

Contents

Entity Framework

Сравнение EF Core и EF6

EF6 и Core EF в одном приложении

Перенос приложений из EF6 в EF Core

Проверка требований

Перенос модели на основе EDMX

Перенос модели на основе кода

Entity Framework Core

Новые возможности

Стратегия развития EF Core

EF Core 2.1

EF Core 2.0

EF Core 1.1

EF Core 1.0

Начало работы

Установка EF Core

.NET Framework (консоль, WinForms, WPF и т. д.)

.NET Framework — новая база данных

.NET Framework — существующая база данных

.NET Core (Windows, OSX, Linux и т. д.)

.NET Core — новая база данных

ASP.NET Core

ASP.NET Core — новая база данных

ASP.NET Core — существующая база данных

EF Core и Razor Pages

Универсальная платформа Windows (UWP)

UWP — новая база данных

Создание модели

Включение и исключение типов

Включение и исключение свойств
Ключи (первичные)
Созданные значения
Обязательные/дополнительные свойства
Максимальная длина
Токены параллелизма
Свойства тени
Отношения
Индексы
Альтернативные ключи
Наследование
Резервные поля
Преобразования значений
Присвоение начальных значений данных
Конструкторы типов сущностей
Собственные типы сущностей
Типы запросов
Переключение моделей с одинаковым DbContext
Моделирование реляционных баз данных
Сопоставление таблиц
Сопоставление столбцов
Типы данных
Первичные ключи
Схема по умолчанию
Вычисляемые столбцы
Последовательности
Значения по умолчанию
Индексы
Ограничения внешнего ключа
Альтернативные ключи (ограничения уникальности)
Наследование (реляционная база данных)
Управление схемами баз данных

Миграции

Командные среды

Пользовательские операции

Использование отдельного проекта

Несколько поставщиков

Настраиваемая таблица журнала

□ API создания и удаления

□ Реконструирование

Запросы к данным

Базовый запрос

Загрузка связанных данных

Оценка клиента и сервера

Сравнение работы с отслеживанием и без него

Необработанные SQL-запросы

Асинхронные запросы

Принцип работы запроса

Глобальные фильтры запросов

Сохранение данных

Базовое сохранение

Связанные данные

Каскадное удаление

Конфликты параллелизма

Транзакции

Асинхронное сохранение

Отключенные сущности

Явные значения для создаваемых свойств

Поддерживаемые реализации .NET

Поставщики баз данных

Microsoft SQL Server

Таблицы, оптимизированные для памяти

SQLite

Ограничения SQLite

- [Выполнение в памяти \(для тестирования\)](#)
- [Создание поставщика баз данных](#)
- [Изменения, влияющие на поставщика](#)
- [Инструменты и расширения](#)
- [Справочник по командной строке](#)
 - [Консоль диспетчера пакетов \(Visual Studio\)](#)
 - [Интерфейс командной строки .NET Core](#)
 - [Создание DbContext во время разработки](#)
 - [Службы времени разработки](#)
- [Прочее](#)
 - [Строки подключения](#)
 - [Ведение журнала](#)
 - [Устойчивость подключений](#)
 - [Тестирование](#)
 - [Тестирование с помощью SQLite](#)
 - [Тестирование с использованием выполнения в памяти](#)
 - [Настройка DbContext](#)
 - [Обновление с версии 1.0 релиз-кандидата 1 до релиз-кандидата 2](#)
 - [Обновление с версии 1.0 релиз-кандидата 2 до RTM](#)
 - [Обновление до EF Core 2.0](#)
 - [□ Справочник по API EF Core](#)
- [Entity Framework 6](#)
 - [Новые возможности](#)
 - [Стратегия развития](#)
 - [Прошлые выпуски](#)
 - [Обновление до EF6](#)
 - [Выпуски Visual Studio](#)
 - [Начало работы](#)
 - [Основы](#)
 - [Установка Entity Framework](#)
 - [Работа с DbContext](#)
 - [Как работают отношения](#)

[Асинхронные запросы и сохранение](#)

[Конфигурация](#)

[На базе кода](#)

[Файл конфигурации](#)

[Строки подключения](#)

[Разрешение зависимостей](#)

[Управление подключениями](#)

[Устойчивость подключений](#)

[Логика повтора](#)

[Проблемы с фиксацией транзакций](#)

[Привязка данных](#)

[WinForms](#)

[WPF](#)

[Отключенные сущности](#)

[Самоотслеживающиеся сущности](#)

[Пошаговое руководство](#)

[Ведение журналов и перехват](#)

[Производительность](#)

[Особенности производительности \(технический документ\)](#)

[Использование NGEN](#)

[Использование заготовленных представлений](#)

[Поставщики](#)

[Модель поставщика EF6](#)

[Поставщики и пространственные данные](#)

[Использование прокси](#)

[Тестирование с EF6](#)

[Использование макетирования](#)

[Написание собственных имитированных реализаций](#)

[Возможности тестирования с EF4 \(статья\)](#)

[Создание модели](#)

[Использование Code First](#)

[Рабочие процессы](#)

[С новой базой данных](#)

[С существующей базой данных](#)

[Заметки к данным](#)

[Наборы DbSet](#)

[Типы данных](#)

[Перечисления](#)

[Пространственные данные](#)

[Соглашения](#)

[Встроенные соглашения](#)

[Настраиваемые соглашения](#)

[Модельные соглашения](#)

[Конфигурация с текущим API](#)

[Отношения](#)

[Типы и свойства](#)

[Использование в Visual Basic](#)

[Сопоставление хранимых процедур](#)

[Миграции](#)

[Автоматическая миграция](#)

[Работа с существующими базами данных](#)

[Настройка журнала миграции](#)

[Использование Migrate.exe](#)

[Миграция в командных средах](#)

[Использование EF Designer](#)

[Рабочие процессы](#)

[Model First](#)

[Database First](#)

[Типы данных](#)

[Сложные типы](#)

[Перечисления](#)

[Пространственные данные](#)

[Разделение сопоставлений](#)

[Разделение сущности](#)

[Разделение таблицы](#)

[Сопоставление с наследованием](#)

[Одна таблица на иерархию](#)

[Одна таблица на тип](#)

[Сопоставление хранимых процедур](#)

[Запрос](#)

[Обновление](#)

[Сопоставление отношений](#)

[Множество диаграмм](#)

[Выбор версии среды выполнения](#)

[Создание кода](#)

[Прошлые версии с ObjectContext](#)

[Дополнительно](#)

[Формат файлов EDMX](#)

[Определяющий запрос](#)

[Множество результирующих наборов](#)

[Функции с табличным значением](#)

[Сочетания клавиш](#)

[Запросы к данным](#)

[Метод Load](#)

[Локальные данные](#)

[Отслеживаемые и неотслеживаемые запросы](#)

[Использование прямых SQL-запросов](#)

[Запрос связанных данных](#)

[Сохранение данных](#)

[Отслеживание изменений](#)

[Автообнаружение изменений](#)

[Состояние сущностей](#)

[Значения свойств](#)

[Обработка конфликтов параллелизма](#)

[Использование транзакций](#)

[Проверка данных](#)

[Дополнительные ресурсы](#)

[Блоги](#)

[Примеры](#)

[Участие](#)

[Получение справки](#)

[Глоссарий](#)

[Образец базы данных School](#)

[Инструменты и расширения](#)

[Лицензии](#)

[EF5](#)

[Китайский \(упрощенное письмо\)](#)

[Китайский \(традиционное письмо\)](#)

[Немецкий](#)

[Английский](#)

[Испанский](#)

[Французский](#)

[Итальянский](#)

[Японский](#)

[Корейский](#)

[Русский](#)

[EF6](#)

[Предварительный выпуск](#)

[Китайский \(упрощенное письмо\)](#)

[Китайский \(традиционное письмо\)](#)

[Немецкий](#)

[Английский](#)

[Испанский](#)

[Французский](#)

[Итальянский](#)

[Японский](#)

[Корейский](#)

[Русский](#)

Документация по Entity Framework

Entity Framework

Entity Framework является объектно-реляционным модулем сопоставления (О/RМ), который позволяет разработчикам .NET работать с базой данных, используя объекты .NET. Это устраняет необходимость в большей части кода для доступа к данным, который разработчикам обычно приходится писать.



Entity Framework Core

EF Core — это упрощенная, расширяемая и кроссплатформенная версия Entity Framework.



Entity Framework 6

EF 6 — это проверенная технология доступа к данным, для которой в течение многих лет велась доработка функций и повышалась стабильность работы.



Выбор

Узнайте, какая версия EF подходит для вас.



Перенос в EF Core

Руководство по переносу приложения EF 6 в EF Core.

EF Core

все

EF Core — это упрощенная, расширяемая и кроссплатформенная версия Entity Framework.



Начало работы

Обзор

Создание модели

Запрос данных

Сохранение данных



Учебники

.NET Framework

.NET Core

[ASP.NET Core](#)

[UWP](#)

[Дополнительно...](#)

□

[Поставщики баз данных](#)

[SQL Server](#)

[MySQL](#)

[PostgreSQL](#)

[SQLite](#)

[Дополнительно...](#)

□

[□ Справочник по API](#)

[DbContext](#)

[DbSet< TEntity >](#)

[Дополнительно...](#)

[EF 6](#)

EF 6 — это проверенная технология доступа к данным, для которой в течение многих лет велась доработка функций и повышалась стабильность работы.

□

[Приступая к работе](#)

[Узнайте, как получить доступ к данным с помощью Entity Framework 6.](#)

□

[□ Справочник по API](#)

Просмотрите список интерфейсов Entity Framework 6, упорядоченный по пространствам имен.

Сравнение EF Core и EF6

28.08.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework предлагается в двух версиях: Entity Framework Core и Entity Framework 6.

Entity Framework 6

Entity Framework 6 (EF6) — это проверенная технология доступа к данным, для которой в течение многих лет велась доработка функций и повышалась стабильность работы. Впервые этот продукт был выпущен в 2008 году в составе пакета обновления 1 (SP1) для платформы .NET Framework 3.5 и пакета обновления 1 (SP1) для Visual Studio 2008. Начиная с выпуска EF4.1, он поставлялся в виде [пакета NuGet EntityFramework](#), который сейчас стал одним из наиболее популярных пакетов на сайте NuGet.org.

Продукт EF6 все еще поддерживается, и для него постепенно будут выходить исправления ошибок и небольшие улучшения.

Entity Framework Core

Entity Framework Core (EF Core) — это упрощенная, расширяемая и кроссплатформенная версия Entity Framework. EF Core отличается от EF6 многочисленными усовершенствованиями и новыми функциями. При этом EF Core является новой базой кода, не настолько зрелой как EF6.

В EF Core взаимодействие с разработчиком осуществляется так же, как и в EF6, кроме того, там сохранена основная часть API верхнего уровня, поэтому EF Core покажется очень знакомым тем, кто уже работал с EF6. При этом EF Core основан на совершенно новом наборе основных компонентов. Это означает, что EF Core не наследуют все функции EF6 автоматически. Хотя некоторые функции появятся в будущих выпусках, другие, реже используемые функции не будут реализованы в EF Core.

Новое расширяемое и упрощенное ядро позволило добавить в EF Core некоторые функции, которые не будут реализованы в EF6 (например, альтернативные ключи, пакетные обновления, а также смешанное вычисление для клиентов и баз данных в запросах LINQ).

Использование EF Core и EF6 в одном приложении

02.09.2018 • 2 minutes to read • [Edit Online](#)

EF и EF6 можно использовать в одном приложении или библиотеке .NET Framework, установив оба пакета NuGet.

Некоторые типы имеют одинаковые имена в EF Core и EF6, различаясь только пространством имен, что может усложнить параллельное использование EF Core и EF6 в одном файле кода. Эту неоднозначность можно легко устранить с помощью директив псевдонима пространства имен. Пример:

```
using Microsoft.EntityFrameworkCore; // use DbContext for EF Core
using EF6 = System.Data.Entity; // use EF6.DbContext for the EF6 version
```

Если вы портируете существующее приложение, в котором есть несколько моделей EF, у вас есть возможность перенести часть этих моделей в EF Core и сохранить EF6 для остальных.

Перенос приложений из EF6 в EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Из-за фундаментальных отличий от EF Core мы не рекомендуем переносить приложения EF6 на EF Core, если для этого нет веской причины. Переход с EF6 на EF Core следует рассматривать как портирование, а не обновление.

Прежде чем перенос приложений из EF6 в EF Core: проверка требования вашего приложения

29.08.2018 • 5 minutes to read • [Edit Online](#)

Перед началом процесса переноса важно проверить EF Core, соответствует ли требований доступа к данным для вашего приложения.

Для отсутствующих компонентов

Убедитесь, что EF Core имеет все функции, которые необходимо использовать в приложении. См. в разделе [сравнение функций](#) подробное сравнение с набором функций из EF Core с EF6. Если все необходимые компоненты отсутствуют, убедитесь, что вы можете компенсировать отсутствие этих функций перед развертыванием в EF Core.

Изменения в поведении

Это неполный список некоторые изменения в поведении между EF6 и EF Core. Важно помнить как порт приложения, так как они могут изменить способ приложение ведет себя, но не будут отображаться как ошибки компиляции после переключения в EF Core.

Поведение `DbSet.Add/Attach` и `graph`

В EF6 вызвав `DbSet.Add()` на результаты сущностей в рекурсивный поиск для всех сущностей, на которые ссылается его свойства навигации. Все сущности, на которые имеются и, не отслеживаются контекстом, можно пометить как добавлен. `DbSet.Attach()` работает так же, за исключением того, все сущности, помечаются как без изменений.

EF Core выполняет рекурсивный поиск аналогичный, но в некоторых правилах немного отличается.

- Корневая сущность всегда указывается в запрошенное состояние (для `DbSet.Add` и не изменяется для `DbSet.Attach`).
- Для сущностей, которые были обнаружены во время рекурсивный поиск свойств навигации:
 - Если первичный ключ сущности сохраняются созданные
 - Если первичный ключ не присвоено значение, состояние устанавливается на добавлена. Значение первичного ключа считается «не установлено» назначена CLR по умолчанию для типа свойства (например, `0` для `int`, `null` для `string` и т. д.).
 - Если первичный ключ, присваивается значение, состояние имеет значение без изменений.
 - Если первичный ключ не сформированный базой данных, сущность помещается в том же состоянии, в качестве привилегированного пользователя.

Код инициализации первой базы данных

EF6 имеет значительный объем magic, выполняемых по выбору подключения к базе данных и инициализации базы данных. Ниже перечислены некоторые из этих правил.

- Если конфигурация не выполняется, EF6 выберет базу данных в SQL Express или LocalDb.
- Если строка подключения с тем же именем, как контекст — в приложениях `App/Web.config` файл, будет

использоваться это соединение.

- Если базы данных не существует, он создается.
- Если ни одной из таблиц из модели существует в базе данных, схема для текущей модели добавляется в базу данных. Если включены миграции, они используются для создания базы данных.
- Если база данных существует и EF6 ранее создали схемы, схемы проверяются на совместимость с текущей моделью. Исключение возникает в том случае, если модель данных была изменена с момента создания схемы.

EF Core не выполняет эти невероятные вещи.

- Подключение к базе данных должна быть настроена явным образом в коде.
- Инициализация не выполняется. Необходимо использовать `DbContext.Database.Migrate()` по применению миграции (или `DbContext.Database.EnsureCreated()` И `EnsureDeleted()` для создания и удаления базы данных без использования миграций).

Код первой таблицы, соглашение об именовании

EF6 выполняется с именем класса сущностей через службу преобразования во множественную форму, для которого требуется вычислить имя таблицы по умолчанию, который сопоставляется сущность.

EF Core использует имя `DbSet` свойства, предоставляемого сущность в производном контексте. Если сущность не содержит `DbSet` используется свойство, а затем имя класса.

Перенос модель на основе EDMX EF6 в EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

EF Core не поддерживает формат файла EDMX для моделей. Перенос этих моделей, лучше создать новую модель на основе кода из базы данных для вашего приложения.

Установить пакеты EF Core NuGet

Установить `Microsoft.EntityFrameworkCore.Tools` пакет NuGet.

Повторное создание модели

Теперь можно использовать функциональные возможности реконструирования, чтобы создать модель, основанную на существующей базы данных.

Выполните следующую команду в консоли диспетчера пакетов (средства → Диспетчер пакетов NuGet → консоль диспетчера пакетов). См. в разделе [консоль диспетчера пакетов \(Visual Studio\)](#) для параметры команд для формирования шаблонов подмножество таблиц и т.д.

```
Scaffold-DbContext "<connection string>" <database provider name>
```

Например вот команду, чтобы сформировать модель из базы данных блогов на локальном экземпляре SQL Server LocalDB.

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer
```

Удаление модели EF6

Теперь вам нужно удалить модель EF6 из приложения.

Это можно оставить EF6 установленный пакет NuGet (`EntityFramework`), как EF Core и EF6 можно использовать side-by-side в одном приложении. Тем не менее если вы не собираетесь использовать EF6 в любой области приложения, то при удалении пакета помогут выдают ошибки компиляции на фрагменты кода, требуют внимания.

Обновление кода

На этом этапе это вопрос адресации ошибки компиляции и проверка кода ли изменения в работе между EF6 и EF Core повлияет на вас.

Протестировать порт

Компилирует приложение, не означает, что он успешно перенесена в EF Core. Необходимо будет проверить все области приложения, чтобы убедиться, что ни одно из изменений поведение негативно повлиять на приложения.

TIP

См. в разделе [Приступая к работе с EF Core в ASP.NET Core с существующей базой данных](#) для ссылки на дополнительные о том, как работать с существующей базой данных

Перенос модель на основе кода EF6 в EF Core

28.08.2018 • 4 minutes to read • [Edit Online](#)

Если вы прочитали все предупреждения, и вы готовы к порту, Вот несколько рекомендаций, которые помогут вам приступить к работе.

Установить пакеты EF Core NuGet

Чтобы использовать EF Core, установите пакет NuGet для поставщика базы данных, которую вы хотите использовать. Например, при разработке для SQL Server, требуется установить

`Microsoft.EntityFrameworkCore.SqlServer`. См. в разделе [поставщики баз данных](#) подробные сведения.

Если вы планируете использовать migrations, то также следует установить

`Microsoft.EntityFrameworkCore.Tools` пакета.

Это можно оставить EF6 установленный пакет NuGet (`EntityFramework`), как EF Core и EF6 можно использовать side-by-side в одном приложении. Тем не менее если вы не собираетесь использовать EF6 в любой области приложения, то при удалении пакета помогут выдать ошибки компиляции на фрагменты кода, требуют внимания.

Переключить пространств имен

Большинство интерфейсов API, используемых в EF6 находятся в `System.Data.Entity` пространства имен (и связанные с ней подпространства имен). Первое изменение кода, поменяйте местами для

`Microsoft.EntityFrameworkCore` пространства имен. Вы бы, обычно начинается с файла кода производного контекста и затем разработать оттуда адресации ошибки компиляции по мере их появления.

Конфигурацию контекста (соединение д.)

Как описано в разделе [обеспечить EF Core будет действий по обеспечению приложения](#), EF Core имеет менее magic вокруг обнаружение для подключения к базе данных. Необходимо переопределить `OnConfiguring` метод в производном контексте и используйте поставщик конкретный API базы данных для настройки подключения к базе данных.

Большинство приложений EF6 хранить строки подключения в приложениях `App/Web.config` файл. В EF Core прочтением этой строки подключения с помощью `ConfigurationManager` API. Может потребоваться добавить ссылку на `System.Configuration` framework сборки, чтобы иметь возможность использовать этот API.

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(ConfigurationManager.ConnectionStrings["BloggingDatabase"].ConnectionString);
    }
}
```

Обновление кода

На этом этапе это вопрос адресации ошибки компиляции и проверки кода, чтобы увидеть, если изменения в работе повлияет на вас.

Существующие миграции

Действительно не представляется возможным способ переноса имеющихся миграций EF6 в EF Core.

Если это возможно лучше всего предполагают, что всех предыдущих операций миграции из EF6 были применены к базе данных, а затем start, миграция схемы из этой точки, с использованием EF Core. Чтобы сделать это, используйте `Add-Migration` команду, чтобы добавить миграцию, когда модель будет перенесена в EF Core. Затем, вам нужно удалить весь код из `Up` и `Down` методы шаблонный миграции. Последующей миграции сравниваются в модель при был сформированные для первоначальной миграции.

Протестировать порт

Компилирует приложение, не означает, что он успешно перенесена в EF Core. Необходимо будет проверить все области приложения, чтобы убедиться, что ни одно из изменений поведение негативно повлиять на приложения.

Entity Framework Core

10.09.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework (EF) — это упрощенная, расширяемая и кроссплатформенная версия популярной технологии для доступа к данным Entity Framework.

EF Core может использоваться как объектно реляционный модуль сопоставления (O/RM), позволяя разработчикам .NET работать с базой данных с помощью объектов .NET и устранивая необходимость в написании большей части кода, требуемого для доступа к данным.

EF Core поддерживает множество систем баз данных. Дополнительные сведения см. в разделе [Поставщики баз данных](#).

Модель

В EF Core доступ к данным осуществляется с помощью модели. Модель состоит из классов сущностей и производного контекста, который представляет сеанс взаимодействия с базой данных, позволяя запрашивать и сохранять данные. Дополнительные сведения см. в разделе [Создание модели](#).

Вы можете создать модель из существующей базы данных, вручную составить код модели, соответствующий базе данных, или использовать миграции EF, чтобы создать базу данных из модели (и модифицировать ее по мере изменения модели).

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=MyDatabase;Trusted_Connection=True;");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }
        public int Rating { get; set; }
        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

Выполнение запросов

Экземпляры классов сущностей извлекаются из базы данных с помощью LINQ. Дополнительные сведения см. в разделе [Запросы к данным](#).

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

Сохранение данных

Для создания, удаления и изменения данных в базе данных используются экземпляры классов сущностей. Дополнительные сведения см. в разделе [Сохранение данных](#).

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

Следующие шаги

Вводные руководства см. в разделе [Начало работы с Entity Framework Core](#).

Новые возможности в EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Дополнительные сведения о новых возможностях в каждом выпуске см. по следующим ссылкам:

Будущие выпуски

- [Стратегия развития EF Core](#)

Последние выпуски

- [EF Core 2.1 \(последняя выпущенная версия\);](#)
- [EF Core 2.0](#)

Прошлые выпуски

- [EF Core 1.1](#)
- [EF Core 1.0](#)

Стратегия развития Entity Framework Core

12.10.2018 • 7 minutes to read • [Edit Online](#)

IMPORTANT

Обратите внимание на то, что наборы возможностей и расписания будущих выпусков могут быть изменены, и, несмотря на то, что мы стараемся поддерживать эту страницу в актуальном состоянии, в определенные моменты времени содержание страницы может не отражать наши текущие планы.

Последний выпуск: EF Core 2.1

Стабильная версия EF Core 2.1 была выпущена 30 мая 2018 г. Дополнительные сведения об этом выпуске см. в статье [Новые возможности в EF Core 2.1](#).

Будущие выпуски

EF Core 2.2

Этот выпуск включает множество исправлений ошибок и сравнительно небольшой набор новых функций. Сведения об этом выпуске включены в [объявление о стратегическом плане по EF Core 2.2](#).

EF Core 3.0

Хотя [процесс планирования выпуска](#), следующего после 2.2, не закончен, сейчас мы планируем опубликовать основной выпуск вместе с .NET Core 3.0 и ASP.NET 3.0.

Рабочие элементы, предварительно назначенные для будущего выпуска, можно просмотреть в [этом запросе в средстве отслеживания проблем](#).

Расписание

Расписание выпусков EF Core синхронизируется с [расписанием .NET Core](#) и [расписанием ASP.NET Core](#).

Невыполненная работа

Мы используем [контрольные точки невыполненной работы](#) при отслеживании проблем для ведения подробного списка проблем и возможностей. Клиенты могут комментировать и голосовать за них.

Как правило, мы не закрываем проблемы, которые мы рассчитываем когда-нибудь решить, или с которыми может справиться кто-то из сообщества, но это не означает намерения разрешить их за конкретный период времени до тех пор, пока мы не добавим их к определенной контрольной точке как часть [процесса планирования выпусков](#).

Если мы не планируем когда-либо реализовать возможность, проблема, скорее всего, будет закрыта. Закрытие проблемы может быть пересмотрено позднее, если у нас появятся новые сведения о ней.

Другими словами, у нас недостаточно сведений о будущем, чтобы иметь возможность сказать, что возможность X будет добавлена за время или в версии Y. Как и в случае с любым программным обеспечением, приоритеты, расписания выпусков и доступные ресурсы могут изменяться в любой момент.

Процесс планирования выпусков

Мы часто получаем вопросы о том, как мы выбираем возможности, которые будут добавлены в конкретный выпуск. Наш список невыполненной работы не становится автоматически планом выпусков. Наличие возможности в EF6 также не означает автоматически, что она должна быть реализована в EF Core.

Сложно подробно описать весь процесс планирования выпуска, отчасти от того, что большую часть занимает обсуждение конкретных функций, возможностей и приоритетов, отчасти от того, что сам процесс обычно эволюционирует от выпуска к выпуску. Тем не менее можно относительно легко привести список наиболее распространенных вопросов, на которые мы пытаемся найти ответ при принятии решения о том, над чем мы будем работать на следующем этапе:

- 1. Сколько разработчиков, по нашему мнению, будет использовать новую возможность, насколько она улучшит их приложения и облегчит процесс разработки?** Для этого мы собираем отзывы из множества источников, комментариев и голосования за проблемы — один из таких источников.
- 2. Какие существуют обходные пути для использования этой возможности, пока мы ее не реализовали?** Например, многие разработчики используют сопоставление соединяемой таблицы для обхода отсутствия встроенной поддержки связи многие ко многим. Очевидно, что не все разработчики могут использовать эту возможность, но многие сумеют, и это является фактором, который влияет на принятие решения.
- 3. Будет ли реализация этой возможности способствовать развитию EF Core, то есть приблизит ли это реализацию других возможностей?** Как правило, мы отдаем предпочтение возможностям, выступающим в качестве строительных блоков для других функций. Например, разделение таблицы для собственных типов продвигает нас в реализации поддержки подхода TPT (одна таблица на тип).
- 4. Является ли возможность точкой расширяемости?** Как правило, мы отдаем предпочтение точкам расширяемости, так как они позволяют разработчикам легко реализовывать собственные алгоритмы и использовать некоторые из нереализованных возможностей. Мы планируем реализовать кое-что из этого в начале работы над отложенной загрузкой.
- 5. Каков синергетический эффект от использования этой возможности в сочетании с другими продуктами?** Как правило, мы отдаем предпочтение возможностям, позволяющим использовать EF Core с другими продуктами или значительно улучшить процесс использования других продуктов, таких как .NET Core, последняя версия Visual Studio, Microsoft Azure и т. д.
- 6. Какова квалификация людей, доступных для работы над этой возможностью, и как лучше всего распорядиться этими ресурсами?** Все члены команды EF и даже участники сообщества обладают разным уровнем опыта в различных областях, и при планировании мы должны это учитывать. Даже если бы мы захотели бросить все силы на работу над конкретной возможностью, такой как преобразование оператора GroupBy или связи многие ко многим, это было бы непрактично.

Как упоминалось ранее, этот процесс развивается от выпуска к выпуску, и в будущем мы хотим добавить больше возможностей членам сообщества разработчиков для участия в разработке плана выпуска, например облегчая рецензирование предложенных проектов возможностей и самого плана выпуска.

Новые возможности в EF Core 2.1

16.09.2018 • 11 minutes to read • [Edit Online](#)

Кроме исправления многочисленных ошибок и небольших улучшений функциональности и производительности, EF Core 2.1 содержит ряд новых интересных функций.

Отложенная загрузка

EF Core теперь содержит стандартные блоки, необходимые для создания классов сущностей, которые могут загружать свои свойства навигации по запросу. Мы также разработали новый пакет Microsoft.EntityFrameworkCore.Proxies, который использует эти стандартные блоки для создания прокси-классов отложенной загрузки на основе минимально измененных классов сущностей (например, классов с виртуальными свойствами навигации).

Дополнительные сведения об этом см. в [разделе, посвященном отложенной загрузке](#).

Параметры в конструкторах сущностей

В качестве одного из стандартных блоков для отложенной загрузки была реализована возможность создавать сущности, конструкторы которых принимают параметры. Параметры можно использовать для вставки значений свойств, делегатов отложенной загрузки и служб.

Дополнительные сведения об этом см. в [разделе, посвященном конструкторам сущностей с параметрами](#).

Преобразования значений

В предыдущих версиях EF Core было возможно сопоставление свойств только тех типов, которые поддерживались в собственном коде основного поставщика базы данных. Значения копировались между столбцами и свойствами без какого-либо преобразования. Начиная с версии EF Core 2.1 возможно преобразование значений, полученных из столбцов, прежде чем они будут применены к свойствам, и наоборот. Мы предлагаем целый ряд преобразований, которые могут при необходимости применяться стандартным способом, а также API явно задаваемой конфигурации, с помощью которого можно регистрировать настраиваемые преобразования между столбцами и свойствами. Варианты применения этой возможности:

- Хранение перечислений в строковом виде
- Сопоставление целочисленных значений со знаком с SQL Server
- Автоматические шифрование и расшифровка значений свойств

Дополнительные сведения об этом см. в [разделе, посвященном преобразованиям значений](#).

Преобразование оператора LINQ GroupBy

В версиях EF Core, предшествовавших версии 2.1, оператор LINQ GroupBy всегда вычислялся в памяти. На данный момент в большинстве типовых случаев поддерживается его преобразование в предложение SQL GROUP BY.

В этом примере показан запрос с оператором GroupBy, используемый для вычисления различных агрегатных функций:

```
var query = context.Orders
    .GroupBy(o => new { o.CustomerId, o.EmployeeId })
    .Select(g => new
    {
        g.Key.CustomerId,
        g.Key.EmployeeId,
        Sum = g.Sum(o => o.Amount),
        Min = g.Min(o => o.Amount),
        Max = g.Max(o => o.Amount),
        Avg = g.Average(o => o.Amount)
    });
});
```

Соответствующее преобразование в SQL будет иметь следующий вид:

```
SELECT [o].[CustomerId], [o].[EmployeeId],
    SUM([o].[Amount]), MIN([o].[Amount]), MAX([o].[Amount]), AVG([o].[Amount])
FROM [Orders] AS [o]
GROUP BY [o].[CustomerId], [o].[EmployeeId];
```

Присвоение начальных значений данных

В новом выпуске будет возможно предоставление начальных значений для заполнения базы данных. В отличие от версии EF6, начальные значения данных связываются с типом сущности в рамках конфигурации модели. В результате система миграции EF Core может автоматически вычислить, какие операции вставки, обновления или удалений необходимо применить при обновлении базы данных с использованием новой версии модели.

Например, чтобы настроить начальные значения данных для отправки в `OnModelCreating`, можно использовать следующее:

```
modelBuilder.Entity<Post>().HasData(new Post{ Id = 1, Text = "Hello World!" });
```

Дополнительные сведения об этом см. в [разделе, посвященном присвоению начальных значений данных](#).

Типы запросов

Модель EF Core теперь может включать типы запросов. В отличие от типов сущностей типы запросов не имеют определенных ключей и не могут вставляться, удаляться или обновляться (то есть доступны только для чтения), однако при этом они могут напрямую возвращаться запросами. Ниже приведены некоторые сценарии использования типов запросов:

- Сопоставление с представлениями без первичных ключей
- Сопоставление с таблицами без первичных ключей
- Сопоставление с запросами, определенными в модели
- Использование в качестве типа возвращаемого значения для запросов `FromSql()`

Дополнительные сведения об этом см. в [разделе, посвященном типам запросов](#).

Использование метода `Include` с производными типами

В новой версии при написании выражений для метода `Include` можно будет указывать только свойства навигации, определенные для производных типов. Для строго типизированной версии метода `Include` будет поддерживаться либо явное приведение, либо оператор `as`. Кроме того, теперь поддерживается использование ссылок на имена свойств навигации, определенных для производных типов в строковой

версии метода `Include`:

```
var option1 = context.People.Include(p => ((Student)p).School);
var option2 = context.People.Include(p => (p as Student).School);
var option3 = context.People.Include("School");
```

Дополнительные сведения об этом см. в [разделе, посвященном использованию метода `Include` с производными типами](#).

Поддержка `System.Transactions`

Мы реализовали возможность работы с функциями `System.Transactions`, такими как `TransactionScope`. Это будет возможно на платформах .NET Framework и .NET Core при использовании поставщиков баз данных, обеспечивающих соответствующую поддержку.

Дополнительные сведения об этом см. в [разделе, посвященном работе с `System.Transactions`](#).

Оптимизация порядка столбцов при первоначальной миграции

На основе отзывов клиентов мы обновили функции миграции таким образом, чтобы изначально столбцы таблиц создавались в том порядке, в котором свойства объявлены в классе. Обратите внимание, что EF Core не позволяет изменять порядок при добавлении новых элементов после создания первоначальной таблицы.

Оптимизация коррелированных вложенных запросов

Мы оптимизировали систему преобразования запросов таким образом, чтобы исключить выполнение SQL-запросов "N+1" в большинстве типовых сценариев, в которых использование свойства навигации в проекции приводит к объединению данных из корневого запроса с данными коррелированного вложенного запроса. Для оптимизации применяется буферизация результатов из вложенного запроса. Чтобы явно согласиться на использование нового поведения, необходимо соответствующим образом изменить запрос.

Например, следующий запрос обычно преобразуется в один запрос к `Customers` и N (где N соответствует возвращаемому числу клиентов) отдельных запросов к `Orders`:

```
var query = context.Customers.Select(
    c => c.Orders.Where(o => o.Amount > 100).Select(o => o.Amount));
```

Включив в нужном месте `ToList()`, вы указываете на необходимость буферизации для `Orders`, что позволяет обеспечить оптимизацию:

```
var query = context.Customers.Select(
    c => c.Orders.Where(o => o.Amount > 100).Select(o => o.Amount).ToList());
```

Обратите внимание, что этот запрос будет преобразован всего в два SQL-запроса: один к `Customers` и еще один к `Orders`.

Атрибут [Owned]

В новой версии можно настраивать [собственные типы сущностей](#), просто аннотируя тип с использованием `[Owned]` с последующей проверкой того, что сущность владельца добавлена в модель:

```
[Owned]
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

Программа командной строки dotnet-ef, входящая в пакет SDK для .NET Core

Теперь команды `dotnet-ef` входят в пакет SDK для .NET Core, поэтому необязательно использовать `DotNetCliToolReference` в проекте, чтобы выполнять перенос или формировать `DbContext` из существующей базы данных.

Дополнительные сведения о включении программ командной строки для разных версий пакета SDK для .NET Core и EF Core см. в разделе об [установке средств](#).

Пакет Microsoft.EntityFrameworkCore.Abstractions

Новый пакет содержит атрибуты и интерфейсы, которые можно использовать в проектах для улучшения функций EF Core без необходимости создавать зависимости от Core EF. Например, здесь находится атрибут `[Owned]` и интерфейс `ILazyLoader`.

События изменения состояния

Новые события `Tracked` и `StateChanged` в `ChangeTracker` можно использовать для записи логики, которая реагирует на ввод сущностей `DbContext` или изменение их состояния.

Анализатор необработанных параметров SQL

Новый анализатор кода входит в состав EF Core для обнаружения потенциально небезопасных случаев использования необработанных API SQL, например `FromSql` или `ExecuteSqlCommand`. Например, в следующем запросе будет отображено предупреждение, так как для `minAge` параметры отсутствуют:

```
var sql = $"SELECT * FROM People WHERE Age > {minAge}";
var query = context.People.FromSql(sql);
```

Совместимость с поставщиками баз данных

Рекомендуется использовать EF Core 2.1 с поставщиками, которые были обновлены и по меньшей мере протестированы для работы с EF Core 2.1.

TIP

Если вы столкнулись с неожиданными проблемами с совместимостью или другими ошибками в новых возможностях, сообщите нам об этом с помощью [средства отслеживания проблем](#).

Новые возможности в EF Core 2.0

02.09.2018 • 16 minutes to read • [Edit Online](#)

.NET Standard 2.0

Сейчас компонент EF Core ориентирован на .NET Standard 2.0, что означает, что он может работать с .NET Core 2.0, .NET Framework 4.6.1 и другими библиотеками, реализующими .NET Standard 2.0. Дополнительные сведения о поддерживаемых компонентах см. в разделе [Поддерживаемые реализации .NET](#).

Моделирование

Разбиение таблиц

Теперь можно сопоставить два типа сущностей с одной таблицей и более, где столбцы первичного ключа будут общими, а каждая строка будет соответствовать двум сущностям или более.

Чтобы использовать разбиение таблиц, нужно настроить идентифицирующее отношение (где свойства внешнего ключа формируют первичный ключ) между всеми типами сущностей, осуществляющими общий доступ к таблице.

```
modelBuilder.Entity<Product>()
    .HasOne(e => e.Details).WithOne(e => e.Product)
    .HasForeignKey<ProductDetails>(e => e.Id);
modelBuilder.Entity<Product>().ToTable("Products");
modelBuilder.Entity<ProductDetails>().ToTable("Products");
```

Принадлежащие типы

Принадлежащий тип сущности может использовать один тип среди CLR совместно с другим принадлежащим типом сущности, но так как его невозможно идентифицировать лишь по типу среди CLR, требуется переход к нему от другого типа сущности. Сущность, содержащая определяющий переход, является владельцем. При запросе владельца принадлежащие типы включаются по умолчанию.

По соглашению, для принадлежащего типа создается теневой первичный ключ, и этот тип сопоставляется с той же таблицей, что и владелец, с помощью разбиения таблицы. Это позволяет использовать принадлежащие типы аналогично сложным типам в EF6.

