Технология доступа к базам данных

ADO.NET

Урок №7

Entity Framework тонкая настройка

Содержание

Entity Framework тонкая настройка

Отложенная загрузка (lazy loading)

К этому моменту вы уже должны понять механизм работы Entity Framework при обращении к БД. А именно — при выполнении запроса к БД фреймворк с помощью созданного контекста БД подключается к БД, читает оттуда требуемую в запросе информацию и заносит прочитанную информацию в соответствующие сущности. Сейчас мы с вами посмотрим, каким образом происходит заполнение сущностей. Entity Framework предоставляет нам несколько разных способов выполнения загрузки данных из таблиц БД в объекты классов приложения (в сущности). Обо всех этих способах загрузки надо знать, чтобы использовать их сильные и слабые стороны в разных конкретных ситуациях.

До сих пор в наших примерах мы использовали одну единственную таблицу Author. Однако чаще всего приходится иметь дело со связанными таблицами. Например, в нашей БД таблица Book связана с таблицами Author и Publisher. Давайте предположим, что, читая таблицу Book, мы также хотим видеть имена и фамилии авторов, а не их идентификаторы. У нас уже есть запрос GetAllBooks(), который отображает информацию и из таблицы Book и из связанной таблицы Author. Мы выяснили, что это

происходит благодаря наличию в сущности Book навигационного свойства Author, в котором содержится информация из одноименной связанной таблицы БД.

Давайте сейчас разберем использование свойств навигации подробнее, а для этого проанализируем, как выполняется наш метод GetAllBooks(). Если вы думаете, что при выполнении этого метода нашей БД был отправлен один запрос, вы ошибаетесь. Сначала был выполнен запрос, сформировавший в оперативной памяти коллекцию названий книг. Однако при этом связанные данные не были загружены в свойства навигации. Связанные данные физически загружаются в свойства навигации только при первом доступе к этим данным. В нашем методе это происходит в цикле foreach в методе Console.WriteLine(), где мы обращаемся к свойству навигации Author.

```
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);
        }
    }
}
```

В этот момент формировался и выполнялся еще один запрос к БД. Этот запрос заполнял свойство навигации Author информацией об авторах для текущей книги из коллекции au, чтобы мы смогли отобразить имя и фамилию автора. Эти действия выполнялись на каждой итерации нашего цикла. Такой способ заполнения свойств навигации выполняется по умолчанию, и называется он «отложенная загрузка» (lazy loading). Запомните отрицательную особенность отложенной загрузки — выполнение большого числа запросов к БД. Правда, надо учитывать и положительную особенность — это простые однотабличные запросы. К положительным качествам отложенной загрузки надо еще отнести тот факт, что код при этом выглядит проще.

Класс DbContext имеет булевское свойство DbContext.Configuration.LazyLoadingEnabled, которое управляет отложенной загрузкой. По умолчанию это свойство равно true и отложенная загрузка активирована. Именно поэтому наш запрос и выполнился так, как мы того ожидали. Однако инициализации одного этого свойства недостаточно для того, чтобы отложенная загрузка работала. Необходимо еще соблюдение двух условий:

- сущность, с которой мы работаем, должна быть public и не должна быть sealed;
- навигационное свойство, к которому мы обращаемся, должно быть virtual;

Попробуйте сейчас в определении сущности Book убрать спецификатор virtual у навигационного свойства

Author и запустите приложение. Вы получите исключительную ситуацию, потому что невиртуальное свойство не поддерживает отложенную загрузку.

> *Давайте поговорим о том, почему свойство навигации должно быть virtual. Дело в том, что во время выполнения вашего кода при использовании отложенной загрузки, Entity Framework не работает с вашим классом сущностью. Он динамически создает прокси-класс, производный от вашей сущности. И в этом прокси-классе Entity Framework переопределяет свойства навигации, чтобы реализовать оптимальный способ извлечения связанных данных. А если свойства навигации не будут virtual, то переопределить их будет невозможно.*

Даже если свойство DbContext.Configuration.LazyLoadingEnabled равно true, отложенная загрузка не будет работать при отсутствии спецификатора virtual у соответствующего навигационного свойства.

У отложенной загрузки есть одна особенность, которая должна вас насторожить. Если вы еще не поняли, то надо повторить: запрос для заполнения свойства навигации создается и выполняется для каждого элемента коллекции, сформированной в главном запросе:

```
var books = db.Book.OrderBy((x) => x.Title).ToList();
```

Другими словами, если у вас в коллекции books будет сто книг, то вам понадобится выполнить сто дополнитель-

ных запросов, чтобы получить имена и фамилии авторов для каждой книги. То есть иногда отложенная загрузка может оказаться не эффективным способом работы с БД. Поэтому рассмотрим альтернативные варианты.

Безотложная загрузка (eager loading)

Безотложная загрузка связанных данных основывается на прямом указании в запросе тех связанных данных, которые необходимо получить. Давайте сконструируем метод, который будет выполнять то же, что и метод GetAllBooks(), но не будет использовать отложенную загрузку.

```
//eager loading
static void GetAllBooks()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        var books = db.Book.Include("Author").
            ToList<Book>();
        foreach (var a in books)
        {
            Console.WriteLine("Книга: " + a.Title + "
            цена: " + a.Price + "  автор: " +
            a.Author.FirstName + "   " +
            a.Author.LastName);
        }
    }
}
```

Применение безотложной загрузки базируется на использовании метода Include(). Именно этот метод позволяет в запросе для одной сущности указать данные

другой, связанной сущности. Вы, конечно же, помните, что все наши запросы, написанные либо в LINQ to Entities, либо в Entity SQL, автоматически переводятся в слое Entity Client data provider в обычные SQL запросы. Так вот, использование метода Include() приводит к созданию SQL запроса, использующего JOIN.

