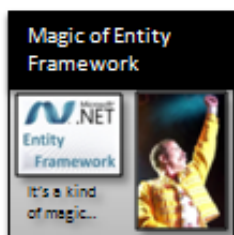


Entity Framework Code First — 2; попытка разобраться как работает эта магия



В данной заметке я постарался рассмотреть Entity Framework и подход к проектированию Code First с точки зрения человека, который не имел дело с ORM и Entity Framework и пытается подробно разобраться, как это все работает.

При работе с БД всегда возникает вопрос, как организовать эту самую работу и где размещать бизнес логику. Способов много, но один умный дядька собрал все воедино, классифицировал, углубил и обострил — и выпустил книгу [Архитектура корпоративных программных приложений](#). С тех пор, в холиварах по размещению бизнес логики появились конкретные термины, в частности два наиболее распространенных из них: [Transaction Script](#) (логика в процедурах) и [Domain Model](#) (логика в объектах модели). Не будем здесь обсуждать, какой поход лучше, только скажу, что мое мнение — обсуждать инструмент в отрыве от задачи бессмысленно.

До недавних пор почти всю логику мне приходилось писать в хранимых процедурах на сервере, но настало время посмотреть более подробно, что же твориться во «вражеском» лагере. А творятся там интересные вещи, работа ведется с объектами, а объекты стыкуются с БД при помощи средств объектно-реляционного отображения (Object Relational Mapping — [ORM](#)). Одной из систем ORM является Entity Framework (EF) от Microsoft, именно ее я выбрал жертвой своих бесчеловечных экспериментов.

Прежде чем продолжить, хочу сказать, что, хотя EF довольно молод, по нему уже есть куча замечательного материала, руководств и tutorial-ов. В частности, рекомендую к ознакомлению руководство по созданию модели EF на сайте [asp.net](#) (eng), которое, кстати, так же доступно в виде [pdf](#) или его русскую адаптацию в [блоге Владимира Юнева](#). Если вы преследуете цель максимально быстро начать использовать EF, не вдаваясь в подробности его работы, то смело переходите по указанным выше ссылкам, а к этой статье вернетесь позже, если будет такой интерес. Если же вам, как и мне, всегда интересно, что скрывается за тем, когда несколько строк кода приводят к созданию запросов, таблиц и целой БД — тогда давайте разбираться дальше вместе.

Версия EF

Да, еще одно важное уточнение, поскольку версии EF выпускаются со скоростью близкой к первой космической, а от версии к версии, возможности расширяются и меняются — отдельно скажу, что написанное ниже относится к версии [EF 4.1 Update 1](#) (хотя, на момент написания этих строк уже есть версия [EF 4.2 Release Candidate](#)).

Итак, если у вас еще не установлен EF 4.1 Update 1, то скачивайте, устанавливайте и приступим!

Небольшое отступление на тему Code First и других подходов

Entity Framework — что это? Чтобы не пересказывать документацию, я просто объясню

основную идею, она состоит в том, чтобы между объектной моделью и БД поместить еще один слой абстракции — модель сущностей или концептуальную модель, [Entity Data Model](#) (EDM), согласно которой и будут определяться правила отражения объектов на БД.

Создать эту модель можно несколькими способами: «*DataBase First*», «*Model First*», «*Code First*». Каждый из способов, по идее, описывает одноименный сценарий — «сначала БД», «сначала модель» и «сначала код» — соответственно. На самом деле, способы вполне могут перекрываться так, что одним способом можно охватить несколько сценариев.

DataBase First — у вас есть БД, по ней вы генерируете модель сущностей и объектную модель. Описание модели сущностей сохраняется в виде xml файла, по которому EF в рантайме создаст саму модель EDM. Визуально посмотреть модель можно в дизайнера модели в Visula Studio (VS).

Model First — вы создаете xml файл модели, например, при помощи того же дизайнера (хотя можно и ручками — путь настоящих джедаев), и по этой модели генерируется БД и объектная модель, т.е. классы.

В обоих перечисленных случаях, автоматически созданные таким способом объекты будут унаследованы от базового класса **EntityObject**. Благодаря этому наследованию EF сможет отслеживать состояние объектов (новый, измененный, и т.д.), и выполнять их отображение в БД. Кстати, автоматическую генерацию классов можно отключить, очистив в дизайнера модели свойство Custom Tool , в котором содержится название инструмента, который EF использует для автоматической генерации классов из концептуальной модели.

Code First — у вас нет ни модели, ни БД, вы сразу пишете код классов предметной области, используете пару хитрых трюков, нажимаете кнопку выполнить и вуаля! Откуда-то появляется БД, модель сущностей EDM (ты видишь модель? нет! и я нет, а она есть!) и все начинает шустро работать. Кроме того, объекты не наследуются ни от какого базового класса! Т.е. являются POCO (меня очень развеселила эта аббревиатура, когда я ее в первый раз увидел, мне подумалось что это какие-то «особые объекты, которые делают особую магию, в рот мне ноги», но это всего лишь сокращение от Plain Old CLR Objects, т.е. старые добрые обычные CLR объекты, и взято это сокращение видимо по аналогии от POJO — Plain Old Java Objects). Собственно наследования нет, xml файла маппинга нет, а все работает – чудо!