```

modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, cb =>
{
    cb.OwnsOne(c => c.BillingAddress);
    cb.OwnsOne(c => c.ShippingAddress);
});

public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}

```

Дополнительные сведения об этой возможности см. в [разделе, посвященном собственным типам сущностей](#).

Фильтры запросов на уровне модели

EF Core 2.0 включает новую функцию, которую мы назвали фильтром запросов на уровне модели. Она позволяет определить предикаты запросов LINQ (логическое выражение, которое обычно передается в оператор запроса LINQ) непосредственно в типах сущностей внутри модели метаданных (обычно в `OnModelCreating`). Такие фильтры автоматически применяются к любым запросам LINQ, связанным с этими типами сущностей, включая типы сущностей, указанные косвенно, например, с помощью оператора `Include` или ссылок на свойства прямой навигации. Ниже приведены некоторые типичные способы применения этой функции.

- Обратимое удаление: тип сущности определяет свойство `IsDeleted`.
- Мультитенантность: тип сущности определяет свойство `TenantId`.

Ниже приведен простой пример, демонстрирующий использование этой функции в двух указанных выше сценариях.

```

public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    public int TenantId { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>().HasQueryFilter(
            p => !p.IsDeleted
            && p.TenantId == this.TenantId );
    }
}

```

Мы определяем фильтр на уровне модели, который реализует мультитенантность и обратимое удаление для экземпляров типа сущности `Post`. Обратите внимание на использование свойства уровня экземпляра `DbContext: TenantId`. Фильтры на уровне модели будут использовать значение из правильного экземпляра контекста (то есть экземпляра контекста, выполняющего запрос).

Можно отключить фильтры для отдельных запросов LINQ с помощью оператора `IgnoreQueryFilters()`.

Ограничения

- Ссылки навигации запрещены. Эта функция может быть добавлена на основе отзывов.
- Фильтры можно определить только для корневого типа сущности в иерархии.

Сопоставление скалярных функций базы данных

В EF Core 2.0 представлено важное дополнение от [Пола Миддлтона \(Paul Middleton\)](#), позволяющее сопоставлять скалярные функции базы данных с заглушками методов, чтобы их можно было использовать в запросах LINQ и преобразовывать в SQL.

Ниже приведено краткое описание использования этой функции.

Объявите статический метод для вашего `DbContext` и аннотируйте его с помощью `DbFunctionAttribute`.

```
public class BloggingContext : DbContext
{
    [DbFunction]
    public static int PostReadCount(int blogId)
    {
        throw new Exception();
    }
}
```

Подобные методы регистрируются автоматически. После регистрации вызовы метода в запросе LINQ могут быть преобразованы в вызовы функций SQL:

```
var query =
    from p in context.Posts
    where BloggingContext.PostReadCount(p.Id) > 5
    select p;
```

Следует отметить несколько аспектов.

- По соглашению, имя метода используется в качестве имени функции (в данном случае пользовательской) при создании кода SQL, но вы можете переопределить имя и схему во время регистрации метод.
- Сейчас поддерживаются только скалярные функции.
- Необходимо создать сопоставимую функцию в базе данных. Она не создается при миграциях EF Core.

Настройка самодостаточных типов для Code First

В EF6 можно было инкапсулировать конфигурацию Code First для определенного типа сущности, создав объект, производный от `EntityTypeConfiguration`. В EF Core 2.0 мы возвращаем эту возможность.

```
class CustomerConfiguration : IEntityTypeConfiguration<Customer>
{
    public void Configure(EntityTypeBuilder<Customer> builder)
    {
        builder.HasKey(c => c.AlternateKey);
        builder.Property(c => c.Name).HasMaxLength(200);
    }
}

...
// OnModelCreating
builder.ApplyConfiguration(new CustomerConfiguration());
```

Высокая производительность

Создание пулов DbContext

Типичное использование EF Core в приложении ASP.NET Core обычно включает в себя регистрацию пользователя типа DbContext в системе внедрения зависимостей и последующее получение экземпляров этого типа через параметры конструктора в контроллерах. Это означает, что для каждого из запросов создается экземпляр DbContext.

В версии 2.0 мы предлагаем новый способ регистрации пользовательских типов DbContext в системе внедрения зависимостей, который прозрачно вводит в обиход пул многоразовых экземпляров DbContext. Для работы с пулами DbContext используйте `AddDbContextPool` вместо `AddDbContext` во время регистрации службы.

```
services.AddDbContextPool<BlogginContext>(  
    options => options.UseSqlServer(connectionString));
```

Если применяется этот метод, при запросе экземпляра DbContext контроллером мы сначала проверим, доступен ли этот экземпляр в пуле. После завершения обработки запроса любое состояние экземпляра сбрасывается, а сам экземпляр возвращается в пул.

По своему принципу этот подход похож на работу пулов подключений в поставщиках ADO.NET, кроме того, он выгоден с позиции снижения затрат на инициализацию экземпляра DbContext.

Ограничения

Новый метод накладывает ряд ограничений на то, что можно сделать в методе `OnConfiguring()` DbContext.

WARNING

Избегайте создания пулов DbContext, если используете собственное состояние (например, закрытые поля) в производном классе DbContext, которое не должно совместно использоваться разными запросами. EF Core сбросит только состояние, о котором известно перед добавлением экземпляра DbContext в пул.

Явным образом скомпилированные запросы

Это вторая функция обеспечения производительности, предоставляющая преимущества в крупномасштабных сценариях.

API вручную или явно скомпилированных запросов были доступны в предыдущих версиях EF, а также в LINQ to SQL, позволяя приложениям кэшировать преобразование запросов, чтобы их можно было вычислить всего один раз, а выполнять многократно.

Хотя в общем случае EF Core может автоматически компилировать и кэшировать запросы на основе хэшированного представления выражений запросов, этот механизм можно использовать для небольшого повышения производительности за счет обхода вычисления хеша и поиска по нему, что позволяет приложению использовать уже скомпилированный запрос с помощью вызова делегата.

```
// Create an explicitly compiled query  
private static Func<CustomerContext, int, Customer> _customerById =  
    EF.CompileQuery((CustomerContext db, int id) =>  
        db.Customers  
            .Include(c => c.Address)  
            .Single(c => c.Id == id));  
  
// Use the compiled query by invoking it  
using (var db = new CustomerContext())  
{  
    var customer = _customerById(db, 147);  
}
```

Отслеживание изменений

Attach может отслеживать граф новых и существующих сущностей.

EF Core поддерживает автоматическое формирование значений ключа посредством разнообразных механизмов. При использовании этой функции значение создается, если для свойства ключа задано значение по умолчанию для среды CLR, которое обычно равно нулю или NULL. Это означает, что граф сущностей можно передать в `DbContext.Attach` или `DbSet.Attach`, и EF Core пометит сущности, для которых ключ уже задан, как `Unchanged`, а сущности без набора ключей — как `Added`. Это упрощает вложение смешанных графов из новых и существующих сущностей при использовании сформированных ключей. `DbContext.Update` и `DbSet.Update` работают аналогично, за исключением того, что сущности с набором ключей помечаются как `Modified` вместо `Unchanged`.

Запрос

Усовершенствованное преобразование LINQ

Позволяет успешно выполнять больше запросов, увеличив при этом долю логики, вычисляемой в базе данных (а не в памяти) и уменьшив объем ненужных данных, извлекаемых из базы данных.

Усовершенствования объединений групп

Эта работа улучшает код SQL, создаваемый для объединений групп. Чаще всего объединения групп возникают в результате выполнения вложенных запросов для необязательных свойств навигации.

Интерполяция строк в `FromSql` и `ExecuteSqlCommand`

В C# 6 появилась интерполяция строк, позволяющая внедрять выражения C# непосредственно в строковые литералы, что очень удобно для создания строк во время выполнения. В EF Core 2.0 мы добавили специальную поддержку интерполированных строк в два основных API, принимающих необработанные строки SQL: `FromSql` и `ExecuteSqlCommand`. Это обеспечивает безопасное использование интерполяции строк C#. То есть этот механизм защищает от типичных ошибок внедрения кода SQL, которые могут возникнуть при динамическом формировании кода SQL во время выполнения.

Пример:

```
var city = "London";
var contactTitle = "Sales Representative";

using (var context = CreateContext())
{
    context.Set<Customer>()
        .FromSql($@"
            SELECT *
            FROM ""Customers"""
            WHERE ""City"" = {city} AND
                  ""ContactTitle"" = {contactTitle}")
        .ToArray();
}
```

В этом примере в строку формата SQL внедрены две переменные. EF Core выдаст следующий код SQL:

```
@p0='London' (Size = 4000)
@p1='Sales Representative' (Size = 4000)

SELECT *
FROM ""Customers"""
WHERE ""City"" = @p0
    AND ""ContactTitle"" = @p1
```

EF.Functions.Like()

Мы добавили свойство EF.Functions, которое EF Core или поставщики могут использовать для определения методов, соответствующих операторам или функциям базы данных, чтобы их можно вызывать в запросах LINQ. Первым примером такого метода является Like().

```
var aCustomers =
    from c in context.Customers
    where EF.Functions.Like(c.Name, "a%")
    select c;
```

Обратите внимание, что Like() предоставляетя в реализации для выполнения в памяти, что может быть удобно при работе с выполняющейся в памяти базой данных или при вычислении предиката на стороне клиента.

Управление базами данных

Обработчик преобразования во множественную форму для формирования DbContext

В EF Core 2.0 появилась новая служба *IPluralizer*, используемая для преобразования имен типов сущности в единственное число и имен DbSet — во множественное. Реализация по умолчанию является холостой, то есть это просто обработчик, куда можно подключить свой собственный преобразователь во множественное число.

Вот как разработчик может подключить свой собственный преобразователь во множественное число.

```
public class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
    {
        services.AddSingleton<IPluralizer, MyPluralizer>();
    }
}

public class MyPluralizer : IPluralizer
{
    public string Pluralize(string name)
    {
        return Inflector.Inflector.Pluralize(name) ?? name;
    }

    public string Singularize(string name)
    {
        return Inflector.Inflector.Singularize(name) ?? name;
    }
}
```

Другие

Перемещение поставщика ADO.NET SQLite в SQLitePCL.raw

Это позволяет получить более надежное решение в Microsoft.Data.Sqlite для распространения собственных двоичных файлов SQLite на разных платформах.

Всего один поставщик на модель

Значительно расширяет возможности взаимодействия поставщиков с моделью и упрощает использование соглашений, заметок и текущих API с разными поставщиками.

EF Core 2.0 теперь будет создавать отдельный объект [IModel](#) для каждого из используемых поставщиков. Обычно это является прозрачным для приложения. Это позволило упростить работу с API метаданных более

низкого уровня, например, добиться того, что любое обращение к основным концепциям реляционных метаданных всегда осуществляется путем вызова `.Relational` вместо `.SqlServer`, `.Sqlite` и т. д.

Консолидация ведения журнала и диагностики

Механизмы ведения журнала (на основе `ILogger`) и диагностики (на основе `DiagnosticSource`) теперь используют еще больше общего кода.

В версии 2.0 были изменены идентификаторы событий для сообщений, отправляемых в `ILogger`. Теперь эти идентификаторы событий являются уникальными в рамках всего кода EF Core. Эти сообщения теперь соответствуют стандартному шаблону для структурированного ведения журнала, используемому, например, MVC.

Кроме того, были изменены категории средства ведения журнала. Теперь используется известный набор категорий, доступных через `DbLoggerCategory`.

События `DiagnosticSource` теперь используют те же имена идентификаторов событий, что и соответствующие сообщения `ILogger`.

Новые возможности в EF Core 1.1

02.09.2018 • 2 minutes to read • [Edit Online](#)

Моделирование

Сопоставление полей

Позволяет настроить резервное поле для свойства. Это может быть полезно для свойств, доступных только для чтения, или данных, для которых вместо свойств используются методы Get и Set.

Сопоставление с оптимизированными для памяти таблицами в SQL Server

Вы можете указать, что сопоставленная с таблицей сущность оптимизирована для памяти. Если вы используете EF Core для создания и обслуживания базы данных на основе используемой модели (с помощью миграций или `Database.EnsureCreated()`), то для таких сущностей будет создана оптимизированная для памяти таблица.

Change tracking

Дополнительные API из EF6 для отслеживания изменений

Например, `Reload`, `GetModifiedProperties`, `GetDatabaseValues` и т. п.

Запрос

Явная загрузка

Позволяет активировать заполнение свойства навигации для сущности, ранее загруженной из базы данных.

DbSet.Find

Предоставляет простой способ получить сущность по значению первичного ключа.

Другое

Устойчивость подключений

Автоматически выполняет повторные попытки для неудачных обращений к базе данных. Это особенно полезно при подключении к базе данных SQL Azure, для которой характерны временные сбои.

Упрощенная замена служб

Позволяет легко заменить внутренние службы, используемые EF.

Возможности, добавленные в версии EF Core 1.0

02.09.2018 • 7 minutes to read • [Edit Online](#)

Платформы

.NET Framework 4.5.1

Включает консоль, WPF, WinForms, ASP.NET 4 и многое другое.

.NET Standard 1.3

Поддержка ASP.NET Core для .NET Framework и .NET Core в операционных системах Windows, OSX и Linux.

Моделирование

Базовое моделирование

На основе сущностей POCO со свойствами get и set для стандартных скалярных типов (`int`, `string` и т. д.).

Связи и свойства навигации

В модели на основе внешнего ключа можно создавать отношения "один ко многим" и "один к не более чем одному". С этими отношениями можно связать свойства навигации для простой коллекции или ссылочных типов.

Встроенные соглашения

В этом варианте исходная модель создается на основе формы классов сущностей.

Текущий API

Позволяет переопределять метод `OnModelCreating` для пользовательского контекста, чтобы настраивать модель, обнаруженную на основе соглашений.

Заметки к данным

Это атрибуты, которые можно добавлять в классы и (или) свойства сущности и которые будут влиять на модели EF. Например, добавление `[Required]` сообщит платформе EF, что соответствующее свойство является обязательным.

Сопоставление в реляционной таблице

Позволяет сопоставить сущности с таблицами и (или) столбцами.

Создание значений ключа

Поддерживается создание на стороне клиентских процессов или в базе данных.

Созданные базой данных значения

Позволяет базе данных создавать значения при вставке (для значений по умолчанию) или обновлении данных (для вычисляемых столбцов).

Последовательности в SQL Server

Позволяет определить в модели объекты последовательностей.

Ограничения UNIQUE

Позволяет определить альтернативные ключи и связи, использующие такие ключи.

Индексы

При определении индексов в модели они автоматически добавляются в базу данных. Поддерживаются

также индексы UNIQUE.

Теневые свойства состояния

Возможность определить в модели свойства, которые не объявляются и не хранятся в классе .NET, но отслеживаются и обновляются с помощью EF Core. Чаще всего применяется для свойств внешнего ключа, которые нежелательно отображать в самом объекте.

Шаблон наследования "одна таблица на иерархию"

Позволяет сохранять несколько сущностей иерархии в одну таблицу, в которой столбец дискриминатора определяет тип сущности для каждой записи в базе данных.

Проверка модели

Обнаруживает в модели неправильные соотношения и предоставляет полезные сообщения об ошибках.

Change tracking

Отслеживание изменений по моментальному снимку

Позволяет автоматически обнаруживать изменения в сущностях, сравнивая их текущее состояние с копией (моментальным снимком) исходного состояния.

Извещения об изменении состояния

Позволяет создавать уведомления для отслеживающей стороны при изменении значений свойств.

Доступ к отслеживаемому состоянию

Осуществляется через `DbContext.Entry` и `DbContext.ChangeTracker`.

Присоединение отсоединенных сущностей и диаграмм

Новый API `DbContext.AttachGraph` позволяет повторно присоединить сущности к контексту, чтобы избежать лишних операций создания или изменения объектов.

Сохранение данных

Основные возможности функции сохранения

Изменения экземпляров сущности могут сохраняться в базу данных.

Оптимистическая блокировка

Зашита от перезаписи изменений, внесенных другим пользователем после получения данных из базы данных.

Асинхронный метод сохранения `SaveChanges`

Позволяет освободить текущий поток для обработки других запросов, пока база данных обрабатывает команды от `SaveChanges`.

Транзакции базы данных

Означает, что `SaveChanges` всегда является атомарной (то есть либо завершается успешно полностью, либо не вносит никаких изменений в базу данных). Также для транзакций предоставлены специальные API, которые позволяют совместно использовать транзакции в нескольких экземплярах контекста и т. д.

Реляционные базы данных: пакетная обработка инструкций

Повышает производительность, выполняя несколько команд INSERT/UPDATE/DELETE за одно обращение к базе данных.

Запрос

Базовая поддержка LINQ

Предоставляет возможность использовать LINQ для извлечения данных из базы данных.

Смешанное вычисление на клиенте и сервере

Позволяет включать в запросы логику, которая не может быть выполнена в базе данных, и обрабатывать ее уже после получения данных в память.

NoTracking

Запросы выполняются быстрее, если контекст не отслеживает изменения экземпляров сущностей (это полезно, если результаты доступны только для чтения).

Безотложная загрузка

Предоставляет методы `Include` и `ThenInclude` для определения смежных данных, которые нужно извлекать вместе с основным запросом.

Асинхронный запрос

Позволяет освободить текущий поток и его ресурсы для обработки других запросов, пока база данных обрабатывает текущий запрос.

Необработанные SQL-запросы

Предоставляет метод `DbSet.FromSql` для получения данных с помощью необработанных SQL-запросов. Эти запросы можно также создавать с помощью LINQ.

Управление схемой базы данных

Интерфейсы API для создания и удаления баз данных

Предназначены в основном для тестирования, при котором часто бывает нужно быстро создать или удалить базу данных без использования миграции.

Миграции реляционной базы данных

Позволяет изменять схему реляционной базы данных параллельно с развитием модели.

Реконструирование из базы данных

Создает модель EF на основе существующей схемы реляционной базы данных.

Поставщики баз данных

SQL Server

Позволяет подключаться к Microsoft SQL Server 2008 и более новых версий.

SQLite

Позволяет подключаться к базе данных SQLite 3.

Выполнение в памяти

Позволяет легко выполнять тестирование без подключения к настоящим базам данных.

Сторонние поставщики

Поддерживаются несколько поставщиков других ядер СУБД. Полный их список можно найти в статье [Database Providers](#) (Поставщики баз данных).

Начало работы с Entity Framework Core

10.09.2018 • 2 minutes to read • [Edit Online](#)

Установка EF Core

Сводка по шагам, необходимым для добавления EF Core в приложение в разных платформах и популярных средах IDE.

Пошаговые учебники

Эти вводные руководства не требуют знакомства с Entity Framework Core или определенным IDE. Они содержат пошаговую процедуру создания простого приложения, которое запрашивает и сохраняет данные из базы данных. Мы предоставили учебники для начала работы с различными операционными системами и типами приложений.

Entity Framework Core позволяет создать модель на основе существующей базы данных или базу данных на основе модели. Доступны учебники, демонстрирующие оба этих подхода.

- .NET Framework (консольные приложения, WinForms, WPF)
 - [Новая база данных](#)
 - [Существующая база данных](#)
- .NET Core (Windows, macOS, Linux)
 - [Новая база данных](#)
- ASP.NET Core
 - [Новая база данных](#)
 - [Существующая база данных](#)
 - [EF Core и Razor Pages](#)
- Универсальная платформа Windows (UWP)
 - [Новая база данных](#)

NOTE

Эти руководства и сопутствующие примеры обновлены для использования EF Core 2.1. Но в большинстве случаев приложения, использующие предыдущие выпуски, можно создать с минимальными отступлениями от приведенных инструкций.

Установка Entity Framework Core

06.10.2018 • 7 minutes to read • [Edit Online](#)

Предварительные требования

- Для разработки приложений, предназначенных для .NET Core 2.1, установите [пакет SDK для .NET Core 2.1](#). Это нужно сделать, даже если у вас установлена последняя версия Visual Studio 2017.
- Для разработки в Visual Studio приложений, предназначенных для .NET Core 2.1, установите Visual Studio 2017 версии 15.7 или новее.
- Использовать Entity Framework 2.1 можно в приложениях ASP.NET Core версии 2.1. Приложения с более ранними версиями ASP.NET Core нужно обновить до версии 2.1.
- Для разработки приложений, предназначенных для .NET Framework 4.6.1 и более поздних версий, можно использовать Visual Studio 2015. Однако вам нужна версия NuGet, поддерживающая .NET Standard 2.0 и совместимые среды. Чтобы получить ее в Visual Studio 2015, [обновите клиент NuGet до версии 3.6.0](#).

Получение среды выполнения Entity Framework Core

Чтобы добавить в приложение библиотеки среды выполнения EF Core, установите пакет NuGet для поставщика базы данных, который вы хотите использовать. Список поддерживаемых поставщиков и имена их пакетов NuGet см. в разделе [Поставщики баз данных](#).

Установить или обновить пакеты NuGet можно в интерфейсе командной строки .NET Core, диалоговом окне диспетчера пакетов Visual Studio или консоли диспетчера пакетов Visual Studio.

Для приложений ASP.NET Core 2.1 поставщики в памяти и поставщики SQL Server включены автоматически и устанавливать их отдельно не требуется.

TIP

Если требуется обновить приложение, использующее сторонний поставщик базы данных, всегда ищите обновление поставщика, совместимое с нужной вам версией EF Core. Например, поставщики баз данных для предыдущих версий несовместимы с версией 2.1 среды выполнения EF Core.

Интерфейс командной строки .NET Core

Следующая команда интерфейса командной строки .NET Core устанавливает или обновляет поставщик SQL Server:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Вы можете указать в команде `dotnet add package` конкретную версию, используя модификатор `-v`. Например, чтобы установить пакеты EF Core 2.1.0, добавьте к команде `-v 2.1.0`.

Диалоговое окно диспетчера пакетов NuGet в Visual Studio

- В меню выберите **Проект > Управление пакетами NuGet**
- Нажмите кнопку **Обзор** или откройте вкладку **Обновления**.

- Чтобы установить или обновить поставщик SQL Server, выберите пакет `Microsoft.EntityFrameworkCore.SqlServer` и подтвердите свой выбор.

Дополнительные сведения: [Диалоговое окно диспетчера пакетов NuGet](#).

Консоль диспетчера пакетов NuGet в Visual Studio

- В меню выберите **Сервис > Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.
- Чтобы установить поставщик SQL Server, в консоли диспетчера пакетов выполните следующую команду:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

- Чтобы обновить поставщик, используйте команду `Update-Package`.
- Чтобы указать конкретную версию, используйте модификатор `-Version`. Например, чтобы установить пакеты EF Core 2.1.0, добавьте в команды `-Version 2.1.0`

Дополнительные сведения: [Консоль диспетчера пакетов](#).

Получение инструментов Entity Framework Core

Помимо библиотек среды выполнения, вы можете установить инструменты для выполнения в проекте определенных задач, связанных с EF Core, во время разработки. Например, вы можете создавать миграции, выполнять их и создавать модель на основе существующей базы данных.

Доступно два набора инструментов:

- [Инструменты интерфейса командной строки \(CLI\)](#) .NET Core можно использовать в Windows, Linux и macOS. Эти команды начинаются с `dotnet ef`.
- [Инструменты консоли диспетчера пакетов](#) работают в Visual Studio 2017 на Windows. Эти команды начинаются с глагола, например `Add-Migration`, `Update-Database`.

Хотя вы можете использовать команды `dotnet ef` в консоли диспетчера пакетов, в Visual Studio удобнее использовать инструменты консоли диспетчера пакетов:

- Они автоматически работают в текущем проекте, выбранном в консоли диспетчера пакетов, без необходимости ручного переключения каталогов.
- Они автоматически открывают файлы, созданные командами в Visual Studio, после завершения соответствующей команды.

Получение инструментов интерфейса командной строки

Команды `dotnet ef` входят в пакет SDK для .NET Core, но чтобы включить их, нужно установить пакет `Microsoft.EntityFrameworkCore.Design`:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

Этот пакет автоматически включается в приложения ASP.NET Core 2.1.

Как говорилось ранее в разделе [Предварительные требования](#), вам также нужно установить пакет SDK для .NET Core 2.1.

IMPORTANT

Всегда используйте версии пакетов инструментов, которые соответствуют основному номеру версии для пакетов среды выполнения.

Получение инструментов консоли диспетчера пакетов

Чтобы получить инструменты консоли диспетчера пакетов для EF Core, установите пакет

`Microsoft.EntityFrameworkCore.Tools` :

```
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

Этот пакет автоматически включается в приложения ASP.NET Core 2.1.

Обновление до EF Core 2.1

При обновлении существующего приложения до EF Core 2.1 некоторые ссылки на более старые пакеты EF Core, возможно, потребуется удалить вручную:

- Пакеты поставщиков баз данных, используемые во время разработки, например `Microsoft.EntityFrameworkCore.SqlServer.Design`, больше не требуются и не поддерживаются в EF Core 2.1, но они не удаляются автоматически при обновлении других пакетов.
- Средства .NET CLI теперь включены в пакет SDK для .NET, следовательно ссылку на этот пакет можно удалить из файла `.csproj`:

```
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
```

Убедитесь, что приложения, предназначенные для .NET Framework и созданные в более ранних версиях Visual Studio, совместимы с библиотеками .NET Standard 2.0:

- Измените файл проекта и включите в группу начальной свойств следующую запись.

```
<AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
```

- Для тестовых проектов также включите следующую запись.

```
<GenerateBindingRedirectsOutputType>true</GenerateBindingRedirectsOutputType>
```

Начало работы с EF Core в .NET Framework

28.08.2018 • 2 minutes to read • [Edit Online](#)

Для использования этих руководств базового уровня не требуется опыта работы с Entity Framework Core или Visual Studio. Они содержат пошаговую процедуру создания простого консольного приложения .NET Framework, которое запрашивает и сохраняет данные из базы данных. Вы можете выбрать учебник, в котором создается модель на основе существующей базы данных или база данных на основе модели.

Вы можете использовать методики, рассмотренные в этих учебниках, в любом приложении, предназначенном для .NET Framework, включая WPF и WinForms.

NOTE

Эти руководства и сопутствующие примеры обновлены для использования EF Core 2.1. Но в большинстве случаев приложения, использующие предыдущие выпуски, можно создать с минимальными отступлениями от приведенных инструкций.

Начало работы с EF Core в .NET Framework с новой базой данных

02.09.2018 • 5 minutes to read • [Edit Online](#)

В этом руководстве вы создадите консольное приложение, которое реализует простейший доступ к базе данных Microsoft SQL Server с помощью Entity Framework. Вы создадите из модели базу данных при помощи миграций.

[Пример для этой статьи на GitHub.](#)

Предварительные требования

- [Visual Studio 2017 версии 15.7 или выше](#)

Создание нового проекта

- Откройте Visual Studio 2017.
- "Файл" > "Создать" > "Проект"...
- В меню слева выберите "Установленные" > **Visual C# > Классическое приложение Windows**
- Выберите шаблон проекта **Консольное приложение (.NET Framework)**.
- Задайте **.NET Framework 4.6.1** или более позднюю версию в качестве целевой платформы проекта
- Назовите проект *ConsoleApp.NewDb* и нажмите кнопку **OK**

Установка Entity Framework

Чтобы использовать EF Core, установите пакеты для поставщиков базы данных, с которыми вы будете работать. В этом руководстве используется SQL Server. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

- Последовательно выберите пункты "Средства" -> "Диспетчер пакетов NuGet" -> "Консоль диспетчера пакетов".
- Запуск `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

Далее в этом руководстве вы воспользуетесь рядом инструментов Entity Framework Tools для обслуживания базы данных. Поэтому также установите пакет инструментов.

- Запуск `Install-Package Microsoft.EntityFrameworkCore.Tools`

Создание модели

Теперь нужно задать контекст и классы сущностей, которые составляют модель.

- "Проект" > "Добавить класс"...
- Введите имя класса *Model.cs* и щелкните **OK**.
- Замените все содержимое этого файла следующим кодом:

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace ConsoleApp.NewDb
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

TIP

В реальном приложении каждый класс помещается в отдельный файл, а строка подключения — в файл конфигурации или переменную среды. В этом руководстве для простоты все занесено в один файл кода.

Создание базы данных

Итак, модель готова, и вы можете создать из нее базу данных с помощью миграций.

- Последовательно выберите пункты **Средства > Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.
- Выполните `Add-Migration InitialCreate`, чтобы сформировать шаблон миграции и создать начальный набор таблиц для модели.
- Запустите `Update-Database`, чтобы применить созданную миграцию к базе данных. Так как база данных еще не существует, она будет создана перед выполнением миграции.

TIP

Если вы внесете в модель изменения, с помощью команды `Add-Migration` вы сможете сформировать новый шаблон миграции, который вносит необходимые изменения в схему базы данных. Проверьте сформированный код шаблона (и внесите правки, если потребуется), а затем выполните команду `Update-Database`, чтобы применить изменения к базе данных.

EF использует таблицу `__EFMigrationsHistory` в базе данных, чтобы отслеживать уже примененные к базе данных миграции.

Использование модели

Теперь вы можете использовать созданную модель для доступа к данным.

- Откройте файл `Program.cs`.
- Замените все содержимое этого файла следующим кодом:

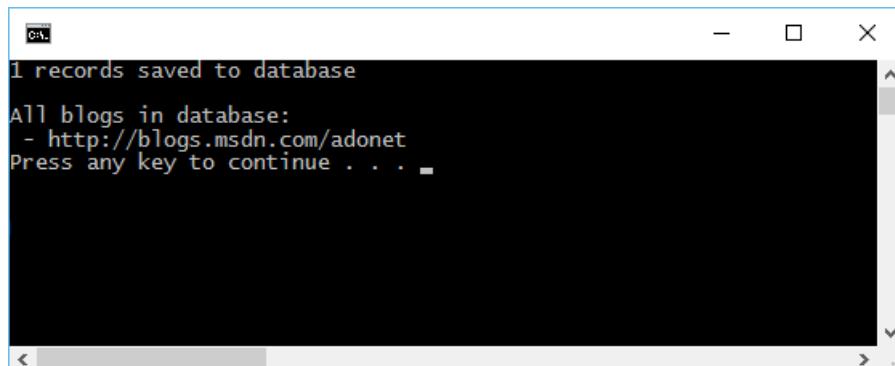
```
using System;

namespace ConsoleApp.NewDb
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BloggingContext())
            {
                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}
```

- **"Отладка" > "Запустить без отладки"**

Вы увидите, как один блог сохраняется в базе данных, а затем сведения обо всех блогах выводятся в консоль.



Дополнительные ресурсы

- [EF Core в .NET Framework с существующей базой данных](#)
- [EF Core в .NET Core с новой базой данных — SQLite. Руководство по кроссплатформенной консоли EF.](#)

Начало работы с EF Core в .NET Framework с существующей базой данных

12.10.2018 • 6 minutes to read • [Edit Online](#)

В этом руководстве вы создадите консольное приложение, которое реализует простейший доступ к базе данных Microsoft SQL Server с помощью Entity Framework. Для создания модели Entity Framework используется реконструирование существующей базы данных.

[Пример для этой статьи на GitHub.](#)

Предварительные требования

- [Visual Studio 2017 версии 15.7 или выше](#)

Создание базы данных Blogging

В этом руководстве в качестве существующей базы данных используется база **Blogging** для ведения блогов, размещенная на локальном экземпляре LocalDb. Если вы уже создали базу данных **Blogging**, работая с другим руководством, пропустите эти шаги.

- Открытие Visual Studio
- "Сервис" > "Подключение к базе данных"...
- Выберите **Microsoft SQL Server** и щелкните **Продолжить**.
- Введите значение **(localdb)\mssqllocaldb** для параметра **Имя сервера**.
- Введите значение **master** для параметра **Имя базы данных**, затем щелкните **OK**.
- Теперь база данных master появится в разделе **Подключения к данным в обозревателе сервера**.
- Щелкните правой кнопкой мыши базу данных в **обозревателе сервера** и выберите действие **Создать запрос**.
- Скопируйте представленный ниже скрипт и вставьте его в редактор запросов
- Щелкните область редактора запросов правой кнопкой мыши и выберите действие **Выполнить**.

```

CREATE DATABASE [Blogging];
GO

USE [Blogging];
GO

CREATE TABLE [Blog] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Post] (
    [PostId] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    [Content] nvarchar(max),
    [Title] nvarchar(max),
    CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog] ([BlogId]) ON DELETE CASCADE
);
GO

INSERT INTO [Blog] (Url) VALUES
('http://blogs.msdn.com/dotnet'),
('http://blogs.msdn.com/webdev'),
('http://blogs.msdn.com/visualstudio')
GO

```

Создание нового проекта

- Откройте Visual Studio 2017.
- "Файл" > "Создать" > "Проект"...
- В меню слева выберите "Установленные" > **Visual C# > Классическое приложение Windows**
- Выберите шаблон проекта **Консольное приложение (.NET Framework)**.
- Задайте **.NET Framework 4.6.1** или более позднюю версию в качестве целевой платформы проекта
- Назовите проект *ConsoleApp.ExistingDb* и нажмите кнопку **OK**

Установка Entity Framework

Чтобы использовать EF Core, установите пакеты для поставщиков базы данных, с которыми вы будете работать. В этом руководстве используется SQL Server. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

- Последовательно выберите пункты **Средства > Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.
- Запуск `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

На следующем этапе вы воспользуетесь рядом инструментов Entity Framework Tools для реконструирования базы данных. Поэтому также установите пакет инструментов.

- Запуск `Install-Package Microsoft.EntityFrameworkCore.Tools`

Реконструирование модели

Теперь пора создать модель EF на основе существующей базы данных.

- Последовательно выберите пункты **Средства -> Диспетчер пакетов NuGet -> Консоль диспетчера пакетов**.
- Выполните следующую команду, чтобы создать модель на основе существующей базы данных.

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer
```

TIP

Вы можете выбрать, для каких таблиц создавать сущности, указав в команде выше аргумент `-Tables`. Например, `-Tables Blog,Post`.

Процесс реконструирования создает классы сущностей (`Blog` и `Post`) и производный контекст (`BloggingContext`) на основе схемы существующей базы данных.

Классы сущностей — это простые объекты C#, которые представляют данные для использования в запросах и командах сохранения. Ниже приведены классы сущностей `Blog` и `Post`:

```
using System;  
using System.Collections.Generic;  
  
namespace ConsoleApp.ExistingDb  
{  
    public partial class Blog  
    {  
        public Blog()  
        {  
            Post = new HashSet<Post>();  
        }  
  
        public int BlogId { get; set; }  
        public string Url { get; set; }  
  
        public ICollection<Post> Post { get; set; }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
  
namespace ConsoleApp.ExistingDb  
{  
    public partial class Post  
    {  
        public int PostId { get; set; }  
        public int BlogId { get; set; }  
        public string Content { get; set; }  
        public string Title { get; set; }  
  
        public Blog Blog { get; set; }  
    }  
}
```

TIP

Чтобы включить отложенную загрузку, можно присвоить свойствам навигации (`Blog.Post` и `Post.Blog`) значение `virtual`.

Контекст представляет сеанс работы с базой данных. Он содержит методы, которые можно использовать для запроса и сохранения экземпляров классов сущностей.

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

namespace ConsoleApp.ExistingDb
{
    public partial class BloggingContext : DbContext
    {
        public BloggingContext()
        {

        }

        public BloggingContext(DbContextOptions<BloggingContext> options)
            : base(options)
        {
        }

        public virtual DbSet<Blog> Blog { get; set; }
        public virtual DbSet<Post> Post { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            if (!optionsBuilder.IsConfigured)
            {
#warning To protect potentially sensitive information in your connection string, you should move it out of
source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.
                optionsBuilder.UseSqlServer("Server=
(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;");
            }
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>(entity =>
            {
                entity.Property(e => e.Url).IsRequired();
            });

            modelBuilder.Entity<Post>(entity =>
            {
                entity.HasOne(d => d.Blog)
                    .WithMany(p => p.Post)
                    .HasForeignKey(d => d.BlogId);
            });
        }
    }
}
```

Использование модели

Теперь вы можете использовать созданную модель для доступа к данным.

- Откройте файл `Program.cs`.
- Замените все содержимое этого файла следующим кодом:

```

using System;

namespace ConsoleApp.ExistingDb
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BloggingContext())
            {
                db.Blog.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blog)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}

```

- Выберите "Отладка" -> "Запустить без отладки".

Вы увидите, как один блог сохраняется в базе данных, а затем сведения обо всех блогах выводятся в консоль.

```

1 records saved to database
All blogs in database:
- http://blogs.msdn.com/dotnet
- http://blogs.msdn.com/webdev
- http://blogs.msdn.com/visualstudio
- http://blogs.msdn.com/adonet
Press any key to continue . . .

```

Следующие шаги

Дополнительные сведения о формировании классов контекста и сущности см. в следующих статьях:

- [Entity Framework Core tools reference - .NET CLI](#) (Справочник по основным инструментам Entity Framework Core — .NET CLI)
- [Entity Framework Core tools reference - Package Manager Console](#) (Справочник по основным инструментам Entity Framework Core — консоль диспетчера пакетов)

Начало работы с EF Core в .NET Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Эти учебники базового уровня не требуют знакомства с Entity Framework Core или Visual Studio. Они содержат пошаговую процедуру создания простого консольного приложения .NET Core, которое запрашивает и сохраняет данные из базы данных. Эти учебники можно выполнять на любой платформе, поддерживаемой .NET Core (Windows, OSX, Linux, и т. д.).