Вверху приведен вариант метода GetAllBooks(), использующего безотложную загрузку и написанного в синтаксисе метода. Однако, как обычно в Linq to Entities, этот метод можно переписать в синтаксисе запроса.

```
//eager loading
static void GetAllBooks()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        var books = (from s in db.Book.
        Include("Author")select s).ToList<Book>();
        foreach (var a in books)
        {
           Console.WriteLine("Книга: " + a.Title + "
           цена: " + a.Price + "  автор: " +
           a.Author.FirstName + "   " +
           a.Author.LastName);
        }
     }
}
```

При использовании безотложной загрузки не происходит увеличения числа обращений к БД, как это имеет место при использовании отложенной загрузки. Это происходит потому, что данные из связанных сущностей загружаются в память при выполнении главного запроса.

Правда, сам запрос при этом несколько сложнее — это многотабличный запрос с инструкцией JOIN. К тому же вы должны избегать применения в своих запросах большого числа вызовов Include(), т.к. в этом случае Entity Framework может сгенерировать «плохо оптимизированный запрос». Какое количество вызовов Include() является «большим» и сколько таких вызовов можно использовать без опасений — нам не сообщают, полагаясь на нашу скромность и благоразумие. Но вы должны понимать, что Include() выбирает все связанные данные и использовать его надо тогда, когда все данные вам действительно необходимы.

Поэтому в каждом конкретном случае надо принимать решение, какой вариант будет более эффективным: либо много простых запросов при отложенной загрузке, либо меньшее число более сложных запросов при безотложной загрузке.

Явная загрузка (explicit loading)

Третий способ загрузки данных из связанных сущностей, который мы рассмотрим, называется «явная загрузка». Он, так же как и безотложная загрузка, базируется на применении некоторых специальных методов. Давайте рассмотрим явную загрузку на примере нашей БД. Если у вас в приложении отложенная загрузка не активирована, вы все равно можете реализовать ее, используя явную загрузку. Но если при использовании отложенной загрузки связанные данные загружаются автоматически, при обращении к ним, то при явной загрузке вы должны загружать их непосредственным вызовом специальных

методов. Явная загрузка обладает некоторыми характеристиками, которые могут сделать ее применение предпочтительным по сравнению с отложенной загрузкой:

- навигационное свойство, данные которого вас интересуют, не обязано быть virtual (иногда у нас просто нет возможности изменять определение сущности, например, чтобы добавить спецификатор virtual);
- если при использовании отложенной загрузки дополнительные запросы генерируются, скажем так, неконтролируемо, то при использовании явной загрузки вы всегда точно знаете, когда и какой запрос создается.

Основные методы, предназначенные для реализации явной загрузки, следующие: Entry(), Reference(), Collection() и Load(). Рассмотрим краткую характеристику каждого из них, а затем применим их в нашем приложении.

Метод Entry() определен в классе DbContext. Этот метод перегруженный и существует в двух вариантах: Entry(Object) и Entry<TEntity>(TEntity). Как видно, он может принимать либо один параметр типа Object, либо один параметр типа сущности TEntity. В любом случае мы передаем этому методу объект сущности. В первом случае метод Entry() возвращает объект типа System.Data.Entity.Infrastructure.DbEntityEntry<TEntity>. Во втором — объект типа System.Data.Entity.Infrastructure.DbEntityEntry. Этот объект, полученный от метода Entry(), предоставляет нам полный доступ ко всей информации о сущности. Речь идет не только о значениях, которые содержатся в свойствах текущей сущности. Информация, предоставляемая нам

методом Entry(), включает в себя состояние сущности (об этом будем говорить позже), а также исходные значения каждого свойства на момент извлечения сущности из БД. Кроме того, объект DbEntityEntry предоставляет нам доступ к некоторым действиям, которые мы можем выполнять с сущностью — в частности, к загрузке данных в навигационные свойства сущности.

Получив объект DbEntityEntry для конкретной сущности, мы можем использовать методы Reference() или Collection(), чтобы добраться к данным для навигационных свойств сущности. Эти методы позволяют управлять выбором данных для навигационных свойств сущности. Чтобы занести данные, отобранные методом Reference() или Collection(), в навигационные свойства сущности используется метод Load().

Зачем нужны два метода, Reference() и Collection(), для выполнения однотипной работы? Есть ли отличие между методами Reference() и Collection()? Да, разница между этими методами есть, и она принципиальна. Когда навигационное свойство сущности связано с одним единственным значением другой сущности, тогда следует использовать метод Reference(). Именно такой случай отображен в методе GetAuthors(), приведенном ниже. Обратите внимание, здесь book — это одна книга, поскольку это значение получено методом FirstOrDefault(). Для этой книги мы хотим получить информацию об авторе. Давайте посмотрим на определение сущности Book, с которой мы работаем. Для этого надо развернуть в обозревателе решений узел LibraryModel1.edmx, затем

в нем развернуть узел LibraryModel1.tt и выбрать файл Book.cs, в котором содержится определение сущности Book. У меня эта сущность 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; }
}
```

Посмотрите на описание свойства навигации Author — это не коллекция, а просто значение. Поэтому запомните: чтобы получить значения для одиночного свойства навигации, надо использовать методы Entry() и Reference(). Сначала для объекта сущности book мы вызываем метод Entry(), а затем, для полученного объекта DbEntityEntry, вызываем метод Reference(), указывая, какие данные нам нужны.

```
//explicit loading для одной книги
static void GetAuthors()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        db.Configuration.LazyLoadingEnabled = false;
        var book = (from b in db.Book where(b.Title.
           StartsWith("w"))select b).
           FirstOrDefault<Book>();
```

```
            db.Entry(book).Reference(a => a.Author).Load();
            Console.WriteLine("Книга: " + book.Title +
                " цена: " + book.Price + " автор: " + book.
                Author.FirstName + "   " + book.Author.
                LastName);
        }
}
```

Теперь рассмотрим другой вариант. Предположим, что нам надо получить информацию обо всех книгах, написанных каким-либо автором. Понятно, что таких книг может быть несколько. В этом случае нам надо работать с сущностью Author и с ее навигационным свойством Book. Сначала посмотрите на определение сущности Author. У меня эта сущность выглядит так:

```
public partial class Author
{
    public Author()
    {
        this.Book = new HashSet<Book>();
    }

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

Посмотрите, как здесь описано свойство навигации Book. Это — коллеция ICollection<Book>. В этой ситуации для получения значений такого свойства надо исполь-

зовать методы Entry() и Collection(), а не Reference(), как в методе GetAuthors(). Этот случай показан в следующем методе GetAllBooks().

```
//explicit loading для коллекции книг
static void GetAllBooks()
{
    using (LibraryEntities db =
       new LibraryEntities())
    {
        db.Configuration.LazyLoadingEnabled = false;
        var author = (from b in db.Author
            where(b.LastName.StartsWith("si"))
            select b).FirstOrDefault<Author>();
        db.Entry(author).Collection("Book").Load();
        foreach (Book book in author.Book)
        {
            Console.WriteLine("Книга: " + book.Title +
              " цена: " + book.Price + "  автор: " +
              author.FirstName + "   " + author.
              LastName);
        }
    }
}
```

Давайте подытожим все о трех способах загрузки данных из связанных сущностей.