Кто все эти люди

Когда начинаешь разбираться и открываешь документацию или [блог разработчиков](#), на тебя сразу вываливается тонна информации: Data Annotations, Fluent API,ObjectContext, DbContext, DbSet...так что возникает ощущение как у героя анекдота и хочется спросить: [...кто все эти люди?](#) Трудно сразу все это запомнить и уж тем более начать ориентироваться, поэтому, давайте немного сузим взгляд и будем разбираться последовательно.

Для этого откроем VS 2010 (так же будет необходим SQL Server Express, обычно он ставится вместе со студией, но если нет, его можно скачать с сайта microsoft бесплатно, у меня установлена версия 2008 R2) и создадим простое консольное приложение, назовем его, например, Heroes. Добавим в проект референсы на классы EF из стандартной сборки System.Data.Entity и на последнюю версию библиотеки Entity Framework (в данном случае 4.1 Update 1), она находится на вкладке .NET и так и называется EntityFramework, 4.1.0.0. Добавим пространство имен System.Data.Entity в секцию using. Создадим класс простой класс Hero, с одним публичным свойством Name. Вот что получается:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Data.Entity;

namespace Heroes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.ReadLine();
        }
    }
    public class Hero
    {
        public string Name { get; set; }
    }
}
```

Наша задача — научиться сохранять и восстанавливать из БД объекты класса **Hero**.

Определение БД

«Точкой входа», если можно так сказать, точкой, откуда начинается взаимодействие с БД является класс **DbContext**. Из названия интуитивно понятно, что это контекст БД, с которой будет вестись работа.

Как задать конкретную базу данных?

У класса **DbContext** есть несколько перегруженных конструкторов:

- 1) публичные: принимают как параметр название БД, строку соединения, существующий объект подключения (**DbConnection**)
 - 2) защищенные: пустой конструктор.
- (есть еще конструкторы, но мы их пока не будем рассматривать, чтобы не тонуть в деталях)

Итак, уже понятно, что БД, сама по себе ни откуда не возникает, а задается, так или иначе, параметрами конструктора, за исключением пустого конструктора. Что будет, если унаследовать класс от **DbContext**, не реализовывать конструктор, и таким образом заставить сработать пустой конструктор **DbContext**?

В таком случае, будет производиться поиск БД в соответствии с соглашениями по умолчанию. Вот она, первая ступень магии. То, что мы видим во всех примерах, класс унаследованный от **DbContext** и БД которая соединяется с этим классом автоматически на основании полного имени класса наследника **DbContext**. Т.е. если бы мы определили такой класс:

```
public class HeroesContext : DbContext{}
```

то искалась бы БД со следующим названием: Heroes.HeroesContext. Давайте выполним скрипт на сервере и создадим БД с таким именем, запустим код и проверим, как работает магия, а именно что даже, несмотря на то, что мы нигде не указали БД, она сцепилась с контекстом автоматически.

```
create database [Heroes.HeroesContext]
go
```

```
class Program
{
    static void Main(string[] args)
    {
        var hc = new HeroesContext();
        if ( hc.Database.Exists() )
            Console.WriteLine("Achtung, Database detected!");
        Console.ReadLine();
    }
}
public class HeroesContext : DbContext
{
}
```

Лично у меня все отработало на ура и на консоль вывелось сообщение: *Achtung, Database detected!*

Тот же эффект будет если мы обойдемся без класса наследника и зададим имя базы явно (т.к. пустой конструктор нам будет уже не доступен).

```
var hc = new DbContext("Heroes.HeroesContext");
```

Либо, зададим строку подключения напрямую:

```
var hc = new DbContext(@"Data Source=.\sqlexpress;Initial Catalog=Heroes.HeroesContext;Integrated Security=True");
```

Либо, поместим строку подключения в файл, и в конструктор передадим название строки:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <connectionStrings>
        <add name="HeroesContextConnectionString" connectionString="Data Source=.\sql
press;Initial Catalog=Heroes.HeroesContext;Integrated Security=True" providerName="
System.Data.SqlClient"/>
    </connectionStrings>
</configuration>
```

```
var hc = new DbContext("name=HeroesContextConnectionString");
```

Либо у вас уже есть объект `DbConnection`, например, традиционный `SqlConnection`

```
SqlConnection sqlCon = new SqlConnection(@"Data Source=.\sqlexpress;Initial Catalog=Heroes.HeroesContext;Integrated Security=True");  
var hc = new DbContext(sqlCon,true);
```

Так же не забываем, что если мы создали класс наследник, вовсе не обязательно использовать только пустой конструктор, мы можем вызвать базовый конструктор с любым из перечисленных параметров:

```
//Любой из параметров, например имя БД.  
public HeroesContext():base("MyDataBaseName"){}
```

Еще отмечу, что строка соединения не обязательно должна быть строкой соединения с БД, это может быть и строка соединения с xml файлом модели, если используется не Code First подход. Подробнее можно почитать в блоге разработчиков EF.

Итак, подытожим, БД может быть определена из:

- 1) Соглашение имени БД по умолчанию (по имени класса наследника DbContext);
- 2) Явным указанием имени БД;
- 3) Явным указанием имени строки подключения;
- 4) Явным указанием непосредственно строки подключения;
- 5) Явным указанием существующего DbConnection подключения.