Документацию по .NET Core см. на сайте docs.microsoft.com/dotnet/articles/core.

Начало работы с EF Core в консольном приложении .NET Core с новой базой данных

05.09.2018 • 5 minutes to read • [Edit Online](#)

В этом руководстве вы создадите консольное приложение .NET Core, которое осуществляет доступ к базе данных SQLite с помощью Entity Framework Core. Вы примените процесс миграций, чтобы создать базу данных из модели. В разделе [ASP.NET Core - новая база данных](#) описана версия для Visual Studio с использованием ASP.NET Core MVC.

Пример для этой статьи на GitHub]

(<https://github.com/aspnet/EntityFramework.Docs/tree/master/samples/core/GetStarted/NetCore/ConsoleApp.SQLite>).

Предварительные требования

- [Пакет SDK для .NET Core 2.1](#)

Создание нового проекта

- Создание консольного проекта:

```
dotnet new console -o ConsoleApp.SQLite
cd ConsoleApp.SQLite/
```

Установка Entity Framework Core

Чтобы использовать EF Core, установите пакеты для поставщиков базы данных, с которыми вы будете работать. В этом пошаговом руководстве используется SQLite. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

- Установите Microsoft.EntityFrameworkCore.Sqlite и Microsoft.EntityFrameworkCore.Design:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
dotnet add package Microsoft.EntityFrameworkCore.Design
```

- Выполните `dotnet restore`, чтобы установить новые пакеты.

Создание модели

Определите контекст и классы сущности, которые должны входить в модель:

- Создайте новый файл `Model.cs` со следующим содержимым:

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace ConsoleApp.SQLite
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=blogging.db");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public ICollection<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

Совет. В реальном приложении каждый класс помещается в отдельный файл, а строка подключения — в файл конфигурации или переменную среды. Для упрощения в этом руководстве все данные содержатся в одном файле.

Создание базы данных

Когда модель будет готова, создайте базу данных с помощью [миграций](#).

- Запустите `dotnet ef migrations add InitialCreate`, чтобы сформировать шаблон миграции и создать начальный набор таблиц для модели.
- Запустите `dotnet ef database update`, чтобы применить созданную миграцию к базе данных. Эта команда создает базу данных и применяет к ней миграции.

База данных SQLite `blogging.db` * находится в каталоге проекта.

Использование модели

- Откройте файл `Program.cs` и замените его содержимое следующим кодом:

```
using System;

namespace ConsoleApp.SQLite
{
    public class Program
    {
        public static void Main()
        {
            using (var db = new BloggingContext())
            {
                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}
```

- Проверьте работу приложения:

```
dotnet run
```

В базе данных будет создан один блог, а затем в консоли появятся сведения обо всех блогах.

```
ConsoleApp.SQLite>dotnet run
1 records saved to database

All blogs in database:
- http://blogs.msdn.com/adonet
```

Изменение модели

- Если вы внесете в модель изменения, с помощью команды `dotnet ef migrations add` вы можете сформировать новую [миграцию](#). Проверьте сформированный код (и внесите необходимые правки), а затем используйте команду `dotnet ef database update`, чтобы применить изменения схемы к базе данных.
- EF Core использует таблицу `__EFMigrationsHistory` в базе данных для отслеживания уже выполнявшихся в ней миграций.
- Ядро СУБД SQLite не поддерживает некоторые изменения схем, поддерживаемые большинством других реляционных баз данных. Например, не поддерживается операция `DropColumn`. При миграциях EF Core будет создан код для этих операций. Но если вы попробуете применить их к базе данных или сгенерировать скрипт, EF Core будет выдавать исключения. Подробнее о них можно узнать [в этой статье](#). Если вы разрабатываете новое приложение, при изменении модели лучше не использовать миграции, а просто отказаться от старой базы данных и создать новую.

Дополнительные ресурсы

- [Введение в ASP.NET Core MVC для Mac и Linux](#)
- [Введение в ASP.NET Core MVC для Visual Studio](#)
- [Начало работы с ASP.NET Core и Entity Framework Core с использованием Visual Studio](#)

Начало работы с EF Core в ASP.NET Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Эти учебники базового уровня не требуют знакомства с Entity Framework Core или Visual Studio. Они содержат пошаговую процедуру создания простого приложения ASP.NET Core, которое запрашивает и сохраняет данные из базы данных. Вы можете выбрать учебник, в котором создается модель на основе существующей базы данных или база данных на основе модели.

Документацию по ASP.NET Core см. здесь: [Введение в ASP.NET Core](#).

NOTE

Эти учебники и соответствующие примеры были обновлены до EF Core 2.0 (кроме учебника UWP, где по-прежнему используется EF Core 1.1). Но в большинстве случаев приложения, использующие предыдущие выпуски, можно создать с минимальными отступлениями от приведенных инструкций.

Начало работы с EF Core в ASP.NET Core с новой базой данных

02.09.2018 • 7 minutes to read • [Edit Online](#)

В этом руководстве вы создадите приложение ASP.NET Core MVC, которое выполняет базовые операции доступа к данным через платформу Entity Framework Core. Вы создадите из модели EF Core базу данных при помощи миграций.

[Пример для этой статьи на GitHub.](#)

Предварительные требования

Установите следующее программное обеспечение:

- [Visual Studio 2017 версии 15.7](#) со следующими рабочими нагрузками:
 - **ASP.NET и веб-разработка** в разделе **Интернет и облако**)
 - **Кроссплатформенная разработка .NET Core** (в разделе **Другие наборы инструментов**)
- [Пакет SDK для .NET Core 2.1.](#)

Создание нового проекта в Visual Studio 2017

- Откройте Visual Studio 2017.
- Последовательно выберите **Файл > Создать > Проект**.
- В меню слева выберите "Установленные" > **Visual C# > .NET Core**.
- Выберите **Новое веб-приложение ASP.NET Core**.
- Введите имя проекта **EFGetStarted.AspNetCore.NewDb** и щелкните **OK**.
- В диалоговом окне **Создание веб-приложения ASP.NET Core** сделайте следующее.
 - Выберите пункты **.NET Core** и **ASP.NET Core 2.1** в раскрывающихся списках
 - Выберите **Web Application (Model-View-Controller)** (Веб-приложение "модель — представление — контроллер").
 - Убедитесь, что для параметра **Проверка подлинности** задано значение **Без проверки подлинности**.
 - Нажмите кнопку **OK**.

Предупреждение. Если для параметра **Проверка подлинности** вы установите значение **Учетные записи отдельных пользователей** вместо **Без проверки подлинности**, в ваш проект будет добавлен файл `Models\IdentityModel.cs` с моделью Entity Framework Core. Используя методы, представленные в этом руководстве, вы можете добавить еще одну модель или расширить уже созданную, включив в нее ваши классы сущностей.

Установка Entity Framework Core

Чтобы установить EF Core, установите пакеты целевых поставщиков базы данных EF Core, с которыми вы будете работать. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

В этом руководстве используется SQL Server, поэтому устанавливать пакет поставщиков не требуется. Пакет поставщиков SQL Server включен в [метапакет Microsoft.AspNetCore.App](#).

Создание модели

Задайте класс контекста и классы сущностей, составляющие модель:

- Щелкните папку **Models** (Модели) правой кнопкой мыши и выберите **Добавить > Класс**.
- Введите имя класса **Model.cs** и щелкните **OK**.
- Замените все содержимое этого файла следующим кодом:

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace EFGetStarted.AspNetCore.NewDb.Models
{
    public class BloggingContext : DbContext
    {
        public BloggingContext(DbContextOptions<BloggingContext> options)
            : base(options)
        { }

        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public ICollection<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

В реальном приложении каждый класс из модели обычно размещается в отдельном файле. В этом руководстве для простоты все классы заносятся в один файл.

Регистрация контекста с помощью внедрения зависимостей

С помощью [внедрения зависимостей](#) службы (например, `BloggingContext`) регистрируются во время запуска приложения. Затем компоненты, которые используют эти службы (например, контроллеры MVC), обращаются к ним через параметры или свойства конструктора.

Чтобы сделать контекст `BloggingContext` доступным MVC-контроллерам, зарегистрируйте его как службу.

- Откройте файл **Startup.cs**.
- Добавьте в него следующие инструкции `using`:

```
using EFGetStarted.AspNetCore.NewDb.Models;
using Microsoft.EntityFrameworkCore;
```

Вызовите метод `AddDbContext`, чтобы зарегистрировать контекст как службу.

- Добавьте следующий выделенный код в метод `ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is needed for a given
        request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    var connection = @"Server=
(localdb)\mssqllocaldb;Database=EFGetStarted.AspNetCore.NewDb;Trusted_Connection=True;ConnectRetryCount
=0";
    services.AddDbContext<BlogginContext>
        (options => options.UseSqlServer(connection));
    // BlogginContext requires
    // using EFGetStarted.AspNetCore.NewDb.Models;
    // UseSqlServer requires
    // using Microsoft.EntityFrameworkCore;
}
}
```

Примечание. В реальном приложении строка подключения обычно помещается в файл конфигурации или переменную среды. В этом руководстве для простоты она задана в коде. Дополнительные сведения вы найдете в статье [Connection Strings](#) (Строки подключения).

Создание базы данных

Когда модель будет готова, из нее можно создать базу данных с помощью [миграций](#):

- Последовательно выберите пункты **Средства > Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.
- Запустите `Add-Migration InitialCreate`, чтобы сформировать шаблон миграции для создания начального набора таблиц для модели. Если появляется сообщение об ошибке `The term 'add-migration' is not recognized as the name of a cmdlet`, закройте и снова откройте Visual Studio.
- Запустите `Update-Database`, чтобы применить созданную миграцию к базе данных. Эта команда создает базу данных и применяет к ней миграции.

Создание контроллера

Добавьте шаблон контроллера и представлений для сущности `Blog`.

- В **обозревателе решений** щелкните папку **Контроллеры** правой кнопкой мыши и выберите пункт **Добавить -> Контроллер**.
- Выберите **Контроллер MVC с представлениями, использующий Entity Framework** и щелкните **Добавить**.
- Для параметра **Класс модели** установите значение **Блог**, а для параметра **Класс контекста данных — BlogginContext**.
- Нажмите кнопку **Добавить**.

Запуск приложения

Нажмите клавишу F5, чтобы запустить приложение для проверки.

- Перейдите к папке `/blogs`.
- С помощью ссылки создания внесите в блог несколько записей. Проверьте также работу ссылок для просмотра и удаления.

The screenshot shows a browser window with the title bar "Create - EFGetStarted.A × +". The address bar displays "localhost:2590/Blogs/Create". The main content area has a header "Create" and a sub-header "Blog". Below this, there is a form field labeled "Url" containing the value "https://blogs.msdn.com/dotnet". A "Create" button is visible below the input field. At the bottom of the page, there is a link "Back to List" and a copyright notice "© 2017 - EFGetStarted.AspNetCore.NewDb".

The screenshot shows a browser window with the title bar "Index - EFGetStarted.A × +". The address bar displays "localhost:2590/Blogs". The main content area has a header "Index" and a "Create New" link. Below this, there is a table-like structure with one row. The row contains the URL "https://blogs.msdn.com/dotnet" and three links: "Edit", "Details", and "Delete". At the bottom of the page, there is a copyright notice "© 2017 - EFGetStarted.AspNetCore.NewDb".

Дополнительные ресурсы

- Руководство по применению кроссплатформенной консоли EF с новой базой данных SQLite
- Введение в ASP.NET Core MVC для Mac и Linux
- Введение в ASP.NET Core MVC для Visual Studio

- Начало работы с ASP.NET Core и Entity Framework Core с использованием Visual Studio

Начало работы с EF Core в ASP.NET Core с существующей базой данных

12.10.2018 • 9 minutes to read • [Edit Online](#)

В этом руководстве вы создадите приложение ASP.NET Core MVC, которое выполняет базовые операции доступа к данным через платформу Entity Framework Core. Для создания модели Entity Framework используется реконструирование существующей базы данных.

[Пример для этой статьи на GitHub.](#)

Предварительные требования

Установите следующее программное обеспечение:

- [Visual Studio 2017 версии 15.7](#) со следующими рабочими нагрузками:
 - **ASP.NET и веб-разработка** в разделе **Интернет и облако**)
 - **Кроссплатформенная разработка .NET Core** (в разделе **Другие наборы инструментов**)
- [Пакет SDK для .NET Core 2.1.](#)

Создание базы данных Blogging

В этом руководстве в качестве существующей базы данных используется база данных **для ведения блогов**, размещенная на локальном экземпляре LocalDb. Если вы уже создали базу данных **Blogging**, работая с другим руководством, пропустите эти шаги.

- Открытие Visual Studio
- Выберите **Инструменты -> Подключиться к базе данных....**
- Выберите **Microsoft SQL Server** и щелкните **Продолжить**.
- Введите значение **(localdb)\mssqllocaldb** для параметра **Имя сервера**.
- Введите значение **master** для параметра **Имя базы данных**, затем щелкните **OK**.
- Теперь база данных master появится в разделе **Подключения к данным в обозревателе сервера**.
- Щелкните правой кнопкой мыши базу данных в **обозревателе сервера** и выберите действие **Создать запрос**.
- Скопируйте представленный ниже скрипт и вставьте его в редактор запросов
- Щелкните область редактора запросов правой кнопкой мыши и выберите действие **Выполнить**.

```

CREATE DATABASE [Blogging];
GO

USE [Blogging];
GO

CREATE TABLE [Blog] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Post] (
    [PostId] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    [Content] nvarchar(max),
    [Title] nvarchar(max),
    CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog] ([BlogId]) ON DELETE CASCADE
);
GO

INSERT INTO [Blog] (Url) VALUES
('http://blogs.msdn.com/dotnet'),
('http://blogs.msdn.com/webdev'),
('http://blogs.msdn.com/visualstudio')
GO

```

Создание нового проекта

- Откройте Visual Studio 2017.
- **"Файл" > "Создать" > "Проект"...**
- В меню слева выберите **"Установленные" > Visual C# > Интернет**
- Выберите шаблон проекта **Веб-приложение ASP.NET Core**
- Введите имя проекта **EFGetStarted.AspNetCore.ExistingDb** и щелкните **OK**.
- Дождитесь появления диалогового окна **Создание веб-приложения ASP.NET Core**.
- В соответствующих раскрывающихся списках должна быть выбрана целевая платформа **.NET Core** и версия **ASP.NET Core 2.1**
- Выберите шаблон **Веб-приложение (модель-представление-контроллер)**
- Убедитесь, что для параметра **Проверка подлинности** задано значение **Без проверки подлинности**.
- Нажмите кнопку **OK**.

Установка Entity Framework Core

Чтобы установить EF Core, установите пакеты целевых поставщиков базы данных EF Core, с которыми вы будете работать. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

В этом руководстве используется SQL Server, поэтому устанавливать пакет поставщиков не требуется. Пакет поставщиков SQL Server включен в [метапакет Microsoft.AspNetCore.App](#).

Реконструирование модели

Теперь пора создать модель EF на основе существующей базы данных:

- Последовательно выберите пункты **Средства -> Диспетчер пакетов NuGet -> Консоль диспетчера**

пакетов.

- Выполните следующую команду, чтобы создать модель на основе существующей базы данных:

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Если появляется сообщение об ошибке

The term 'Scaffold-DbContext' is not recognized as the name of a cmdlet , закройте и снова откройте Visual Studio.

TIP

Вы можете выбрать, для каких таблиц нужно создать сущности, указав в команде выше аргумент `-Tables`.

Например, `-Tables Blog,Post`.

Процесс реконструирования создает классы сущностей (`Blog.cs` & `Post.cs`) и производный контекст (`BloggingContext.cs`) на основе схемы существующей базы данных.

Классы сущностей — это простые объекты C#, которые представляют данные для использования в запросах и командах сохранения. Ниже приведены классы сущностей `Blog` и `Post`:

```
using System;  
using System.Collections.Generic;  
  
namespace EFGetStarted.AspNetCore.ExistingDb.Models  
{  
    public partial class Blog  
    {  
        public Blog()  
        {  
            Post = new HashSet<Post>();  
        }  
  
        public int BlogId { get; set; }  
        public string Url { get; set; }  
  
        public ICollection<Post> Post { get; set; }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
  
namespace EFGetStarted.AspNetCore.ExistingDb.Models  
{  
    public partial class Post  
    {  
        public int PostId { get; set; }  
        public int BlogId { get; set; }  
        public string Content { get; set; }  
        public string Title { get; set; }  
  
        public Blog Blog { get; set; }  
    }  
}
```

TIP

Чтобы включить отложенную загрузку, можно присвоить свойствам навигации (`Blog.Post` и `Post.Blog`) значение `virtual`.

Контекст соответствует сеансу взаимодействия с базой данных и позволяет запрашивать и сохранять экземпляры классов сущностей.

```
public partial class BloggingContext : DbContext
{
    public BloggingContext()
    {
    }

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    {
    }

    public virtual DbSet<Blog> Blog { get; set; }
    public virtual DbSet<Post> Post { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            #warning To protect potentially sensitive information in your connection string, you should move it
            out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection
            strings.
            optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;");
        }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>(entity =>
        {
            entity.Property(e => e.Url).IsRequired();
        });

        modelBuilder.Entity<Post>(entity =>
        {
            entity.HasOne(d => d.Blog)
                .WithMany(p => p.Post)
                .HasForeignKey(d => d.BlogId);
        });
    }
}
```

Регистрация контекста с помощью внедрения зависимостей

Внедрение зависимостей является ключевой концепцией для ASP.NET Core. С помощью внедрения зависимостей службы (например, `BloggingContext`) регистрируются во время запуска приложения. Затем компоненты, которые используют эти службы (например, контроллеры MVC), обращаются к ним через параметры или свойства конструктора. Дополнительные сведения о внедрении зависимостей вы найдете в статье [Introduction to Dependency Injection in ASP.NET Core](#) (Знакомство с концепцией внедрения зависимостей в ASP.NET Core) на сайте документации по этой платформе.

Регистрация и настройка контекста в файле Startup.cs

Чтобы сделать контекст `BloggingContext` доступным MVC-контроллерам, зарегистрируйте его как службу.

- Откройте файл **Startup.cs**.
- Добавьте в начало этого файла следующие инструкции `using`.

```
using EFGetStarted.AspNetCore.ExistingDb.Models;
using Microsoft.EntityFrameworkCore;
```

Теперь вы можете использовать метод `AddDbContext(...)`, чтобы зарегистрировать его как службу.

- Найдите в коде метод `ConfigureServices(...)`.
- Добавьте в него следующий выделенный код, который регистрирует контекст в качестве службы

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is needed for a given
        request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    var connection = @"Server=
(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;ConnectRetryCount=0";
    services.AddDbContext<BloggingContext>(options => options.UseSqlServer(connection));
}
```

TIP

В реальном приложении строка подключения обычно помещается в файл конфигурации или переменную среды. Для простоты в этом руководстве она будет задана в коде. Дополнительные сведения см. в статье [Connection Strings](#) (Строки подключения).

Создание контроллера и представлений

- В **обозревателе решений** щелкните папку **Контроллеры** правой кнопкой мыши и выберите пункт **Добавить -> Контроллер**.
- Выберите **Контроллер MVC с представлениями, использующий Entity Framework** и щелкните **OK**.
- Для параметра **Класс модели** установите значение **Блог**, а для параметра **Класс контекста данных** — **BloggingContext**.
- Щелкните **Добавить**.

Запуск приложения

Теперь вы можете запустить приложение и увидеть, как оно работает:

- Выберите **Отладка -> Запустить без отладки**.
- Будет выполнена сборка приложения, и оно откроется в браузере
- Перейдите к папке `/blogs`.

- Щелкните **Создать**.
- Введите **URL-адрес** для нового блога и щелкните **Создать**.

Create

Blog

Url

https://blogs.msdn.com/dotnet

[Create](#)

[Back to List](#)

© 2017 - EFGetStarted.AspNetCore.NewDb

Url	Action
http://blogs.msdn.com/dotnet	Edit Details Delete
http://blogs.msdn.com/webdev	Edit Details Delete
http://blogs.msdn.com/visualstudio	Edit Details Delete
http://blogs.msdn.com/dotnet	Edit Details Delete
http://blogs.msdn.com/webdev	Edit Details Delete
http://blogs.msdn.com/visualstudio	Edit Details Delete
http://blogs.msdn.com/dotnet	Edit Details Delete
http://blogs.msdn.com/webdev	Edit Details Delete
http://blogs.msdn.com/visualstudio	Edit Details Delete

© 2017 - EFGetStarted.AspNetCore.ExistingDb

Следующие шаги

Дополнительные сведения о формировании классов контекста и сущности см. в следующих статьях:

- [Entity Framework Core tools reference - .NET CLI](#) (Справочник по основным инструментам Entity Framework Core — .NET CLI)
- [Entity Framework Core tools reference - Package Manager Console](#) (Справочник по основным инструментам Entity Framework Core — консоль диспетчера пакетов)

Начало работы с EF Core на универсальной платформе Windows (UWP)

12.10.2018 • 2 minutes to read • [Edit Online](#)

Для использования этих руководств базового уровня не требуется опыт работы с Entity Framework Core или Visual Studio. Они содержат пошаговую процедуру создания простого приложения для универсальной платформы Windows (UWP), которое запрашивает и сохраняет данные из базы данных с помощью EF Core.

Дополнительные сведения о разработке приложений UWP можно найти в [документации по универсальной платформе Windows](#).

Начало работы с EF Core на универсальной платформе Windows (UWP) с новой базой данных

05.09.2018 • 10 minutes to read • [Edit Online](#)

В этом руководстве вы создадите приложение универсальной платформы Windows (UWP), которое осуществляет простейший доступ к локальной базе данных SQLite с помощью Entity Framework Core.

[Пример для этой статьи на GitHub.](#)

Предварительные требования

- [Windows 10 Fall Creators Update \(10.0, сборка 16299\)](#) или более поздняя версия.
- [Visual Studio 2017 версии 15.7](#) или более поздняя версия с рабочей нагрузкой **Разработка приложений для универсальной платформы Windows**.
- [Пакет SDK для .NET Core 2.1](#) или более поздняя версия.

Создание проекта модели

IMPORTANT

Из-за ограниченных возможностей взаимодействия инструментов .NET Core с проектами UWP, если вы хотите запускать команды миграции в **консоли диспетчера пакетов** (PMC), модель следует поместить в проект не для UWP

- Открытие Visual Studio
- Последовательно выберите **Файл > Создать > Проект**.
- В меню слева выберите "**Установленные**" > **Visual C# > .NET Standard**.
- Выберите шаблон **Библиотека классов (.NET Standard)**.
- Назовите проект *Blogging.Model*.
- Назовите решение *Blogging*.
- Нажмите кнопку **OK**.

Установка Entity Framework Core

Чтобы использовать EF Core, установите пакеты для поставщиков базы данных, с которыми вы будете работать. В этом руководстве используется SQLite. Список доступных поставщиков вы найдете в разделе [Database Providers](#) (Поставщики базы данных).

- Выберите "**Сервис**" > "**Диспетчер пакетов NuGet**" > "**Консоль диспетчера пакетов**".
- Запуск `Install-Package Microsoft.EntityFrameworkCore.Sqlite`

Далее в этом руководстве вы воспользуетесь рядом инструментов Entity Framework Core для обслуживания базы данных. Поэтому также установите пакет инструментов.

- Запуск `Install-Package Microsoft.EntityFrameworkCore.Tools`

Создание модели

Теперь нужно задать контекст и классы сущностей, которые составляют модель.

- Удалите `Class1.cs`.
- Создайте файл `Model.cs` со следующим кодом:

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Blogging.Model
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=blogging.db");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

Создание нового проекта UWP

- В **обозревателе решений** щелкните решение правой кнопкой мыши и выберите "**Добавить**" > "**Новый проект**".
- В меню слева выберите "**Установленные**" > **Visual C#** > "**Универсальные приложения для Windows**".
- Выберите шаблон проекта **Пустое приложение (универсальное приложение для Windows)**.
- Назовите проект `Blogging.UWP` и нажмите кнопку **OK**
- Задайте в качестве целевой и минимальной версий как минимум **Windows 10 Fall Creators Update (10.0, сборка 16299.0)**.

Создание первоначальной миграции

Теперь, когда у вас есть модель, настройте приложение на создание базы данных при его первом запуске. В этом разделе вы создадите начальную миграцию. В следующем разделе вы добавите код, который выполняет эту миграцию при запуске приложения.

Инструментам миграции требуется автозагружаемый проект, не являющийся проектом UWP, поэтому сначала создайте его.

- В **обозревателе решений** щелкните решение правой кнопкой мыши и выберите **"Добавить" > "Новый проект"**.
- В меню слева выберите **"Установленные" > Visual C# > .NET Core**.
- Выберите шаблон проекта **Консольное приложение (.NET Core)**.
- Назовите проект *Blogging.Migrations.Startup* и нажмите кнопку **OK**.
- Добавьте ссылку на проект из *Blogging.Migrations.Startup* в *Blogging.Model*.

Теперь вы можете создать начальную миграцию.

- Последовательно выберите пункты **Средства > Диспетчер пакетов NuGet > Консоль диспетчера пакетов**.
- Выберите проект *Blogging.Model* в качестве **проекта по умолчанию**.
- В **обозревателе решений** задайте проект *Blogging.Migrations.Startup* как автозагружаемый.
- Запустите `Add-Migration InitialCreate`.

Эта команда формирует миграцию, которая создает начальный набор таблиц для вашей модели.

Создание базы данных при запуске приложения

Так как вы хотите создать базу данных на устройстве, где работает приложение, добавьте код, который при запуске приложения выполнит все незавершенные миграции для локальной базы данных. Таким образом локальная база данных будет создана автоматически при первом запуске приложения.

- Добавьте ссылку на проект из *Blogging.UWP* в *Blogging.Model*.
- Откройте файл *App.xaml.cs*.
- Добавьте выделенный код, который выполняет незавершенные миграции.

```
using Blogging.Model;
using Microsoft.EntityFrameworkCore;
using System;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace Blogging.UWP
{
    /// <summary>
    /// Provides application-specific behavior to supplement the default Application class.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Initializes the singleton application object. This is the first line of authored code
        /// executed, and as such is the logical equivalent of main() or WinMain().
        /// </summary>
        public App()
    }
}
```

```

public App()
{
    this.InitializeComponent();
    this.Suspending += OnSuspending;

    using (var db = new BloggingContext())
    {
        db.Database.Migrate();
    }
}

/// <summary>
/// Invoked when the application is launched normally by the end user. Other entry points
/// will be used such as when the application is launched to open a specific file.
/// </summary>
/// <param name="e">Details about the launch request and process.</param>
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;

    // Do not repeat app initialization when the Window already has content,
    // just ensure that the window is active
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation context and navigate to the first page
        rootFrame = new Frame();

        rootFrame.NavigationFailed += OnNavigationFailed;

        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            //TODO: Load state from previously suspended application
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }

    if (e.PrelaunchActivated == false)
    {
        if (rootFrame.Content == null)
        {
            // When the navigation stack isn't restored navigate to the first page,
            // configuring the new page by passing required information as a navigation
            // parameter
            rootFrame.Navigate(typeof(MainPage), e.Arguments);
        }
        // Ensure the current window is active
        Window.Current.Activate();
    }
}

/// <summary>
/// Invoked when Navigation to a certain page fails
/// </summary>
/// <param name="sender">The Frame which failed navigation</param>
/// <param name="e">Details about the navigation failure</param>
void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
{
    throw new Exception("Failed to load Page " + e.SourcePageType.FullName);
}

/// <summary>
/// Invoked when application execution is being suspended. Application state is saved
/// without knowing whether the application will be terminated or resumed with the contents
/// of memory still intact.
/// </summary>
/// <param name="sender">The source of the suspend request.</param>
/// <param name="e">Details about the suspend request.</param>

```

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background activity
    deferral.Complete();
}
```

TIP

Если вы внесете в модель изменения, с помощью команды `Add-Migration` вы сможете сформировать новую миграцию, которая внесет соответствующие изменения в базу данных. Теперь при каждом запуске приложения к локальной базе данных на каждом устройстве будут применяться все незавершенные миграции.

EF использует таблицу `__EFMigrationsHistory` в базе данных, чтобы отслеживать уже примененные к базе данных миграции.

Использование модели

Теперь вы можете использовать созданную модель для доступа к данным.

- Откройте файл `MainPage.xaml`.
- Добавьте обработчик загрузки страницы и содержимое пользовательского интерфейса, которые выделены ниже.

```
<Page
    x:Class="Blogging.UWP.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Blogging.UWP"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Loaded="Page_Loaded">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel>
            <TextBox Name="NewBlogUrl"></TextBox>
            <Button Click="Add_Click">Add</Button>
            <ListView Name="Blogs">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding Url}" />
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackPanel>
    </Grid>
</Page>
```

Теперь добавьте код, который свяжет пользовательский интерфейс с базой данных

- Откройте файл `MainPage.xaml.cs`.
- Добавьте в него код, выделенный в следующем листинге:

```

using Blogging.Model;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace Blogging.UWP
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        private void Page_Loaded(object sender, RoutedEventArgs e)
        {
            using (var db = new BloggingContext())
            {
                Blogs.ItemsSource = db.Blogs.ToList();
            }
        }

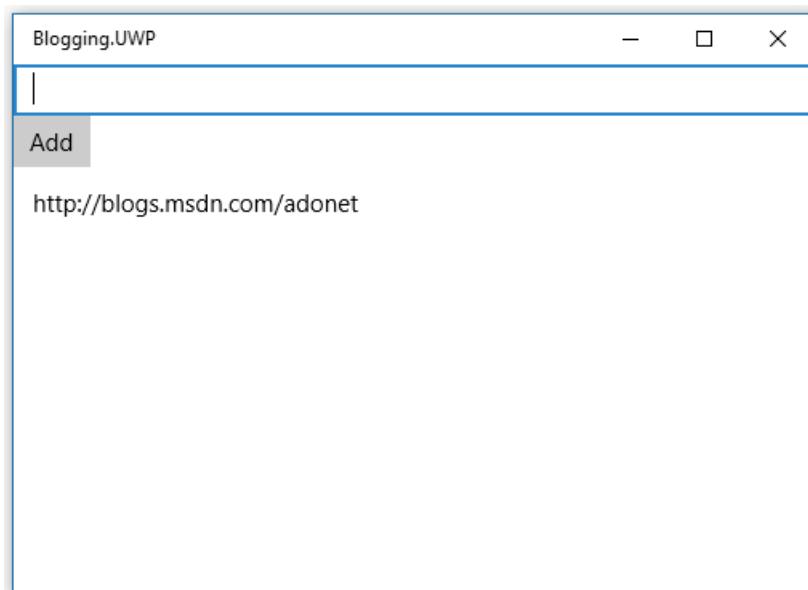
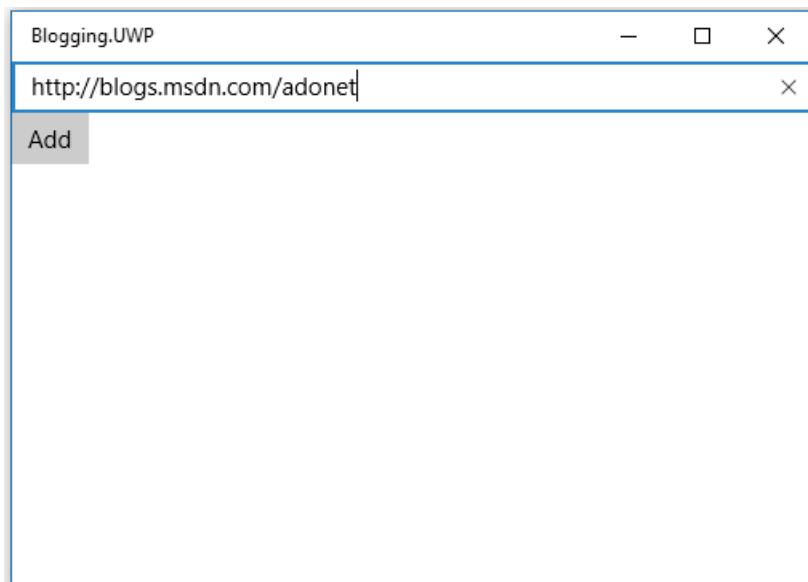
        private void Add_Click(object sender, RoutedEventArgs e)
        {
            using (var db = new BloggingContext())
            {
                var blog = new Blog { Url = NewBlogUrl.Text };
                db.Blogs.Add(blog);
                db.SaveChanges();

                Blogs.ItemsSource = db.Blogs.ToList();
            }
        }
    }
}

```

Теперь вы можете запустить приложение и увидеть, как оно работает:

- В **обозревателе решений** щелкните правой кнопкой мыши проект *Blogging.UWP* и выберите **Развернуть**.
- Задайте *Blogging.UWP* в качестве автозагружаемого проекта.
- **"Отладка" > "Запустить без отладки"**
Произойдет сборка и запуск приложения.
- Введите URL-адрес и нажмите кнопку **Добавить**.



Готово! Теперь у вас есть простое приложение UWP, в котором работает Entity Framework Core.

Следующие шаги

Сведения о совместимости и производительности, которые нужно знать при использовании EF Core с UWP, см. в разделе [Реализации .NET, поддерживаемые EF Core](#).

Другие статьи этой документации помогут вам расширить свои знания о возможностях Entity Framework Core.

Создание и настройка модели

12.10.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework использует набор соглашений для создания модели на основе формы классов сущностей. Вы можете указать дополнительную конфигурацию, чтобы дополнить и (или) переопределить данные, обнаруженные соглашением.

Эта статья описывает конфигурацию, которую можно применить к модели, ориентированной на любое хранилище данных, а также при ориентации на любую реляционную базу данных. Поставщики также могут включить конфигурацию, предназначенную для конкретного хранилища данных. Документацию по конфигурации для конкретного поставщика см. в разделе [Поставщики баз данных](#).

TIP

Для этой статьи вы можете просмотреть [пример](#) в репозитории GitHub.

Использование текущего API для настройки модели

Вы можете переопределить метод `OnModelCreating` в производном контексте и использовать `ModelBuilder API` для настройки модели. Это наиболее эффективный метод настройки, который позволяет задать конфигурацию без изменения ваших классов сущностей. Конфигурация текущего API имеет самый высокий приоритет и переопределяет соглашения и заметки к данным.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .IsRequired();
    }
}
```

Использование заметок к данным для настройки модели

Вы также можете применять для классов и свойств атрибуты (называемые заметками к данным). Заметки к данным переопределяют соглашения, но они будут перезаписаны конфигурацией текущего API.

```
public class Blog
{
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
```

Включение и исключение типов

28.08.2018 • 2 minutes to read • [Edit Online](#)

В модели означает EF содержит метаданные о, введите и будет предпринята попытка чтения и записи экземпляров в базе данных, включая тип.

Соглашения

По соглашению, типов, которые доступны в `DbSet` свойства в контексте, включены в модель. Кроме того, типы, упомянутые в `OnModelCreating` метод также будут включены. Наконец все типы, которые можно найти, рекурсивно, изучение свойства навигации обнаруженных типов также включены в модели.

Например в следующем примере кода обнаруживаются всех трех типов:

- `Blog` так как она предоставляется в `DbSet` свойство контекста
- `Post` Поскольку обнаружения с помощью `Blog.Posts` свойство навигации
- `AuditEntry` так как оно встречается в `OnModelCreating`

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

Заметки к данным

Чтобы исключить тип из модели можно использовать заметки к данным.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogMetadata Metadata { get; set; }
}

[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Текущий API

Можно использовать Fluent API, чтобы исключить тип из модели.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogMetadata Metadata { get; set; }
}

public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

Включение и исключение свойств

28.08.2018 • 2 minutes to read • [Edit Online](#)

Включая свойства в модели означает, что EF содержит метаданные об этом свойстве и будет предпринята попытка чтения и записи значений в базе данных.

Соглашения

По соглашению будут включены свойства getter и setter в модели.

Заметки к данным

Чтобы исключить свойство из модели можно использовать заметки к данным.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

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

Текущий API

Fluent API можно использовать, чтобы исключить свойство из модели.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Ignore(b => b.LoadedFromDatabase);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public DateTime LoadedFromDatabase { get; set; }
}
```

Ключи (первичные)

29.08.2018 • 2 minutes to read • [Edit Online](#)

Ключ служит в качестве основного уникальный идентификатор для каждого экземпляра сущности. При использовании реляционной базы данных это значение соответствует значению концепции *первичный ключ*. Вы также можете настроить уникальный идентификатор, который не является первичным ключом (см. в разделе [альтернативные ключи](#) Дополнительные сведения).