- Отложенная загрузка не достает сразу все связанные данные, а доставляет их только тогда, когда выполняется их явный запрос, и только те данные, которые требуются в запросе. Использовать отложенную загрузку имеет смысл тогда, когда вам не нужны все связанные данные, а лишь некоторые из них.

- Безотложная загрузка является противоположностью отложенной загрузки. Она сразу доставляет данные из основной сущности и из связанных сущностей. Этот способ загрузки полезен тогда, когда вам нужны все данные из связанных сущностей.
- Явная загрузка похожа на отложенную. Но она позволяет точно управлять загрузкой связанных данных вызовом метода Load(), т.к. до явного вызова метода Load() данные из связанных сущностей не загружаются.

Почему свойства навигации могут быть либо просто значениями, либо коллекциями, мы поговорим прямо сейчас.

Новые термины:

- ***Lazy loading, eager loading, explicit loading*** — разные способы заполнения данными объектов сущностей

Три вида связей в EDM

Из предыдущего материала вы узнали, что при создании сущностей и свойств навигации в них Entity Framework может создавать свойства навигации по-разному — либо как одиночное значение, либо как коллекцию ICollection< >. От чего это зависит? Это зависит от типа связи между таблицами БД, для которых созданы сущности. Чтобы проверить это, надо посмотреть на диаграмму БД, созданную Entity Framework. Кликните по узлу LibraryModel1.edmx — и вы увидите графическую диаграмму нашей БД.

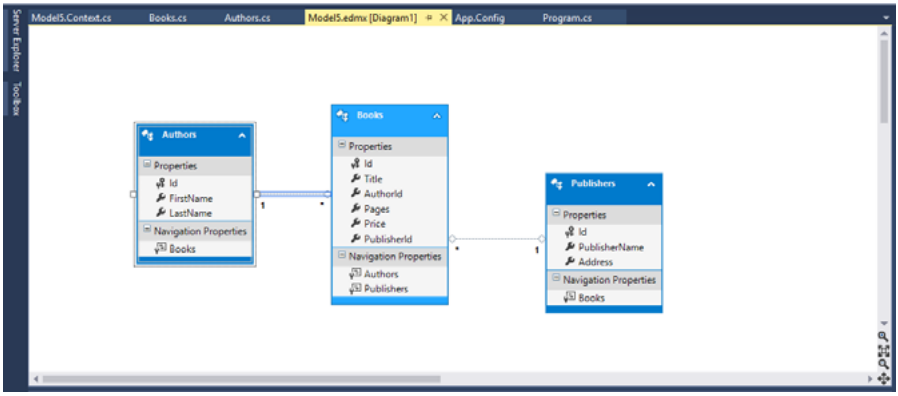

Рис.1. Диаграмма БД

На этой диаграмме отображены сущности и связи между ними. В нашей БД присутствуют только связи типа «один ко многим». Кроме этого еще могут быть связи типа «один к одному» и «многие ко многим». Посмотрим, каким образом создаются свойства навигации в сущностях для каждого из трех указанных типов связей.

Как показывает наша диаграмма, между сущностями Author и Book существует связь типа «один ко многим». В сущности Author эта связь реализуется первичным ключом Id, а в сущности Book — внешним ключом IdAuthor. И вот теперь вступает в действие Entity Framework. Чтобы иметь возможность в сущности Author хранить связанные данные из сущности Book, Entity Framework создал в сущности Author навигационное свойство в виде ICollection< >. Коллекция нужна потому, что одной сущности Author может потребоваться хранить несколько сущностей Book.

Посмотрим на эту же связь с другой стороны, со стороны другой сущности. В нашей БД у одной книги может быть только один автор. Такое приближение не соответствует действительности, но мы договорились, что наша БД будет максимально упрощенной. Поэтому, со стороны сущности Book, каждой сущности Book может соответствовать только одно значение сущности Author. И поэтому навигационное свойство Author, предназначенное для хранения связанных значений сущности Author, создано в виде простого значения.

Для случая связи «один к одному» навигационные свойства в обеих связанных сущностях являются простыми значениями. В нашей БД таких связей нет, но вы легко можете представить такую связь.

Суммируем все о видах навигационных свойств в зависимости от типов связей между таблицами:

- ***связь «один к одному»:*** соответствующие навигационные свойства представляют собой простые значения;

- ***связь «один ко многим»:*** в одной сущности используется внешний ключ, а в другой — коллекция типа *ICollection<T>:*
- ***связь «многие ко многим»:*** обе сущности содержат коллекции *ICollection<T>:*

Жизненный цикл сущности

Когда мы посредством объекта контекста БД выполняем необходимые запросы, мы изменяем состояния сущностей как задействуемых в запросах напрямую, так и связанных с ними. В течение жизненного цикла каждая сущность характеризуется определенным состоянием. Эти состояния определены в перечислении System.Data.EntityState, которое содержит следующие значения:

```
public enum EntityState
{
    Detached = 1,
    Unchanged = 2,
    Added = 4,
    Deleted = 8,
    Modified = 16,
}
```

Рассмотрим случаи явного использования разных состояний. При работе с сущностями важную роль выполняет метод SaveChanges(), предназначенный для синхронизации состояния сущности и соответствующих таблиц в БД. Однако этот метод выполняет синхронизацию в зависимости от состояния сущности. Сущность может находиться в одном из таких состояний:

- *Состояние **Detached** — сущность не отслеживается контекстом БД (не присоединена к контексту).*
- *Состояние **Unchanged** — сущность отслеживается контекстом БД и находится в БД, и значения всех ее свойств не изменены по сравнению со значениями в БД.*
- *Состояние **Added** — сущность уже отслеживается (управляется) контекстом БД, но еще не добавлена в БД.*
- *Состояние **Deleted** — сущность еще отслеживается контекстом БД и еще находится в БД, но она уже была помечена на удаление и будет удалена при следующем вызове метода SaveChanges().*
- *Состояние **Modified** — сущность отслеживается контекстом БД и находится в БД, но значения некоторых ее свойств были изменены по сравнению со значениями в БД.*

Так вот, метод SaveChanges() совсем не обрабатывает сущности в состоянии Unchanged. Сущности в состоянии Added добавляются в БД, после чего метод SaveChanges() изменяет состояние сущностей в Unchanged. Сущности в состоянии Modified изменяются в БД, после чего метод SaveChanges() изменяет состояние сущностей в Unchanged. Сущности в состоянии Deleted удаляются из БД, после чего отсоединяются от контекста БД, т.е. их состояние переводится в значение Detached. Рассмотрим использование состояний сущностей. С сущностью удобно работать, когда она присоединена к контексту БД. И все, что выше было сказано о методе SaveChanges(), справедливо только для сущностей, присоединенных к контексту БД, т.е. для сущностей, состояние которых не равно Detached.

Рассмотрим все эти состояния в коде.

*Состояние **Detached** — сущность не отслеживается контекстом БД (не присоединена к контексту).*

Если вы создали новый объект сущности, то он еще не присоединен к контексту БД и не управляется им. Состояние такого объекта есть Detached.

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = new Author { FirstName = "Ray",
       LastName = "Bradbury" };
    var authorEntry = db.Entry<T>(author); //получаем
    //объект DbEntityEntry
    //для созданной сущности author, чтобы посмотреть
    //на состояние author
    Console.WriteLine(authorEntry.State)
    //EntityState.Detached
}
```

Другая ситуация, когда сущность может находиться в состоянии Detached — удаление сущности. После выполнения удаления состояние сущности становится Detached.

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = db.Authors.Find(1); //находим
    //автора с Id=1
    db.Authors.Remove(author); //помечаем на удаление
    //автора с Id=1
    db.SaveChanges(); //удаляем автора с Id=1 и
    //выставляем его состояние в Detached
}
```

Эти же действия можно выполнить таким способом:

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = db.Authors.Find(1); //находим
    //автора с Id=1
    db.Entry(author).State = System.Data.EntityState.
    Deleted;
    db.SaveChanges(); //удаляем автора с Id=1 и
    //выставляем его состояние в Detached
}
```

*Состояние **Unchanged** — сущность отслеживается контекстом БД и находится в БД, и значения всех ее свойств не изменены по сравнению со значениями в БД.*

Сущность будет находиться в состоянии Unchanged после вызова метода Attach() или после явного выставления состояния в значение Unchanged. Метод Attach() используется для присоединения к контексту БД существующих значение из БД. Посмотрим, как можно присоединить к контексту БД существующую в базе данных информацию. До сих пор для этого мы адресовали к БД запрос, позволявший получить в контекст БД требуемую информацию. Предположим, вы знаете, что в БД есть запись о каком-то писателе, но к контексту БД эта запись не присоединена. Вы можете присоединить эту запись таким образом:

```
//создаем сущность, соответствующую существующей
//записи в БД
Author author = new Author { id = 4, FirstName =
  "Ray", LastName = "Bradbury" };
using (LibraryEntities db = new LibraryEntities())
```

```
{
    db.Authors.Attach(author); //присоединяем
    //сущность к контексту БД, ее состояние есть Detached
    db.SaveChanges(); //в БД изменения не происходят,
    //а состояние сущности выставляется Unchanged
}
```

Надо отметить, что в этом случае при вызове метода SaveChanges() с БД никаких изменений не произойдет, т.к. сущность находится в состоянии Unchanged. При этом, если присоединенная сущность имеет ссылки на другие сущности, не присоединенные к контексту, эти связанные сущности также присоединяются к контексту БД и получают состояние Unchanged.

Присоединение существующей сущности к контексту БД можно выполнить и другим способом, явно используя состояние сущности:

```
Author author = new Author { id = 4, FirstName =
"Ray", LastName = "Bradbury" };
using (LibraryEntities db = new LibraryEntities())
{
    db.Entry(author).State = System.Data.EntityState.
      Unchanged;
    db.SaveChanges();
}
```

*Состояние **Added** — сущность уже отслеживается (управляется) контекстом БД, но еще не добавлена в БД.*

Рассмотрим добавление новой сущности в контекст БД таким способом:

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = new Author { FirstName =
    "Ray", LastName = "Bradbury" };
    db.Authors.Add(author); //сущности присоединяется
    //к контексту, а ее состояние есть Added
    db.SaveChanges(); //сущность добавлена в БД,
    //а ее состояние изменяется на Unchanged
}
```

В данном случае для добавления данных в сущность используется метод Add() класса DbSet, а для синхронизации сущности с БД — метод SaveChanges(). Именно так мы поступали в наших примерах. Однако выполнить эти же действия можно и другим способом, явно используя состояние сущности:

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = new Author { FirstName = "Ray",
      LastName = "Bradbury" };
    db.Entry(author).State = System.Data.EntityState.
      Added;
    db.SaveChanges();
}
```

Как видите, в этом случае мы явным образом изменяем состояние сущности на значение Added, после чего вызываем метод SaveChanges(), который добавит сущность author в БД и изменит ее состояние на Unchanged.

*Состояние **Deleted** — сущность еще отслеживается контекстом БД и еще находится в БД, но она уже была помечена на удаление и будет удалена при следующем вызове метода SaveChanges().*

Сущность может находиться в состоянии Deleted либо после вызова метода Remove(), либо после явной инициализации состояния значением Deleted. Мы уже видели это в предыдущем примере.

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = db.Authors.Find(1);
    //находим автора с Id=1
    db.Authors.Remove(author);
    //помечаем на удаление автора с Id=1
    db.SaveChanges();
    //удаляем автора с Id=1 и выставляем его
    //состояние в Detached
}
```