Главный вывод из всего этого такой, что программисту не надо следовать никакой магии, чтобы все работало, он свободен в выборе, так что не обязательно наследовать класс DbContext, как и не обязательно использовать БД с автоматически назначенным именем. Тем не менее, теперь, когда мы знаем, как это работает, давайте оставим класс наследник HeroesContext с пустым конструктором. И посмотрим дальше, на процесс создания БД, если ее еще нет.

Создание БД

Что происходит, если поиск БД с определенным именем закончился неудачей? Опять же когда смотришь презентации или самоучители, никто не прогоняет на сервере никакой скрипт по созданию БД, но тем не менее БД создается. Тут мы подходим ко второй ступени магии — автоматическому созданию БД. А что вы хотели, Code First все-таки! Лично я привык создавать БД и таблицы руками при помощи скриптов или дизайнера Sql Server Management Studio (SSMS) и весьма скептически отношусь к возможности автоматического создания БД. Но врага надо знать в лицо, и поэтому этот момент заслуживает отдельного рассмотрения.

Итак:

- 1) Во-первых, и в главных: как мы уже знаем Code First не навязывает автоматическое создание базы, всегда можно создать базу самому, определить какой угодно скрипт, со всеми настройками, нюансами, collation-ами, размещением файлов лога и данных и т.д. — что мы и проделали в предыдущем разделе в упрощенном виде. Эта база прекрасно сцепится, и будет работать. Вообще, как пишут в своем блоге сами разработчики EF, название «Code First» не совсем точно отражает суть. Это не значит, что первым обязательно надо писать код. БД вполне может уже существовать, и тогда на совести разработчика останется только сцепить ее с кодом. Более подходящее название, по их же словам «Code Only», т.е. только код, без дополнительных xml файлов.
- 2) В подходе Code First предусмотрен функционал по автоматическому созданию и даже наполнению БД. И этот функционал по-умолчанию включен. Так что когда вы создаете

пустой контекст и обращаетесь к какой-либо сущности, если БД не будет найдена, она будет автоматически создана.

Существуют три основных сценария автоматического создания БД:

CreateDatabaseIfNotExists, **DropCreateDatabaseAlways**,

DropCreateDatabaseIfModelChanges. Как они работают не сложно догадаться из названия. По умолчанию включено соглашение **DropCreateDatabaseIfModelChanges**. Давайте удалим созданную ранее БД, и посмотрим, что произойдет, когда мы запустим приложение.

```
drop database [Heroes.HeroesContext]
go
```

```
class Program
{
    static void Main(string[] args)
    {
        var hc = new HeroesContext();
        Console.ReadLine();
    }
}
public class HeroesContext : DbContext
{
}
```

Обновим в SSMS список БД, и посмотрим, появилась ли там наша БД? Нет, не появилась. Почему? Наверное потому что мы просто создали инстанс, но не вызвали ни один его метод, попробуем добавить вызов метода, который сохраняет изменения:

```
var hc = new HeroesContext();
hc.SaveChanges();
```

Запустим и снова проверим список БД. И снова, БД не появилась! Вах, шайтан. Может что-то не так со стратегией инициализации и по умолчанию она не включена. На самом деле, причина тут в отложенной инициализации (lazy initialization) и, как объясняет в своем блоге один из разработчиков, даже вызов метода **SaveChanges()**, который по идее должен приводить к сохранению, не возымеет эффекта, если не было ни одного реального обращения, ни к одной из сущностей БД. Раз мы ни разу не обращались, то нечего и сохранять — именно так реализовано. И это хорошо, как говорится. Инициализацию можно вызвать принудительно, вызвав метод **Initialize()** БД контекста, что мы сейчас и сделаем, чтобы пока на этом этапе не отвлекаться на добавление сущностей и запросы к ним.

```
var hc = new HeroesContext();
hc.Database.Initialize(false);
Console.ReadLine();
```

параметр метода определяет, следует ли вызывать инициализацию повторно, если она уже была вызвана, в данном случае, нам не важно, какое у него будет значение, т.к. мы ее

вызываем всего один раз. После выполнения программы, обновим список БД и о чудо, наконец-то БД создана автоматически. Причем, если мы раскроем раздел таблицы, то мы увидим, что БД отличается от той, что мы создавали руками.



В созданной автоматически есть таблица **EdmMetadata**, в которой хранятся метаданные модели, именно по этой таблице определяется, соответствует ли концептуальная модель базе данных или БД устарела.