Соглашения

По соглашению, свойство с именем `Id` или `<type name>Id` будет настроен в качестве ключа сущности.

```
class Car
{
    public string Id { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

```
class Car
{
    public string CarId { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Заметки к данным

Заметки к данным можно использовать для настройки одно свойство в качестве ключа сущности.

```
class Car
{
    [Key]
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки одно свойство в качестве ключа сущности.

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => c.LicensePlate);
    }
}

class Car
{
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Fluent API также можно настроить несколько свойств в качестве ключа сущности (известный как составной ключ). Составные ключи можно настроить только с помощью Fluent API — соглашения никогда не настроит составной ключ и заметки к данным нельзя использовать для его настройки.

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => new { c.State, c.LicensePlate });
    }
}

class Car
{
    public string State { get; set; }
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

Созданные значения

10.09.2018 • 7 minutes to read • [Edit Online](#)

Шаблоны для создания значения

Существует три шаблона создания значения, которые могут использоваться для свойства.

- Без создания значение
- Добавить значение, созданное в
- Значение, созданное на добавление или обновление

Без создания значение

Без создания значение означает, что всегда будет указано допустимое значение сохраняется в базе данных. Это допустимое значение должны быть назначены новые сущности, прежде чем они добавляются в контекст.

Добавить значение, созданное в

Значение, созданное на добавление означает, что значение, создаваемое для новых сущностей.

В зависимости от используемого поставщика баз данных, значения может быть создан на стороне клиента с EF или в базе данных. Если значение создается в базе данных, EF может назначить это временное значение при добавлении сущности в контекст. Это временное значение затем заменено значение сформированный базой данных во время `SaveChanges()`.

При добавлении сущности в контекст, имеет значение, присваиваемое свойству, EF попытается вставить это значение вместо создания нового. Свойство видимости, имеет значение, если не назначено значение по умолчанию среды CLR (`null` для `string`, `0` для `int`, `Guid.Empty` для `Guid` и т. д.). Дополнительные сведения см. в разделе [явные значения для создаваемых свойств](#).

WARNING

Способ создания значения для добавленных сущностей будет зависеть от используемого поставщика базы данных. Поставщики баз данных может автоматически настроить создание значений для некоторых типов свойств, но другие могут потребовать ручной настройки, как создается значение.

Например, при использовании SQL Server, значения будут создаваться автоматически для `GUID` свойства (с помощью `GUID` алгоритм последовательной обработки SQL Server). Тем не менее если указать, что `DateTime` создается свойство при добавлении, то необходимо настроить способ создать значения. — Это один из способов сделать это, чтобы настроить значение по умолчанию `GETDATE()`, см. в разделе [значения по умолчанию](#).

Значение, созданное на добавление или обновление

Значение, созданное на добавление или обновление означает, что новое значение формируется каждый раз при сохранении записи (`insert` или `update`).

Как и `value generated on add`, если указать значение для свойства на вновь добавленный экземпляр сущности, что значение будет вставлен вместо того чтобы создаваемый значение. Это также можно задать явно заданное значение при обновлении. Дополнительные сведения см. в разделе [явные значения для создаваемых свойств](#).

WARNING

Способ создания значения для добавленных и измененных сущностей будет зависеть от используемого поставщика базы данных. Поставщики баз данных может автоматически настраивает Создание значений для некоторых типов свойств, а другие будут требовать ручной настройки, как создается значение.

Например, при использовании SQL Server `byte[]` свойства, заданные при создании на добавление или обновление и помечен как маркеры параллелизма, будет настроена с `rowversion` тип данных — так, что значения будут создаваться в базе данных. Тем не менее если указать, что `DateTime` создается свойство добавить или обновить, а затем необходимо настроить способ создать значения. — Это один из способов сделать это, чтобы настроить значение по умолчанию `GETDATE()` (см. в разделе [значения по умолчанию](#)) для формирования значений для новых строк. Затем можно использовать триггер базы данных для формирования значений во время обновления (например, следующий триггер пример).

```
CREATE TRIGGER [dbo].[Blogs_UPDATE] ON [dbo].[Blogs]
    AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF ((SELECT TRIGGER_NESTLEVEL()) > 1) RETURN;

    DECLARE @Id INT

    SELECT @Id = INSERTED.BlogId
    FROM INSERTED

    UPDATE dbo.Blogs
    SET LastUpdated = GETDATE()
    WHERE BlogId = @Id
END
```

Соглашения

По соглашению не являющиеся составными первичных ключей типа short, int, long или Guid будет настроена для значения, создаваемые на добавление. Все остальные свойства будет настроена с без создания значение.

Заметки к данным

Без создания значение (заметки к данным)

```
public class Blog
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Значение, созданное на добавление (заметки к данным)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public DateTime Inserted { get; set; }
}
```

WARNING

Это просто позволяет платформе EF, что значения, сформированные для добавленных сущностей, он не гарантирует, что EF действия по настройке реальные механизмы для создания значений. См. в разделе [добавить значение, созданное в более подробные сведения](#).

Значение, созданное на добавление или обновление (заметки к данным)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime LastUpdated { get; set; }
}
```

WARNING

Это просто позволяет платформе EF, что значения, сформированные для добавленных или обновленных сущностей, он не гарантирует, что EF действия по настройке реальные механизмы для создания значений. См. в разделе [значение, созданное в добавлении или обновлении](#) более подробные сведения.

Текущий API

Fluent API можно использовать для изменения шаблона создания значение для заданного свойства.

Без создания значение (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.BlogId)
    .ValueGeneratedNever();
```

Значение, созданное на добавление (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.Inserted)
    .ValueGeneratedOnAdd();
```

WARNING

`ValueGeneratedOnAdd()` дает платформе EF, что значения, сформированные для добавленных сущностей, он не гарантирует, что EF действия по настройке реальные механизмы для создания значений. См. в разделе [добавить значение, созданное в более подробные сведения](#).

Значение, созданное на добавление или обновление (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.LastUpdated)
    .ValueGeneratedOnAddOrUpdate();
```

WARNING

Это просто позволяет платформе EF, что значения, сформированные для добавленных или обновленных сущностей, он не гарантирует, что EF действия по настройке реальные механизмы для создания значений. См. в разделе [значение, созданное в добавлении или обновлении](#) более подробные сведения.

Обязательные и необязательные свойства

28.08.2018 • 2 minutes to read • [Edit Online](#)

Свойство не является обязательным, если он является допустимым, чтобы содержать `null`. Если `null` не является допустимым значением для связанной со свойством, то он считается обязательным свойством.

Соглашения

По соглашению, свойство, тип которого CLR может содержать значение `null` будет настроен как необязательный (`string`, `int?`, `byte[]` и т. д.). Свойства, типом CLR не может содержать значение `null`, будут задаваться при необходимости (`int`, `decimal`, `bool` и т. д.).

NOTE

Свойство CLR, тип которого не может содержать значение `null`, нельзя настроить как необязательные. Свойство будет расцениваться как требуется для Entity Framework.

Заметки к данным

Заметки к данным можно использовать для указания, что свойство является обязательным.

```
public class Blog
{
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
```

Текущий API

Fluent API можно использовать для указания, что свойство является обязательным.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .IsRequired();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Максимальная длина

28.08.2018 • 2 minutes to read • [Edit Online](#)

Настройка максимальной длиной предоставляет подсказку в хранилище данных о соответствующего типа данных для заданного свойства. Максимальная длина применяется только к типы данных массивов, такие как `string` и `byte[]`.

NOTE

Платформа Entity Framework не выполняет проверку максимальной длиной перед передачей данных к поставщику. Возлагается поставщик или хранилище данных для проверки при необходимости. Например когда исключение, так как тип данных базового столбца для различных версий SQL Server, превышающего максимальную длину приведет к не позволит избыточные данные для сохранения.

Соглашения

По соглашению он остается до поставщик базы данных, чтобы выбрать соответствующий тип данных для свойства. Для свойств, которые имеют длину поставщик базы данных обычно будет выбирать тип данных, который позволяет самая длинная длина данных. Например, Microsoft SQL Server будет использовать `nvarchar(max)` для `string` свойства (или `nvarchar(450)` Если столбец используется в качестве ключа).

Заметки к данным

Заметки данных можно использовать для настройки Максимальная длина для свойства. В этом примере для различных версий SQL Server, в результате `nvarchar(500)` используемого типа данных.

```
public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки Максимальная длина для свойства. В этом примере для различных версий SQL Server, в результате `nvarchar(500)` используемого типа данных.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasMaxLength(500);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Маркеры параллелизма

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

На этой странице приводятся способы настройки маркеры параллелизма. См. в разделе [обработка конфликтов параллелизма](#) подробное описание того, как работает управление параллелизмом в EF Core и примеры для обработки конфликтов параллелизма в приложении.

Свойства, настроенные как маркеры параллелизма, используемые для реализации управления оптимистичным параллелизмом.

Соглашения

По соглашению свойства никогда не настраиваются как маркеры параллелизма.

Заметки к данным

Заметки данных можно использовать для настройки свойства в качестве маркера параллелизма.

```
public class Person
{
    public int PersonId { get; set; }

    [ConcurrencyCheck]
    public string LastName { get; set; }

    public string FirstName { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки свойства в качестве маркера параллелизма.

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .Property(p => p.LastName)
            .IsConcurrencyToken();
    }
}

public class Person
{
    public int PersonId { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Метка времени или версии

Метка времени — это свойство, где новое значение формируется по базе данных каждый раз при вставке или обновлении строки. Свойство также рассматриваются как маркер параллелизма. Это гарантирует, что вы получите исключение, если кто-то еще изменил строку, которую вы пытаетесь обновить, так как запросы для данных.

Как это можно сделать, зависит от используемого поставщика базы данных. Для SQL Server, метки времени обычно используется на `byte[]` свойство, которое будет настроить как `ROWVERSION` столбца в базе данных.

Соглашения

По соглашению свойства никогда не настраиваются как отметки времени.

Заметки к данным

Заметки к данным можно использовать для настройки свойства в качестве метки времени.

```
public class Blog
{
    public int BlogId { get; set; }

    public string Url { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки свойства в качестве метки времени.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(p => p.Timestamp)
            .IsRowVersion();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public byte[] Timestamp { get; set; }
}
```

Свойства тени

29.08.2018 • 3 minutes to read • [Edit Online](#)

Теневые свойства являются свойствами, которые не определены в классе сущностей .NET, но определены для этого типа сущности в модель EF Core. Значение и состояние этих свойств сохраняется исключительно в объекте отслеживания изменений.

Теневые свойства полезны в тех случаях, когда в базе данных, которые не должны быть предоставлены для типов сущностей, сопоставленных данных. Чаще всего они используются для свойств внешнего ключа, связь между двумя сущностями, представленного значение внешнего ключа в базе данных, когда связь осуществляется в типах сущностей, с помощью свойств навигации между типами сущностей.

Значения свойств тени можно получить и изменить с помощью `ChangeTracker` API.

```
context.Entry(myBlog).Property("LastUpdated").CurrentValue = DateTime.Now;
```

Теневые свойства можно ссылаться в запросах LINQ через `EF.Property` статический метод.

```
var blogs = context.Blogs
    .OrderBy(b => EF.Property<DateTime>(b, "LastUpdated"));
```

Соглашения

Теневые свойства могут создаваться по соглашению, если обнаружены связи, но не свойство внешнего ключа не найден в классе зависимой сущности. В этом случае будут вводиться теневые свойства внешнего ключа. Теневые свойства внешнего ключа будет называться

`<navigation property name><principal key property name>` (Навигация по зависимой сущности, которое указывает основной сущности, используется для именования). Если имя участника ключевое свойство содержит имя свойства навигации, а затем имя будет просто `<principal key property name>`. Если ни одно свойство навигации на зависимой сущности, вместо него используется имени основного типа.

Например, следующий листинг кода приведет к `BlogId` свойство тени, представленной в `Post` сущности.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

Заметки к данным

Теневые свойства не могут создаваться с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки свойств тени. После вызова перегрузки строки `Property` можно объединять в цепочку любой конфигурации вызовов, что и для других свойств.

Если передано имя `Property` метода совпадает с именем существующего свойства (свойство тени или одно определенное в классе сущности), то код будет настраивать существующие свойства, а не обучаться новое свойство тени.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property<DateTime>("LastUpdated");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Отношения

28.08.2018 • 21 minutes to read • [Edit Online](#)

Отношение определяет, каким образом две сущности связаны друг с другом. В реляционной базе данных оно представлено ограничение внешнего ключа.

NOTE

Большинство примеров в этой статье для демонстрации понятия использовать отношение один ко многим. Примеры использования одному "и" многие ко многим "см. [другие шаблоны связь](#) в конце статьи.

Определение терминов

Существует несколько терминов, используемых для описания связей

- **Зависимой сущности:** это сущность, которая содержит свойства внешнего ключа. Иногда называют «дочерний» связи.
- **Основной сущности:** это сущность, которая содержит ключевые свойства основного или альтернативное. Иногда называют «parent» связи.
- **Внешний ключ:** свойства в зависимой сущности, который используется для хранения значений субъекта ключевое свойство, сущность связана с.
- **Основной ключ:** свойства, которое однозначно определяет основной сущности. Это может быть первичный ключ или резервный ключ.
- **Свойство навигации:** свойство, определенное для основного и/или зависимой сущности, содержащей ссылки на связанные объекты.
 - **Свойство навигации по коллекции:** свойство навигации, которое содержит ссылки на множество связанных сущностей.
 - **Свойство навигации по ссылке:** свойство навигации, которое содержит ссылку на одну связанную сущность.
 - **Свойство навигации обратный:** при обсуждении конкретным свойством навигации, этот термин относится к свойству навигации на другом конце связи.

В следующем примере кода показаны отношение "один ко многим" между `Blog` и `Post`

- `Post` является зависимой сущности
- `Blog` — Это основной сущности
- `Post.BlogId` внешний ключ
- `Blog.BlogId` является главным ключом (в данном случае это первичный ключ, а не альтернативного ключа)
- `Post.Blog` — Это свойство навигации ссылки
- `Blog.Posts` является свойством навигации коллекции
- `Post.Blog` свойство навигации обратный `Blog.Posts` (и наоборот)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Соглашения

По соглашению связь будет создан при наличии свойства навигации, обнаруженных на тип. Свойство считается свойство навигации, если тип, который указывает не могут сопоставляться как скалярный тип текущим поставщиком базы данных.

NOTE

Связи, которые обнаруживаются в соответствии с соглашением, всегда будут обрабатываться первичный ключ основной сущности. Для альтернативного ключа, дополнительная настройка выполняется с помощью Fluent API.

Полностью определенные связи

Самый распространенный шаблон для связи — имеют свойства навигации, определенные на обоих концах связи и свойство внешнего ключа, определенные в классе зависимой сущности.

- При обнаружении пары свойств навигации между двумя типами они будут настроены как свойства навигации обратный той же связи.
- Если зависимая сущность содержит свойство с именем <primary key property name>, <navigation property name><primary key property name>, или <principal entity name><primary key property name> то он будет настроен в качестве внешнего ключа.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

WARNING

Если есть несколько свойств навигации, определенных между двумя типами (то есть несколько уникальных пар переходов, которые указывают друг с другом), затем связи не будут созданы по соглашению, и необходимо будет вручную настроить их для идентификации как Пара свойств навигации вверх.

Нет свойство внешнего ключа

Рекомендуется иметь свойство внешнего ключа, определенные в классе зависимой сущности, однако он не требуется. При обнаружении не свойство внешнего ключа, свойство внешнего ключа тени познакомитесь с именем `<navigation property name><principal key property name>` (см. в разделе [замещения свойств](#) Дополнительные сведения).

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

Одного свойства навигации

Включая только одно свойство навигации (не обратное навигации и не свойство внешнего ключа) достаточно, чтобы иметь связи, определенной по соглашению. Также возможно одного свойства навигации и свойство внешнего ключа.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
```

Каскадное удаление

По соглашению каскадное удаление будет присвоено `Cascade` для необходимые связи и `ClientSetNull` необязательно связей. `CASCADE` означает, что зависимые сущности также удаляются. `ClientSetNull` означает, что зависимые сущности, которые не были загружены в память останется без изменений и должны быть вручную удалены или обновлены с допустимым основной сущности. Для сущностей, которые загружаются в память EF Core попытается установить для свойств внешнего ключа значение null.

См. в разделе [обязательных и необязательных связи](#) раздел для разницы между обязательных и

необязательных связей.

См. в разделе [каскадное удаление](#) для поведения и значения по умолчанию, используемый в соответствии с соглашением об удалении Дополнительные сведения о различных.

Заметки к данным

Существуют два заметок к данным, которые могут использоваться для настройки связей, `[ForeignKey]` и `[InverseProperty]`.

`[ForeignKey]`

Заметки данных можно использовать для настройки, какое свойство следует использовать как свойство внешнего ключа для конкретной связи. Обычно это делается свойства внешнего ключа не обнаружен по соглашению.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }

    [ForeignKey("BlogForeignKey")]
    public Blog Blog { get; set; }
}
```

TIP

`[ForeignKey]` Заметки можно поместить в свойство навигации, либо в связи. Он не должен перейти в свойстве навигации в классе зависимой сущности.

`[InverseProperty]`

Заметки данных можно использовать для настройки, как сопоставить свойства навигации в зависимой и основной сущности. Обычно это делается при наличии более чем одна пара свойств навигации между двумя типами сущностей.

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int AuthorUserId { get; set; }
    public User Author { get; set; }

    public int ContributorUserId { get; set; }
    public User Contributor { get; set; }
}

public class User
{
    public string UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [InverseProperty("Author")]
    public List<Post> AuthoredPosts { get; set; }

    [InverseProperty("Contributor")]
    public List<Post> ContributedToPosts { get; set; }
}
```

Текущий API

Чтобы настроить отношение в Fluent API, начните с определение свойства навигации, составляющих связи.

`HasOne` или `HasMany` определяет свойство навигации типа сущности, в начале конфигурации. Затем вы цепочку вызов `WithOne` или `WithMany` для идентификации обратной навигации. `HasOne` / `WithOne` используются для свойства навигации по ссылке и `HasMany` / `WithMany` используются для свойства навигации по коллекции.

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

```

Одного свойства навигации

Если имеется только одно свойство навигации, то существуют перегрузки без параметров `WithOne` и `WithMany`. Это означает, что имеется по сути, является ссылкой или коллекцией на другом конце связи, но нет свойства навигации, включенные в классе сущности.

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasMany(b => b.Posts)
            .WithOne();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}

```

Внешний ключ

Fluent API можно использовать для настройки того, какое свойство следует использовать как свойство внешнего ключа для конкретной связи.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogForeignKey);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
```

В следующем примере кода показаны способы настройки составного внешнего ключа.

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasKey(c => new { c.State, c.LicensePlate });

        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => new { s.CarState, s.CarLicensePlate });
    }
}

public class Car
{
    public string State { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public List<RecordOfSale> SaleHistory { get; set; }
}

public class RecordOfSale
{
    public int RecordOfSaleId { get; set; }
    public DateTime DateSold { get; set; }
    public decimal Price { get; set; }

    public string CarState { get; set; }
    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}

```

Можно использовать перегрузку строки `HasForeignKey(...)` для настройки теневого свойство как внешний ключ (см. в разделе [замещения свойств](#) Дополнительные сведения). Мы рекомендуем явным образом добавить в модель теневые свойства перед его использованием в качестве внешнего ключа (как показано ниже).

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Add the shadow property to the model
        modelBuilder.Entity<Post>()
            .Property<int>("BlogForeignKey");

        // Use the shadow property as a foreign key
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey("BlogForeignKey");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

```

Ключ субъекта-службы

Если требуется, чтобы внешний ключ для ссылки на свойство, отличный от первичного ключа, можно использовать Fluent API для настройки свойство основного ключа для связи. Свойство, настроенное в качестве основного ключа будет автоматически иметь установки в качестве альтернативного ключа (см. в разделе [альтернативные ключи](#) Дополнительные сведения).

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => s.CarLicensePlate)
            .HasPrincipalKey(c => c.LicensePlate);
    }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public List<RecordOfSale> SaleHistory { get; set; }
}

public class RecordOfSale
{
    public int RecordOfSaleId { get; set; }
    public DateTime DateSold { get; set; }
    public decimal Price { get; set; }

    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}
```

В следующем коде демонстрируется настройка составного ключа субъекта.

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => new { s.CarState, s.CarLicensePlate })
            .HasPrincipalKey(c => new { c.State, c.LicensePlate });
    }
}

public class Car
{
    public int CarId { get; set; }
    public string State { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public List<RecordOfSale> SaleHistory { get; set; }
}

public class RecordOfSale
{
    public int RecordOfSaleId { get; set; }
    public DateTime DateSold { get; set; }
    public decimal Price { get; set; }

    public string CarState { get; set; }
    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}

```

WARNING

Порядок, в котором указываются свойства основного ключа должен соответствовать порядку, в котором они указаны для внешнего ключа.

Обязательные и необязательные связей

Fluent API позволяет настроить, является ли связь обязательными или необязательными. В конечном счете определяет, является ли свойство внешнего ключа обязательными или необязательными. Это может пригодиться при использовании внешнего ключа состояния тени. Если у вас есть свойство внешнего ключа в классе сущностей, а затем requiredness связи определяется в зависимости от свойства внешнего ключа обязательными или нет (см. в разделе [обязательные и необязательные свойства](#) для получения дополнительных сведений о).

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .IsRequired();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

Каскадное удаление

Fluent API можно использовать для настройки поведения cascade delete для конкретной связи явно.

См. в разделе [каскадное удаление](#) на разделе Сохранение данных подробное описание каждого параметра.

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .OnDelete(DeleteBehavior.Cascade);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int? BlogId { get; set; }
    public Blog Blog { get; set; }
}

```

Другие шаблоны связи

Один к одному

Связи один к одному имеют свойства навигации ссылки на обеих сторонах. Они следуют тем же правилам как отношения один ко многим, но появилось в свойстве внешнего ключа, чтобы убедиться, что каждому участнику связан только один зависит от уникального индекса.

```

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}

```

NOTE

EF выберет одной сущности к зависимым его возможностям для обнаружения свойство внешнего ключа. Если выбрать неправильный сущности в качестве зависимого, Fluent API можно использовать для устранения этой проблемы.

Настраивая связь с Fluent API, используйте `HasOne` и `WithOne` методы.

При настройке внешнего ключа, необходимо указать тип зависимой сущности — Обратите внимание, что универсальный параметр, переданный `HasForeignKey` в списке ниже. В отношении один ко многим становится ясно, что зависит от сущности с навигации ссылки на и с коллекции является участником. Однако это не является таким образом, в взаимно-однозначной связи — таким образом необходимость явно определить его.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<BlogImage> BlogImages { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasOne(p => p.BlogImage)
            .WithOne(i => i.Blog)
            .HasForeignKey<BlogImage>(b => b.BlogForeignKey);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
```

Многие ко многим

Связи многие ко многим без класса сущности для представления таблицы соединения еще не поддерживаются. Тем не менее вы можете представить связь многие ко многим, включив класс сущности таблицы соединения и две отдельные связи один ко многим сопоставления.

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PostTag>()
            .HasKey(t => new { t.PostId, t.TagId });

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Post)
            .WithMany(p => p.PostTags)
            .HasForeignKey(pt => pt.PostId);

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Tag)
            .WithMany(t => t.PostTags)
            .HasForeignKey(pt => pt.TagId);
    }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class PostTag
{
    public int PostId { get; set; }
    public Post Post { get; set; }

    public string TagId { get; set; }
    public Tag Tag { get; set; }
}
```

Индексы

28.08.2018 • 2 minutes to read • [Edit Online](#)

Индексы являются понятием распространенных в многих хранилищах данных. Хотя их реализации в хранилище данных могут отличаться, они используются превратить уточняющие запросы, на основании столбца (или набор столбцов), более эффективным.

Соглашения

По соглашению индекс создается в каждом свойство (или набор свойств), которые используются в качестве внешнего ключа.

Заметки к данным

Индексы могут не создаваться с помощью заметок к данным.

Текущий API

Fluent API можно использовать для указания индекса по одному свойству. По умолчанию индексы не уникальны.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Можно также указать что индекса должны быть уникальными, это означает, что две сущности не может иметь же значения для заданного свойства.

```
modelBuilder.Entity<Blog>()
    .HasIndex(b => b.Url)
    .IsUnique();
```

Можно также указать индекс по нескольким столбцам.

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .HasIndex(p => new { p.FirstName, p.LastName });
    }
}

public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

TIP

Есть только один индекс отдельный набор свойств. Если вы используете Fluent API для настройки индекса на набор свойств, которые уже есть индекс, определенный, по соглашению или предыдущей конфигурации, затем подлежащий изменению определения этого индекса. Это полезно, если вы хотите продолжить настройку индекса, который был создан в соответствии с соглашением.

Альтернативные ключи

28.08.2018 • 3 minutes to read • [Edit Online](#)

Дополнительный ключ служит в качестве альтернативного уникального идентификатора для каждого экземпляра сущности, помимо первичного ключа. Альтернативные ключи можно использовать в качестве целевого объекта связи. При использовании реляционной базы данных сопоставляется концепцию индекс или ограничение уникальности на альтернативный столбца первичного ключа и один или несколько ограничений внешнего ключа, которые ссылаются на столбцы.

TIP

Если требуется обеспечить уникальность данных столбца, то требуется уникальный индекс, а не альтернативного ключа, см. в разделе [индексы](#). В EF альтернативные ключи обеспечивают больше возможностей, чем уникальные индексы, так как они могут использоваться в качестве целевого объекта внешнего ключа.

Альтернативные ключи обычно вводятся автоматически, когда требуется и необходимо настроить их вручную. См. в разделе [соглашения](#) для получения дополнительных сведений.

Соглашения

По соглашению альтернативного ключа вводится автоматически при идентификации свойством, которое не является первичным ключом, что целевым объектом отношения.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogUrl)
            .HasPrincipalKey(b => b.Url);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public string BlogUrl { get; set; }
    public Blog Blog { get; set; }
}
```

Заметки к данным

Не альтернативные ключи можно настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки единственное свойство являться и резервным ключом.

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasAlternateKey(c => c.LicensePlate);
    }
}

class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

Fluent API также можно настроить несколько свойств для являться и резервным ключом (известный как альтернативный составного ключа).

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasAlternateKey(c => new { c.State, c.LicensePlate });
    }
}

class Car
{
    public int CarId { get; set; }
    public string State { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

Наследование

28.08.2018 • 2 minutes to read • [Edit Online](#)

Наследование в модели EF используется для управления представление наследования в классы сущностей в базе данных.

Соглашения

По соглашению возлагается поставщик базы данных, чтобы определить, как наследование, будет представлено в базе данных. См. в разделе [наследование \(реляционная база данных\)](#) это обработки с помощью поставщика реляционной базы данных.

EF будет Настройка наследования только в том случае, если два или более наследуемых типов явно включены в модель. EF не будет выполнять сканирование для базовых или производных типов, в противном случае не включенные в модели. Можно включить типы в модели, предоставляя *DbSet* для каждого типа в иерархии наследования.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<RssBlog> RssBlogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

Если вы не хотите предоставлять *DbSet* для одной или нескольких сущностей в иерархии, можно использовать Fluent API для обеспечения того, они включаются в модель. Если вы не полагаться на соглашения можно указать базовый тип явно с помощью `.HasBaseType`.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RssBlog>().HasBaseType<Blog>();
    }
}
```

NOTE

Можно использовать `.HasBaseType((Type)null)` для удаления типа сущности в иерархии.

Заметки к данным

Заметки к данным нельзя использовать для настройки наследования.

Текущий API

Fluent API для наследования зависит от поставщика базы данных, которую вы используете. См. в разделе [наследование \(реляционная база данных\)](#) для конфигурации, можно выполнить для поставщика реляционной базы данных.

Резервные поля

29.08.2018 • 4 minutes to read • [Edit Online](#)

NOTE

Этот компонент впервые появился в EF Core 1.1.

Резервные поля Разрешить EF на чтение или запись в поле, а не свойство. Это может быть полезно, когда инкапсуляция в классе используется для ограничения использования и улучшения семантики вокруг доступа к данным в коде приложения, но значение следует чтения или записи в базу данных без использования этих ограничений / Улучшенные возможности.

Соглашения

По соглашению будут обнаружены следующие поля как резервные поля для заданного свойства (перечисленные в порядке приоритетности). Поля обнаруживаются только для свойств, которые включаются в модели. Дополнительные сведения, на котором включены свойства в модели, см. в разделе [Включение и исключение свойств](#).

- `_<camel-cased property name>`
- `_<property name>`
- `m_<camel-cased property name>`
- `m_<property name>`

```
public class Blog
{
    private string _url;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```

При настройке резервное поле EF записи непосредственно к этому полю при материализации экземпляров сущности из базы данных (а не с помощью метода задания свойства). Если EF должен считывать или записывать значения в других случаях, она будет по возможности использовать свойство. Например если EF необходимо обновить значение для свойства, он если он был определен будет использовать метод задания свойства. Если свойство доступно только для чтения, он запишет к полю.

Заметки к данным

Резервные поля нельзя настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки резервное поле для свойства.

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasField("_validatedUrl");
    }
}

public class Blog
{
    private string _validatedUrl;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _validatedUrl; }
    }

    public void SetUrl(string url)
    {
        using (var client = new HttpClient())
        {
            var response = client.GetAsync(url).Result;
            response.EnsureSuccessStatusCode();
        }

        _validatedUrl = url;
    }
}

```

Управление, если используется поле

Вы можете настроить, когда EF использует поле или свойство. См. в разделе [PropertyAccessMode перечисления](#) поддерживаемые параметры.

```

modelBuilder.Entity<Blog>()
    .Property(b => b.Url)
    .HasField("_validatedUrl")
    .UsePropertyAccessMode(PropertyAccessMode.Field);

```

Поля типа без свойства

Также можно создать свойство концептуальной модели, не имеет соответствующего свойства среди CLR в классе сущности, но вместо этого использует поле для хранения данных в сущности. Это отличается от [замещения свойств](#), где данные хранятся в объекте отслеживания изменений. Это обычно будет использоваться, если класс сущностей использует методы для получения и задания значений.

Можно предоставить EF имя поля в `Property(...)` API. Если отсутствует свойство с заданным именем, EF будет выглядеть для поля.

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(" _validatedUrl");
    }
}

public class Blog
{
    private string _validatedUrl;

    public int BlogId { get; set; }

    public string GetUrl()
    {
        return _validatedUrl;
    }

    public void SetUrl(string url)
    {
        using (var client = new HttpClient())
        {
            var response = client.GetAsync(url).Result;
            response.EnsureSuccessStatusCode();
        }

        _validatedUrl = url;
    }
}

```

Можно также дать свойству имя, отличное от имени поля. Это имя используется при создании модели, прежде всего он будет использоваться для имени столбца, который сопоставляется в базе данных.

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property<string>("Url")
        .HasField("_validatedUrl");
}

```

Если отсутствует свойство в классе сущности, можно использовать `EF.Property(...)` метода в запросе LINQ для ссылки на свойство, которое по существу является частью модели.

```
var blogs = db.blogs.OrderBy(b => EF.Property<string>(b, "Url"));
```

Преобразования значений

29.08.2018 • 7 minutes to read • [Edit Online](#)

NOTE

Это новая возможность в EF Core 2.1.

Преобразователи значений позволяет значениям свойств преобразования при чтении или записи в базу данных. Это преобразование может быть от одного значения к другому того же типа (например, шифрование строки) или из значения одного типа в значение другого типа (например, преобразование перечисления значения строки в базе данных и обратно.)

Основы

Преобразователи значений указываются в виде `ModelClrType` и `ProviderClrType`. Тип модели является типом .NET свойства в типе сущности. Тип поставщика является типом .NET, подразумевается поставщик базы данных. Например, чтобы сохранить перечислений в виде строк в базе данных, тип модели — тип перечисления и поставщик является `String`. Эти два типа могут совпадать.

Преобразования определяются с использованием двух `Func` деревьев выражений: один из `ModelClrType` для `ProviderClrType`, а другой из `ProviderClrType` для `ModelClrType`. Деревья выражений используются, чтобы их можно скомпилировать в код доступа базы данных для эффективного преобразования. Для выполнения сложных преобразований дерева выражения может быть простой вызов метода, который выполняет преобразование.

Настройка преобразователя значений

Преобразования значений определяются в свойствах в `OnModelCreating` из вашей `DbContext`. Например рассмотрим enum и сущности типа, определенного как:

```
public class Rider
{
    public int Id { get; set; }
    public EquineBeast Mount { get; set; }
}

public enum EquineBeast
{
    Donkey,
    Mule,
    Horse,
    Unicorn
}
```

Затем можно определить преобразования в `OnModelCreating` для хранения значений enum как строки (например, «Спецов», «Mule»,...) в базе данных:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(
            v => v.ToString(),
            v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));
}
```

NOTE

Объект `null` значение никогда не будут передаваться преобразователь значений. Это облегчает реализацию преобразования и позволяет им быть общим для свойства допускает значения NULL и не допускающие значения NULL.

Класс ValueConverter

Вызов `HasConversion` как показано выше, создаст `ValueConverter` экземпляра и задать его свойству.

`ValueConverter` Вместо этого можно явным образом создать. Пример:

```
var converter = new ValueConverter<EquineBeast, string>(
    v => v.ToString(),
    v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

Это может быть полезно, когда несколько свойств используйте то же самое преобразование.

NOTE

В настоящее время нет способа указать в одном месте, что каждое свойство данного типа должны использовать один и тот же преобразователь значения. Эта функция будет рассматриваться в будущих выпусках.

Встроенные конвертеры

EF Core поставляется с набором предварительно определенных `ValueConverter` классами, имеющимися в `Microsoft.EntityFrameworkCore.Storage.ValueConversion` пространства имен. Эти особые значения приведены ниже.

- `BoolToZeroOneConverter` -Bool в 0 и 1
- `BoolToStringConverter` -Bool для строк, таких как «Y» и «N»
- `BoolToTwoValuesConverter` -Bool любые два значения
- `BytesToStringConverter` -Массив байтов строку в кодировке Base64
- `CastingConverter` -Преобразования, которые требуется только приведение типа
- `CharToStringConverter` -Char в одну символьную строку
- `DateTimeOffsetToBinaryConverter` -DateTimeOffset в двоичной кодировке 64-разрядное значение
- `DateTimeOffsetToBytesConverter` -DateTimeOffset байтовый массив
- `DateTimeOffsetToStringConverter` -DateTimeOffset строку

- `DateTimeToBinaryConverter` -DateTime на 64-разрядное значение, включая DateTimeKind
- `DateTimeToStringConverter` -DateTime в строку
- `DateTimeToTicksConverter` -Дата и время в тактах
- `EnumToNumberConverter` -Enum основной номер
- `EnumToStringConverter` -Enum строку
- `GuidToBytesConverter` -Guid байтовый массив
- `GuidToStringConverter` -Guid строку
- `NumberToBytesConverter` — Любое числовое значение, массив байтов
- `NumberToStringConverter` — Любое числовое значение в строку
- `StringToBytesConverter` -Строка для UTF8 байты
- `TimeSpanToStringConverter` -TimeSpan в строку
- `TimeSpanToTicksConverter` -TimeSpan меткам

Обратите внимание, что `EnumToStringConverter` включается в этот список. Это означает, что нет необходимости для преобразования явно указать, как показано выше. Вместо этого используйте встроенный преобразователь:

```
var converter = new EnumToStringConverter<EquineBeast>();

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

Обратите внимание, что все встроенные преобразователи не имеют состояния, и поэтому один экземпляр можно безопасно использовать их совместно с несколькими свойствами.

Предопределенные преобразования

Для общих преобразований, для которых существует встроенный преобразователь нет необходимости явно указать преобразователь. Вместо этого достаточно настроить следует использовать тип поставщика и EF будет автоматически использовать соответствующий встроенный преобразователь. Перечисление для преобразования строк используются как в примере выше, но EF фактически сделает это автоматически при настройке тип поставщика:

```
modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion<string>();
```

То же самое достигается путем явного указания типа столбца. Например, если определен тип сущности, например так:

```
public class Rider
{
    public int Id { get; set; }

    [Column(TypeName = "nvarchar(24)")]
    public EquineBeast Mount { get; set; }
}
```

Затем значения перечисления будет сохранен как строки в базе данных без дополнительной настройки в `OnModelCreating`.

Ограничения

Существует несколько известных ограничений системы преобразования значения.

- Как отмечалось выше, `null` не может быть преобразован.
- В настоящее время нет способа распространения преобразование одного из свойств для нескольких столбцов или наоборот.
- Использование преобразования значений может повлиять на возможность преобразования выражений SQL EF Core. В таких случаях будет зарегистрировано предупреждение. Удаление этих ограничений считается в будущих выпусках.

Присвоение начальных значений данных

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Это новая возможность в EF Core 2.1.

Заполнение данными позволяет предоставить начальные данные для заполнения базы данных. В отличие от в EF6 в EF Core, присвоение начальных значений данных, связанный с типом сущности как часть конфигурации модели. Затем EF Core [миграций](#) может автоматически вычислить, что вставить, обновить или удалить операции необходимо применить при обновлении базы данных до новой версии модели.

Например, можно использовать это для настройки начальные данные для `Blog` в `OnModelCreating`:

```
modelBuilder.Entity<Blog>().HasData(new Blog {BlogId = 1, Url = "http://sample.com"});
```

Добавление сущностей, которые имеют отношение значения внешнего ключа необходимо указать. Часто свойств внешнего ключа находятся в состоянии тени, чтобы иметь возможность установить значения анонимный класс следует использовать:

```
modelBuilder.Entity<Post>().HasData(
    new {BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1"},
    new {BlogId = 1, PostId = 2, Title = "Second post", Content = "Test 2"});
```

После добавления сущности, рекомендуется использовать [миграций](#) для применения изменений.

Кроме того, можно использовать `context.Database.EnsureCreated()` для создания новой базы данных, содержащей начальные данные, например для тестовой базы данных или при использовании поставщика в памяти. Обратите внимание, что если база данных уже существует, `EnsureCreated()` не обновляет схему и начальные данные в базе данных.

Типы сущностей с помощью конструкторов

29.08.2018 • 10 minutes to read • [Edit Online](#)

NOTE

Это новая возможность в EF Core 2.1.

Начиная с EF Core 2.1, стало возможным определять конструктор с параметрами и посредством вызова этого конструктора, при создании экземпляра сущности EF Core. Параметры конструктора могут быть привязаны к сопоставленным свойствам или на различные виды услуг, которые помогают такое поведение, как отложенной загрузки.

NOTE

В версии EF Core 2.1 все привязки конструктора являются по соглашению. Конфигурация из определенных конструкторов для использования планируется в будущих выпусках.

Привязка к сопоставленным свойствам

Рассмотрим типичный модель сообщение в блоге:

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}
```

Когда EF Core создает экземпляры этих типов, например для результатов запроса, он сначала вызвать конструктор без параметров по умолчанию и затем в базе данных присвоено значение каждого свойства. Тем не менее если EF Core находит параметризованный конструктор с имена параметров и типы, которые совпадают с сопоставлены свойства, то он вместо этого будет вызывать параметризованный конструктор со значениями этих свойств и не будет устанавливать каждое свойство явным образом. Пример:

```

public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}

```

Некоторые другие примечания:

- Не все свойства должны иметь параметры конструктора. Например свойство Post.Content не задано каким-либо конструктором параметром, поэтому EF Core установит его после вызова конструктора обычным способом.
- Типы параметров и имена должны совпадать имена и типы свойств, за исключением того, что свойства могут быть стилем Pascal хотя параметры являются в стиле Camel.
- EF Core не удалось установить свойства навигации (например, блогов или сообщений выше) с помощью конструктора.
- Конструктор может быть открытым, закрытым, или иметь любой уровень доступа.