Эти же действия можно выполнить таким способом:

```
using (LibraryEntities db = new LibraryEntities())
{
    Author author = db.Authors.Find(1);
    //находим автора с Id=1
    db.Entry(author).State = System.Data.EntityState.
    Deleted;
    db.SaveChanges(); //удаляем автора с Id=1 и
    //выставляем его состояние в Detached
}
```

*Состояние **Modified** — сущность отслеживается контекстом БД и находится в БД, но значения некоторых ее свойств были изменены по сравнению со значениями в БД.*

Использовать метод Attach() для присоединения существующей сущности к контексту БД можно только

в том случае, если вы уверены, что с присоединяемой сущностью не выполнялось никаких изменений. Если же с такой сущностью могли выполняться какие-либо изменения, то присоединять ее к контексту устройства надо таким образом:

```
Author author = new Author { id = 4,
    FirstName = "Ray", LastName = "Bradbury" };
using (LibraryEntities db = new LibraryEntities())
{
    db.Entry(author).State = System.Data.EntityState.
    Modified;
    db.SaveChanges();
}
```

В этом случае сущность будет присоединена к контексту, а затем, при вызове метода SaveChanges(), синхронизирована с БД.

За последними примерами проглядывает следующая аналогия. Вызов метода Add() — это добавление новой записи (Insert), а вызов метода Attach() — это изменение существующей (Update).

Соглашения Entity Framework

Entity Framework, как и многие другие фреймворки, автоматически выполняет большой объем рутинной работы. Однако для этого фреймворк предлагает пользователю придерживаться некоторых соглашений. Давайте поговорим об этих соглашениях. Надо отметить, что соглашения могут зависеть от способа создания EDM.

Например, при использовании подхода «код сначала», имена таблиц, создаваемых фреймворком, соответствуют именам классов. Такое поведение можно переопределить путем использования специальных атрибутов. Мы рассмотрим, как это делать, в уроке, посвященном технологии «код сначала». Свойства класса с именами Id или ИмяКлассаId (регистр при этом не учитывается), считаются первичными ключами создаваемых таблиц. Внешние ключи таблиц описываются двумя свойствами:

1) обычным свойством с именем ИмяСвязанной Таблицы+ИмяПервичногоКлючаСвязаннойТаблицы;
2) навигационным свойством для хранения связанных значений.

Как и всегда с соглашениями, их соблюдение не является обязательным. Но если все же соблюдать соглашения, то работа с фреймворком будет более продуктивной.

Отслеживание действий Entity Framework

Согласитесь, что очень полезно было бы иметь механизм, позволяющий отслеживать все действия фреймворка с БД. Для того чтобы была возможность вести такой лог, даже было разработано несколько инструментов сторонними разработчиками. Но, начиная с Entity Framework 6.0, необходимость в таких инструментах отпала, поскольку в самом Entity Framework теперь встроен замечательный механизм ведения лога. Давайте ознакомимся с этим механизмом, позволяющим отслеживать все действия, выполняемые контекстом БД. Здесь все очень просто. У контекста есть свойство Database, у которого в свою очередь, есть свойство

Log. Тип этого свойства Action<string>. Если вы хотите управлять процессом логгирования, все что вам надо сделать — это создать метод с одним стринговым параметром и типом возвращаемого значения void и инициализировать адресом этого метода свойство контекста database.Log. Теперь все действия, выполняемые контекстом БД, будут записываться в лог этим вашим методом.

Измените наш метод AddAuthor(), добавив в блок using одну строку, инициализирующую указанное свойство:

```
static void AddAuthor(Author author)
{
    using (LibraryEntities db = new LibraryEntities())
    {
        db.Database.Log = Console.Write;
        Author a = db.Author.Where((x) =>
          x.FirstName == author.FirstName &&
          x.LastName == author.LastName).
          FirstOrDefault();
        if (a == null)
        {
            db.Author.Add(author);
            db.SaveChanges();
            Console.WriteLine("New author added:" +
              author.LastName);
        }
    }
}
```

Теперь перестройте и выполните наше приложение. Вы получите в консольном окне что-то подобное рисунку 2. Там вы увидите описание всех подключений к БД, выполненных запросов, отключений от БД, статусов за-

вершения действий и т.п. Очень часто такая информация может быть чрезвычайно полезной, особенно на этапе отладки приложений.

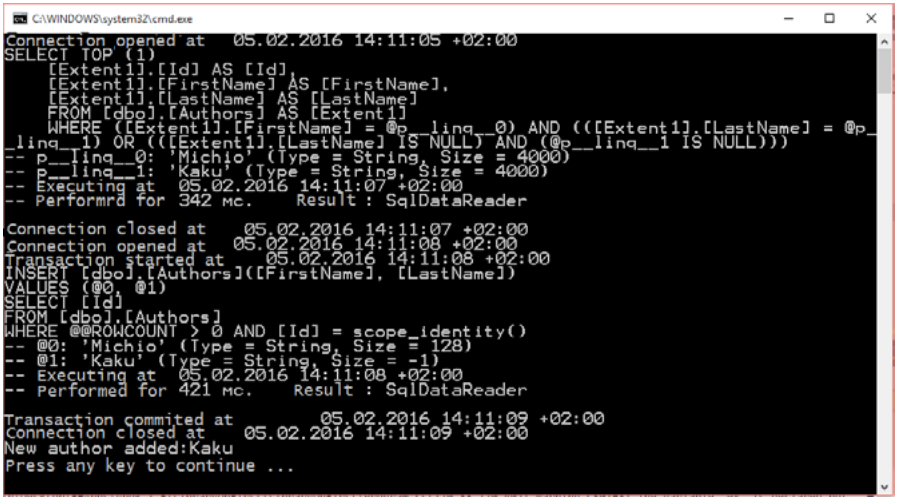

Рис. 2. Лог нашего приложения методом Console.Write()