Вот что отправляется серверу при создании такой БД.

```
SELECT Count(*) FROM sys.databases WHERE [name]=N'Heroes.HeroesContext'
--
create database [Heroes.HeroesContext]
create table [dbo].[EdmMetadata] (
    [Id] [int] not null identity,
    [ModelHash] [nvarchar](max) null,
    primary key ([Id])
);
--
exec sp_executesql N'insert [dbo].[EdmMetadata]([ModelHash])
values (@@0)
select [Id]
from [dbo].[EdmMetadata]
where @@ROWCOUNT > 0 and [Id] = scope_identity()',N'@@0 nvarchar(max) ',@@0=N'B8F62AA
BEADFA7D4C6855DFBCEA0BCD540A7BF1D3226CF8DF9421CC06FD1ABD6'
```

Т.е. мы видим, что из всего многообразия опций и настроек команды create database реально не используется ни одна, т.е. БД создается со всеми параметрами по умолчанию. Возникает вопрос, можно ли, помимо трех встроенных инициализаторов, написать свой инициализатор, с блек джеком и... ну вы в курсе. Ответ, да, можно. Для этого нужно всего лишь написать класс, который реализует интерфейс **IDatabaseInitializer**, и в нем реализовать всего один метод **InitializeDatabase()**. Прежде чем приступить к написанию своего инициализатора, давайте посмотрим, как устроен стандартный инициализатор, например соглашение **CreateDatabaseIfNotExists**.

```
public class CreateDatabaseIfNotExists<TContext> : IDatabaseInitializer<TContext> where TContext: DbContext
{
    public void InitializeDatabase(TContext context)
    {
        bool flag;
        using (new TransactionScope(TransactionScopeOption.Suppress))
        {
            flag = context.Database.Exists();
        }
        if (flag)
        {
            bool throwIfNoMetadata = false;
```



```
        if (!context.Database.CompatibleWithModel(throwIfNoMetadata))
        {
            throw Error.DatabaseInitializationStrategy_ModelMismatch(context.Get
GetType().Name);
        }
    }
    else
    {
        context.Database.Create();
        this.Seed(context);
        context.SaveChanges();
    }
}

protected virtual void Seed(TContext context)
{
}
}
```

** код из рефлексора приведен только с ознакомительной и исследовательской целью*

В принципе, все понятно, простой класс, реализующий метод **InitializeDatabase()**, в котором проверяется наличие БД и используются стандартные методы по созданию БД. Так же видим что есть «точка входа» для разработчиков в виде метода **Seed()**, который будет вызван для заполнения БД. Таким образом, можно перегрузив данный метод заполнить БД по своему усмотрению, ну или выполнить еще какие-либо действия. Ничего не мешает написать свой подобный класс. Для простоты, уберем всю работу с транзакциями и метод заполнения, т.к. он нас пока не интересует. И реализуем логику, при которой БД будет создаваться каждый раз при вызове инициализатора (для примера, реализуем для конкретного контекста **HeroesContext**). Таким образом, класс выглядит вот так:

```
public class MyHeroicInitializer : IDatabaseInitializer<HeroesContext>
{
    public void InitializeDatabase(HeroesContext context)
    {
        if (context.Database.Exists())
            context.Database.Delete();
        context.Database.Create();
    }
}
```

Как установить инициализатор?

Для установки есть отдельный метод (да-да, здесь придется запомнить еще одно название класса), это статический метод класса **Database**. Его нужно вызывать при старте приложения. Соответственно, если у вас веб-приложение, то вызывайте его, например, в `Globals.Application_Start`, мы же вызовем его в методе `Main`. Выглядит это так:

```
class Program
{
    static void Main(string[] args)
    {
```



```
Database.SetInitializer(new MyHeroicInitializer());
var hc = new HeroesContext();
hc.Database.Initialize(false);
Console.ReadLine();
}
}
```

Запустим и посмотрим, как отработает.

Лично у меня все отработало как надо, БД создалась. Теперь возникает вопрос, что еще мы можем сделать в инициализаторе? В принципе, благодаря тому, что у **DbContext** есть свойство **Database**, а у него есть метод **ExecuteSqlCommand()**, мы можем как угодно alter-ить нашу БД после создания. Это хорошая команда и она дает много возможностей ([ALTER DATABASE](#) - Изменяет базу данных или файлы и файловые группы, связанные с базой данных. Добавляет или удаляет файлы и файловые группы из базы данных, изменяет атрибуты базы данных или ее файлов и файловых групп, изменяет параметры сортировки базы данных и устанавливает параметры базы данных. Моментальные снимки базы данных изменить нельзя.). Хотелось бы конечно иметь возможность повлиять на генерацию опций команды create database непосредственно. К сожалению, просто пути, чтобы сделать это – я не нашел. Поиски по рефлектору приводят к такому коду создания БД:

```
internal static string CreateDatabaseScript(string databaseName, string dataFileName, string logFileName)
{
    SqlDdlBuilder builder = new SqlDdlBuilder();
    builder.AppendSql("create database ");
    builder.AppendIdentifier(databaseName);
    if (dataFileName != null)
    {
        builder.AppendSql(" on primary ");
        builder.AppendFileName(dataFileName);
        builder.AppendSql(" log on ");
        builder.AppendFileName(logFileName);
    }
    return builder.unencodedStringBuilder.ToString();
}
```

* код из рефлектора приведен только с ознакомительной и исследовательской целью

который находится в **System.Data.SqlClient**, в классе **internal sealed class SqlDdlBuilder** и как мы видим, наследовать от него нельзя, все зашито достаточно жестко.