Свойства только для чтения

После установки свойств с помощью конструктора имеет смысл сделать некоторые из них только для чтения. EF Core поддерживает это, но есть несколько моментов, нужно обратить внимание:

- Свойства без методов задания не сопоставлены по соглашению. (Таким образом, как правило, для сопоставления свойства, которые не должны сопоставляться, например вычисляемых свойств.)
- С помощью автоматического формирования значений ключей требуется ключевое свойство, чтение и запись, так как значение ключа необходимо задать генератором ключа при вставке новых сущностей.

Для использования закрытых методов задания — простой способ избежать такие вещи. Пример:

```
public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; private set; }

    public string Name { get; private set; }
    public string Author { get; private set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; private set; }

    public string Title { get; private set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; private set; }

    public Blog Blog { get; set; }
}
```

EF Core воспринимает свойство с закрытым методом задания как чтения и записи, что означает, что все свойства сопоставляются как и раньше, и ключ можно по-прежнему создаваться хранилища.

Альтернативой использованию закрытых методов задания является сделать свойства действительно только для чтения и добавление более явное сопоставление в `OnModelCreating`. Аналогично некоторые свойства можно полностью удалить и заменить только поля. Например рассмотрим эти типы сущностей:

```

public class Blog
{
    private int _id;

    public Blog(string name, string author)
    {
        Name = name;
        Author = author;
    }

    public string Name { get; }
    public string Author { get; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    private int _id;

    public Post(string title, DateTime postedOn)
    {
        Title = title;
        PostedOn = postedOn;
    }

    public string Title { get; }
    public string Content { get; set; }
    public DateTime PostedOn { get; }

    public Blog Blog { get; set; }
}

```

И эта конфигурация в OnModelCreating:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Author);
            b.Property(e => e.Name);
        });

    modelBuilder.Entity<Post>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Title);
            b.Property(e => e.PostedOn);
        });
}

```

Сведения:

- Теперь поле является ключ «свойство». Это не `readonly` таким образом, чтобы сформированные хранилищем ключи могут использоваться.
- Другие свойства являются только для чтения свойства, заданные только в конструкторе.
- Если значение первичного ключа является только заданные EF или прочитать из базы данных, нет необходимости включать его в конструкторе. Это оставляет ключ «свойство» как простое поле и позволяет понять, что он не должен задаваться явным образом при создании нового блогов или

сообщений.

NOTE

Этот код приведет к компилятору выдает предупреждение "169", указывающее, что поле никогда не используется. Это сообщение можно проигнорировать, так как на самом деле EF Core использует поле extralinguistic образом.

Добавление служб

EF Core также можно внедрить «службы» в конструктор типа сущности. Например ниже можно ввести:

- `DbContext` — текущий экземпляр контекста, можно также ввести производный тип `DbContext`
- `ILazyLoader` — отложенная загрузка службы — см. в разделе [отложенной загрузки документации](#) подробности
- `Action<object, string>` — Делегат отложенной загрузки — см. в разделе [отложенной загрузки документации](#) подробности
- `IEntityType` — EF Core метаданные, связанные с этим типом сущности

NOTE

Начиная с EF Core 2.1 можно внедрить только те службы, которые известны EF Core. Поддержка добавления службы приложений будет рассмотрен в будущих выпусках.

Например можно использовать внедренного `DbContext` выборочный доступ к базе данных для получения сведений о связанных сущностях, не загружая их все. В следующем примере это используется для получения числа записей в блог без загрузки в записях:

```

public class Blog
{
    public Blog()
    {
    }

    private Blog(BloggingContext context)
    {
        Context = context;
    }

    private BloggingContext Context { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; set; }

    public int PostsCount
        => Posts?.Count
        ?? Context?.Set<Post>().Count(p => Id == EF.Property<int?>(p, "BlogId"))
        ?? 0;
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}

```

Несколько замечания об этом.

- Конструктор является закрытым, так как только вызывается с помощью EF Core, а другой открытый конструктор для общего использования.
- Код, с помощью внедренного службы (то есть контекст) имеет защищенный характер: на ее основе, `null` для обработки случаев, где EF Core не создает экземпляра.
- Так как службы хранится в свойстве чтения и записи будет изменено при присоединении сущности в новый экземпляр контекста.

WARNING

Внедрение `DbContext`, как это часто считается антишаблоном так, как он связывает типах сущностей непосредственно в EF Core. Внимательно рассмотрите все параметры, прежде чем с помощью внесения службы следующим образом.

Собственные типы сущностей

13.09.2018 • 9 minutes to read • [Edit Online](#)

NOTE

Этот компонент впервые появился в EF Core 2.0.

EF Core позволяет модели типы сущностей, которые могут появляться только когда-либо других типов сущностей свойства навигации. Они называются *собственные типы сущностей*. Сущность, содержащая принадлежащий Тип сущности является его владельцем.

Явной настройки

Объект, к которому типы никогда не включаются по EF Core в модели по соглашению, которым владеет. Можно использовать `OwnsOne` метод в `OnModelCreating` или указать для типа `OwnedAttribute` (новое в EF Core 2.1) для настройки типа как принадлежащего типа.

В этом примере `StreetAddress` — это тип с помощью нет свойство `identity`. Он используется как свойство сущности `Order` для указания адреса доставки конкретного заказа. В `OnModelCreating`, мы используем `OwnsOne` метод, чтобы указать, что свойство `ShippingAddress` принадлежит сущности типа `Order`.

```
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

// OnModelCreating
modelBuilder.Entity<Order>().OwnsOne(p => p.ShippingAddress);
```

Если свойство `ShippingAddress` является закрытым в типе `Order`, можно использовать строковую версию `OwnsOne` метод:

```
modelBuilder.Entity<Order>().OwnsOne(typeof(StreetAddress), "ShippingAddress");
```

В этом примере мы используем `OwnedAttribute` для достижения одной цели:

```
[Owned]
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}

public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

Неявные ключи

В EF Core 2.0 и 2.1 принадлежащие типы можно указать только свойства навигации по ссылке. Коллекциями принадлежащих типов не поддерживаются. Эти ссылки, которым владеет типы всегда иметь взаимно-однозначной связи с владельцем, поэтому им не нужны собственные значения ключа. В предыдущем примере тип `StreetAddress` не определяет ключевое свойство.

Чтобы понять, как EF Core отслеживает эти объекты, имеет смысл подумать, что первичный ключ создается как [затемнения свойства](#) для принадлежащего типа. Значение ключа экземпляра принадлежащего типа будет таким же, как значение ключа владельца экземпляра.

Сопоставление, принадлежащие типы с помощью разбиения таблицы

При использовании реляционных баз данных, по соглашению принадлежащие типы сопоставляются с той же таблице, как владелец. Для этого требуется разделение таблицы в двух: некоторые столбцы будут использоваться для хранения данных, владельца, а некоторые столбцы будут использоваться для хранения данных, принадлежащий сущности. Это общие возможности, известной как разделение таблицы.

TIP

Принадлежащие типы, хранимые с помощью разбиения таблицы можно использовать почти так же, как сложные типы используются в EF6.

По соглашению EF Core будет имен столбцов базы данных для свойств типа принадлежащих сущностей следующий шаблон `EntityProperty_OwnedEntityProperty`. Таким образом `StreetAddress` свойства отобразятся в таблице `Orders` с именами `ShippingAddress_Street` и `ShippingAddress_City`.

После добавления `HasColumnName` Метод переименовать эти столбцы. В случае, когда `StreetAddress` является открытым свойством сопоставление будет выполняться

```
modelBuilder.Entity<Order>().OwnsOne(
    o => o.ShippingAddress,
    sa =>
    {
        sa.Property(p=>p.Street).HasColumnName("ShipsToStreet");
        sa.Property(p=>p.City).HasColumnName("ShipsToCity");
    });
});
```

Совместное использование одного типа .NET среди нескольких принадлежащие типы

Принадлежащий Тип сущности может быть того же типа .NET как другим принадлежащим типом сущности, поэтому тип .NET может оказаться недостаточно для определения принадлежащего типа.

В таких случаях свойство, указывающее от владельца принадлежащих сущностей становится **задания перехода** типа принадлежащих сущностей. С точки зрения EF Core определяющую навигацию является частью идентификатора типа вместе с типом .NET.

Например в следующем классе ShippingAddress и BillingAddress имеют тот же тип .NET, StreetAddress:

```
public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
    public StreetAddress BillingAddress { get; set; }
}
```

Чтобы понять, как EF Core позволит отличить отслеживаемые экземпляры этих объектов, может быть полезно подумать, что определение навигации стало частью ключа экземпляра со значением ключа владельца и тип .NET, принадлежащего типа.

Вложенные принадлежащие типы

В этом примере OrderDetails владеет BillingAddress и ShippingAddress, который представляют собой типы StreetAddress. Затем OrderDetails принадлежит типу.

```
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
    public OrderStatus Status { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public enum OrderStatus
{
    Pending,
    Shipped
}

public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

Существует возможность цепочки `OwnsOne` метод текущего сопоставления для настройки этой модели:

```
modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, od =>
{
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
});
```

Это можно добиться той же самое с использованием `OwnedAttribute` OrderDetails и StreetAddress.

В дополнение к вложенные принадлежащие типы принадлежащий тип может ссылаться на обычной сущности. В следующем примере выбранной страной является обычной сущности, не принадлежащими:

```
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
    public Country Country { get; set; }
}
```

Эта возможность устанавливает собственные типы сущностей помимо сложных типов в EF6.

Хранение принадлежащие типы в отдельных таблицах

Также в отличие от сложных типов EF6, принадлежащие типы могут храниться в отдельной таблице от владельца. Чтобы переопределить соглашение, которое сопоставляется с той же таблице, как владелец принадлежащих типов, можно просто вызвать `ToTable` и укажите другое имя таблицы. Следующий пример будет сопоставлен OrderDetails и двумя адресами отдельную таблицу из заказа:

```
modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, od =>
{
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
    od.ToTable("OrderDetails");
});
```

Запрос принадлежащие типы

При запросе владельца принадлежащие типы включаются по умолчанию. Нет необходимости использовать `Include` метод, даже если принадлежащие типы хранятся в отдельной таблице. Исходя из модели, описанной выше, следующий запрос извлечет, заказа, OrderDetails и два собственных StreetAddresses все ожидающие заказы из базы данных:

```
var orders = context.Orders.Where(o => o.Status == OrderStatus.Pending);
```

Ограничения

Некоторые из этих ограничений лежат в основе как принадлежащий рабочих типов сущности, но некоторые другие являются ограничения, что мы можем удалить в последующих выпусках:

Недостатки в предыдущих версиях

- В EF Core 2.0 принадлежащие переходы на типы сущностей не может объявляться в производные типы сущностей, если собственные типы сущностей, явным образом сопоставляются отдельную таблицу из иерархии владельца. Это ограничение было снято в EF Core 2.1

Нынешних изъянов

- Иерархии наследования, содержащие принадлежащих типов сущностей, не поддерживаются
- Собственные типы сущностей не может быть, на который ссылается свойство навигации коллекции (только одну ссылку переходы в настоящее время поддерживаются)
- Переходы на собственных типов сущностей не может быть null, если они явным образом сопоставлены с отдельной таблицей от владельца
- Экземпляров принадлежащих типов сущностей, которые не могут совместно использоваться несколько владельцем (это хорошо известных сценариев для объектов значений, которые не может быть реализован

с помощью принадлежащих типов сущностей)

Ограничения нормальное поведение

- Не удается создать `DbSet<T>`
- Нельзя вызывать `Entity<T>()` с принадлежащего типа в `ModelBuilder`

Типы запросов

13.09.2018 • 6 minutes to read • [Edit Online](#)

NOTE

Эта функция находится в EF Core 2.1

В дополнение к типам сущностей, могут содержать модель EF Core *типы запросов*, который может использоваться для выполнения запросов к базе данных для данных, которые не сопоставлены с типами сущностей.

Сравнение типов запросов к типам сущностей

Типы запросов похожи на типы сущностей, в который они:

- Можно добавить в модель либо в `OnModelCreating` или с помощью свойства «набор» на производный `DbContext`.
- Поддержка множества тех же возможности сопоставления, как наследование сопоставления и свойств навигации. В реляционных хранилищах их можно настроить объекты целевой базы данных и столбцы с помощью текущего API методы или заметки к данным.

Тем не менее, они отличаются от сущностей тем, что в них типы:

- Ключ должен быть определен не требуется.
- Никогда не отслеживаются для изменения на `DbContext` и поэтому никогда не вставке, удалении или обновлении в базе данных.
- Никогда не обнаруживаются по соглашению.
- Поддерживают только подмножество возможностей навигации сопоставления — в частности:
 - Они никогда не может выступать в качестве основной конец связи.
 - Могут содержать только свойства навигации ссылки на сущности.
 - Сущности не может содержать свойства навигации к типам запросов.
- Посвящена `ModelBuilder` с помощью `Query` метод вместо `Entity` метод.
- Сопоставляются на `DbContext` через свойства типа `DbQuery<T>` вместо `DbSet<T>`
- Сопоставляются с объектами базы данных с помощью `ToView` метод, а не `ToTable`.
- Могут быть сопоставлены с запросом — определение запроса — это дополнительный запрос, объявленные в модели, используемый источник данных для типа запроса.

Сценарии использования

Ниже приведены некоторые из основных варианта применения типы запросов.

- Выступая в качестве возвращаемого типа для нерегламентированных `FromSql()` запросов.
- Сопоставление с представлениями базы данных.
- Сопоставление таблиц, у которых не определен первичный ключ.
- Сопоставление с запросами, определенными в модели.

Сопоставление объектов базы данных

Сопоставление типа запроса к объекту базы данных достигается с помощью `ToView` текущего API. С точки зрения EF Core, является объект базы данных, указанный в этом методе *представление*, это означает, что она обрабатывается как источник запроса только для чтения и не может быть целевым объектом update, insert или delete операций. Тем не менее это не означает, что объект базы данных требуется фактически быть представлениями базы данных - также может быть таблицей базы данных, которое будет интерпретироваться как доступный только для чтения. И наоборот, для типов сущностей, EF Core предполагает, что объект базы данных, указанные в `ToTable` метод может рассматриваться как *таблицы*, это означает, что он может использоваться как источник запроса, но также предназначено обновление, удаление и вставка операции. На самом деле, можно указать имя представления базы данных в `ToTable` и все должно работать нормально, пока представление настроено как обновляемые в базе данных.

Пример

В следующем примере показано, как использовать тип запроса для запроса представления базы данных.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Во-первых определим простую модель блога и Post.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
}
```

Далее мы определим представление простой базы данных, которые позволят нам запросить число сообщений, связанных с каждого блога:

```
db.Database.ExecuteSqlCommand(
    @"CREATE VIEW View_BlogPostCounts AS
        SELECT Name, Count(p.PostId) as PostCount from Blogs b
        JOIN Posts p on p.BlogId = b.BlogId
        GROUP BY b.Name");
```

Далее мы определим класс для хранения результата из представления базы данных:

```
public class BlogPostsCount
{
    public string BlogName { get; set; }
    public int PostCount { get; set; }
}
```

Далее мы настраиваем тип запроса в *OnModelCreating* с помощью `modelBuilder.Query<T>` API. Мы используем стандартную конфигурацию fluent API-интерфейсы для настройки сопоставления для типа

запроса:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Query<BlogPostsCount>().ToView("View_BlogPostCounts")
        .Property(v => v.BlogName).HasColumnName("Name");
}
```

Наконец можно запросить представления базы данных обычным образом:

```
var postCounts = db.BlogPostCounts.ToList();

foreach (var postCount in postCounts)
{
    Console.WriteLine($"{postCount.BlogName} has {postCount.PostCount} posts.");
    Console.WriteLine();
}
```

TIP

Обратите внимание, что мы определили свойство контекста запроса уровня (DbQuery) в качестве корневого для запросов этого типа.

Переключаясь между несколькими моделями с тем же типом DbContext

28.08.2018 • 2 minutes to read • [Edit Online](#)

Модель, построенная `OnModelCreating` использует свойство контекста для изменения построение модели. Например он может использоваться для исключения определенное свойство:

```
public class DynamicContext : DbContext
{
    public bool? IgnoreIntProperty { get; set; }

    public DbSet<ConfigurableEntity> Entities { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder
            .UseInMemoryDatabase("DynamicContext")
            .ReplaceService<IModelCacheKeyFactory, DynamicModelCacheKeyFactory>();

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        if (IgnoreIntProperty.HasValue)
        {
            if (IgnoreIntProperty.Value)
            {
                modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.IntProperty);
            }
            else
            {
                modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.StringProperty);
            }
        }
    }
}
```

IModelCacheKeyFactory

Однако если вы попробовал сделать выше без дополнительных изменений получится той же модели каждый раз при создании нового контекста для любого значения `IgnoreIntProperty`. Это связано с моделью EF использует для повышения производительности путем вызова только механизм кэширования `OnModelCreating` один раз и кэширования в модели.

По умолчанию EF считает, что для любого заданного контекста типа модели будут совпадать. Для выполнения этой задачи по умолчанию реализация `IModelCacheKeyFactory` Возвращает ключ, который просто содержит тип контекста. Изменить этот параметр необходимо заменить `IModelCacheKeyFactory` службы. Новая реализация должен возвращать объект, который можно сравнивать для других ключей модели, с помощью `Equals` метод, который учитывает все переменные, которые влияют на модель:

```
public class DynamicModelCacheKeyFactory : IModelCacheKeyFactory
{
    public object Create(DbContext context)
    {
        if (context is DynamicContext dynamicContext)
        {
            return (context.GetType(), dynamicContext.IgnoreIntProperty);
        }
        return context.GetType();
    }
}
```

Моделирование реляционных баз данных

28.08.2018 • 2 minutes to read • [Edit Online](#)

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Сопоставление таблиц

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Сопоставление таблиц определяет, какие данные таблицы следует оттуда и сохранены в базе данных.

Соглашения

По соглашению каждая сущность будет настроен для сопоставления с таблицей с тем же именем, что `DbSet< TEntity >` свойство, которое предоставляет сущность в производном контексте. Если нет `DbSet< TEntity >` включается для заданной сущности, используется имя класса.

Заметки к данным

Заметки к данным можно использовать для настройки в таблице, которая сопоставляется с типом.

```
using System.ComponentModel.DataAnnotations.Schema;
```

```
[Table("blogs")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Можно также указать схему, которой принадлежит таблица.

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки в таблице, которая сопоставляется с типом.

```
using Microsoft.EntityFrameworkCore;
```

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .ToTable("blogs");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Можно также указать схему, которой принадлежит таблица.

```
modelBuilder.Entity<Blog>()
    .ToTable("blogs", schema: "blogging");
```

Сопоставление столбцов

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Сопоставление столбцов определяет, какие данные столбца следует оттуда и сохранены в базе данных.

Соглашения

По соглашению каждое свойство будет настраиваться для сопоставления со столбцом с тем же именем, как свойство.

Заметки к данным

Заметки к данным можно использовать для настройки столбца, с которым сопоставлено свойство.

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Текущий API

Fluent API можно использовать для настройки столбца, с которым сопоставлено свойство.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.BlogId)
            .HasColumnName("blog_id");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Типы данных

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Тип данных относится к конкретному типу базы данных столбца, с которым сопоставлено свойство.

Соглашения

По соглашению поставщик базы данных выбирает тип данных, на основе типа CLR свойства. Он также принимает во внимание другие метаданные, например настроенного [Максимальная длина](#), является ли свойство частью первичного ключа и т. д.

Например, SQL Server использует `datetime2(7)` для `DateTime` свойства, и `nvarchar(max)` для `string` свойства (или `nvarchar(450)` для `string` свойствах, которые используются в качестве ключа).

Заметки к данным

Заметки к данным можно использовать для указания свой тип данных для столбца.

Например, приведенный ниже код настраивает `Url` как строка не в Юникоде с максимальной длиной `200` и `Rating` как тип `decimal` с точностью `5` и масштабирование объекта `2`.

```
public class Blog
{
    public int BlogId { get; set; }
    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }
    [Column(TypeName = "decimal(5, 2)")]
    public decimal Rating { get; set; }
}
```

Текущий API

Fluent API также можно указать разные типы данных для столбцов.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>(eb =>
        {
            eb.Property(b => b.Url).HasColumnType("varchar(200)");
            eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");
        });
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public decimal Rating { get; set; }
}
```

Первичные ключи

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Ограничение первичного ключа введен для ключа каждого типа сущности.

Соглашения

По соглашению, первичный ключ в базе данных будет называться `PK_<type name>`.

Заметки к данным

Нет конкретных аспектов реляционной базы данных первичного ключа можно настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки имени ограничения первичного ключа в базе данных.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasKey(b => b.BlogId)
            .HasName("PrimaryKey_BlogId");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Схема по умолчанию

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Схема по умолчанию — схему базы данных, объекты будут созданы в, если схема не настроена явным образом для этого объекта.

Соглашения

По соглашению поставщик базы данных будет выбирать наиболее подходящий схемы по умолчанию. Например, Microsoft SQL Server будет использовать `dbo` схемы и SQLite не будет использовать схему (поскольку схемы не поддерживаются в SQLite).

Заметки к данным

Не удается установить схему по умолчанию, с помощью заметок к данным.

Текущий API

Fluent API можно использовать для указания схемы по умолчанию.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultSchema("blogging");
    }
}
```

Вычисляемые столбцы

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Вычисляемый столбец является столбцом, значение которого вычисляется в базе данных. Вычисляемый столбец могут использоваться другие столбцы в таблице, для которого требуется вычислить его значение.

Соглашения

По соглашению вычисляемые столбцы не создаются в модели.

Заметки к данным

Вычисляемые столбцы можно настроить не с заметками к данным.

Текущий API

Fluent API можно использовать для указания, что свойство должно сопоставляться вычисляемый столбец.

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .Property(p => p.DisplayName)
            .HasComputedColumnSql("[LastName] + ' ' + [FirstName]");
    }
}

public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string DisplayName { get; set; }
}
```

Последовательности

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Последовательность создает последовательных числовых значений в базе данных. Последовательности не связаны с определенной таблицей.

Соглашения

По соглашению последовательностей не добавляются в модель.

Заметки к данным

Вы можете настроить не последовательности с использованием заметок к данным.

Текущий API

Fluent API можно использовать для создания последовательности в модели.

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers");
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
    public string Url { get; set; }
}
```

Также можно настроить дополнительные аспекты последовательности, например ее схему, начальное значение и приращение.

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
            .StartsAt(1000)
            .IncrementsBy(5);
    }
}
```

Как только вводится последовательности, его можно использовать для формирования значений для свойств в модели. Например, можно использовать [значения по умолчанию](#) для вставки следующего значения из последовательности.

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
            .StartsAt(1000)
            .IncrementsBy(5);

        modelBuilder.Entity<Order>()
            .Property(o => o.OrderNo)
            .HasDefaultValueSql("NEXT VALUE FOR shared.OrderNumbers");
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
    public string Url { get; set; }
}
```

Значения по умолчанию

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего пакета *Microsoft.EntityFrameworkCore.Relational*).

Значение по умолчанию столбца является значением, который будет вставлен при вставке новой строки, но значение не указано для столбца.

Соглашения

По соглашению значение по умолчанию не настроен.

Заметки к данным

Значение по умолчанию, с помощью заметок к данным нельзя будет сделать.

Текущий API

Fluent API можно использовать для указания значения по умолчанию для свойства.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Rating)
            .HasDefaultValue(3);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
}
```

Можно также указать фрагмент SQL, который используется для вычисления значения по умолчанию.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Created)
            .HasDefaultValueSql("getdate()");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public DateTime Created { get; set; }
}
```

Индексы

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Совпадает с концепцией индекса в ядро Entity Framework сопоставляет индекса в реляционной базе данных.

Соглашения

По соглашению, называются индексы `IX_<type name>_<property name>`. Для индексов являются составные индексы `<property name>` становится список имен свойств, разделенных символом подчеркивания.

Заметки к данным

Не индексов можно настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки имени индекса.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .WithName("Index_Url");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Можно также указать фильтр.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .HasFilter("[Url] IS NOT NULL");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

При использовании поставщика SQL Server EF добавляет фильтрации «IS NOT NULL» для всех столбцов допускает значения NULL, которые являются частью уникального индекса. Чтобы переопределить это соглашение, можно указать `null` значение.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .IsUnique()
            .HasFilter(null);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Ограничения внешнего ключа

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Ограничение внешнего ключа введен для каждой связи в модели.

Соглашения

По соглашению с именем ограничения внешнего ключа

`FK_<dependent type name>_<principal type name>_<foreign key property name>`. Для составных ключей внешнего `<foreign key property name>` становится подчеркивания запятыми список имен свойств внешнего ключа.

Заметки к данным

Имена ограничение внешнего ключа нельзя настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки имени ограничения внешнего ключа для связи.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogId)
            .HasConstraintName("ForeignKey_Post_Blog");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Альтернативные ключи (ограничения уникальности)

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего пакета *Microsoft.EntityFrameworkCore.Relational*).

Для каждого альтернативного ключа в модели введено ограничение уникальности.

Соглашения

По соглашению, индекс и ограничения, введенные для альтернативного ключа будет называться `AK_<type name>_<property name>`. Для составных ключей альтернативный `<property name>` становится списком имен свойств, разделенных символом подчеркивания.

Заметки к данным

Не ограничения UNIQUE можно настроить с помощью заметок к данным.

Текущий API

Fluent API можно использовать для настройки имени индексов и ограничений для альтернативного ключа.

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Car>()
            .HasAlternateKey(c => c.LicensePlate)
            .HasName("AlternateKey_LicensePlate");
    }
}

class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

Наследование (реляционная база данных)

28.08.2018 • 3 minutes to read • [Edit Online](#)

NOTE

Описанная в этом разделе конфигурации применяется к реляционным базам данных в общем случае. Показанные здесь методы расширения будут доступны после установки поставщика реляционной базы данных (посредством общего *пакета Microsoft.EntityFrameworkCore.Relational*).

Наследование в модели EF используется для управления представление наследования в классы сущностей в базе данных.

NOTE

В настоящее время только шаблон таблица на иерархию (ТРН) реализована в EF Core. Другие распространенные шаблоны, такие как одна таблица на тип (ТРТ) и таблица на конкретный тип (ТРС) еще не доступны.

Соглашения

По соглашению наследования будут сопоставлены с помощью шаблона одна таблица на иерархию (ТРН). ТРН использует одну таблицу для хранения данных для всех типов в иерархии. Столбец дискриминатора используется для определения подходящего, каждая строка представляет.

Если два или более наследуемых типов явно включены в модель EF Core только процедура позволяет настроить наследование (см. в разделе [наследования](#) подробности).

Ниже приведен пример, показывающий простые сценарии и данные, хранящиеся в таблице реляционной базы данных, используя шаблон ТРН. *Дискриминатора* столбца определяет, какой тип блог хранится в каждой строке.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<RssBlog> RssBlogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

Results				
	BlogId	Discriminator	Url	RssUrl
1	1	Blog	http://blogs.msdn.com/dotnet	NULL
2	2	RssBlog	http://blogs.msdn.com/adonet	http://blogs.msdn.com/b/adonet/atom.aspx

Заметки к данным

Заметки к данным нельзя использовать для настройки наследования.

Текущий API

Чтобы настроить имя и тип столбца дискриминатора и значений, которые используются для идентификации каждого типа в иерархии можно использовать Fluent API.

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasDiscriminator<string>("blog_type")
            .HasValue<Blog>("blog_base")
            .HasValue<RssBlog>("blog_rss");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

Настройка свойства дискриминатора

В приведенных выше примерах дискриминатора создается как [затемнения свойства](#) на базовая сущность иерархии. Так как это свойство в модели, его можно настроить так же, как и другие свойства. Например, чтобы задать Максимальная длина, когда используется по умолчанию, по соглашению дискриминатора:

```
modelBuilder.Entity<Blog>()
    .Property("Discriminator")
    .HasMaxLength(200);
```

Дискриминатора может быть сопоставлено Фактическое свойство CLR в сущности. Пример:

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasDiscriminator<string>("BlogType");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public string BlogType { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

Объединяя эти две вещи можно сопоставить настоящее Свойство дискриминатора и настройте его:

```
modelBuilder.Entity<Blog>(b =>
{
    b.HasDiscriminator<string>("BlogType");

    b.Property(e => e.BlogType)
        .HasMaxLength(200)
        .HasColumnName("blog_type");
});
```

Управление схемами баз данных

28.08.2018 • 2 minutes to read • [Edit Online](#)

EF Core предоставляет два основных способа синхронизации схемы базы данных и модели EF Core. Чтобы выбрать один из них, определите, что является истинным — ваша схема базы данных или модель EF Core.

Если истинной должна быть модель EF Core, используйте [миграции](#). При внесении изменений в модель EF Core этот подход постепенно применяет соответствующие изменения схемы к базе данных, чтобы она оставалась совместимой с вашей моделью EF Core.

Используйте [реконструирование](#), если хотите сделать истинной схему базы данных. Этот подход позволяет формировать DbContext и классы типов сущностей, реконструируя схему базы данных в модель EF Core.

NOTE

API [создания и удаления](#) также позволяют создать схему базы данных из модели EF Core. Но они предназначены главным образом для тестирования, создания прототипов и других сценариев, где допустимо удаление базы данных.

Миграции

12.10.2018 • 9 minutes to read • [Edit Online](#)

Модель данных в процессе разработки может измениться и перестанет соответствовать базе данных. Вы всегда можете удалить базу данных, и EF создаст для вас новую версию, в точности соответствующую модели, но такая процедура приводит к потере текущих данных. Функция миграции в EF Core позволяет последовательно применять изменения схемы к базе данных, чтобы синхронизировать ее с моделью данных в приложении без потери существующих данных.

Миграции включают средства командной строки и API-интерфейсы, которые помогают в решении следующих задач:

- [Создание миграции](#). Создайте код, который обновляет базу данных для синхронизации с набором изменений модели данных.
- [Обновление базы данных](#). Примените ожидающие миграции, чтобы обновить схему базы данных.
- [Настройка кода миграции](#). Иногда созданный код приходится изменять или дополнять.
- [Удаление миграции](#). Удалите созданный код.
- [Отмена миграции](#). Отмените примененные к базе данных изменения.
- [Создание скриптов SQL](#). Иногда бывает нужно создать скрипт для обновления рабочей базы данных или для устранения неполадок в коде миграции.
- [Применение миграции во время выполнения](#). Если обновления во время разработки и выполнение скриптов в вашем сценарии работы подходят плохо, вызовите метод `Migrate()`.

Установка инструментов

Установите [средства командной строки](#).

- Для Visual Studio мы рекомендуем [Инструменты консоли диспетчера пакетов](#).
- Для других сред разработки выберите [Средства интерфейса командной строки .NET Core](#).

Создание миграции

После того, как вы [определите начальную модель](#), можно переходить к созданию базы данных. Чтобы добавить первоначальную миграцию, выполните следующую команду.

```
Add-Migration InitialCreate
```

```
dotnet ef migrations add InitialCreate
```

В каталог **Migrations** проекта добавляются три файла.

- **0000000000000000_InitialCreate.cs** — главный файл миграций. Содержит операции, необходимые для применения миграции (в `Up()`) и ее отмены (в `Down()`).
- **0000000000000000_InitialCreate.Designer.cs** — файл метаданных миграций. Содержит сведения, используемые EF.
- **MyContextModelSnapshot.cs** — моментальный снимок текущей модели. Используется для определения появившихся изменений при добавлении следующей миграции.

Метка времени в именах файлов позволяет расположить их в хронологическом порядке, чтобы вы могли

отслеживать ход изменений.

TIP

Вы можете свободно перемещать файлы миграций и изменять их пространство имен. Новые миграции создаются в виде элементов того же уровня, что и последняя миграция.

Обновление базы данных

После этого примените миграцию к базе данных, чтобы создать схему.

```
Update-Database
```

```
dotnet ef database update
```

Настройка кода миграции

После внесения изменений в модель EF Core может нарушиться синхронизация схемы базы данных. Чтобы сделать ее актуальной, добавьте еще одну миграцию. Имя миграции можно использовать как сообщение фиксации в системе управления версиями. Например, вы можете выбрать имя *AddProductReviews*, если суть изменений заключается в создании нового класса сущностей для обзоров.

```
Add-Migration AddProductReviews
```

```
dotnet ef migrations add AddProductReviews
```

После создания кода миграции проверьте, правильно ли он создан. Если потребуется, добавьте, удалите или измените любые операции для правильного применения изменений.

Например, миграция может содержать следующие операции.

```
migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");

migrationBuilder.AddColumn<string>(
    name: "Name",
    table: "Customer",
    nullable: true);
```

Хотя эти операции обеспечивают совместимость схемы базы данных, они не сохраняют существующие имена клиентов. Чтобы улучшить миграцию, перепишите ее следующим образом.

```
migrationBuilder.AddColumn<string>(
    name: "Name",
    table: "Customer",
    nullable: true);

migrationBuilder.Sql(
@"
    UPDATE Customer
    SET Name = FirstName + ' ' + LastName;
");

migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");
```

TIP

Процесс создания новой миграции предупреждает, если операция может привести к потере данных (например, содержит удаление столбца). Увидев это предупреждение, проверьте точность кода миграции с особой тщательностью.

Примените миграцию к базе данных с помощью соответствующей команды.

```
Update-Database
```

```
dotnet ef database update
```

Пустые миграции

Иногда бывает удобно добавить миграцию, не внося изменения в модель. В этом случае при добавлении новой миграции создаются файлы кода с пустыми классами. Вы можете настроить ее для выполнения операций, которые не связаны напрямую с моделью EF Core. Ниже приведен ряд вещей, которыми вам может потребоваться управлять.

- Полнотекстовый поиск
- Функции
- Хранимые процедуры
- Триггеры
- Представления

Удаление миграции

Иногда, добавив миграцию, вы понимаете, что нужно внести дополнительные изменения в модель EF Core перед ее применением. Для удаления последней миграции используйте приведенную ниже команду.

```
Remove-Migration
```

```
dotnet ef migrations remove
```

Удалив миграцию, вы можете внести в модель дополнительные изменения и снова добавить ее.

Отмена миграции

Если вы уже применили одну миграцию для базы данных или несколько и хотите отменить их, можно использовать ту же команду, что и для применения миграций, но указав имя миграции, к которой надо откатить изменения.

```
Update-Database LastGoodMigration
```

```
dotnet ef database update LastGoodMigration
```

Создание скриптов SQL

При отладке миграций или их развертывании в рабочей базе данных бывает полезно создать скрипт SQL. Такой скрипт можно дополнительно проверить на точность и настроить в соответствии с потребностями рабочей базы данных. Кроме того, его можно использовать в сочетании с технологией развертывания. Базовая команда имеет следующий вид.

```
Script-Migration
```

```
dotnet ef migrations script
```

Для этой команды доступно несколько параметров.

Миграция **from** должна быть последней миграцией, применяемой к базе данных перед выполнением скрипта. Если не было применено ни одной миграции, укажите `0` (это значение по умолчанию).

Миграция **to** является последней миграцией, применяемой к базе данных после выполнения скрипта. По умолчанию она является последней миграцией в проекте.

При необходимости можно создать **идемпотентный** скрипт. Он применяет миграции только в том случае, если они еще не были применены к базе данных. Это удобно, если точно неизвестно, какая последняя миграция была применена к базе данных, или вы развертываете несколько баз данных, каждая из которых может иметь отдельную миграцию.

Применение миграции во время выполнения

Некоторым приложениям может потребоваться применить миграции во время выполнения — при запуске или первом выполнении. Для этого можно использовать метод `Migrate()`.

Этот метод основан на службе `IMigrator`, которую можно применять в более сложных сценариях. Для доступа к нему используйте `DbContext.GetService<IMigrator>()`.

```
myDbContext.Database.Migrate();
```

WARNING

- Такой подход не является универсальным. Хотя он отлично подходит для приложений с локальной базой данных, большинству приложений требуется более надежная стратегия развертывания, такая как создание скриптов SQL.
- Не вызывайте `EnsureCreated()` перед `Migrate()`. `EnsureCreated()` обходит миграции, чтобы создать схему, что приводит к сбою `Migrate()`.

Следующие шаги

Дополнительные сведения см. в разделе [Справочник по инструментам Entity Framework Core — EF Core](#).

Миграция в командных средах

28.08.2018 • 2 minutes to read • [Edit Online](#)

При работе с Миграциями в командных средах, особое внимание с файлом моментального снимка модели. Этот файл может сообщить, если ваш коллега миграции четко объединяет с вашими или если вам требуется устраниТЬ конфликт, повторно создав миграции прежде чем использовать их.