Если вас не устраивает результат работы метода Console.Write(), вы можете написать свой собственный метод для ведения лога. Единственное требование к такому методу — соответствие делегату Action<string>. Например, можно добавить в наше приложение такой класс:

```
public class MyLogger
{
    public static void EFLog(string message)
    {
        Console.WriteLine("Action performed: {0} ",
message);
    }
}
```

Как вы видите, здесь снова используется метод Console.WriteLine(), но это просто потому, что мы работаем в консольном приложении. Теперь надо заменить строку инициализации свойства Database.Log в методе AddAuthor() на такую строку:

```
db.Database.Log = MyLogger.EFLog;
```

Если вы запустите наше приложение теперь, то лог будет выглядеть так:

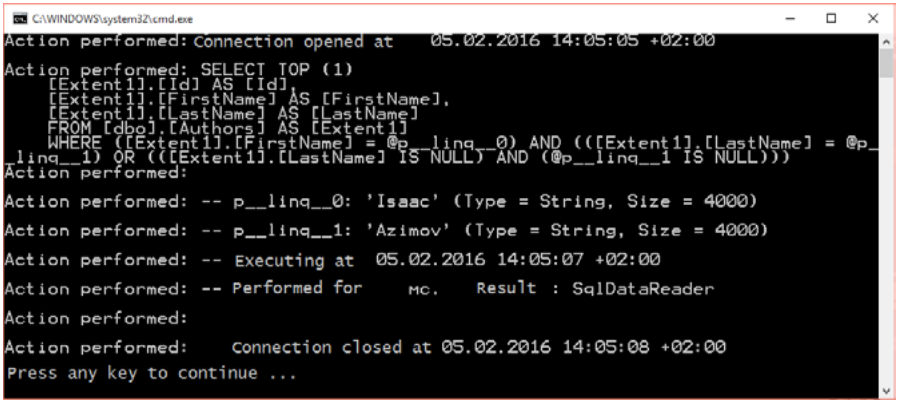

Рис. 3. Лог нашего приложения методом MyLogger.EFLog ()

Поддержка транзакций в Entity Framework

Вы должны понимать, что при работе с БД очень часто необходимо иметь возможность выполнять действия в режиме транзакций. Entity Framework поддерживает механизм транзакций, и сейчас мы рассмотрим, как этим механизмом пользоваться. Изначально Entity Framework по умолчанию выполняет

такие операции, как Insert, Update и Delete, в режиме транзакции, при выполнении метода SaveChanges(). Хотя «транзакционности» метода SaveChanges() бывает достаточно в большинстве случаев, разработчики Entity Framework решили добавить механизм, позволяющий более тонко управлять транзакциями. Такой механизм появился, начиная с версии Entity Framework 6. Этот механизм реализован методами Database.BeginTransaction() и Database.UseTransaction() и позволяет, в частности, выполнять в рамках одной транзакции несколько разных действий.

Добавим в наше приложение метод MyTransaction(), в котором и продемонстрируем выполнение нескольких действий в одной транзакции. Главным для нас будет не то, какие действия выполняются в этом методе, а то, как реализовать транзакцию. Метод может выглядеть таким образом:

```
static void MyTransaction()
{
    using (LibraryEntities db = new LibraryEntities())
    {
        using (System.Data.Entity.DbContextTransaction
               dbTran = db.Database.BeginTransaction())
        {
            try
            {
                Author author = new Author {
                   FirstName = "Stanislaw",
                   LastName = "Lem" };
                db.Author.Add(author);
```

```
                db.Author.Remove(author);
                db.SaveChanges();
                dbTran.Commit();
            }
            catch (Exception ex)
            {
                dbTran.Rollback();
            }
        }
    }
}
```

Весь код, включенный в блок try, будет выполняться в рамках одной транзакции. В случае нормального завершения всех действий в try блоке, управление дойдет до метода dbTran.Commit(), который и выполнит транзакцию. При возникновении исключительной ситуации — выполнится метод dbTran.Rollback(), который откатит транзакцию. Все, что надо реализовать для такой множественной транзакции — это блок using для переменной типа DbContextTransaction, которую возвращает метод Database.BeginTransaction(). Именно в классе DbContextTransaction и реализованы методы Commit() и Rollback().

Как вы видите из этого примера, транзакция начинается и завершается в нашем контексте БД. Кроме того, Entity Framework поддерживает работу с транзакциями, начатыми вне Entity Framework. Для этого используется метод Database.UseTransaction(). Мы не будем рассматривать такие транзакции в ходе нашего разговора.

Наследование в Entity Framework

Одним из базовых принципов ООП является наследование — создание новых классов на основе уже существующих. Entity Framework должен отражать иерархии классов при наследовании. Как можно поеазать в БД, что одна сущность является производной от другой сущности? Entity Framework применяет для этого три способа. Давайте предположим, что в модели нашего приложения существуют классы, объединенные наследованием. Например, класс Point (*точка*), являющийся базовым, и два производных от него класса: Rectangle (*прямоугольник*) и Circle (*окружность*). В классе прямоугольника точка описывает позицию левого верхнего угла, а в классе окружности точка описывает позицию центра.

```
abstract public class Point
{
    protected double x;
    protected double y;
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public abstract double Area();
    public abstract double Perimeter();
}

abstract public class Point
{
    protected double x;
    protected double y;
   public Point(double x, double y)
```

```
    {
        this.x = x;
        this.y = y;
    }
    public abstract double Area();
    public abstract double Perimeter();
}

public class Rectangle : Point
{
    protected double width;
    protected double height;
    public Rectangle(double x, double y,
      double width, double height): base(x,y)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area()
    {
        return this.width* this.height;
    }

    public override double Perimeter()
    {
       return 2 * (this.width + this.height);
    }
}

public class Circle : Point
{
    protected double radius;
    protected double height;
    public Circle(double x, double y,
      double radius) : base(x, y)
    {
        this.radius = radius;
    }
```

```
    public override double Area()
    {
        return Math.PI * this.radius * this.radius;
    }

    public override double Perimeter()
    {
        return Math.PI * this.radius * 2;
    }
}
```

Первый способ отображения такой иерархии классов в Entity Framework называется Table per concrete type (или TPC). Это можно перевести как «таблица для каждого конкретного типа». Обратите внимание на слово «конкретного». Дело в том, что для абстрактных классов при этом таблицы не создаются. При таком подходе для каждого класса в БД будет создана отдельная таблица, и мы в итоге получим три таблицы.. Это приведет к тому, что в некоторых случаях, когда нас будут интересовать все фигуры (и прямоугольники и окружности), в запросе будут задействованы все три таблицы.