Но, например, чтобы просто поменять collation или реализовать свою логику создания БД вполне сойдет и простой инициализатор. Давайте в качестве примера напишем класс, который всегда пересоздает БД и меняет ей collation.

```
public class MyHeroicInitializer : IDatabaseInitializer<HeroesContext>
{
    public void InitializeDatabase(HeroesContext context)
    {
        if (context.Database.Exists())
            context.Database.Delete();
        context.Database.Create();
    }
}
```

```
var alterCommand = new StringBuilder();
string DbName = context.Database.Connection.Database;
alterCommand.AppendFormat("alter database [{0}] collate SQL_Latin1_General_CP1251_CI_AS;", DbName);
context.Database.ExecuteSqlCommand(alterCommand.ToString());
}
```

Тот же самый код по изменению collation, можно было бы написать и в типизированном классе стандартного инициализатора, реализовав перегруженный метод **Seed()**.

Как определяется, какие объекты участвуют в модели

Третьей ступенью магии для меня был вопрос, как определяется, какие классы EF будет использовать в модели, а какие нет, и как я могу этим управлять.

При разработке Domain Model, в так называемом стиле Domain Driven Development (DDD) часто используется такой паттерн как [репозиторий](#) (на русский его переводят как хранилище, но лично мне понятнее репозиторий). Суть вкратце можно описать так, паттерн репозиторий предоставляет доступ к объектам таким образом, как если бы это была коллекция в памяти, а так же предоставляет методы по поиску, добавлению и удалению объектов в коллекции. Обычно в DDD кандидатами на роль объектов для которых создаются репозитории являются так называемые «корни агрегации». Объекты, которые способны группировать по смыслу вокруг себя другие объекты и являющиеся для них корневыми. Т.е. например объект «автомобиль» и объект «колесо», и если колесо нас интересует только в контексте автомобиля, тогда объект «автомобиль» будет являться корнем агрегации. Извиняюсь если пример не очень понятный, но DDD это отдельная большая тема по которой есть много литературы и не хотелось бы тут ее пересказывать, делая заметку еще больше, по этому, для более детального изучения, я просто дам ссылку на небольшую обзорную статью по этой теме [Введение в проблемно-ориентированное проектирование](#) (и большую книжку Эрика Эванса – [Предметно ориентированное проектирование](#)).

Зная о **DbContext** как о начальной точке работы с EF, а теперь зная еще и то, что, по сути, работа с любым корневым объектом, в идеологии DDD должна начинаться с извлечения его из репозитория, уже можно догадаться, откуда именно EF мог бы получить информацию о сущностях. Довольно логичный путь — просмотреть все типы хранилищ, зарегистрированных в контексте БД, и по ним определить типы сущностей. Однако какие именно типы хранилищ ищет EF? Понятно, что если мы напишем такой код и запустим его на выполнение, то никаких таблиц в БД не появится:

```
public class HeroesContext : DbContext
{
    private ICollection<Hero> Heroes { get; set; }
}
```

А вот если мы вместо **ICollection<>** напишем **IDbSet<>** или просто **DbSet<>**, то мы уже увидим попытку EF создать сущность для нашего класса, которая пока правда закончится неудачно. Ошибка свидетельствует о попытке EF создать сущность в модели и в ней говорится о том, что для сущности не удалось определить первичный ключ (все-таки есть отличие от «классических» объектов, когда объекты у нас только в памяти – мы всегда

можем отличить один от другого, но как только требуется удалить из памяти и восстановить объект из постоянного хранилища, требуется постоянный ключ). Задать ключ можно несколькими способами, ограничимся пока лишь самым простым (но не самым очевидным), добавим поле **HeroID**, которое (еще одна неочевидная магия), благодаря включенным соглашениям по-умолчанию, автоматически отразится в БД в виде поля первичного ключа с признаком **Identity**.

```
public class Hero
{
    public int HeroID { get; set; }
    public string Name { get; set; }
}
```

Если мы теперь выполним код и при помощи запроса в SSMS посмотрим на список таблиц в БД

```
use master
go
select * from [Heroes.HeroesContext].sys.tables
```

(Я пишу полное имя БД и выполняю запросы в БД **master**, а не открываю БД в дизайнера SSMS потому, что EF спокойно пересоздает БД если к ней нет ни одного активного подключения, но если у вас открытое соединение с БД например в той же SSMS, удаление закончится неудачей и это просто неудобно для демонстрации.)

Мы увидим, что появилась таблица **Heroes**, в которой и предполагается хранение объектов **Hero**. Давайте добавим еще пару классов, которые будут описывать героя, допустим базовые навыки и коллекция воинов.

Spoiler:

```
namespace Heroes
{
    class Program
    {
        static void Main(string[] args)
        {
            var hc = new HeroesContext();
            Database.SetInitializer(new MyHeroicInitializer());
            hc.Database.Initialize(false);
            Console.ReadLine();
        }
    }
    public class HeroesContext : DbContext
    {
        public DbSet<Hero> Heroes { get; set; }
    }
    public class MyHeroicInitializer : IDatabaseInitializer<HeroesContext>
    {

```

```
public void InitializeDatabase(HeroesContext context)
{
    if (context.Database.Exists())
        context.Database.Delete();
    context.Database.Create();

    var alterCommand = new StringBuilder();
    string dbName = context.Database.Connection.Database;
    alterCommand.AppendFormat("alter database [{0}] collate SQL_Latin1_General_CP1251_CI_AS;", dbName);
    context.Database.ExecuteSqlCommand(alterCommand.ToString());
}

public class Hero
{
    public int HeroID { get; set; }
    public string Name { get; set; }
    public BasicSkills BasicSkills { get; set; }
    public ICollection<Warrior> Warriors { get; set; }
}

public class BasicSkills
{
    public int Attack { get; set; }
    public int Defence { get; set; }
    public int Power { get; set; }
    public int Knowledge { get; set; }
}

public class Warrior
{
    public int WarriorID { get; set; }
    public string Name { get; set; }
}
```