СЛИЯНИЕ

При миграции слияние от коллег, в файле моментального снимка модели может появиться конфликтов. Если оба изменения не связаны, слияние является тривиальным, и две операции миграции могут сосуществовать. Например можно получить конфликта слияния в конфигурации типа сущности `customer`, выглядит следующим образом:

```
<<<<< Mine
b.Property<bool>("Deactivated");
=====
b.Property<int>("LoyaltyPoints");
>>>>> Theirs
```

Так как оба эти свойства должны существовать в конечной модели, Завершение слияния, добавив оба свойства. Во многих случаях системы управления версиями может автоматически объединить такие изменения автоматически.

```
b.Property<bool>("Deactivated");
b.Property<int>("LoyaltyPoints");
```

В этих случаях миграции и миграции ваш коллега независимы друг от друга. Так как сначала любого из них может применяться, не нужно вносить дополнительные изменения миграции перед предоставлением общего доступа со своей командой.

Разрешение конфликтов

Иногда возникают true конфликтов при слиянии модель моментального снимка. Например вы и ваш партнер по группе каждого было изменено и то же свойство.

```
<<<<< Mine
b.Property<string>("Username");
=====
b.Property<string>("Alias");
>>>>> Theirs
```

Если вы столкнулись с такого рода конфликт, устраниТЬ ее повторное создание миграции. Выполните следующие действия.

1. Отменить слияние "и" rollback в рабочий каталог до слияния
2. Удалить переход (но сохранить изменения модели)
3. Слияние изменений ваш коллега в рабочий каталог
4. Повторно добавить переход

После этого, две операции миграции могут применяться в правильном порядке. Миграцию применяется в первую очередь, переименование столбца *псевдоним*, после миграции переименовывает его, чтобы *Username*.

Миграцию можно безопасно использовать с остальными участниками команды.

Миграция пользовательских операций

27.09.2018 • 3 minutes to read • [Edit Online](#)

MigrationBuilder API дает возможность выполнять различные виды операций во время миграции, но это далеко не исчерпывающий. Тем не менее API также является расширяемой, что позволяет определять собственные операции. Существует два способа для расширения API: с помощью `Sql()` метод, или определения пользовательских `MigrationOperation` объектов.

Чтобы продемонстрировать, давайте взглянем на реализации операции, которая создает пользователя базы данных с помощью каждого подхода. В нашей миграции нам нужно включить написание следующего кода:

```
migrationBuilder.CreateUser("SQLUser1", "Password");
```

С помощью `MigrationBuilder.Sql()`

Самый простой способ реализовать пользовательскую операцию, — определить метод расширения, который вызывает `MigrationBuilder.Sql()`. Ниже приведен пример, который создает соответствующие Transact-SQL.

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
=> migrationBuilder.Sql($"CREATE USER {name} WITH PASSWORD '{password}';");
```

Если миграции необходима поддержка нескольких поставщиков базы данных, можно использовать `MigrationBuilder.ActiveProvider` свойство. Ниже приведен пример, которые поддерживают Microsoft SQL Server и PostgreSQL.

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
{
    switch (migrationBuilder.ActiveProvider)
    {
        case "Npgsql.EntityFrameworkCore.PostgreSQL":
            return migrationBuilder
                .Sql($"CREATE USER {name} WITH PASSWORD '{password}';");

        case "Microsoft.EntityFrameworkCore.SqlServer":
            return migrationBuilder
                .Sql($"CREATE USER {name} WITH PASSWORD = '{password}';");
    }

    return migrationBuilder;
}
```

Этот подход работает, только если известно, чтобы каждый поставщик применения вашей пользовательской операции.

С помощью `MigrationOperation`

Для отделения пользовательской операции из SQL, можно определить собственные `MigrationOperation` для его отображения. Операция затем передается поставщику, чтобы можно было определить подходящий код SQL для создания.

```
class CreateUserOperation : MigrationOperation
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

При таком подходе метод расширения просто нужно добавить одну из этих операций для `MigrationBuilder.Operations`.

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
{
    migrationBuilder.Operations.Add(
        new CreateUserOperation
        {
            Name = name,
            Password = password
        });

    return migrationBuilder;
}
```

Этот подход требует каждому поставщику, чтобы знать, как создавать код SQL для этой операции в их `IMigrationsSqlGenerator` службы. Ниже приведен пример переопределения генератора SQL Server для обработки новой операции.

```

class MyMigrationsSqlGenerator : SqlServerMigrationsSqlGenerator
{
    public MyMigrationsSqlGenerator(
        MigrationsSqlGeneratorDependencies dependencies,
        IMigrationsAnnotationProvider migrationsAnnotations)
        : base(dependencies, migrationsAnnotations)
    {
    }

    protected override void Generate(
        MigrationOperation operation,
        IModel model,
        MigrationCommandListBuilder builder)
    {
        if (operation is CreateUserOperation createUserOperation)
        {
            Generate(createUserOperation, builder);
        }
        else
        {
            base.Generate(operation, model, builder);
        }
    }

    private void Generate(
        CreateUserOperation operation,
        MigrationCommandListBuilder builder)
    {
        var sqlHelper = Dependencies.SqlGenerationHelper;
        var stringMapping = Dependencies.TypeMappingSource.FindMapping(typeof(string));

        builder
            .Append("CREATE USER ")
            .Append(sqlHelper.DelimitIdentifier(operation.Name))
            .Append(" WITH PASSWORD = ")
            .Append(stringMapping.GenerateSqlLiteral(operation.Password))
            .AppendLine(sqlHelper.StatementTerminator)
            .EndCommand();
    }
}

```

Замените обновленный службы генератор миграций sql по умолчанию.

```

protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IMigrationsSqlGenerator, MyMigrationsSqlGenerator>();

```

Использование отдельного проекта

29.09.2018 • 2 minutes to read • [Edit Online](#)

Вам может потребоваться сохранить миграции в сборку, содержащую один вашей `DbContext`. Также можно использовать эту стратегию, чтобы поддерживать несколько миграций, например, один для разработки, а другой для обновления до выпуска к выпуску.

Действие

1. Создайте новую библиотеку классов.
2. Добавьте ссылку на сборку `DbContext`.
3. Переместите миграции и файлы моментальных снимков модели к библиотеке классов.

TIP

Если у вас нет имеющихся миграций, создать его в проект, содержащий `DbContext`, а затем переместите ее. Это важно, так как миграция не содержит существующие миграции, команда `Add-Migration` будет не удастся найти `DbContext`.

4. Настройка сборки миграции:

```
options.UseSqlServer(  
    connectionString,  
    x => x.MigrationsAssembly("MyApp.Migrations"));
```

5. Добавьте ссылку на сборку миграции из запуска сборки.

- Если это приводит к циклической зависимости, обновите путь к выходной библиотеке классов:

```
<PropertyGroup>  
    <OutputPath>..\\MyStartupProject\\bin\\$(Configuration)\\</OutputPath>  
</PropertyGroup>
```

Если все делается правильно, можно добавить в проект новые миграции.

```
Add-Migration NewMigration -Project MyApp.Migrations
```

```
dotnet ef migrations add NewMigration --project MyApp.Migrations
```

Миграция с несколькими поставщиками

13.09.2018 • 2 minutes to read • [Edit Online](#)

[Инструменты EF Core] 1 только сформировать шаблон миграции для поставщика active Directory. Иногда тем не менее, можно использовать более одного поставщика (например, Microsoft SQL Server и SQLite) с DbContext. Существует два способа решения этой с Миграциями. Вы можете поддерживать две миграции — один для каждого поставщика--или объединения их в один набор, можно работать с обеими.

Два набора миграции

Первый способ вы создаете две операции миграции для каждого изменения модели.

Один из способов сделать это для размещения каждого набора миграции [в отдельной сборке] 2 и переключения между добавлением две операции миграции поставщика активной (и миграции сборки).

Другой подход, который упрощает работу со средствами — создать новый тип, который является производным от DbContext и переопределяет активный поставщик. Этот тип используется на этапе разработки времени при добавлении или применение миграций.

```
class MysqliteDbContext : MyDbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=my.db");
}
```

NOTE

Так как каждый набор миграции использует собственные типы DbContext, этот подход не требует использования отдельных миграций сборки.

При добавлении новой миграции, укажите типы контекстов.

```
Add-Migration InitialCreate -Context MyDbContext -OutputDir Migrations\SqlServerMigrations
Add-Migration InitialCreate -Context MysqliteDbContext -OutputDir Migrations\SqliteMigrations
```

```
dotnet ef migrations add InitialCreate --context MyDbContext --output-dir Migrations/SqlServerMigrations
dotnet ef migrations add InitialCreate --context MysqliteDbContext --output-dir Migrations/SqliteMigrations
```

TIP

Не нужно указать выходной каталог для последующей миграции, так как они создаются как элементы одного уровня до последнего.

Набор одной миграции

Если вас не устраивает, наличие двух наборов миграций, их можно вручную объединить в один набор, который может применяться для обоих поставщиков.

Заметки могут существовать, так как поставщик не учитывает любые заметки, которые не понимает.

Например столбец первичного ключа, который работает с Microsoft SQL Server и SQLite может выглядеть следующим образом.

```
Id = table.Column<int>(nullable: false)
    .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn)
    .Annotation("Sqlite:Autoincrement", true),
```

Если операции могут применяться только на один поставщик (или если они по-разному между поставщиками), используйте `ActiveProvider` свойство, чтобы определить, какой поставщик является активным.

```
if (migrationBuilder.ActiveProvider == "Microsoft.EntityFrameworkCore.SqlServer")
{
    migrationBuilder.CreateSequence(
        name: "EntityFrameworkHiLoSequence");
}
```

Таблица миграции пользовательского журнала

13.09.2018 • 2 minutes to read • [Edit Online](#)

По умолчанию EF Core следит за миграций, которые были применены к базе данных, записав их в таблицу с именем `__EFMigrationsHistory`. По различным причинам можно настроить эту таблицу для удовлетворения ваших потребностей.

IMPORTANT

Если вы настраиваете таблице журнала миграции *после* применение миграций, вы несете ответственность за обновление существующей таблицы в базе данных.

Имя схемы и таблицы

Можно изменить схему и имя таблицы с помощью `.MigrationsHistoryTable()` метод в `OnConfiguring()` (или `ConfigureServices()` на ASP.NET Core). Ниже приведен пример с помощью поставщика SQL Server EF Core.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlServer(
        connectionString,
        x => x.MigrationsHistoryTable("__MyMigrationsHistory", "mySchema"));
```

Прочие изменения

Чтобы настроить дополнительные аспекты класса таблицы, переопределение и замените специфический для поставщика `IHistoryRepository` службы. Ниже приведен пример изменения имени столбца `MigrationId` для *идентификатор* на сервере SQL Server.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IHistoryRepository, MyHistoryRepository>();
```

WARNING

`SqlServerHistoryRepository` находится внутри внутреннего пространства имен и может измениться в будущих версиях.

```
class MyHistoryRepository : SqlServerHistoryRepository
{
    public MyHistoryRepository(HistoryRepositoryDependencies dependencies)
        : base(dependencies)
    {
    }

    protected override void ConfigureTable(EntityTypeBuilder<HistoryRow> history)
    {
        base.ConfigureTable(history);

        history.Property(h => h.MigrationId).HasColumnName("Id");
    }
}
```

□ API создания и удаления

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Этот раздел еще не написан! Можно отслеживать состояние данного объекта [проблема] [1](#) через средства открытый отслеживания проблем GitHub. Узнайте, как можно [contribute] [2](#) на сайте GitHub.

□ Реконструирование

29.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Этот раздел еще не написан! Можно отслеживать состояние данного объекта [проблема] [1](#) через средства открытый отслеживания проблем GitHub. Узнайте, как можно [contribute] [2](#) на сайте GitHub.

Запросы к данным

28.08.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework Core использует LINQ для запроса данных из базы данных. LINQ позволяет использовать C# (или предпочитаемый вами язык .NET) для написания строго типизированных запросов, основанных на производном контексте и классах сущностей. Представление запроса LINQ передается поставщику базы данных, который преобразует его в язык запросов конкретной базы данных (например, SQL для реляционной базы данных). Дополнительные сведения по обработке запроса см. в разделе [Принцип работы запроса](#).

Базовые запросы

02.09.2018 • 2 minutes to read • [Edit Online](#)

Узнайте, как загружать сущности из базы данных с помощью LINQ.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

101 пример LINQ

На этой странице показано несколько примеров выполнения типичных задач с использованием Entity Framework Core. Широкий набор примеров, демонстрирующих возможности LINQ, см. в разделе со [101 примером LINQ](#).

Загрузка всех данных

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

Загрузка отдельной сущности

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Фильтрация

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Загрузка связанных данных

02.09.2018 • 13 minutes to read • [Edit Online](#)

Entity Framework Core позволяет использовать свойства навигации в модели для загрузки связанных сущностей. Для загрузки связанных данных используются три общих шаблона объектно-реляционного сопоставления (O/RM).

- **Безотложная загрузка** означает, что данные загружены из базы данных как часть исходного запроса.
- **Явная загрузка** означает, что связанные данные явно загружаются из базы данных позже.
- **Отложенная загрузка** означает, что связанные данные прозрачно загружаются из базы данных при доступе к свойству навигации.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Безотложная загрузка

Вы можете использовать метод `Include`, чтобы указать связанные данные, которые будут включены в результаты запроса. В следующем примере блоги, возвращенные в результатах, будут иметь свойство `Posts`, заполненное соответствующими записями.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

TIP

EF Core автоматически исправляет свойства навигации для других сущностей, которые были ранее загружены в экземпляр контекста. Даже если данные для свойства навигации не включены явно, свойство все равно можно заполнить при условии, что ранее были загружены некоторые или все связанные сущности.

Связанные данные из нескольких связей можно включить в один запрос.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
}
```

Включение нескольких уровней

Вы можете детализировать отношения, чтобы включить несколько уровней связанных данных, используя метод `ThenInclude`. В следующем примере загружаются все блоги, связанные с ними записи и автор каждой записи.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
        .ToList();
}
```

NOTE

Текущие версии Visual Studio предлагают неправильные варианты завершения кода и могут быть причиной того, что правильные выражения могут быть помечены как содержащие синтаксические ошибки при использовании метода `ThenInclude` после свойства навигации коллекции. Это признак ошибки IntelliSense, отслеживаемой в <https://github.com/dotnet/roslyn/issues/8237>. Эти ложные синтаксические ошибки можно игнорировать, если код верен и может быть успешно скомпилирован.

Вы можете связать в цепочку несколько вызовов `ThenInclude`, чтобы продолжить включение уровней связанных данных.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .ToList();
}
```

Вы можете объединить все это, чтобы включить связанные данные из нескольких уровней и нескольких корней в один и тот же запрос.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
            .ThenInclude(owner => owner.Photo)
        .ToList();
}
```

Может потребоваться включить несколько связанных сущностей для одной включенной сущности.

Например, при запросе `blog` вы включаете `Posts`, а затем хотите включить `Author` и `Tags` из `Posts`. Для этого вам нужно указать каждый путь включения, начиная с корня. Например, `Blog -> Posts -> Author` и `Blog -> Posts -> Tags`. Это не означает, что вы получите избыточные соединения, в большинстве случаев EF будет консолидировать объединения при создании SQL.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Tags)
    .ToList();
}
```

Использование метода `Include` с производными типами

Вы можете включать связанные данные из навигаций, определенных только с производным типом, используя `Include` и `ThenInclude`.

Рассмотрим следующую модель:

```
public class SchoolContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<School> Schools { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<School>().HasMany(s => s.Students).WithOne(s => s.School);
    }
}

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Student : Person
{
    public School School { get; set; }
}

public class School
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Student> Students { get; set; }
}
```

Содержимое навигации `School` всех людей, которые являются учениками, может быть активно загружено с помощью нескольких шаблонов:

- С помощью приведения.

```
context.People.Include(person => ((Student)person).School).ToList()
```

- С помощью оператора `as`.

```
context.People.Include(person => (person as Student).School).ToList()
```

- С помощью перегрузки `Include`, принимающей параметр типа `string`.

```
context.People.Include("Student").ToList()
```

Игнорируемые включения

Если вы измените запрос так, чтобы он больше не возвращал экземпляры типа сущности, с которого начинался запрос, то операторы `include` игнорируются.

В следующем примере операторы `include` основаны на `Blog`, а затем оператор `Select` используется, чтобы изменить запрос для возврата анонимного типа. В этом случае операторы `include` не действуют.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Select(blog => new
    {
        Id = blog.BlogId,
        Url = blog.Url
    })
    .ToList();
}
```

По умолчанию EF Core будет регистрировать предупреждение, если операторы `include` игнорируются.

Дополнительные сведения о просмотре выходных данных журнала см. в [этой статье](#). Вы можете изменить поведение, когда оператор `include` будет игнорироваться, чтобы либо не возвращать данные, либо ничего не делать. Это можно указать при настройке параметров для вашего контекста — обычно в

`DbContext.OnConfiguring` или в `Startup.cs`, если вы используете ASP.NET Core.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFQuerying;Trusted_Connection=True;ConnectRetryCount=0")
        .ConfigureWarnings(warnings => warnings.Throw(CoreEventId.IncludeIgnoredWarning));
}
```

Явная загрузка

NOTE

Эта возможность появилась в EF Core 1.1.

Вы можете явно загрузить свойство навигации с помощью API `DbContext.Entry(...)`.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

Вы также можете явно загрузить свойство навигации, выполнив отдельный запрос, который возвращает связанные сущности. Если отслеживание изменений включено, то при загрузке сущности EF Core автоматически установит свойства навигации вновь загруженной сущности для ссылки на любые уже загруженные сущности и задаст свойства навигации уже загруженных сущностей, чтобы ссылаться на вновь загруженные сущности.

Запрос связанных сущностей

Можно также получить запрос LINQ, который представляет содержимое свойства навигации.

Это позволяет выполнять такие задачи, как запуск оператора агрегирования в связанных сущностях без их загрузки в память.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

Можно также фильтровать связанные сущности, которые загружаются в память.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```

Отложенная загрузка

NOTE

Эта возможность появилась в EF Core 2.1.

Самый простой способ использовать отложенную загрузку — установить пакет [Microsoft.EntityFrameworkCore.Proxies](#) и включить его с помощью вызова `UseLazyLoadingProxies`. Пример:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

При использовании `AddDbContext`:

```
.AddDbContext<BlogContext>(
    b => b.UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString));
```

Затем EF Core включит отложенную загрузку любого свойства навигации, которое может быть переопределено, то есть оно должно быть типа `virtual` и быть в классе, который может быть унаследован. Например, в следующих сущностях свойства навигации `Post.Blog` и `Blog.Posts` будут загружены отложено.

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

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

Отложенная загрузка без прокси-серверов

Прокси-серверы с отложенной загрузкой работают, внедряя службу `ILazyLoader` в сущность, как описано в статье [Entity Type Constructors](#) (Конструкторы типов сущностей). Пример:

```

public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}

```

Не требуется, чтобы наследуемые типы сущностей или свойства навигации были виртуальными, и экземплярам сущностей, созданным с помощью `new`, можно отложено загружаться после прикрепления к контексту. Однако для этого требуется ссылка на службу `ILazyLoader`, которая определена в пакете `Microsoft.EntityFrameworkCore.Abstractions`. Этот пакет содержит минимальный набор типов, поэтому он не оказывает большого влияния. Тем не менее, чтобы полностью избежать зависимости от любых пакетов EF Core в типах сущностей, можно ввести метод `ILazyLoader.Load` в качестве делегата. Пример:

```

public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}

```

В приведенном выше коде используется метод расширения `Load`, чтобы упростить использование делегата:

```
public static class PocoLoadingExtensions
{
    public static TRelated Load<TRelated>(
        this Action<object, string> loader,
        object entity,
        ref TRelated navigationField,
        [CallerMemberName] string navigationName = null)
        where TRelated : class
    {
        loader?.Invoke(entity, navigationName);

        return navigationField;
    }
}
```

NOTE

Параметр конструктора для делегата с отложенной загрузкой должен называться "lazyLoader". Конфигурация для использования отличного от этого имени запланирована в будущей версии.

Связанные данные и сериализация

Так как EF Core автоматически исправляет свойства навигации, вы можете завершить циклы в графе объектов. Например, загрузка блога и связанных записей приведет к созданию объекта блога, который ссылается на коллекцию записей. Каждая из этих записей будет ссылкой на блог.

Некоторые платформы сериализации не допускают такие циклы. Например Json.NET вызовет следующее исключение при обнаружении цикла.

```
Newtonsoft.Json.JsonSerializationException: самоссылающийся цикл обнаружен для свойства "Blog" с типом MyApplication.Models.Blog.
```

Если вы используете ASP.NET Core, вы можете настроить Json.NET для игнорирования циклов, которые он находит в графе объектов. Это делается в методе `ConfigureServices(...)` в `Startup.cs`.

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddMvc()
        .AddJsonOptions(
            options => options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );

    ...
}
```

Выполнение в клиенте и на сервере

12.10.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework Core поддерживает выполнение части запроса в клиенте и передачу второй части в базу данных. Поставщик базы данных определяет, какие части запроса могут выполняться в базе данных.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Выполнение в клиенте

В следующем примере вспомогательный метод используется для стандартизации URL-адресов блогов, которые возвращаются из базы данных SQL Server. Так как поставщик SQL Server не знает, как реализован этот метод, его невозможно преобразовать в SQL. Все остальные аспекты запроса выполняются в базе данных, но передача возвращаемого `URL` с помощью этого метода выполняется в клиенте.

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(blog => new
    {
        Id = blog.BlogId,
        Url = StandardizeUrl(blog.Url)
    })
    .ToList();
```

```
public static string StandardizeUrl(string url)
{
    url = url.ToLower();

    if (!url.StartsWith("http://"))
    {
        url = string.Concat("http://", url);
    }

    return url;
}
```

Проблемы с производительностью при вычислении в клиенте

Хотя выполнение в клиенте может быть очень полезным, в некоторых случаях это может привести к снижению производительности. Рассмотрим следующий запрос, где вспомогательный метод теперь используется в фильтре. Так как этот запрос невозможно выполнить в базе данных, все данные извлекаются в память, а затем фильтр применяется в клиенте. В зависимости от объема данных и степени их фильтрования это может привести к низкой производительности.

```
var blogs = context.Blogs
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

Журналирование вычислений в клиенте

По умолчанию EF Core будет регистрировать предупреждение при выполнении запроса в клиенте.

Дополнительные сведения о просмотре выходных данных журнала см. в [этой статье](#).

Необязательное поведение: создание исключения при вычислении в клиенте

Вы можете изменить поведение при выполнении запроса в клиенте: либо создавать предупреждение, либо ничего не делать. Это можно указать при настройке параметров для вашего контекста — обычно в

`DbContext.OnConfiguring` или в `Startup.cs`, если вы используете ASP.NET Core.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EFQuerying;Trusted_Connection=True;")
        .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
}
```

Сравнение запросов с отслеживанием и без отслеживания

02.09.2018 • 3 minutes to read • [Edit Online](#)

Поведение отслеживания контролирует, будет ли Entity Framework Core хранить информацию об экземпляре сущности в своем средстве отслеживания изменений. Если сущность отслеживается, любые изменения, обнаруженные в сущности, будут сохраняться в базе данных во время операции `SaveChanges()`. Entity Framework Core также будет адаптировать свойства навигации между сущностями, полученными из запроса отслеживания, и сущностями, которые ранее были загружены в экземпляр `DbContext`.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Отслеживания запросов

По умолчанию запросы, возвращающие типы сущностей, отслеживаются. Это означает, что вы можете внести изменения в эти экземпляры сущностей и сохранить эти изменения с помощью операции `SaveChanges()`.

В следующем примере изменение рейтинга блогов будет обнаружено и сохранено в базе данных во время операции `SaveChanges()`.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
    blog.Rating = 5;
    context.SaveChanges();
}
```

Отключение отслеживания запросов

Некакие запросы отслеживания не полезны, когда результаты используются в сценарии только для чтения. Они быстрее выполняются, потому что нет необходимости настраивать информацию об отслеживании изменений.

Вы можете отключить отслеживание для отдельного запроса:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .AsNoTracking()
        .ToList();
}
```

Вы также можете изменить поведение отслеживания по умолчанию на уровне экземпляра контекста:

```
using (var context = new BloggingContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var blogs = context.Blogs.ToList();
}
```

NOTE

Запросы отслеживания не выполняют разрешение идентичности в исполняемом запросе. Если результирующий набор содержит одну и ту же сущность несколько раз, тот же экземпляр класса сущности будет возвращен для каждого вхождения в результирующем наборе. Однако для отслеживания сущностей, которые уже были возвращены, используются слабые ссылки. Если предыдущий результат с тем же идентификатором выходит за пределы области, а сборка мусора выполняется, вы можете получить новый экземпляр сущности. Подробнее см. в статье о [принципе работы запросов](#).

Отслеживание и проекции

Даже если тип результата запроса не является типом сущности, если результат содержит типы сущностей, они по-прежнему будут отслеживаться по умолчанию. В следующем запросе, который возвращает анонимный тип, будут отслеживаться экземпляры `Blog` в результирующем наборе.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Blog = b,
                Posts = b.Posts.Count()
            });
}
```

Если результирующий набор не содержит типов сущностей, то отслеживание не выполняется. В следующем запросе, который возвращает анонимный тип с некоторыми значениями из сущности (но не экземплярами фактического типа сущности), отслеживание не выполняется.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Id = b.BlogId,
                Url = b.Url
            });
}
```

Необработанные SQL-запросы

02.09.2018 • 6 minutes to read • [Edit Online](#)

Entity Framework Core позволяет вам переходить на уровень необработанных SQL-запросов при работе с реляционной базой данных. Это может быть полезно, если запрос, который вы хотите выполнить, не может быть выражен с помощью LINQ или если использование запроса LINQ приводит к отправке незэффективных SQL-запросов в базу данных. Необработанные SQL-запросы могут возвращать типы сущностей или, начиная с EF Core 2.1, [типы запросов](#), которые являются частью модели.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Ограничения

Есть несколько ограничений, которые следует учитывать при использовании необработанных SQL-запросов:

- SQL-запрос должен возвращать данные для всех свойств сущности или типа запроса.
- Имена столбцов в результирующем наборе должны совпадать с именами столбцов, с которыми сопоставляются свойства. Обратите внимание, что это отличается от EF6, где сопоставление свойств или столбцов игнорировалось для необработанных SQL-запросов, а имена столбцов в результирующем наборе должны были соответствовать именам свойств.
- SQL-запрос не может содержать связанные данные. Однако во многих случаях вы можете использовать метод `compose` поверх запроса с помощью оператора `Include` для возврата связанных данных (см. раздел [Включение связанных данных](#)).
- Операторы `SELECT`, переданные этому методу, обычно должны быть составными: если EF Core должен вычислять дополнительные операторы запросов на сервере (например, для преобразования операторов LINQ, применяемых после `FromSql`), передаваемый SQL-запрос будет рассматриваться как вложенный запрос. Это означает, что переданный SQL-запрос не должен содержать символы и параметры, которые не допускаются во вложенных запросах, такие как:
 - конечная точка с запятой;
 - на сервере SQL Server конечное указание на уровне запроса (например, `OPTION (HASH JOIN)`);
 - на сервере SQL Server предложение `ORDER BY`, которое не сопровождается `TOP 100 PERCENT` в предложении `SELECT`.
- Операторы SQL, отличные от `SELECT`, автоматически распознаются как несоставные. Как следствие, полные результаты хранимых процедур всегда возвращаются клиенту, а любые операторы LINQ, применяемые после `FromSql`, вычисляются в памяти.

Основные необработанные SQL-запросы

Вы можете использовать метод расширения `FromSql`, чтобы начать запрос LINQ на основе необработанного SQL-запроса.

```
var blogs = context.Blogs
    .FromSql("SELECT * FROM dbo.Blogs")
    .ToList();
```

Необработанные SQL-запросы могут использоваться для выполнения хранимой процедуры.

```
var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

Передача параметров

Как и для любого API, который принимает SQL, важно параметризовать любые входные данные пользователя для защиты от атак путем внедрения кода SQL. Вы можете включить заполнители параметров в строку SQL-запроса, а затем указать значения параметров в качестве дополнительных аргументов. Значения всех указанных параметров будут автоматически преобразованы в `SqlParameter`.

В следующем примере в хранимую процедуру передается один параметр. Хотя это может выглядеть как синтаксис `String.Format`, предоставленное значение упаковано в параметр и созданное имя параметра вставляется вместо указанного заполнителя `{0}`.

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

Это тот же запрос, но с использованием синтаксиса интерполяции строк, который поддерживается в EF Core версии 2.0 и выше:

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
    .ToList();
```

Вы также можете создать `SqlParameter` и указать его как значение параметра. Это позволяет использовать именованные параметры в строке SQL-запроса.

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser @user", user)
    .ToList();
```

Создание с помощью LINQ

Если SQL-запрос может быть составлен в базе данных, вы можете использовать метод `compose` поверх исходного необработанного SQL-запроса с помощью операторов LINQ. SQL-запросы, поверх которых можно использовать метод `compose`, содержат ключевое слово `SELECT`.

В следующем примере используется необработанный SQL-запрос, который выбирает из функции с табличным значением (TVF), а затем выполняет компоновку на ее основе с использованием LINQ для

выполнения фильтрации и сортировки.

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```

Включение связанных данных

Компоновка с операторами LINQ может использоваться для включения связанных данных в запрос.

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Include(b => b.Posts)
    .ToList();
```

WARNING

Всегда используйте параметризацию для необработанных SQL-запросов. API, которые принимают необработанную строку SQL, такую как `FromSql` и `ExecuteSqlCommand`, позволяют легко передавать значения в качестве параметров. В дополнение к проверке вводимых пользователем данных всегда используйте параметры для любых значений, используемых в необработанном SQL-запросе или команде. Если вы используете объединение строк для динамической сборки любой части запроса, вам необходимо проверить все входные данные для защиты от атак путем внедрения кода SQL.

Асинхронные запросы

02.09.2018 • 2 minutes to read • [Edit Online](#)

Асинхронные запросы позволяют избежать блокирования потока при выполнении запроса в базе данных. Они помогают предотвратить замораживание пользовательского интерфейса многофункционального клиентского приложения. Асинхронные операции могут также увеличить пропускную способность веб-приложения, где можно высвободить поток для обработки других запросов во время завершения операции базы данных. Дополнительные сведения см. в статье об [асинхронном программировании на C#](#).

WARNING

EF Core не поддерживает выполнение нескольких параллельных операций в одном экземпляре контекста. Следует подождать завершения одной операции, прежде чем запускать следующую. Для этого обычно нужно указать ключевое слово `await` в каждой асинхронной операции.

Entity Framework Core предоставляет набор асинхронных методов расширения, которые можно использовать вместо методов LINQ, вызывающих выполнение запроса и возвращение результатов. Примеры: `ToListAsync()`, `ToArrayAsync()`, `SingleAsync()` и т. д. Так как эти методы предназначены только для создания дерева выражений LINQ и не вызывают выполнение запроса в базе данных, асинхронные версии операторов LINQ отсутствуют (например, `Where(...)`, `OrderBy(...)` и т. д.).

IMPORTANT

Асинхронные методы расширения EF Core определяются в пространстве имен `Microsoft.EntityFrameworkCore`. Это пространство имен необходимо импортировать, чтобы методы стали доступными.

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

Как работают запросы

12.10.2018 • 4 minutes to read • [Edit Online](#)

Entity Framework Core использует LINQ для запроса данных из базы данных. LINQ позволяет использовать C# (или предпочтительный вами язык .NET) для написания строго типизированных запросов, основанных на производном контексте и классах сущностей.

Срок действия запроса

Ниже приведен общий обзор процесса, через который проходит каждый запрос.

1. LINQ-запрос обрабатывается Entity Framework Core для создания представления, которое готово для обработки поставщиком данных.
 - a. Результат кэшируется, чтобы не выполнять эту обработку каждый раз при выполнении запроса.
2. Результат передается поставщику базы данных.
 - a. Поставщик базы данных определяет, какие части запроса могут выполняться в базе данных.
 - b. Эти части запроса преобразуются в язык запроса базы данных (например, SQL для реляционной базы данных).
 - c. Один или несколько запросов отправляются в базу данных, и возвращается результирующий набор (результатами являются значения из базы данных, а не экземпляры сущностей).
3. Для каждого элемента в результирующем наборе выполняются следующие действия:
 - a. Если это запрос отслеживания, EF проверяет, не предоставляют ли данные сущность, которая уже находится в объекте отслеживания изменений для экземпляра контекста.
 - Если да, возвращается имеющаяся сущность.
 - Если нет, создается другая сущность, устанавливается объект отслеживания изменений и возвращается новая сущность.
 - b. Если это не запрос отслеживания, EF проверяет, не предоставляют ли данные сущность, которая уже находится в результирующем наборе для этого запроса.
 - Если да, возвращается имеющаяся сущность⁽¹⁾.
 - Если нет, создается и возвращается другая сущность.

⁽¹⁾ В запросах, отличных от запросов отслеживания, используются слабые ссылки для отслеживания сущностей, которые уже были возвращены. Если предыдущий результат с тем же идентификатором выходит за пределы области, а сборка мусора выполняется, вы можете получить новый экземпляр сущности.

Когда выполняются запросы

При вызове операторов LINQ вы просто создаете представление запроса в памяти. Запрос отправляется в базу данных только после обработки результатов.

Ниже приведены наиболее распространенные операции, которые приводят к отправке запроса в базу данных.

- Итерация результатов в цикле `for`.
- Использование оператора, например `ToList`, `ToDictionary`, `Single`, `Count`.
- Привязка данных результатов запроса к пользовательскому интерфейсу.

WARNING

Всегда проверяйте входные данные пользователя. Так как платформа EF Core с помощью параметров и экранирования литералов в запросах обеспечивает защиту от атак путем внедрения кода SQL, она не проверяет входные данные. Поэтому необходимо выполнять проверку в соответствии с требованиями отдельного приложения, прежде чем значения из непроверенного источника будут использоваться в LINQ-запросах, назначенные свойствам сущности или передаваемые другим API EF Core. Сюда входят все входные данные пользователя, используемые для динамического формирования запросов. Даже при использовании LINQ, если вы принимаете входные данные пользователя для создания выражений, необходимо убедиться, что можно создать только предполагаемые выражения.

Глобальные фильтры запросов

02.09.2018 • 3 minutes to read • [Edit Online](#)

Глобальные фильтры запросов являются предикатами запросов LINQ (логическое выражение, которое обычно передается в оператор запроса LINQ *Where*), которые применяются непосредственно в типах сущностей в модели метаданных (обычно в *OnModelCreating*). Такие фильтры автоматически применяются к любым запросам LINQ, связанным с этими типами сущностей, включая типы сущностей, указанные косвенно, например, с помощью оператора *Include* или ссылок на свойства прямой навигации. Ниже приведены некоторые типичные способы применения этой функции.

- **Обратимое удаление.** Тип сущности определяет свойство *IsDeleted*.
- **Мультитенантность.** Тип сущности определяет свойство *TenantId*.

Пример

В следующем примере показано, как использовать глобальные фильтры запросов для реализации такого поведения запроса, как обратимое удаление и мультитенантность, в простой модели ведения блогов.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Сначала определите сущности:

```
public class Blog
{
    private string _tenantId;

    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public bool IsDeleted { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

Запишите объявление поля *_tenantId* в сущности *Blog*. Оно будет использоваться для связывания каждого экземпляра блога с конкретным клиентом. Также определено свойство *IsDeleted* в типе сущности *Post*. Оно используется, чтобы проверять, был ли экземпляр *Post* удален "обратимо". То есть экземпляр помечен как удаленный без физического удаления базовых данных.

Затем настройте фильтры запросов в *OnModelCreating*, используя API `HasQueryFilter`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().Property<string>("TenantId").HasField("_tenantId");

    // Configure entity filters
    modelBuilder.Entity<Blog>().HasQueryFilter(b => EF.Property<string>(b, "TenantId") == _tenantId);
    modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
}
```

Выражения предиката, поступающие в вызовы `HasQueryFilter`, будут автоматически применяться ко всем запросам LINQ для этих типов.

TIP

Обратите внимание на использование поля уровня экземпляра `DbContext`. `_tenantId` используется для установки текущего клиента. Фильтры на уровне модели будут использовать значение из правильного экземпляра контекста (то есть экземпляра, выполняющего запрос).

Отключение фильтров

Фильтры можно отключить для отдельных запросов LINQ, используя оператор `IgnoreQueryFilters()`.

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

Ограничения

Глобальные фильтры запросов имеют следующие ограничения:

- Фильтры не должны содержать ссылки на свойства навигации.
- Фильтры можно определить только для корневого типа сущности в иерархии наследования.

Сохранение данных

28.08.2018 • 2 minutes to read • [Edit Online](#)

Каждый экземпляр контекста имеет `ChangeTracker`, отвечающий за отслеживание изменений, которые требуется записать в базу данных. При внесении изменений в экземпляры классов сущностей эти изменения регистрируются в `ChangeTracker` и затем записываются в базу данных при вызове `SaveChanges`. Поставщик баз данных обеспечивает преобразование этих изменений в операции для конкретной базы данных (например, команды `INSERT`, `UPDATE` и `DELETE` для реляционной базы данных).

Базовое сохранение

02.09.2018 • 2 minutes to read • [Edit Online](#)

Узнайте, как добавлять, изменять и удалять данные, используя контекст и классы сущностей.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Добавление данных

Добавлять новые экземпляры классов сущностей можно с помощью метода `DbSet.Add`. Данные будут вставлены в базу данных при вызове метода `SaveChanges`.

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    context.Blogs.Add(blog);
    context.SaveChanges();

    Console.WriteLine(blog.BlogId + ": " + blog.Url);
}
```

TIP

Методы `Add`, `Attach` и `Update` работают с полным графом переданных им сущностей, как описано в статье [Сохранение связанных данных](#). В качестве альтернативы свойство `EntityEntry.State` можно использовать для установки состояния только одной сущности. Например, `context.Entry(blog).State = EntityState.Modified`.

