
    {
        var au = db.Author.Where((x) =>
            x.LastName.StartsWith("А")).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+" "
                +a.LastName);
        }
    }
}
```

## В формате запросов:

```
static void GetAllAuthors1()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = (from a in db.Author
        where a.LastName.StartsWith("A")
        select a).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+" "+a.
            LastName);
        }
    }
}
```

## OrderBy()

Если вы хотите отсортировать полученный результат, можете использовать метод `OrderBy()`. Например, мы хотим вывести список всех авторов, отсортированный по фамилиям.

## В формате запросов:

```
static void GetAllAuthors()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = (from a in db.Author orderby
        a.LastName ascending select a).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+"
            "+a.LastName);
        }
    }
}
```

В формате методов:

```
static void GetAllAuthors1()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Author.OrderBy((x) =>
            x.LastName).ToList();
        foreach (var a in au)
        {
            Console.WriteLine(a.FirstName+"
            "+a.LastName);
        }
    }
}
```

## Find()

Если вы хотите найти и получить один конкретный объект, можете использовать метод `Firs()`. Например, мы хотим вывести имя автора, по его идентификатору.

В формате методов:

```
static Author GetAuthorById(int id)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Author.Find(id);
        Console.WriteLine(au.FirstName + " " +
            au.LastName);
        return au;
    }
}
```

В нашем примере метод `Find()` ищет автора по заданному уникальному ключу. Если объект по заданному уникальному ключу не найден, метод `Find()` возвращает `null`. Об этом методе надо знать одну интересную особенность. Вы уже должны

понимать, что между созданием объекта какой-нибудь сущности и добавлением этого объекта в БД может существовать определенный перерыв. Сущность добавляется в БД при вызове метода `db.SaveChanges()`. Так вот, метод `Find()` выполняет поиск объектов не только в БД, но также и в оперативной памяти, т.е. поиск объектов, еще не добавленных в БД.

Знакомства с этими методами нам будет достаточно для создания тех приложений, с помощью которых мы будем знакомиться с использованием Entity Framework. Еще с некоторыми методами мы познакомимся дальше.

# Заполнение БД

Рассмотренных методов достаточно для того, чтобы занести записи во все три таблицы нашей БД. Для этого добавьте в приложение методы, приведенные ниже.

Добавление нового издательства:

```
static void AddPublisher(Publisher publisher)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        Publisher a = db.Publisher.Where((x) =>
            x.PublisherName == publisher.
            PublisherName).FirstOrDefault();
        if (a == null)
        {
            db.Publisher.Add(publisher);
            db.SaveChanges();
            Console.WriteLine("New publisher
                               added:" + publisher.PublisherName);
        }
    }
}
```

Добавление новой книги:

```
static void AddBook(Book book)
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        Book a = db.Book.Where((x) => x.Title ==
            book.Title).FirstOrDefault();
        if (a == null)
        {
            db.Book.Add(book);
        }
    }
}
```

```

        db.SaveChanges();
        Console.WriteLine("New book added:" +
            book.Title);
    }
}

```

## Заполнение всех таблиц:

```

static void Init()
{
    Author author = new Author { FirstName =
        "Ray", LastName = "Bradbury" };
    AddAuthor(author);
    author = new Author { FirstName = "Harry",
        LastName = "Harrison" };
    AddAuthor(author);
    author = new Author { FirstName = "Clifford",
        LastName = "Simak" };
    AddAuthor(author);

    Publisher publisher = new Publisher {
        PublisherName = "Rainbow", Address = "Kyiv" };
    AddPublisher(publisher);
    publisher = new Publisher { PublisherName =
        "Exlibris", Address = "Kyiv" };
    AddPublisher(publisher);
    Book book = new Book { Title = "Way Station",
        IdPublisher = 1, IdAuthor = 4,
        Pages = 350, Price = 85 };
    AddBook(book);
    book = new Book { Title = "Ring Around the
        Sun", IdPublisher = 1, IdAuthor = 4,
        Pages = 420, Price = 99 };
    AddBook(book);
    book = new Book { Title = "The Martian
        Chronicles", IdPublisher = 2, IdAuthor = 2,
        Pages = 410, Price = 105 };
}

```

```
AddBook(book);  
book = new Book { Title = "I, Robot",  
    IdPublisher = 3, IdAuthor = 1,  
    Pages = 378, Price = 100 };  
AddBook(book);  
}
```

Теперь надо реализовать вызов метода Init() в методе Main(), и все таблицы нашей БД будут заполнены, а мы получим возможность углубить понимание принципов работы Entity Framework. Прodelайте указанные действия, перестройте и выполните модифицированное приложение, чтобы заполнить все таблицы.

Давайте поговорим о том приложении, которое мы сделали. Это приложение выполнено по технологии Database First. Вы должны понимать, что совершенно не важно, где создана ваша БД, важно лишь то, что она уже существует и приложение использует существующую БД для создания классов, которые будут соответствовать таблицам БД. Вы должны отметить, что все сущности, класс контекста БД и строка подключения к БД были созданы Entity Framework. Мы, как разработчики, создали только прикладную логику. В нашем случае эта логика очень простая, но вы теперь самостоятельно можете реализовать и другие (use cases) прецеденты.

# Свойства навигации

Теперь давайте посмотрим на информацию, хранящуюся в таблице Book. Для этого можно создать такой простой метод:

```
static void GetAllBooks()
{
    using (LibraryEntities db = new LibraryEntities ())
    {
        var au = db.Book.OrderBy((x) => x.Title).
            ToList();

        foreach (var a in au)
        {
            Console.WriteLine("Book: "+a.Title + "
                                price: " + a.Price + "   author: "+
                                a.Author.FirstName + "      "+a.Author.
                                LastName);
        }
    }
}
```

Добавьте вызов этого метода в Main() и запустите наше приложение. Вы можете не бояться того, что при выполнении метода Init() в БД повторно будет занесена дублирующаяся информация в таблицы Author, Publisher и Book. Наши методы AddXXX() теперь проверяют добавляемую информацию и не допускают ее дублирования в БД.

```
static void Main(string[] args)
{
    Init();
    GetAllBooks();
}
```



При выполнении приложения, вы увидите такое окно:

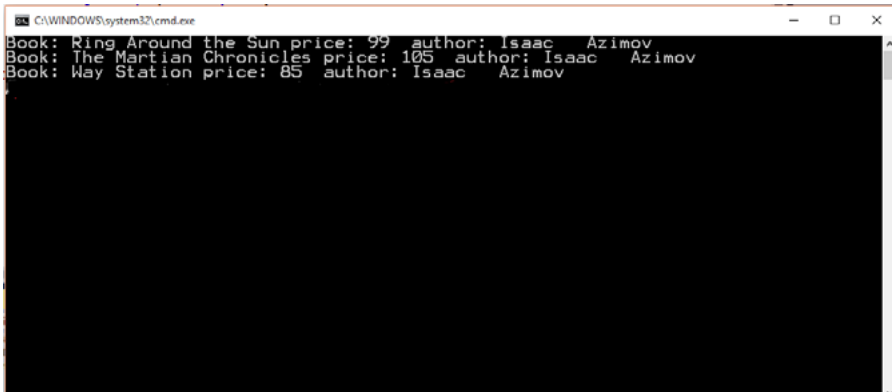


Рис. 14. Вывод метода GetAllBooks()

Посмотрите внимательно на этот рисунок и на код метода GetAllBooks(). Вы не находите ничего странного? Обратите внимание, мы выводим информацию из таблицы Book, но, тем не менее, видим имя и фамилию авторов. Но ведь информации об имени и фамилии автора в таблице Book нет. Эта информация хранится в таблице Author, а в таблице Book хранится только Id автора. Запомните этот момент. Дело в том, что информации об имени и фамилии автора нет в таблице БД с именем Book, но наш запрос работает с сущностью Book, т.е. с классом, созданным на основе таблицы Book. А вот в сущности Book информация об авторе присутствует. Причем вся информация. Поднимитесь немного выше по тексту, к рисунку 4, отображающему диаграмму EDM. На этой диаграмме отображены созданные сущности, и вы можете видеть, что у сущности Book есть так называемые «свойства навигации» — Publisher и Author. Свойства навигации — это данные из связанных таблиц,

которые целиком переносятся и хранятся в сущности. Создаются эти свойства на основании внешних ключей таблицы, т.е. на основе анализа связей между таблицами. Свойства навигации позволяют нам избегать использования многотабличных запросов, чтобы получать данные из связанных таблиц.

Интересно, как свойства навигации описаны в определении сущности? Разверните в обозревателе решений узел LibraryModel1.edmx, затем разверните узел LibraryModel1.tt и, наконец, кликните по файлу Book.cs. У меня определение сущности Book выглядит так:

```
public partial class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int IdAuthor { get; set; }
    public Nullable<int> Pages { get; set; }
    public Nullable<int> Price { get; set; }
    public int IdPublisher { get; set; }

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

Обратите внимание на то, что свойства навигации Author и Publisher описаны как виртуальные. Также интересно отметить описание свойств Pages и Price — их тип не просто int, а Nullable<int>. Это сделано в расчете на тот случай, если подходящего значения в таблице не окажется и придется возвращать null. При этом тип свойства IdPublisher не расширен до Nullable<int>. Это потому,

что данное поле является в БД внешним ключом и, следовательно, не может содержать null. Во всем остальном созданная сущность точно соответствует таблице Book нашей БД.

### Новые термины:

- **Свойства навигации** — это свойства в сущности, содержащие данные из связанных таблиц. Эти свойства создаются Entity Framework автоматически

## Домашнее задание

---

Создайте Windows Forms приложение для работы с нашей БД с использованием Entity Framework по технологии Database First. В главном окне приложения должен содержаться элемент TabControl с тремя вкладками: Books, Authors и Publishers. Каждая вкладка должна обслуживать одну из таблиц нашей БД и выполнять добавление, удаление и редактирование записей в своей таблице. Для добавления, редактирования и удаления записей использовать формы. Для отображения результатов методов GetAllAuthors(), GetAllBooks() и GetAllPublishers() использовать DataGridView.