Запустим приложение и после этого, посмотрим в БД. Вот что мы там увидим:

	name	object_id	principal_id	schema_id	parent_object_id	type
1	Heroes	21575115	NULL	1	0	U
2	Warriors	85575343	NULL	1	0	U
3	EdmMetadata	2105058535	NULL	1	0	U

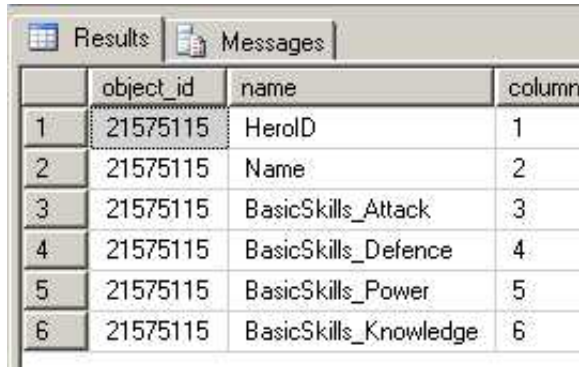
Во-первых, несмотря на то, что для прибавленных типов мы не создали отдельных **DbSet**-ов, а просто включили их в класс **Hero** – таблиц все равно прибавилось.

Во-вторых, хотя мы добавили два класса, прибавилась только одна таблица.

Опять магия. Но если подумать, то не совсем. Очевидный вывод, что классы, включенные в корневой класс, но не имеющие собственных репозиторий так же отражаются в модели (мы

ведь не создавали отдельный репозиторий для класса **Warrior**, а таблица появилась). Теперь если мы, при помощи запроса, посмотрим на таблицу **Heroes**, то мы увидим, во что превратился класс **BasicSkills**

```
select * from [Heroes.HeroesContext].sys.columns where object_id = object_id('[Heroes.HeroesContext]..Heroes')
```



	object_id	name	column
1	21575115	HeroID	1
2	21575115	Name	2
3	21575115	BasicSkills_Attack	3
4	21575115	BasicSkills_Defence	4
5	21575115	BasicSkills_Power	5
6	21575115	BasicSkills_Knowledge	6

Он, как бы, просто «разложился» в таблицу своего класса контейнера. Почему так произошло и в чем разница – мы поговорим в отдельной статье. А пока это только демонстрация того, что все типы включаются в модель рекурсивно, начиная с корневого и не обязательно что под каждый тип будет создана своя таблица.

Отдельно скажу про атрибут **SuppressDbSetInitializationAttribute**, который находится в пространстве имен **System.Data.Entity.Infrastructure**. Он отвечает за автоматическую инициализацию **DbSet**-ов. Им можно пометить как отдельные **DbSet**, так и весь класс контекста. Этот атрибут **не** отвечает за то, будет ли включен отмеченный тип в модель, он будет включен в любом случае, он отвечает именно за инициализацию свойств. Действие этого атрибута, можно быстро посмотреть, пометив им свойство **Heroes**.

```
class Program
{
    static void Main(string[] args)
    {
        var hc = new HeroesContext();
        Database.SetInitializer(new MyHeroicInitializer());
        hc.Database.Initialize(false);
        if (hc.Heroes == null)
        {
            SuppressDbSetInitializationAttribute Console.WriteLine("Heroes is not initialized");
        }
        else
        {
            Console.WriteLine("Heroes is initialized");
            Console.ReadLine();
        }
    }
}

public class HeroesContext : DbContext
{
    [SuppressDbSetInitialization]
    public DbSet<Hero> Heroes { get; set; }
}
```

Выведется сообщение: «*Heroes is not initialized*». Если теперь закомментировать `//[SuppressDbSetInitialization]`, и выполнить код повторно, то на консоль выведется «*Heroes is initialized*».

Итак, при инициализации контекста, происходит рекурсивный поиск типов, начиная с корневых, зарегистрированных при помощи **DbSet**-ов и включение в модель соответствующих сущностей. Далее, происходит автоматическая инициализация **DbSet**-ов, если они специально не помечены атрибутом **SuppressDbSetInitializationAttribute**. Можно ли обойтись без автоматической инициализации и не открывать публичный акцессор **set**? Можно, для этого нужно вызвать метод **DbContext**-а **Set<>()**.