Обновление данных

EF автоматически обнаружит изменения, внесенные в существующую сущность, которая отслеживается контекстом. Сюда входят сущности, которые вы загружаете или запрашиваете из базы данных, и сущности, которые ранее были добавлены и сохранены в базе данных.

Просто измените значения, присвоенные свойствам, а затем вызовите метод `SaveChanges`.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    blog.Url = "http://sample.com/blog";
    context.SaveChanges();
}
```

Удаление данных

С помощью метода `DbSet.Remove` можно удалять экземпляры классов сущностей.

Если сущность уже существует в базе данных, она будет удалена во время выполнения метода `SaveChanges`. Если сущность еще не сохранена в базе данных (т. е. она отслеживается как добавленная), она будет удалена

из контекста и больше не будет вставляться при вызове метода *SaveChanges*.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

Несколько операций в одном методе *SaveChanges*

Вы можете объединить несколько операций *Add*, *Update* и *Remove* в один вызов метода *SaveChanges*.

NOTE

Для большинства поставщиков баз данных метод *SaveChanges* является транзакционным. Это означает, что все операции будут либо успешными, либо неудачными и никогда не будут частично применены.

```
using (var context = new BloggingContext())
{
    // seeding database
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog" });
    context.Blogs.Add(new Blog { Url = "http://sample.com/another_blog" });
    context.SaveChanges();
}

using (var context = new BloggingContext())
{
    // add
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog_two" });

    // update
    var firstBlog = context.Blogs.First();
    firstBlog.Url = "";

    // remove
    var lastBlog = context.Blogs.Last();
    context.Blogs.Remove(lastBlog);

    context.SaveChanges();
}
```

Сохранение связанных данных

02.09.2018 • 4 minutes to read • [Edit Online](#)

В дополнение к изолированным сущностям вы также можете использовать связи, определенные в вашей модели.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Добавление графа новых сущностей

Если вы создадите несколько новых связанных сущностей, добавление одной из них в контекст приведет к добавлению других.

В следующем примере блог и три связанные записи вставляются в базу данных. Записи обнаруживаются и добавляются, потому что они доступны через свойство навигации `Blog.Posts`.

```
using (var context = new BloggingContext())
{
    var blog = new Blog
    {
        Url = "http://blogs.msdn.com/dotnet",
        Posts = new List<Post>
        {
            new Post { Title = "Intro to C#" },
            new Post { Title = "Intro to VB.NET" },
            new Post { Title = "Intro to F#" }
        }
    };

    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

TIP

Используйте свойство `EntityEntry.State`, чтобы установить состояние только одной сущности. Например,

```
context.Entry(blog).State = EntityState.Modified.
```

Добавление связанной сущности

Если вы ссылаетесь на новую сущность из свойства навигации сущности, которая уже отслеживается контекстом, сущность будет обнаружена и вставлена в базу данных.

В следующем примере вставляется сущность `post`, так как она добавлена в свойство `Posts` сущности `blog`, которая была получена из базы данных.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = new Post { Title = "Intro to EF Core" };

    blog.Posts.Add(post);
    context.SaveChanges();
}
```

Изменение связей

Если вы измените свойство навигации для сущности, соответствующие изменения будут внесены в столбец внешнего ключа в базе данных.

В следующем примере сущность `post` обновляется таким образом, чтобы принадлежать к новой сущности `blog`, потому что ее свойство навигации `Blog` указывает на `blog`. Обратите внимание, что `blog` также будет вставлена в базу данных, так как это новая сущность, на которую ссылается свойство навигации сущности, которая уже отслеживается контекстом (`post`).

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://blogs.msdn.com/visualstudio" };
    var post = context.Posts.First();

    post.Blog = blog;
    context.SaveChanges();
}
```

Удаление связей

Вы можете удалить связи, установив для свойства навигации по ссылке значение `null` или удалив связанную сущность из навигации коллекции.

Удаление связи может иметь побочные эффекты для зависимой сущности в соответствии с поведением каскадного удаления, настроенным в связи.

По умолчанию для обязательных связей настроено поведение каскадного удаления, и дочерняя или зависимая сущность будет удалена из базы данных. Для необязательных связей каскадное удаление по умолчанию не настроено, но для свойства внешнего ключа будет установлено значение `null`.

Дополнительные сведения о настройке обязательных и необязательных связей см. в [этом разделе](#).

Дополнительные сведения о том, как работают поведения каскадного удаления, как они могут быть настроены явно и как они выбираются по соглашению, см. в [этой статье](#).

В следующем примере каскадное удаление настраивается в связи между `Blog` и `Post`, поэтому сущность `post` удаляется из базы данных.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = blog.Posts.First();

    blog.Posts.Remove(post);
    context.SaveChanges();
}
```

Каскадное удаление

16.09.2018 • 21 minutes to read • [Edit Online](#)

Каскадное удаление обычно используется в терминологии базы данных, чтобы описать характеристики, которые позволяют удалить строку, чтобы автоматически инициировать удаление связанных строк. Тесно связанная концепция, которая также охватывается поведением удаления EF Core, — это автоматическое удаление дочерней сущности, когда ее связь с родительской сущностью разорвана. Часто это называется "удалением потерянных объектов".

EF Core реализует несколько поведений удаления и предоставляет конфигурацию поведений удаления отдельных связей. Эта платформа также реализует соглашения, с помощью которых можно автоматически настроить полезные стандартные поведения удаления для каждой связи на основе [необходимости связей](#).

Поведения удаления

Поведения удаления определены в типе перечислителя `DeleteBehavior`. Их можно передать в текущий API `OnDelete`, чтобы определить, будет ли удаление основной или родительской сущности или разрыв связей с зависимыми или дочерними сущностями иметь побочный эффект в зависимых или дочерних сущностях.

Существует три действия, которые EF может предпринять при удалении основной или родительской сущности, а также при разрыве связи с дочерней сущностью:

- дочерняя или зависимая сущность может быть удалена;
- значения дочернего внешнего ключа могут быть NULL;
- дочерняя сущность остается неизменной.

NOTE

Поведение удаления, настроенное в модели EF, применяется, только когда основная сущность удалена с помощью EF Core, а зависимые сущности загружены в память (т. е. для отслеживаемых зависимостей). В базе данных необходимо настроить соответствующее каскадное поведение, чтобы убедиться, что к данным, не отслеживаемым контекстом, применены необходимые действия. Если вы создаете базу данных с помощью EF Core, каскадное поведение будет установлено автоматически.

Для второго действия, указанного выше, устанавливать для внешнего ключа значение NULL недопустимо, если он не допускает этого значения. (Внешний ключ, не допускающий значения NULL, соответствует обязательной связи.) В этих случаях EF Core отслеживает, чтобы свойство внешнего ключа было отмечено как NULL, до вызова метода `SaveChanges`. Иначе возникнет исключение, так как изменение нельзя сохранить в базе данных. Это похоже на получение нарушения ограничения из базы данных.

Есть четыре поведения удаления, как показано в таблицах ниже.

Необязательные связи

Для необязательных связей (с внешним ключом, допускающим значение NULL) можно сохранить значение NULL внешнего ключа. Это имеет такие последствия:

Имя поведения	Влияние на зависимую или дочернюю сущность в памяти	Влияние на зависимую или дочернюю сущность в базе данных
<code>Cascade</code>	Сущности удаляются	Сущности удаляются

ИМЯ ПОВЕДЕНИЯ	ВЛИЯНИЕ НА ЗАВИСИМОУ ИЛИ ДОЧЕРНЮЮ СУЩНОСТЬ В ПАМЯТИ	ВЛИЯНИЕ НА ЗАВИСИМОУ ИЛИ ДОЧЕРНЮЮ СУЩНОСТЬ В БАЗЕ ДАННЫХ
ClientSetNull (по умолчанию)	Свойства внешнего ключа имеют значения NULL	Нет
SetNull	Свойства внешнего ключа имеют значения NULL	Свойства внешнего ключа имеют значения NULL
Restrict	Нет	Нет

Обязательные связи

Для обязательных связей (внешний ключ, не допускающий значения NULL) невозможно сохранить значение NULL внешнего ключа. Это имеет такие последствия:

ИМЯ ПОВЕДЕНИЯ	ВЛИЯНИЕ НА ЗАВИСИМОУ ИЛИ ДОЧЕРНЮЮ СУЩНОСТЬ В ПАМЯТИ	ВЛИЯНИЕ НА ЗАВИСИМОУ ИЛИ ДОЧЕРНЮЮ СУЩНОСТЬ В БАЗЕ ДАННЫХ
Cascade (по умолчанию)	Сущности удаляются	Сущности удаляются
ClientSetNull	Строки SaveChanges	Нет
SetNull	Строки SaveChanges	Строки SaveChanges
Restrict	Нет	Нет

Указанное в таблице выше значение *Нет* может привести к нарушению ограничения. Например, если основная или дочерняя сущность удалена, но никаких действий, чтобы изменить внешний ключ зависимой или дочерней сущности не выполняется, то база данных, вероятно, вызовет метод `SaveChanges` из-за нарушения ограничения внешнего ключа.

На высоком уровне:

- Если у вас есть сущности, которые не могут существовать без родительской сущности, и вы хотите, чтобы EF автоматически удалил дочернюю сущность, воспользуйтесь поведением *Cascade*.
 - Сущности, которые не могут существовать без родительского элемента, обычно используют обязательные связи, в которых по умолчанию используется поведение *Cascade*.
- Если у вас есть сущности, у которых может быть или отсутствовать родительский элемент, и вы хотите, чтобы EF автоматически обнулил внешний ключ, используйте поведение *ClientSetNull*.
 - Сущности, которые могут существовать без родительского элемента, обычно используют необязательные связи, в которых по умолчанию используется поведение *ClientSetNull*.
 - Если требуется, чтобы база данных также пыталась распространять значения NULL для дочерних внешних ключей, даже когда дочерняя сущность не загружена, используйте поведение *SetNull*. Тем не менее, обратите внимание, что база данных должна поддерживать эту возможность. Настройка базы данных таким образом может привести к другим ограничениям, которые на практике часто делают этот вариант непрактичным. Поэтому *SetNull* не используется по умолчанию.
- Если вы хотите, чтобы платформа EF Core никогда автоматически не удаляла сущность или не обнуляла внешний ключ, используйте поведение *Restrict*. Обратите внимание, что для этого необходимо вручную синхронизировать в коде дочерние сущности и их значения внешнего ключа, в противном случае возникнут исключения ограничения.

NOTE

В EF Core, в отличие от EF6, каскадные эффекты не происходят немедленно, а только при вызове метода `SaveChanges`.

NOTE

Изменения в EF Core 2.0. В предыдущих выпусках поведение `Restrict` присваивало для необязательных свойств внешнего ключа в отслеживаемых зависимых сущностях значение NULL. Это было стандартное поведение удаления для необязательных связей. В EF Core 2.0 предоставлено `ClientSetNull`, которое имеет аналогичное поведение и является стандартным для необязательных связей. Поведение `Restrict` было изменено, чтобы никогда не иметь побочных эффектов в зависимых сущностях.

Примеры удаления сущности

Приведенный ниже код является частью [примера](#), который можно скачать и выполнить. В примере показано, что происходит с каждым поведением удаления для обязательных и необязательных связей при удалении родительской сущности.

```
var blog = context.Blogs.Include(b => b.Posts).First();
var posts = blog.Posts.ToList();

DumpEntities(" After loading entities:", context, blog, posts);

context.Remove(blog);

DumpEntities($" After deleting blog '{blog.BlogId}':", context, blog, posts);

try
{
    Console.WriteLine();
    Console.WriteLine(" Saving changes:");

    context.SaveChanges();

    DumpSql();

    DumpEntities(" After SaveChanges:", context, blog, posts);
}
catch (Exception e)
{
    DumpSql();

    Console.WriteLine();
    Console.WriteLine($" SaveChanges threw {e.GetType().Name}: {(e is DbUpdateException ? e.InnerException.Message : e.Message)}");
}
```

Рассмотрим каждый вариант, чтобы понять, что происходит.

DeleteBehavior.Cascade с обязательной или необязательной связью

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After deleting blog '1':  
Blog '1' is in state Deleted with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
Saving changes:  
DELETE FROM [Posts] WHERE [PostId] = 1  
DELETE FROM [Posts] WHERE [PostId] = 2  
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

```
After SaveChanges:  
Blog '1' is in state Detached with 2 posts referenced.  
Post '1' is in state Detached with FK '1' and no reference to a blog.  
Post '2' is in state Detached with FK '1' and no reference to a blog.
```

- Блог помечен как удаленный.
- Записи изначально остаются без изменений, так как каскадные удаления не выполняются до вызова метода SaveChanges.
- SaveChanges отправляет операции удаления в зависимые или дочерние сущности (записи), а затем в основные или родительские (блог).
- После сохранения все сущности отсоединяются, так как они были удалены из базы данных.

DeleteBehavior.ClientSetNull или DeleteBehavior.SetNull с обязательной связью

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After deleting blog '1':  
Blog '1' is in state Deleted with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
Saving changes:  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1
```

```
SaveChanges threw DbUpdateException: Cannot insert the value NULL into column 'BlogId', table  
'EFSaving.CascadeDelete.dbo.Posts'; column does not allow nulls. UPDATE fails. The statement has been  
terminated.
```

- Блог помечен как удаленный.
- Записи изначально остаются без изменений, так как каскадные удаления не выполняются до вызова метода SaveChanges.
- SaveChanges пытается установить для внешнего ключа записи значение NULL. Однако эта попытка завершается неудачей, так как внешний ключ не допускает этого значения.

DeleteBehavior.ClientSetNull или DeleteBehavior.SetNull с необязательной связью

```

After loading entities:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

After deleting blog '1':
Blog '1' is in state Deleted with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

Saving changes:
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 2
DELETE FROM [Blogs] WHERE [BlogId] = 1

After SaveChanges:
Blog '1' is in state Detached with 2 posts referenced.
Post '1' is in state Unchanged with FK 'null' and no reference to a blog.
Post '2' is in state Unchanged with FK 'null' and no reference to a blog.

```

- Блог помечен как удаленный.
- Записи изначально остаются без изменений, так как каскадные удаления не выполняются до вызова метода SaveChanges.
- SaveChanges пытается установить для внешнего ключа зависимых или дочерних сущностей (записей) значение NULL, прежде чем удалить основные или родительские (блог).
- После сохранения основная или родительская сущность (блог) удаляется, но зависимые или дочерние сущности (записи) по-прежнему отслеживаются.
- Отслеживаемые зависимые и дочерние сущности (записи) теперь имеют NULL в качестве значений внешних ключей, а их связь с удаленными основными и родительскими сущностями (блог) удалена.

DeleteBehavior.Restrict с обязательной или необязательной связью

```

After loading entities:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

After deleting blog '1':
Blog '1' is in state Deleted with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

Saving changes:
SaveChanges threw InvalidOperationException: The association between entity types 'Blog' and 'Post' has
been severed but the foreign key for this relationship cannot be set to null. If the dependent entity should
be deleted, then setup the relationship to use cascade deletes.

```

- Блог помечен как удаленный.
- Записи изначально остаются без изменений, так как каскадные удаления не выполняются до вызова метода SaveChanges.
- Так как *Restrict* указывает EF не устанавливать автоматически для внешнего ключа значение NULL, он не имеет этого значения и метод SaveChanges вызывается без сохранения.

Примеры удаления потерянных объектов

Приведенный ниже код является частью [примера](#), который можно скачать и выполнить. В примере показано, что происходит с каждым поведением удаления для обязательных и необязательных связей при разрыве связи между родительскими или основными сущностями и их дочерними или зависимыми

сущностями. В этом примере связь разрывается путем удаления зависимых или дочерних сущностей (записей) из свойства навигации коллекции в основных или родительских сущностях (блог). Тем не менее поведение остается прежним, если ссылка из зависимой или дочерней сущности к основной или родительской сущности обнулена.

```
var blog = context.Blogs.Include(b => b.Posts).First();
var posts = blog.Posts.ToList();

DumpEntities(" After loading entities:", context, blog, posts);

blog.Posts.Clear();

DumpEntities(" After making posts orphans:", context, blog, posts);

try
{
    Console.WriteLine();
    Console.WriteLine(" Saving changes:");

    context.SaveChanges();

    DumpSql();

    DumpEntities(" After SaveChanges:", context, blog, posts);
}
catch (Exception e)
{
    DumpSql();

    Console.WriteLine();
    Console.WriteLine($" SaveChanges threw {e.GetType().Name}: {(e is DbUpdateException ? e.InnerException.Message : e.Message)}");
}
```

Рассмотрим каждый вариант, чтобы понять, что происходит.

DeleteBehavior.Cascade с обязательной или необязательной связью

```
After loading entities:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

After making posts orphans:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Modified with FK '1' and no reference to a blog.
Post '2' is in state Modified with FK '1' and no reference to a blog.

Saving changes:
DELETE FROM [Posts] WHERE [PostId] = 1
DELETE FROM [Posts] WHERE [PostId] = 2

After SaveChanges:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Detached with FK '1' and no reference to a blog.
Post '2' is in state Detached with FK '1' and no reference to a blog.
```

- Записи помечены как измененные, так как из-за разрыва связи внешний ключ помечен как NULL.
 - Если внешний ключ не допускает значения NULL, то фактическое значение не будет изменено, даже если оно помечено как NULL.
- SaveChanges отправляет операции удаления в зависимые или дочерние сущности (записи).
- После сохранения зависимые или дочерние сущности (записи) отсоединяются, так как они были удалены

из базы данных.

DeleteBehavior.ClientSetNull или DeleteBehavior.SetNull с обязательной связью

After loading entities:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

After making posts orphans:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK 'null' and no reference to a blog.  
Post '2' is in state Modified with FK 'null' and no reference to a blog.
```

Saving changes:

```
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1
```

```
SaveChanges threw DbUpdateException: Cannot insert the value NULL into column 'BlogId', table  
'EFSaving.CascadeDelete.dbo.Posts'; column does not allow nulls. UPDATE fails. The statement has been  
terminated.
```

- Записи помечены как измененные, так как из-за разрыва связи внешний ключ помечен как NULL.
 - Если внешний ключ не допускает значения NULL, то фактическое значение не будет изменено, даже если оно помечено как NULL.
- SaveChanges пытается установить для внешнего ключа записи значение NULL. Однако эта попытка завершается неудачей, так как внешний ключ не допускает этого значения.

DeleteBehavior.ClientSetNull или DeleteBehavior.SetNull с необязательной связью

After loading entities:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

After making posts orphans:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK 'null' and no reference to a blog.  
Post '2' is in state Modified with FK 'null' and no reference to a blog.
```

Saving changes:

```
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 2
```

After SaveChanges:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK 'null' and no reference to a blog.  
Post '2' is in state Unchanged with FK 'null' and no reference to a blog.
```

- Записи помечены как измененные, так как из-за разрыва связи внешний ключ помечен как NULL.
 - Если внешний ключ не допускает значения NULL, то фактическое значение не будет изменено, даже если оно помечено как NULL.
- SaveChanges устанавливает для внешнего ключа зависимых или дочерних сущностей (записей) значение NULL.
- После сохранения зависимые или дочерние сущности (записи) теперь имеют NULL в качестве значений внешних ключей, а их связь с удаленными основными или родительскими сущностями (блог) удалена.

DeleteBehavior.Restrict с обязательной или необязательной связью

```
After loading entities:
```

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After making posts orphans:
```

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK '1' and no reference to a blog.  
Post '2' is in state Modified with FK '1' and no reference to a blog.
```

```
Saving changes:
```

```
SaveChanges threw InvalidOperationException: The association between entity types 'Blog' and 'Post' has  
been severed but the foreign key for this relationship cannot be set to null. If the dependent entity should  
be deleted, then setup the relationship to use cascade deletes.
```

- Записи помечены как измененные, так как из-за разрыва связи внешний ключ помечен как NULL.
 - Если внешний ключ не допускает значения NULL, то фактическое значение не будет изменено, даже если оно помечено как NULL.
- Так как *Restrict* указывает EF не устанавливать автоматически для внешнего ключа значение NULL, он не имеет этого значения и метод SaveChanges вызывается без сохранения.

Использование каскадных функций в неотслеживаемых сущностях

При вызове метода *SaveChanges* правила каскадного удаления будут применяться ко всем сущностям, которые отслеживаются контекстом. Такая ситуация приведена в примерах выше. Вот почему был создан код SQL, чтобы удалить основные или родительские сущности (блог) и все зависимые или дочерние сущности (записи).

```
DELETE FROM [Posts] WHERE [PostId] = 1  
DELETE FROM [Posts] WHERE [PostId] = 2  
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

Если загружается только субъект, например, когда запрос, выполненный для блога без `Include(b => b.Posts)`, также содержит записи, то *SaveChanges* создаст код SQL, чтобы удалить основную или родительскую сущность.

```
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

Зависимые или дочерние сущности (записи) будут удалены, только если в базе данных настроено соответствующее каскадное поведение. Если вы создаете базу данных с помощью EF, каскадное поведение будет установлено автоматически.

Обработка конфликтов параллелизма

02.09.2018 • 6 minutes to read • [Edit Online](#)

NOTE

На этой странице описывается, как работает параллелизм в EF Core и как обрабатывать конфликты параллелизма в вашем приложении. Подробнее о том, как настроить маркеры параллелизма в вашей модели, см. в [этой статье](#).

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Параллелизм базы данных относится к ситуациям, в которых несколько процессов или пользователей получают доступ или одновременно изменяют одни и те же данные в базе данных. Управление параллелизмом относится к конкретным механизмам, используемым для обеспечения согласованности данных при наличии одновременных изменений.

EF Core реализует *оптимистичный контроль параллелизма*, который позволяет нескольким процессам или пользователям самостоятельно вносить изменения без накладных расходов на синхронизацию или блокировку. В идеальной ситуации эти изменения не будут мешать друг другу и, следовательно, смогут завершиться успешно. В худшем случае два или более процесса будут пытаться вносить конфликтующие изменения, и только один из них сможет преуспеть.

Как работает управление параллелизмом в EF Core

Свойства, настроенные как маркеры параллелизма, используются для реализации оптимистического контроля параллелизма: всякий раз, когда операция обновления или удаления выполняется во время `SaveChanges`, значение маркера параллелизма в базе данных сравнивается с исходным значением, считанным EF Core.

- Если значения совпадают, операция может завершиться.
- Если значения не совпадают, EF Core предполагает, что другой пользователь выполнил конфликтную операцию и прервал текущую транзакцию.

Ситуация, когда другой пользователь выполнил операцию, конфликтующую с текущей операцией, называется *конфликтом параллелизма*.

Поставщики баз данных отвечают за реализацию сравнения значений маркеров параллелизма.

В реляционных базах данных EF Core включает проверку значения маркера параллелизма в предложении `WHERE` любых инструкций с `UPDATE` или `DELETE`. После выполнения операторов EF Core считывает количество затронутых строк.

Если никакие строки не затронуты, конфликт параллелизма обнаруживается и EF Core вызывает `DbUpdateConcurrencyException`.

Например, мы можем настроить `Lastname` в `Person` как маркер параллелизма. Тогда любая операция обновления для пользователя будет содержать проверку параллелизма в предложении `WHERE`:

```
UPDATE [Person] SET [FirstName] = @p1  
WHERE [PersonId] = @p0 AND [LastName] = @p2;
```

Разрешение конфликтов параллелизма

Продолжая рассматривать предыдущий пример, если один пользователь пытается сохранить некоторые изменения в `Person`, но другой пользователь уже изменил `LastName`, будет вызвано исключение.

На этом этапе приложение может просто проинформировать пользователя о том, что обновление не было успешным из-за противоречивых изменений, и продолжить выполнение. Но может быть желательно запросить пользователя убедиться, что запись по-прежнему представляет одного и того же фактического пользователя, и повторить операцию.

Этот процесс является примером *разрешения конфликта параллелизма*.

Разрешение конфликта параллелизма включает в себя объединение ожидающих изменений из текущего контекста `DbContext` со значениями в базе данных. Объединяемые значения отличаются в зависимости от приложения, и их может указать пользователь, введя входные данные.

Доступны три набора значений для разрешения конфликта параллелизма:

- **Текущие значения** — это значения, которые приложение пыталось записать в базу данных.
- **Исходные значения** — это значения, которые изначально были получены из базы данных до того, как были сделаны какие-либо изменения.
- **Значения базы данных** — это значения, которые в настоящее время хранятся в базе данных.

Общий подход к обработке конфликтов параллелизма:

1. Получите `DbUpdateConcurrencyException` в процессе `SaveChanges`.
2. Используйте `DbUpdateConcurrencyException.Entries`, чтобы подготовить новый набор изменений для затронутых объектов.
3. Обновите исходные значения маркера параллелизма, чтобы отразить текущие значения в базе данных.
4. Повторяйте процесс, пока не возникнут конфликты.

В следующем примере `Person.FirstName` и `Person.LastName` устанавливаются как маркеры параллелизма. В месте, где вы указываете конкретную логику приложения, есть комментарий `// TODO:`, позволяющий выбрать значение, которое нужно сохранить.

```

using (var context = new PersonContext())
{
    // Fetch a person from database and change phone number
    var person = context.People.Single(p => p.PersonId == 1);
    person.PhoneNumber = "555-555-5555";

    // Change the person's name in the database to simulate a concurrency conflict
    context.Database.ExecuteSqlCommand(
        "UPDATE dbo.People SET FirstName = 'Jane' WHERE PersonId = 1");

    var saved = false;
    while (!saved)
    {
        try
        {
            // Attempt to save changes to the database
            context.SaveChanges();
            saved = true;
        }
        catch (DbUpdateConcurrencyException ex)
        {
            foreach (var entry in ex.Entries)
            {
                if (entry.Entity is Person)
                {
                    var proposedValues = entry.CurrentValues;
                    var databaseValues = entry.GetDatabaseValues();

                    foreach (var property in proposedValues.Properties)
                    {
                        var proposedValue = proposedValues[property];
                        var databaseValue = databaseValues[property];

                        // TODO: decide which value should be written to database
                        // proposedValues[property] = <value to be saved>;
                    }

                    // Refresh original values to bypass next concurrency check
                    entry.OriginalValues.SetValues(databaseValues);
                }
                else
                {
                    throw new NotSupportedException(
                        "Don't know how to handle concurrency conflicts for "
                        + entry.Metadata.Name);
                }
            }
        }
    }
}

```

Использование транзакций

02.09.2018 • 8 minutes to read • [Edit Online](#)

Транзакции позволяют обрабатывать несколько операций с базой данных атомарным способом. Если транзакция зафиксирована, все операции успешно применяются к базе данных. Если транзакция отменяется, ни одна из операций не применяется к базе данных.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Поведение транзакции по умолчанию

По умолчанию, если поставщик базы данных поддерживает транзакции, все изменения в одном вызове `SaveChanges()` применяются в транзакции. Если какое-либо из изменений завершается ошибкой, транзакция откатывается и ни одно из изменений не применяется к базе данных. Это означает, что операция `SaveChanges()` гарантированно либо будет выполнена, либо оставит базу данных без изменений, если возникла ошибка.

Для большинства приложений это поведение по умолчанию является достаточным. Транзакциями нужно управлять вручную только в том случае, если этого требует приложение.

Управление транзакциями

Вы можете использовать API `DbContext.Database` для начала, фиксации и отката транзакций. В следующем примере показаны две операции `SaveChanges()` и запрос LINQ, выполняемый в одной транзакции.

Не все поставщики баз данных поддерживают транзакции. Некоторые поставщики могут вызывать исключение или не работать при вызове API транзакций.

```

using (var context = new BloggingContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context.SaveChanges();

            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
            context.SaveChanges();

            var blogs = context.Blogs
                .OrderBy(b => b.Url)
                .ToList();

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (Exception)
        {
            // TODO: Handle failure
        }
    }
}

```

Межконтекстная транзакция (только для реляционных баз данных)

Вы можете использовать транзакцию в нескольких экземплярах контекста. Эта функция доступна только при использовании поставщика реляционной базы данных, так как для этого требуется использование транзакции `DbTransaction` и подключения `DbConnection`, которые относятся к реляционным базам данных.

Для совместного использования транзакции в контекстах должны применяться одинаковые `DbConnection` и `DbTransaction`.

Разрешение подключения извне

Для совместного использования подключения `DbConnection` требуется возможность передать подключение в контекст при его создании.

Самый простой способ разрешить предоставление `DbConnection` извне — это прекратить использование метода `DbContext.OnConfiguring`, чтобы настроить контекст и создать параметры `DbContextOptions` извне и передать их конструктору контекста.

TIP

`DbContextOptionsBuilder` — это API, который вы использовали в `DbContext.OnConfiguring` для настройки контекста. Теперь вы будете использовать его извне для создания параметров `DbContextOptions`.

```

public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}

```

Как вариант, можно использовать `DbContext.OnConfiguring`, но принимать подключение `DbContext`, которое сохраняется, а затем используется в `DbContext.OnConfiguring`.

```
public class BloggingContext : DbContext
{
    private DbConnection _connection;

    public BloggingContext(DbConnection connection)
    {
        _connection = connection;
    }

    public DbSet<Blog> Blogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connection);
    }
}
```

Совместное использование подключения и транзакции

Теперь вы можете создать несколько экземпляров контекста, которые используют одно и то же подключение. Затем используйте API `DbContext.Database.UseTransaction(DbTransaction)`, чтобы включить оба контекста в одну транзакцию.

```
var options = new DbContextOptionsBuilder<BloggingContext>()
    .UseSqlServer(new SqlConnection(connectionString))
    .Options;

using (var context1 = new BloggingContext(options))
{
    using (var transaction = context1.Database.BeginTransaction())
    {
        try
        {
            context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context1.SaveChanges();

            using (var context2 = new BloggingContext(options))
            {
                context2.Database.UseTransaction(transaction.GetDbTransaction());

                var blogs = context2.Blogs
                    .OrderBy(b => b.Url)
                    .ToList();
            }
        }

        // Commit transaction if all commands succeed, transaction will auto-rollback
        // when disposed if either commands fails
        transaction.Commit();
    }
    catch (Exception)
    {
        // TODO: Handle failure
    }
}
```

Использование внешних транзакций базы данных (только для реляционных баз данных)

Если вы используете несколько технологий доступа к данным для доступа к реляционной базе данных, вам может потребоваться обмениваться транзакциями между операциями, выполняемыми с помощью различных технологий.

В следующем примере показано, как выполнить операцию ADO.NET SqlConnection и операцию Entity Framework Core в одной транзакции.

```
using (var connection = new SqlConnection(connectionString))
{
    connection.Open();

    using (var transaction = connection.BeginTransaction())
    {
        try
        {
            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.Transaction = transaction;
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseSqlServer(connection)
                .Options;

            using (var context = new BloggingContext(options))
            {
                context.Database.UseTransaction(transaction);
                context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context.SaveChanges();
            }

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (System.Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

Использование System.Transactions

NOTE

Это новая возможность в EF Core 2.1.

Можно использовать внешние транзакции, если вам нужно координировать действия в более широком диапазоне.

```
using (var scope = new TransactionScope(
    TransactionScopeOption.Required,
    new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted }))
{
    using (var connection = new SqlConnection(connectionString))
    {
        connection.Open();

        try
        {
            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseSqlServer(connection)
                .Options;

            using (var context = new BloggingContext(options))
            {
                context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context.SaveChanges();
            }

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            scope.Complete();
        }
        catch (System.Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

Также можно использовать явную транзакцию.

```

using (var transaction = new CommittableTransaction(
    new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted }))
{
    var connection = new SqlConnection(connectionString);

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggingContext(options))
        {
            context.Database.OpenConnection();
            context.Database.EnlistTransaction(transaction);

            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context.SaveChanges();
            context.Database.CloseConnection();
        }

        // Commit transaction if all commands succeed, transaction will auto-rollback
        // when disposed if either commands fails
        transaction.Commit();
    }
    catch (System.Exception)
    {
        // TODO: Handle failure
    }
}

```

Ограничения System.Transactions

- Реализация поддержки System.Transactions в EF Core зависит от поставщиков баз данных. Хотя поддержка распространена среди поставщиков ADO.NET для .NET Framework, API только недавно был добавлен в .NET Core, поэтому на этой платформе поддержка не столь распространена. Если поставщик не реализует поддержку System.Transactions, вызовы этих API могут полностью игнорироваться. SqlClient для .NET Core поддерживает эту функцию, начиная с версии 2.1. SqlClient для .NET Core 2.0 вызовет исключение, если вы попытаетесь использовать эту функцию.

IMPORTANT

Рекомендуется проверить, что API правильно работает с поставщиком, прежде чем использовать его для управления транзакциями. Если он работает неправильно, рекомендуется связаться с представителем поставщика базы данных.

- Начиная с версии 2.1, реализация System.Transactions в .NET Core не включает в себя поддержку распределенных транзакций, поэтому вы не можете использовать `TransactionScope` или `CommittableTransaction` для координации транзакций в нескольких диспетчерах ресурсов.

Асинхронное сохранение

02.09.2018 • 2 minutes to read • [Edit Online](#)

Асинхронное сохранение позволяет избежать блокировки потока при записи изменений в базу данных. Эта функция помогает предотвратить замораживание пользовательского интерфейса многофункционального клиентского приложения. Асинхронные операции могут также увеличить пропускную способность веб-приложения, где можно высвободить поток для обработки других запросов во время завершения операции базы данных. Дополнительные сведения см. в статье об [асинхронном программировании на C#](#).

WARNING

EF Core не поддерживает выполнение нескольких параллельных операций в одном экземпляре контекста. Следует подождать завершения одной операции, прежде чем запускать следующую. Для этого обычно нужно указать ключевое слово `await` в каждой асинхронной операции.

Entity Framework Core предоставляет `DbContext.SaveChangesAsync()` в качестве асинхронной альтернативы `DbContext.SaveChanges()`.