При TPC отображении для абстрактных классов таблицы не создаются. Если в вашем базовом абстрактном классе объявлены какие-либо свойства, то эти свойства будут перенесены в производные классы. Надо отметить, что отображение наследования способом TPC поддерживается только при подходе код сначала (Code First). Мы вернемся к этому вопросу в будущем.

Второй способ отображения в БД связанных классов называется Table per type (или TPT), что можно перевести

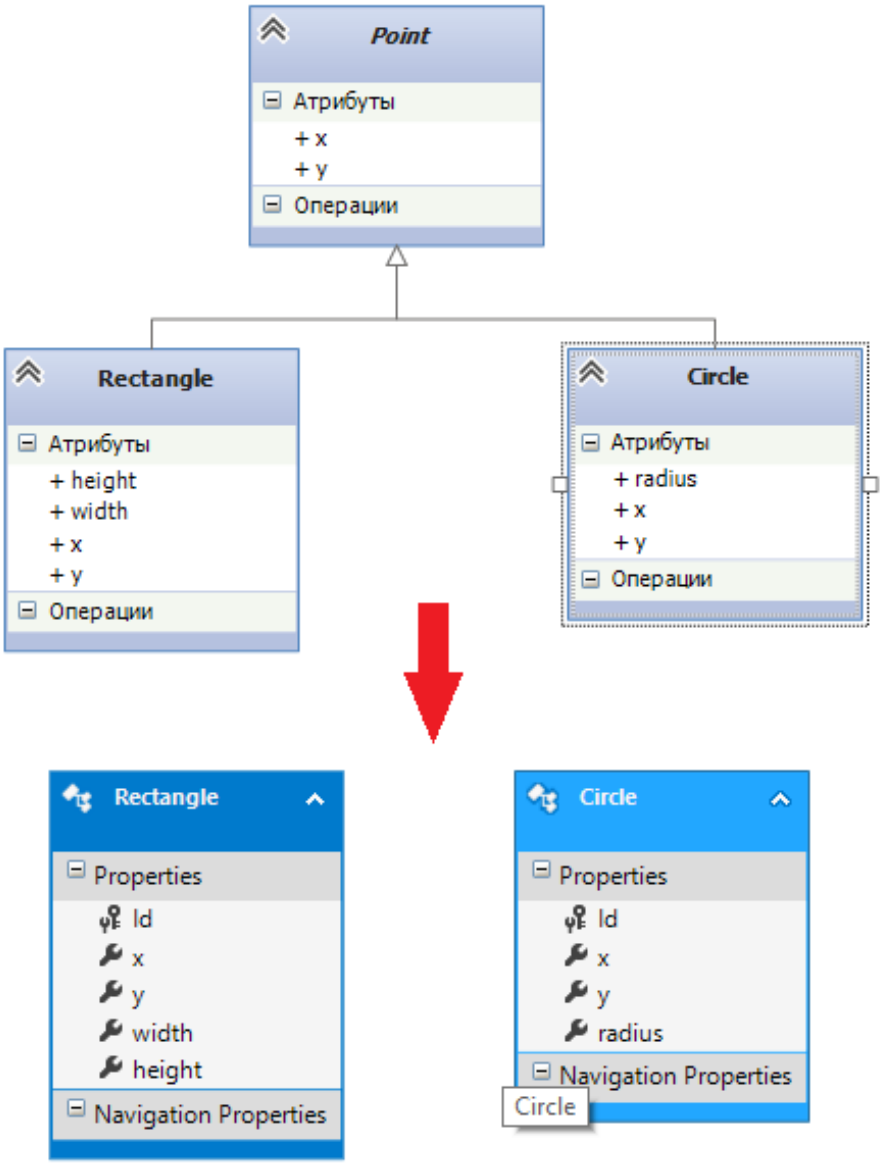

Рис. 4. Диаграмма TPC отображения

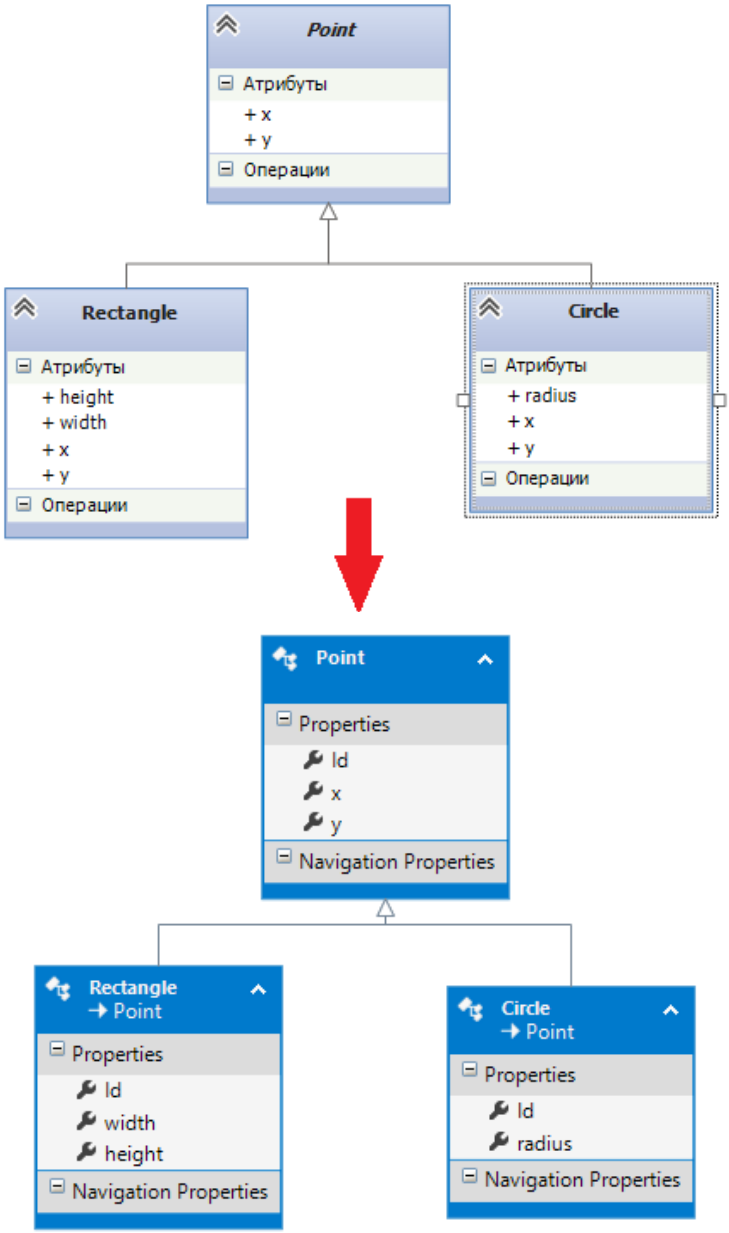

Рис. 5. Диаграмма TPT отображения

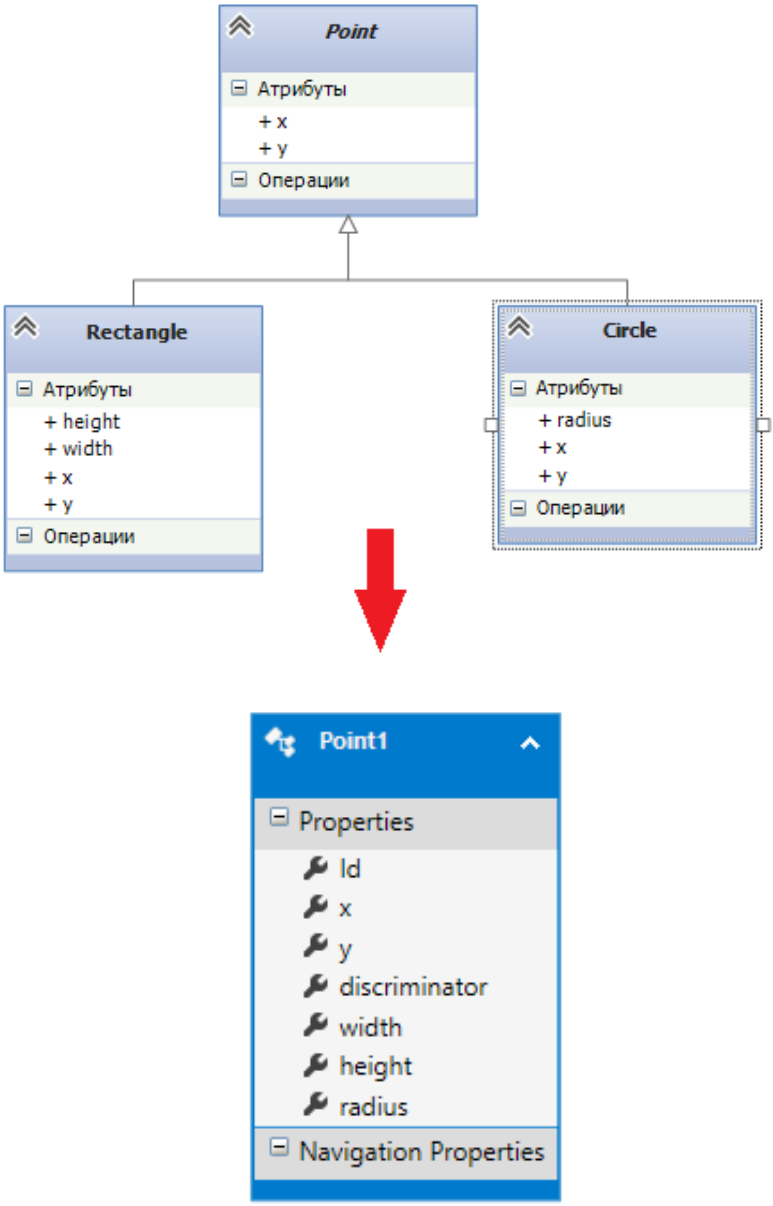

Рис. 6. Диаграмма TPH отображения

как «таблица для каждого типа». В этом случае для нашего примера будет создано три таблицы. Одна таблица будет содержать информацию, соответствующую базовому классу Point и общие для всех классов данные. А дополнительные таблицы, соответствующие классам Rectangle и Circle, будут содержать только данные, специфичные для каждого производного класса. Такой подход удобен в том случае, когда чаще всего при обращении к БД будет требоваться информация, относящаяся к базовому типу (рис. 5).

Третий способ называется Table per hierarchy (или TPH), что переводится как «таблица для иерархии». Этот способ заключается в том, что фреймворк создает одну таблицу для своей группы связанных классов. В такой таблице создается специальное поле Discriminator, в котором указывается, к какому из классов относится данная строка. При таком подходе необходимо следить за тем, чтобы все добавленные свойства производных классов принадлежали к nullable типам (рис. 6).

Новые термины:

- ***Table per concrete type, Table per type, Table per hierarchy*** — разные способы отображения наследования в БД, созданных Entity Framework

Домашнее задание

Создайте Windows Forms приложение для работы с нашей БД с использованием Entity Framework по технологии Database First. В главном окне приложения должно содержаться текстовое поле с именем tbFind, кнопка btFind (изначально неактивная) и элемент DataGridView.

В поле tbFind пользователь должен вводить поисковый контекст, т.е. строку, содержащую часть фамилии автора либо полную фамилию автора. После заполнения поля tbFind должна стать активной кнопка btFind. При нажатии на эту кнопку в DataGridView должны быть отображены книги всех авторов, фамилии которых совпадают с введенным поисковым контекстом. Все запросы в приложении написать с использованием LINQ в формате запросов.