Например, так:

```
class Program
{
    static void Main(string[] args)
    {
        var hc = new HeroesContext();
        Database.SetInitializer(new MyHeroicInitializer());
        hc.Database.Initialize(false);
        if (hc.Heroes == null)
            Console.WriteLine("Heroes is not initialized");
        else
            Console.WriteLine("Heroes is initialized");
        if (hc.Warriors == null)
            Console.WriteLine("Warriors is not initialized");
        else
            Console.WriteLine("Warriors is initialized");
        Console.ReadLine();
    }
}

public class HeroesContext : DbContext
{
    [SuppressDbSetInitialization]
    public DbSet<Hero> Heroes { get; set; }

    [SuppressDbSetInitialization]
    public DbSet<Warrior> Warriors { get { return Set<Warrior>(); } }
}
```

Если запустить данный код, то выведется:

Heroes is not initialized

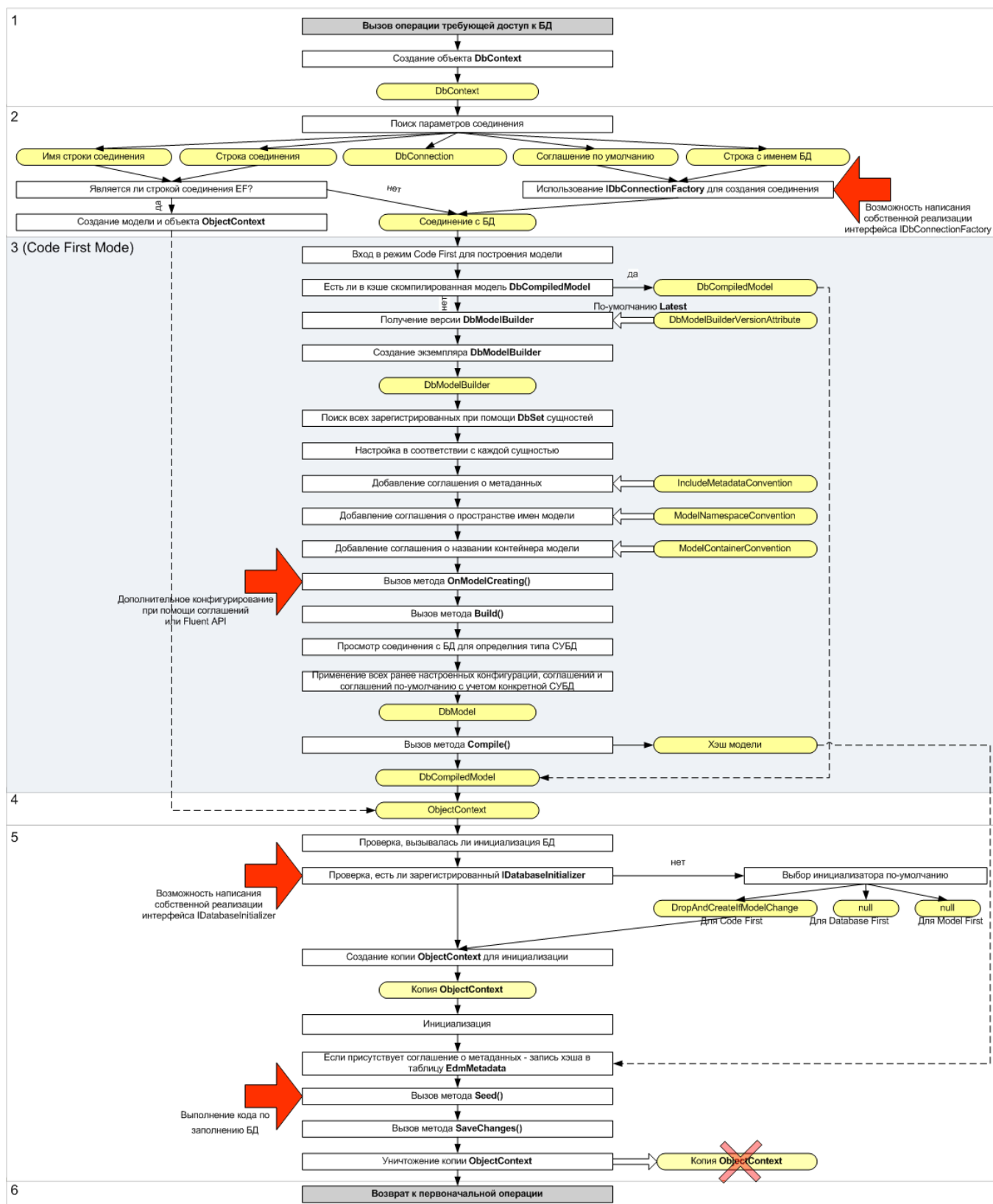
Warriors is initialized

Что логично, т.к. никакой автоматической инициализации **Warriors** не происходит, и атрибут никак не влияет на вызов метода `get`. При этом стоит еще отметить, что результат вызова метода кэшируется контекстом. И в последующих обращениях к свойству **Warriors**, объект **DbSet** будет взят из кэша.

Ну и последнее замечание для полноты представления картины инициализации. Принципиально любой подход Code First, Model First и Database First приходят к единой точке — концептуальной модели или модели сущностей (EDM модели), работа с которой осуществляется при помощи класса **ObjectContext**. Этот класс знаком тем, кто использовал

ранее EF, но не подход Code First. Разработчики в своем блоге характеризуют DbContext как упрощенную, более интуитивно понятную и более оптимизированную под типичные задачи надстройку надObjectContext. По этому, важно понимать, что подход Code First, не является какой-то принципиально отличной веткой в EF, он элегантно встраивается в общую схему.

Для того, чтобы подытожить и пронаблюдать процесс инициализации в целом, ниже представлена схема основных шагов этого процесса. Описание взято из блога разработчика, так что я думаю оно достаточно достоверно, схемка составлена мной по нему, ниже описание шагов.



Пояснения по схеме:

1. Создание экземпляра DbContext.

2. Инициализация. DbContext инициализируется, когда в первый раз используется

2.1. Контекст пытается найти соединение или строку соединения.

2.2. Проверяется, является ли строка соединения — строкой соединения Entity Framework содержащей подробности по использованию EDM или же это просто строка соединения с БД.
- если это строка соединения EF, то создается нижележащий объектObjectContext режиме «Model First/Database First» используя EDM из строки соединения (CSDL, MSL, и SSDL из EDMX)
- если строка — это строка соединения с БД, тогда контекст входит в режим Code First и пытается построить модель согласно подходу Code First.

3. Построение модели.

Модель для определенного типа контекста кешируется в app-domain как экземпляр DbCompiledModel, это сделано для того чтобы весь путь по построению модели Code First был проделан лишь однажды, при первом использовании контекста. Итак, DbContext проверяет, есть ли уже кешированный экземпляр DbCompiledModel, если нет то:

3.1. DbContext создает экземпляр DbModelBuilder. По умолчанию, соглашение, используемое для создания DbModelBuilder это Latest. Другое соглашение можно установить, используя атрибут DbModelBuilderVersionAttribute. Этот атрибут определяет, какие версии соглашений DbContext-а DbModelBuilder-а использовать при построении модели из кода.

3.2. Model Builder настраивается в соответствии с каждым типом сущности, для которой было найдено свойство DbSet. имена свойств используются как имена сущностей, что удобно в сервисах OData.

3.3. Добавляется соглашение IncludeMetadataConvention, которое включит помимо прочего еще и сущность EdmMetadata (используемую для отслеживания изменений модели), если конечно позже это соглашение не исключат.

3.4. Добавляются соглашения ModelContainerConvention и ModelNamespaceConvention, благодаря им имя контекста будет использоваться как имя контейнера Edm, и пространство имен контекста, как пространство имен EDM. И снова, это бывает удобно для сервисов основанных на нижележащей EDM. (например OData — <http://msdn.microsoft.com/ru-ru/magazine/ff714561.aspx>)

3.5. Вызывается метод OnModelCreating, чтобы разработчик мог сделать дополнительное конфигурирование модели.

3.6 Вызывается метод Build. model builder строит внутреннее представление модели EDM основанное на сконфигурированных типах и запускает все соглашения Code First, которые в дальнейшем изменяют конфигурацию модели. В этом процессе используется строка соединения, т.к. часть модели SSDL зависит от конкретной БД.

3.7 Для DbModel вызывается метод Compile, который создает DbCompiledModel. (В настоящее время DbCompiledModel является оберткой MetadataWorkspace). На данном этапе так же создается хэш модели.

4. Скомпилированный объект DbCompiledModel используется для создания нижележащегоObjectContext.

5. На данном этапе мы имеем объектObjectContext, созданный либо с помощью Code First, либо EDM в строке подключения. Теперь DbContext проверяет, была ли произведена инициализация БД в домене приложения для унаследованного класса контекста и строки подключения. Если ее не было то

5.1. Проверяется, был ли для данного экземпляра зарегистрирован IDatabaseInitializer. Если не было зарегистрировано ни одного инициализатора, то регистрируется инициализатор по-умолчанию. В подходе Code First, инициализатором по умолчанию является CreateDatabaseIfNotExists. В подходе Database/Model First mode — инициализатором по умолчанию является null, что означает, что по умолчанию, никакой инициализации БД происходить не будет (потому что, при таком подходе, БД почти всегда уже существует).

5.2. Если найден не нулевой инициализатор, то создается копия объектаObjectContext, которая будет использоваться вместо реального объекта на протяжении всего процесса инициализации, а затем будет уничтожена. Это делается для того чтобы исключить всяческие воздействия и утечки кода инициализации при дальнейшей работе с

приложением.

5.3. Запускается инициализатор. Для примера использующийся по умолчанию DropCreateDatabaseIfModelChanges.

5.4 Вызывается метод Seed().

5.5 Вызывается метод SaveChanges()

5.6. Если б.д. существует, то выполняется проверка, была ли включена в модель сущность EdmMetadata, и если была, то есть ли в БД соответствующая таблица с хешем модели.

5.7. временныйObjectContext уничтожается.

6. Контроль возвращается к той операции, для которой DbContext был использован в первый раз.

И последнее, **обязательно уберите инициализатор из production версии**. Это вроде очевидно, но можно про это забыть. Не стоит говорить, что будет, если вдруг у вас на реальной БД сработает стратегия «Удалить И Создать», и при этом не будет ни одного подключения к БД =).

Надеюсь, теперь ни у кого не осталось ощущения магии, при работе с Entity Framework Code First. И хотя еще остались некоторые загадочные места в работе (например, по каким правилам объекты отображаются в сущности и затем в таблицы, и как мы можем этим управлять), но в целом хотя бы идея и реализация стали более менее ясны.

Полезные ссылки

[ADO.NET team blog](#)

[Документация Entity Framework 4.1](#)

[Блог разработчика из Entity Framework team](#)

[EF 4.1 Code First Walkthrough](#)

[модель Entity Data](#)

SomewhereSomehow's notes