```
public static async Task AddBlogAsync(string url)
{
    using (var context = new BloggingContext())
    {
        var blog = new Blog { Url = url };
        context.Blogs.Add(blog);
        await context.SaveChangesAsync();
    }
}
```

Отключенные сущности

02.09.2018 • 13 minutes to read • [Edit Online](#)

Экземпляр DbContext будет автоматически отслеживать сущности, возвращенные из базы данных. Изменения, внесенные в эти сущности, затем будут обнаружены при вызове SaveChanges, и база данных будет обновлена при необходимости. Дополнительные сведения см. в статье о [базовом сохранении и связанных данных](#).

Тем не менее иногда сущности запрашиваются с помощью одного экземпляра контекста, а затем сохраняются с помощью другого экземпляра. Это часто происходит в автономных сценариях, например в веб-приложении, в котором сущности запрашиваются, отправляются клиенту, изменяются, возвращаются назад на сервер в запросе, а затем сохраняются. В этом случае второй экземпляр контекста должен знать, что нужно сделать: добавить новые сущности или обновить имеющиеся.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

TIP

EF Core может отслеживать только один экземпляр любой сущности с использованием указанного значения первичного ключа. Наилучший способ избежать этой проблемы — использовать кратковременный контекст для каждой единицы работы, чтобы контекст был пустым при запуске, имел связанные с ним сущности, сохранял эти сущности, а затем удалялся и отклонялся.

Определение новых сущностей

Определение новых сущностей клиентом

Самый простой способ определения — это когда клиент сообщает серверу, является ли сущность новой или нет. Например, часто запрос для вставки новой сущности отличается от запроса для обновления имеющейся сущности.

Далее в этом разделе описаны случаи, когда сущности необходимо определить способами, отличными от вставки или обновления.

Использование автоматически созданных ключей

Значение автоматически созданного ключа часто может использоваться, чтобы определить, что нужно сделать: вставить или обновить сущность. Если ключ не задан (например, он по-прежнему имеет стандартное значение среди выполнения NULL, ноль и т. д.), то сущность должна быть новой и необходимо выполнить операцию вставки. С другой стороны, если значение ключа задано, то оно должно было быть сохранено ранее и теперь его нужно обновить. Другими словами, если ключ имеет значение, то сущность была запрошена, отправлена клиенту и теперь вернулась для обновления.

Очень просто проверить, задан ли ключ, если известен тип сущности:

```
public static bool IsItNew(Blog blog)
    => blog.BlogId == 0;
```

Однако EF также имеет встроенный способ выполнить это действие для любого типа сущности и ключа:

```
public static bool IsItNew(DbContext context, object entity)
=> !context.Entry(entity).IsKeySet;
```

TIP

Ключи задаются сразу же после того, как сущности начинают отслеживаться контекстом, даже если сущность находится в добавленном состоянии. Это помогает при обходе графа сущностей. Кроме того, это позволяет решить, что нужно делать с каждым из них, например, при использовании API TrackGraph. Значение ключа можно использовать только показанным здесь способом, *прежде чем* будет выполнен вызов для отслеживания сущности.

Использование других ключей

Если значения ключей не создаются автоматически, необходимо использовать другой механизм для определения новых сущностей. Для этого есть два общих подхода:

- запросить сущность;
- передать флаг от клиента.

Запросить сущность можно с помощью метода Find:

```
public static bool IsItNew(BloggingContext context, Blog blog)
=> context.Blogs.Find(blog.BlogId) == null;
```

Мы не можем показать в этом документе полный код для передачи флага от клиента. В веб-приложении это часто означает выполнение разных запросов для разных действий или передачу некоторого состояния в запросе, а затем извлечение его в контроллере.

Сохранение одной сущности

Если неизвестно, следует выполнять вставку или обновление, то можно должным образом использовать метод Add или Update:

```
public static void Insert(DbContext context, object entity)
{
    context.Add(entity);
    context.SaveChanges();
}

public static void Update(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

Тем не менее, если в сущности используются автоматически созданные значения ключей, в обоих случаях можно использовать метод Update:

```
public static void InsertOrUpdate(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

Метод Update обычно помечает сущность для обновления, не для вставки. Тем не менее, если сущность имеет автоматически созданный ключ и значение ключа не было задано, то вместо этого сущность

автоматически помечается для вставки.

TIP

Это поведение появилось в EF Core 2.0. Для более ранних выпусков всегда необходимо явно выбрать метод Add или Update.

Если в сущности не используются автоматически созданные ключи, то приложение должно решить, следует ли вставить или обновить сущность. Например:

```
public static void InsertOrUpdate(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs.Find(blog.BlogId);
    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
    }

    context.SaveChanges();
}
```

Ниже приведены нужные действия.

- Если метод Find возвращает значение NULL, то база данных не содержит блогов с этим идентификатором. Поэтому мы вызываем метод Add, чтобы пометить сущность для вставки.
- Если метод Find возвращает сущность, то она имеется в базе данных и контекст теперь отслеживает имеющуюся сущность.
 - Затем мы используем метод SetValues, чтобы задать для всех свойств этой сущности значения, которые переданы от клиента.
 - При вызове метода SetValues будет помечена сущность, которая будет обновлена при необходимости.

TIP

SetValues будет помечать сущности только при изменении свойств, значения которых отличаются от значений в отслеживаемой сущности. Это означает, что при отправке операции обновления, будут обновлены только столбцы, которые изменились. (Если ничего не изменилось, операция обновления не будет отправлена.)

Работа с графиками

Разрешение идентификатора

Как отмечалось выше, EF Core может отслеживать только один экземпляр любой сущности с использованием указанного значения первичного ключа. При работе с графиками граф должен в идеале создаваться таким способом, чтобы этот инвариант поддерживался. Контекст должен использоваться только для одной единицы работы. Если граф не содержит дубликатов, понадобится обработать его, прежде чем отправить в EF для объединения нескольких экземпляров в один. Это может быть сложной задачей, если экземпляры имеют конфликтующие значения и связи. Поэтому следует как можно быстрее объединить дубликаты в конвейере приложения, чтобы избежать разрешения конфликтов.

Все новые или имеющиеся сущности

Пример работы с графами — обновление или вставка блога вместе с его коллекцией связанных записей. Если необходимо вставить или обновить все сущности в графе, этот процесс аналогичен описанному выше процессу для одной сущности. Например, граф блогов и записей, созданных таким образом

```
var blog = new Blog
{
    Url = "http://sample.com",
    Posts = new List<Post>
    {
        new Post {Title = "Post 1"},
        new Post {Title = "Post 2"},
    }
};
```

можно вставить следующим образом:

```
public static void InsertGraph(DbContext context, object rootEntity)
{
    context.Add(rootEntity);
    context.SaveChanges();
}
```

Вызов метода Add отметит блог и все записи, которые будут вставлены.

Аналогично, если все сущности в графе необходимо обновить, можно использовать метод Update:

```
public static void UpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

Блог и все записи будут помечены для обновления.

Сочетание новых и имеющихся сущностей

Если у вас есть автоматически созданные ключи, можно снова использовать метод Update для операций вставки и обновления, даже если график содержит сочетание сущностей, для которых требуется выполнить вставку и обновление:

```
public static void InsertOrUpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

Метод Update отметит все сущности в графике, блоге или записи для вставки, если для него не задано значения ключа. Все другие записи будут помечены для обновления.

Как и ранее, если не используются автоматически созданные ключи, можно выполнить запрос и некоторую обработку:

```
public static void InsertOrUpdateGraph(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs
        .Include(b => b.Posts)
        .FirstOrDefault(b => b.BlogId == blog.BlogId);

    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
        foreach (var post in blog.Posts)
        {
            var existingPost = existingBlog.Posts
                .FirstOrDefault(p => p.PostId == post.PostId);

            if (existingPost == null)
            {
                existingBlog.Posts.Add(post);
            }
            else
            {
                context.Entry(existingPost).CurrentValues.SetValues(post);
            }
        }
    }

    context.SaveChanges();
}
```

Обработка удалений

Выполнение удаления может оказаться сложной задачей, так как очень часто отсутствие сущности означает, что ее необходимо удалить. Для решения этой проблемы можно выполнить обратимые удаления. Например, пометить сущность как удаленную, а не удалять ее. Таким образом удаление идентично обновлению.

Обратимые удаления можно реализовать с помощью [фильтров запросов](#).

Для необратимых удалений обычно используется расширение шаблона запроса, чтобы выполнить действия, которые фактически изменяют граф. Пример:

```

public static void InsertUpdateOrDeleteGraph(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs
        .Include(b => b.Posts)
        .FirstOrDefault(b => b.BlogId == blog.BlogId);

    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
        foreach (var post in blog.Posts)
        {
            var existingPost = existingBlog.Posts
                .FirstOrDefault(p => p.PostId == post.PostId);

            if (existingPost == null)
            {
                existingBlog.Posts.Add(post);
            }
            else
            {
                context.Entry(existingPost).CurrentValues.SetValues(post);
            }
        }

        foreach (var post in existingBlog.Posts)
        {
            if (!blog.Posts.Any(p => p.PostId == post.PostId))
            {
                context.Remove(post);
            }
        }
    }

    context.SaveChanges();
}

```

TrackGraph

В методах `Internally`, `Add`, `Attach` и `Update` используется обход графа с определением, выполненным для каждой сущности, независимо от того, следует ли ее пометить как `Added` (для вставки), `Modified` (для обновления), `Unchanged` (не выполнять никаких действий) или `Deleted` (для удаления). Этот механизм предоставляет API `TrackGraph`. Например, предположим, что когда клиент отправляет обратно график сущностей, он устанавливает некоторые флаги на каждой сущности, указывая, как ее следует обрабатывать. `TrackGraph` затем может использоваться для обработки этого флага:

```
public static void SaveAnnotatedGraph(DbContext context, object rootEntity)
{
    context.ChangeTracker.TrackGraph(
        rootEntity,
        n =>
    {
        var entity = (EntityBase)n.Entry.Entity;
        n.Entry.State = entity isNew
            ? EntityState.Added
            : entity.IsChanged
                ? EntityState.Modified
                : entity.IsDeleted
                    ? EntityState.Deleted
                    : EntityState.Unchanged;
    });
    context.SaveChanges();
}
```

Флаги отображаются только в составе сущности для простоты примера. Обычно флаги входят в состав объекта передачи данных или других состояний, включенных в запросе.

Установка явных значений для создаваемых свойств

02.09.2018 • 5 minutes to read • [Edit Online](#)

Создаваемое свойство — это свойство, значение которого создается (либо EF, либо базой данных), когда сущность добавляется или обновляется. Дополнительные сведения о создаваемых свойствах см. в [этой статье](#).

Могут возникать ситуации, когда вы хотите установить явное значение для созданного свойства, вместо того чтобы оно было генерировано.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Модель

Модель, используемая в этой статье, содержит единственную сущность `Employee`.

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public DateTime EmploymentStarted { get; set; }
    public int Salary { get; set; }
    public DateTime? LastPayRaise { get; set; }
}
```

Сохранение явного значения во время добавления

Свойство `Employee.EmploymentStarted` настроено принимать значения, сгенерированные базой данных для новых сущностей (с использованием значения по умолчанию).

```
modelBuilder.Entity<Employee>()
    .Property(b => b.EmploymentStarted)
    .HasDefaultValueSql("CONVERT(date, GETDATE())");
```

Следующий код вставляет записи о двух сотрудниках в базу данных.

- Для первого сотрудника значение свойству `Employee.EmploymentStarted` не присваивается, поэтому для `DateTime` остается установленным значение CLR по умолчанию.
- Для второго сотрудника мы установили явное значение `1-Jan-2000`.

```

using (var context = new EmployeeContext())
{
    context.Employees.Add(new Employee { Name = "John Doe" });
    context.Employees.Add(new Employee { Name = "Jane Doe", EmploymentStarted = new DateTime(2000, 1, 1) });
    context.SaveChanges();

    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name + ", " + employee.EmploymentStarted);
    }
}

```

В выходных данных показано, что база данных создала значение для первого сотрудника, а наше явное значение использовалось для второго.

```

1: John Doe, 1/26/2017 12:00:00 AM
2: Jane Doe, 1/1/2000 12:00:00 AM

```

Явные значения в столбцах IDENTITY SQL Server

По соглашению свойство `Employee.EmployeeId` — это столбец `IDENTITY`, сгенерированный хранилищем.

Для большинства ситуаций описанный выше подход будет работать для ключевых свойств. Однако, чтобы вставить явные значения в столбец SQL Server `IDENTITY`, вам необходимо вручную включить `IDENTITY_INSERT` перед вызовом `SaveChanges()`.

NOTE

В нашем журнале заказов есть [запрос на функцию](#), которая позволяет сделать это автоматически в пределах поставщика SQL Server.

```

using (var context = new EmployeeContext())
{
    context.Employees.Add(new Employee { EmployeeId = 100, Name = "John Doe" });
    context.Employees.Add(new Employee { EmployeeId = 101, Name = "Jane Doe" });

    context.Database.OpenConnection();
    try
    {
        context.Database.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.Employees ON");
        context.SaveChanges();
        context.Database.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.Employees OFF");
    }
    finally
    {
        context.Database.CloseConnection();
    }

    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name);
    }
}

```

Выходные данные показывают, что предоставленные идентификаторы были сохранены в базе данных.

```
100: John Doe  
101: Jane Doe
```

Установка явного значения во время обновления

Свойство `Employee.LastPayRaise` настроено принимать значения, сгенерированные базой данных во время обновления.

```
modelBuilder.Entity<Employee>()  
    .Property(b => b.LastPayRaise)  
    .ValueGeneratedOnAddOrUpdate();  
  
modelBuilder.Entity<Employee>()  
    .Property(b => b.LastPayRaise)  
    .Metadata.AfterSaveBehavior = PropertySaveBehavior.Ignore;
```

NOTE

По умолчанию EF Core создаст исключение, если вы попытаетесь сохранить явное значение для свойства, для которого настроено создание во время обновления. Чтобы этого избежать, вам нужно перейти к API метаданных нижнего уровня и установить `AfterSaveBehavior` (как показано выше).

NOTE

Изменения в EF Core 2.0. В предыдущих версиях поведение после сохранения контролировалось с помощью флага `IsReadOnlyAfterSave`. Этот флаг устарел и заменен `AfterSaveBehavior`.

Существует также триггер базы данных для генерации значений столбца `LastPayRaise` во время операций `UPDATE`.

```

CREATE TRIGGER [dbo].[Employees_UPDATE] ON [dbo].[Employees]
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF ((SELECT TRIGGER_NESTLEVEL()) > 1) RETURN;

    IF UPDATE(Salary) AND NOT Update(LastPayRaise)
    BEGIN
        DECLARE @Id INT
        DECLARE @OldSalary INT
        DECLARE @NewSalary INT

        SELECT @Id = INSERTED.EmployeeId, @NewSalary = Salary
        FROM INSERTED

        SELECT @OldSalary = Salary
        FROM deleted

        IF @NewSalary > @OldSalary
        BEGIN
            UPDATE dbo.Employees
            SET LastPayRaise = CONVERT(date, GETDATE())
            WHERE EmployeeId = @Id
        END
    END
END

```

Следующий код увеличивает зарплату двух сотрудников в базе данных.

- Для первого сотрудника значение свойству `Employee.LastPayRaise` не присваивается, поэтому оно остается нулевым.
- Для второго сотрудника мы установили явное значение "неделя назад" (до повышения зарплаты).

```

using (var context = new EmployeeContext())
{
    var john = context.Employees.Single(e => e.Name == "John Doe");
    john.Salary = 200;

    var jane = context.Employees.Single(e => e.Name == "Jane Doe");
    jane.Salary = 200;
    jane.LastPayRaise = DateTime.Today.AddDays(-7);

    context.SaveChanges();

    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name + ", " + employee.LastPayRaise);
    }
}

```

В выходных данных показано, что база данных создала значение для первого сотрудника, а наше явное значение использовалось для второго.

```

1: John Doe, 1/26/2017 12:00:00 AM
2: Jane Doe, 1/19/2017 12:00:00 AM

```

Реализации .NET, поддерживаемые EF Core

29.08.2018 • 3 minutes to read • [Edit Online](#)

Мы стремимся сделать EF Core доступным везде, где вы можете писать код .NET, и продолжаем работать в этом направлении. Поддержка EF Core в .NET Core и .NET Framework проверена в рамках автоматизированного тестирования и множества случаев успешного применения, однако в Mono, Xamarin и на универсальной платформе Windows по-прежнему присутствуют некоторые проблемы.

Обзор

В следующей таблице приводятся рекомендации для каждой реализации .NET:

РЕАЛИЗАЦИЯ .NET	STATUS	ТРЕБОВАНИЯ ДЛЯ EF CORE 1.X	ТРЕБОВАНИЯ ДЛЯ EF CORE 2.X ⁽¹⁾
.NET Core (ASP.NET Core, консоль и т. д.)	Полностью поддерживается и рекомендуется	Пакет SDK 1.x для .NET Core	Пакет SDK 2.x для .NET Core
.NET Framework (WinForms, WPF, ASP.NET, консоль и т. д.)	Полностью поддерживается и рекомендуется. Также доступна версия EF6 ⁽²⁾	.NET Framework 4.5.1	.NET Framework 4.6.1
Mono и Xamarin	В работе ⁽³⁾	Mono 4.6 Xamarin.iOS 10 Xamarin.Mac 3 Xamarin.Android 7	Mono 5.4 Xamarin.iOS 10.14 Xamarin.Mac 3.8 Xamarin.Android 7.5
Универсальная платформа Windows	Рекомендуется версия EF Core 2.0.1 ⁽⁴⁾	Пакет .NET Core UWP 5.x	Пакет .NET Core UWP 6.x

⁽¹⁾ Версия EF Core 2.0 ориентируется на платформу [.NET Standard 2.0](#) и поэтому нуждается в поддерживающих ее реализациях .NET.

⁽²⁾ Рекомендации по выбору правильной технологии см. в разделе [Сравнение EF Core и EF6](#).

⁽³⁾ В Xamarin присутствуют некоторые проблемы и ограничения, которые могут препятствовать правильной работе некоторых приложений, разработанных с использованием EF Core 2.0. Обходные решения см. в списке [активных проблем](#).

⁽⁴⁾ См. раздел [Универсальная платформа Windows](#) этой статьи.

Универсальная платформа Windows

В ранних версиях EF Core и .NET UWP присутствовали некоторые проблемы с совместимостью, особенно в отношении приложений, скомпилированных с использованием цепочки инструментов .NET Native. В новой версии универсальной платформы Windows .NET добавлена поддержка платформы .NET Standard 2.0 и содержит .NET Native 2.0, где исправлено большинство ранее выявленных проблем с совместимостью. Было проведено более тщательное тестирование EF Core 2.0.1 с универсальной платформой Windows, однако эти тесты не были автоматизированными.

При использовании EF Core в UWP:

- Чтобы оптимизировать производительность запросов, не используйте анонимные типы в запросах LINQ. Чтобы развернуть приложение UWP в магазине приложений, оно должно быть скомпилировано в .NET Native. Запросы с анонимными типами плохо работают в среде .NET Native.
- Чтобы оптимизировать производительность `SaveChanges()`, используйте `ChangeTrackingStrategy.ChangingAndChangedNotifications` и реализуйте `INotifyPropertyChanged`, `INotifyPropertyChanging` и `INotifyCollectionChanged` в типах сущностей.

Сообщить о проблеме

Если какое-либо сочетание работает неправильно, мы рекомендуем добавить новые вопросы в [средство отслеживания вопросов EF Core](#). Для проблем, относящихся к Xamarin, используйте средство отслеживания проблем [Xamarin.Android](#) или [Xamarin.iOS](#).

Поставщики баз данных

13.09.2018 • 7 minutes to read • [Edit Online](#)

Entity Framework Core поддерживает доступ к множеству разных баз данных с использованием библиотек подключаемых модулей, которые называются поставщиками баз данных.

Текущие поставщики

IMPORTANT

Поставщики EF Core поступают из самых разных источников. Не все поставщики разрабатываются в рамках проекта Entity Framework Core. Выбирая поставщика, обязательно оцените качество, лицензирование, поддержку и другие показатели на соответствие вашим требованиям. Также обязательно ознакомьтесь подробными сведениями о совместимости версий, представленными в документации по каждому поставщику.

ПАКЕТ NUGET	ПОДДЕРЖИВАЕМЫЕ ЯДРА СУБД	ПРОГРАММА ОБСЛУЖИВАНИЯ ИЛИ ПОСТАВЩИК	ПРИМЕЧАНИЯ И ТРЕБОВАНИЯ	ПОЛЕЗНЫЕ ССЫЛКИ
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2008 и выше	Проект EF Core (Майкрософт)		Документы
Microsoft.EntityFrameworkCore.Sqlite	SQLite 3.7 и выше	Проект EF Core (Майкрософт)		Документы
Microsoft.EntityFrameworkCore.InMemory	Выполняющаяся в памяти база данных EF Core	Проект EF Core (Майкрософт)	Только для тестирования	Документы
Npgsql.EntityFrameworkCore.PostgreSQL	PostgreSQL	Команда разработчиков Npgsql		Документы
Pomelo.EntityFrameworkCore.MySql	MySQL, MariaDB	Проект Pomelo Foundation		Файл сведений
Pomelo.EntityFrameworkCore.MyCat	Сервер MyCAT	Проект Pomelo Foundation	Предварительная версия до EF Core 1.1	Файл сведений
EntityFrameworkCore.SqlServerCompact40	SQL Server Compact 4.0	Эрик Эйлсков Йенсен (Erik Ejlskov Jensen)	.NET Framework	Вики-сайт
EntityFrameworkCore.SqlServerCompact35	SQL Server Compact 3,5	Эрик Эйлсков Йенсен (Erik Ejlskov Jensen)	.NET Framework	Вики-сайт
MySql.Data.EntityFrameworkCore	MySQL	Проект MySQL (Oracle)	Предварительная версия	Документы

ПАКЕТ NUGET	ПОДДЕРЖИВАЕМЫЕ ЯДРА СУБД	ПРОГРАММА ОБСЛУЖИВАНИЯ ИЛИ ПОСТАВЩИК	ПРИМЕЧАНИЯ И ТРЕБОВАНИЯ	ПОЛЕЗНЫЕ ССЫЛКИ
FirebirdSql.EntityFrameworkCore.Firebird	Firebird 2.5 и 3.x	Jiří Činčura	EF Core 2.0 и выше	Документы
EntityFrameworkCore.FirebirdSql	Firebird 2.5 и 3.x	Рафаэл Алмейда (Rafael Almeida)	EF Core 2.0 и выше	Вики-сайт
IBM.EntityFrameworkCore	Db2, Informix	IBM	Версия Windows	Блог
IBM.EntityFrameworkCore-Inx	Db2, Informix	IBM	Версия Linux	Блог
IBM.EntityFrameworkCore-osx	Db2, Informix	IBM	Версия macOS	Блог
Devart.Data.Oracle.EntityFrameworkCore	Oracle 9.2.0.4 и выше	DevArt	Оплаченный	Документы
Devart.Data.PostgreSql.EntityFrameworkCore	PostgreSQL 8.0 и выше	DevArt	Оплаченный	Документы
Devart.Data.SQLite.EntityFrameworkCore	SQLite 3 и выше	DevArt	Оплаченный	Документы
Devart.Data.MySql.EntityFrameworkCore	MySQL 5 и выше	DevArt	Оплаченный	Документы
EntityFrameworkCore.Jet	Файлы Microsoft Access	Bubi	EF Core 2.0, .NET Framework	Файл сведений

Будущие поставщики

Cosmos DB

Мы ведем разработку поставщика EF Core для API DocumentDB в Cosmos DB. Это будет первый разработанный нами полноценный поставщик базы данных, ориентированный на документы. Опираясь на полученный в ходе разработки опыт, мы будем вносить в проект дальнейшие изменения в версиях после 2.1. На данный момент планируется публикация ранней предварительной версии этого поставщика (2.1).

Oracle

Команда разработчиков Oracle .NET объявила о планах по выпуску основного поставщика для EF Core 2.0 приблизительно в третьем квартале 2018 года. Дополнительные сведения см. в соответствующем [заявлении о направлении разработки для .NET Core и Entity Framework Core](#). Направляйте любые вопросы об этом поставщике, включая сроки выпуска, на [веб-сайт сообщества Oracle](#).

Тем временем команда разработчиков EF представила [образец поставщика EF Core для баз данных Oracle](#). Этот проект не был предназначен для разработки поставщика EF Core, принадлежащего Майкрософт. Его целью было помочь нам выявить недостатки в реляционных и базовых функциях EF Core, которые необходимо устранить для улучшения поддержки Oracle, а также ускорить разработку других поставщиков Oracle для EF Core силами Oracle или других компаний.

Мы рассмотрим предложения по улучшению реализации этого образца. Мы также с радостью примем и поддержим усилия сообщества по созданию поставщика Oracle с открытым исходным кодом для EF Core на основе предлагаемого образца.

Добавление поставщика базы данных в приложение

Большинство поставщиков баз данных для EF Core распространяется в виде пакетов NuGet. Это значит, что для их установки можно использовать средство командной строки `dotnet`:

```
dotnet add package provider_package_name
```

В Visual Studio для этого также можно использовать консоль диспетчера пакетов NuGet:

```
install-package provider_package_name
```

После установки настройка поставщика осуществляется в `DbContext` с использованием либо метода `OnConfiguring`, либо метода `AddDbContext` (если применяется контейнер внедрения зависимостей).

Например, в следующей строке настраивается поставщик SQL Server с использованием переданной строки подключения:

```
optionsBuilder.UseSqlServer(  
    "Server=(localdb)\\mssqllocaldb;Database=MyDatabase;Trusted_Connection=True;");
```

Поставщики баз данных позволяют расширить возможности EF Core, реализуя уникальные функции для конкретных баз данных. Некоторые концепции являются общими для большинства баз данных и включены в основной набор компонентов EF Core. К ним относятся выражение запросов с помощью LINQ, транзакции и отслеживание изменений объектов при их загрузке из базы данных. Некоторые концепции характерны для определенного поставщика. Например, поставщик SQL Server позволяет [настроить таблицы, оптимизированные для памяти](#) (функция, относящаяся к SQL Server). Другие концепции характерны для класса поставщиков. Например, поставщики EF Core для реляционных баз данных основаны на общей библиотеке `Microsoft.EntityFrameworkCore.Relational`, которая предоставляет API для настройки сопоставлений столбцов и таблиц, ограничения внешнего ключа и т. п. Поставщики обычно распространяются в виде пакетов NuGet.

IMPORTANT

Выпускаемые исправления EF Core часто содержат обновления пакета `Microsoft.EntityFrameworkCore.Relational`. При добавлении поставщика реляционной базы данных этот пакет становится транзитивной зависимостью вашего приложения. Тем не менее многие поставщики выпускаются независимо от EF Core и могут не обновляться при выпуске новых исправлений этого пакета. Чтобы гарантировать исправление всех обнаруженных ошибок, рекомендуется добавлять исправления `Microsoft.EntityFrameworkCore.Relational` в приложение в виде прямых зависимостей.

Поставщик базы данных Microsoft SQL Server EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Этот поставщик базы данных позволяет использовать Entity Framework Core с Microsoft SQL Server (включая SQL Azure). Работы над этим поставщиком ведутся в рамках [проекта Entity Framework Core](#).

Установка

Установите [пакет NuGet Microsoft.EntityFrameworkCore.SqlServer](#).

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

Приступая к работе

Для начала работы с поставщиком используйте указанные ниже ресурсы.

- [Начало работы в .NET Framework \(консоль, WinForms, WPF и т. д.\)](#)
- [Начало работы в ASP.NET Core](#)
- [Пример приложения UnicornStore](#)

Поддерживаемые ядра СУБД

- Microsoft SQL Server (2008 и выше)

Поддерживаемые платформы

- .NET Framework (4.5.1 и выше)
- .NET Core
- Mono (4.2.0 и выше)

Caution: Using this provider on Mono will make use of the Mono SQL Client implementation, which has a number of known issues. For example, it does not support secure connections (SSL).

Оптимизированные для памяти таблицы поддерживают в поставщик базы данных SQL Server EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Эта возможность появилась в EF Core 1.1.

[Оптимизированные для памяти таблиц](#) — это компонент SQL Server, где вся таблица находится в памяти. Второй копии данных таблицы сохраняется на диске, но только с целью увеличения устойчивости. Данные в таблицах, оптимизированных для памяти толькочитываются с диска во время восстановления базы данных. Например после перезапуска сервера.

Настройка оптимизированной для памяти таблицы

Вы можете указать, что сопоставленная с таблицей сущность оптимизирована для памяти. Если вы используете EF Core для создания и обслуживания базы данных на основе используемой модели (с помощью миграций или `Database.EnsureCreated()`), то для таких сущностей будет создана оптимизированная для памяти таблица.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ForSqlServerIsMemoryOptimized();
}
```

Поставщик базы данных SQLite EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Этот поставщик базы данных позволяет использовать Entity Framework Core с SQLite. Работы над этим поставщиком ведутся в рамках [проекта Entity Framework Core](#).

Установка

Установите [пакет NuGet Microsoft.EntityFrameworkCore.Sqlite](#).

```
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

Приступая к работе

Для начала работы с поставщиком используйте указанные ниже ресурсы.

- [Локальный SQLite на UWP](#)
- [Приложение .NET Core для новой базы данных SQLite](#)
- [Пример приложения Unicorn Clicker](#)
- [Пример приложения Unicorn Packer](#)

Поддерживаемые ядра СУБД

- SQLite (3.7 и выше)

Поддерживаемые платформы

- .NET Framework (4.5.1 и выше)
- .NET Core
- Mono (4.2.0 и выше)
- Универсальная платформа Windows

Ограничения

Сведения о некоторых важных ограничениях поставщика SQLite см. в статье [Ограничения SQLite](#).

Ограничения функций поставщика базы данных SQLite EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Поставщик SQLite имеет ряд ограничений миграции. Большинство из этих ограничений представляют собой результат применения ограничения в ядро СУБД SQLite, а не только к EF.

Ограничения моделирования

Общей библиотекой реляционных (совместно используемые поставщиками реляционной базы данных Entity Framework) определяет API-интерфейсы для моделирования основные понятия, которые являются общими для большинства реляционных СУБД. Несколько из этих понятий, не поддерживаются поставщиком SQLite.

- Схемы
- Последовательности

Ограничения миграции

Ядро СУБД SQLite поддерживает ряд операций схемы, которые поддерживаются в большинстве других реляционных баз данных. При попытке применить одну из неподдерживаемых операций с базой данных SQLite то `NotSupportedException` будет создано.

ОПЕРАЦИЯ	ПОДДЕРЖИВАЕТСЯ?	ТРЕБУЕТСЯ ВЕРСИЯ
AddColumn	✓	1.0
AddForeignKey	✗	
AddPrimaryKey	✗	
AddUniqueConstraint	✗	
AlterColumn	✗	
CreateIndex	✓	1.0
CreateTable	✓	1.0
DropColumn	✗	
DropForeignKey	✗	
DropIndex	✓	1.0
DropPrimaryKey	✗	
DropTable	✓	1.0

ОПЕРАЦИЯ	ПОДДЕРЖИВАЕТСЯ?	ТРЕБУЕТСЯ ВЕРСИЯ
DropUniqueConstraint	✗	
RenameColumn	✗	
RenameIndex	✓	2.1
RenameTable	✓	1.0
EnsureSchema	✓ (нет-оп)	2.0
DropSchema	✓ (нет-оп)	2.0
Insert	✓	2.0
Обновление	✓	2.0
Удаление	✓	2.0

Инструкции по решению ограничения миграции

Вы можете устранить некоторые из этих ограничений, написав код в миграции для выполнения таблицы вручную перестроить. Перестройка таблицы включает в себя переименование существующей таблицы, создание новой таблицы, копирование данных в новую таблицу и удалить старую таблицу. Необходимо будет использовать `Sql(string)` метод для выполнения некоторых из этих действий.

См. в разделе [внесения других типов из таблицы изменений схемы](#) в SQLite документации для получения дополнительных сведений.

В будущем EF может поддерживать некоторые из этих операций, используя подход перестроение таблицы, на самом деле. Вы можете [отслеживать эту функцию на портале GitHub](#).

Поставщик выполняющейся в памяти базы данных EF Core

28.08.2018 • 2 minutes to read • [Edit Online](#)

Этот поставщик базы данных позволяет использовать Entity Framework Core с выполняющейся в памяти базой данных. Это может быть полезно для тестирования несмотря на то, что поставщик SQLite в режиме выполнения в памяти может в большей степени подходить для тестирования реляционных баз данных. Работы над этим поставщиком ведутся в рамках [проекта Entity Framework Core](#).

Установка

Установите [пакет NuGet Microsoft.EntityFrameworkCore.InMemory](#).

```
Install-Package Microsoft.EntityFrameworkCore.InMemory
```

Приступая к работе

Для начала работы с поставщиком используйте указанные ниже ресурсы.

- [Тестирование с использованием выполнения в памяти](#)
- [Тесты примера приложения UnicornStore](#)

Поддерживаемые ядра СУБД

- Встроенная выполняющаяся в памяти база (предназначена исключительно для тестирования)

Поддерживаемые платформы

- .NET Framework (4.5.1 и выше)
- .NET Core
- Mono (4.2.0 и выше)
- Универсальная платформа Windows

Разработка поставщика базы данных

29.08.2018 • 2 minutes to read • [Edit Online](#)

Сведения о написании поставщика базы данных Entity Framework Core, см. в разделе [требуется написать поставщика EF Core](#) по [Vickers Артур](#).

NOTE

Эти сообщения не были обновлены с момента EF Core 1.1 и с этого момента были сделаны существенные изменения [681 проблема](#) отслеживает изменения в эту документацию.

База кода EF Core имеет открытый исходный код и содержит несколько поставщиков базы данных, которые могут использоваться в качестве ссылки. Можно найти исходный код в <https://github.com/aspnet/EntityFrameworkCore>. Также это может быть полезно посмотреть на код для часто используемых сторонних поставщиков, таких как [Npgsql](#), [Pomelo MySQL](#), и [SQL Server Compact](#). В частности эти проекты, программу установки, чтобы расширить и запуск функциональных тестов, которые мы публикуем в NuGet. Настоятельно рекомендуется такого типа установки.

Обновление с изменениями поставщика

Начиная с рабочих после 2.1 выпуска, мы создали [журнала изменений](#), может потребоваться соответствующие изменения в код поставщика. Это призвано помочь при обновлении существующего поставщика для работы с новой версии EF Core.

Прежде чем 2.1, мы использовали `providers-beware` и `providers-fyi` меток на наши вопросы GitHub и запросов на Вытягивание с одинаковой целью. Мы выполним `continuie` использование этих меток по проблемам, для предоставления значение, указывающее какие рабочие элементы в определенном выпуске также может потребоваться должны быть выполнены в поставщиках. Объект `providers-beware` метки обычно означает, что реализации рабочего элемента может нарушить работу поставщиков, хотя `providers-fyi` метки обычно означает, что поставщики не будет нарушена, но код может понадобиться изменить в любом случае, например, чтобы задействовать новые функциональные возможности.

Предлагаемые именования сторонних поставщиков

Мы советуем использовать следующие имена пакетов NuGet. Это согласуется с именами пакетов, предоставленные группе EF Core.

`<Optional project/company name>.EntityFrameworkCore.<Database engine name>`

Пример:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Npgsql.EntityFrameworkCore.PostgreSQL`
- `EntityFrameworkCore.SqlServerCompact40`

Изменения, влияющие на поставщика

04.10.2018 • 5 minutes to read • [Edit Online](#)

Эта страница содержит ссылки на запросы, внесенные в репозитории EF Core, может потребовать авторы других поставщиков баз данных для реагирования на Вытягивание. Планируется представляют собой отправную точку для авторов существующей базы данных сторонних поставщиков, при обновлении до новой версии своего поставщика.

Этот журнал начинается с изменениями из 2.1 и 2.2. Прежде чем 2.1 мы использовали `providers-beware` и `providers-fyi` меток на наши проблемы и запросы на включение внесенных изменений.

2.1--> 2.2

Только для тестирования изменений

- <https://github.com/aspnet/EntityFrameworkCore/pull/12057> — Разрешите разделителей групп разрядов настраиваемый SQL в тестах
 - Проверить изменения, обеспечивающие нестрогом с плавающей запятой сравнения в `BuiltInDataTypesTestBase`
 - Изменения тестирования, обеспечивающие тестов запросов для многократного использования с помощью различных разделителей групп разрядов SQL
- <https://github.com/aspnet/EntityFrameworkCore/pull/12072> -Добавьте тесты `DbFunction` реляционных спецификации тесты
 - Таким образом, эти тесты могут выполняться для всех поставщиков базы данных
- <https://github.com/aspnet/EntityFrameworkCore/pull/12362> -Очистка тестовой Async
 - Удалить `Wait` вызовы, ненужные `async` и переименовать некоторые методы теста
- <https://github.com/aspnet/EntityFrameworkCore/pull/12666> -Унифицировать инфраструктуру ведения журнала тестирования
 - Добавлен `CreateListLoggerFactory` и удалены некоторые ранее инфраструктурой ведения журнала, потребуются поставщиков, использующих эти тесты для реагирования
- <https://github.com/aspnet/EntityFrameworkCore/pull/12500> — Запустите дополнительные тесты запрос синхронно и асинхронно
 - Имена тестов и факторинга изменилось, что подразумевает работу поставщиков с помощью этих тестов реагирования на них
- <https://github.com/aspnet/EntityFrameworkCore/pull/12766> — Переименование переходов в модели `ComplexNavigations`
 - Поставщики, с помощью этих тестов может потребоваться реагировать
- <https://github.com/aspnet/EntityFrameworkCore/pull/12141> — Изменяет контекст в пул, вместо того чтобы избавляться в функциональных тестов
 - Это изменение включает в себя оптимизация тестирования которого может потребоваться поставщиков реагирования на них

Изменения кода теста и продукта

- <https://github.com/aspnet/EntityFrameworkCore/pull/12109> -Консолидировать `RelationalTypeMapping.Clone` методы
 - Изменения в 2.1 `RelationalTypeMapping`, разрешенное для упрощения в производных классах. Мы не считают, что это критическим изменением для поставщиков, а поставщики можно воспользоваться преимуществами этого изменения, в их производные типы сопоставление классов.

- <https://github.com/aspnet/EntityFrameworkCore/pull/12069> -Тегами или именованные запросы
 - Добавляет инфраструктуру для добавления тегов запросов LINQ и необходимости эти теги отображаются в виде комментариев в код SQL. Это может потребовать поставщиков реагировать на создание кода SQL.
- <https://github.com/aspnet/EntityFrameworkCore/pull/13115> -Поддерживают Пространственные данные с помощью NTS
 - Позволяет преобразователи для регистрации за пределами поставщика сопоставления типов и членов
 - Поставщики должны вызывать `base.FindMapping()` `ITypeMappingSource` реализации для работы
 - Следуйте этому шаблону, чтобы добавить поддержку пространственных к поставщику, который остается согласованным на всех поставщиков.
- <https://github.com/aspnet/EntityFrameworkCore/pull/13199> — Добавление расширенных возможностей отладки для создания поставщика службы
 - Позволяет `DbContextOptionsExtensions` реализовать новый интерфейс, который может помочь пользователям понять, почему доступ к внутренней службе, повторного построения
- <https://github.com/aspnet/EntityFrameworkCore/pull/13289> — Добавляет `CanConnect` API для использования путем проверки работоспособности
 - Этот запрос на Вытягивание добавляет понятие `CanConnect` которого будет использоваться служба работоспособности ASP.NET Core проверяет, чтобы определить, доступна ли база данных. По умолчанию реляционных реализация просто вызывает `Exist`, но поставщики могут реализовывать другое значение при необходимости. Нереляционные поставщиков необходимо реализовать новый интерфейс API в порядке для проверки работоспособности, чтобы можно было использовать.
- <https://github.com/aspnet/EntityFrameworkCore/pull/13306> -Обновление базового `RelationalTypeMapping` не задать размер `SqlParameter`
 - Остановите задание размера по умолчанию, так как это может привести к усечению. Поставщики может потребоваться добавить свою собственную логику, если необходимо задать размер.
- <https://github.com/aspnet/EntityFrameworkCore/pull/13372> -RevEng: Всегда указывайте тип столбца для десятичных столбцов
 - Всегда можно настройте тип столбца для десятичных столбцов в шаблонном коде, а не настраивать в соответствии с соглашением.
 - Поставщики должны не требуется вносить изменения с их стороны.
- <https://github.com/aspnet/EntityFrameworkCore/pull/13469> — Добавляет `CaseExpression` для создания выражений регистр SQL

Инструменты и расширения EF Core

13.09.2018 • 6 minutes to read • [Edit Online](#)

Инструменты и расширения предоставляют дополнительные возможности для Entity Framework Core.

IMPORTANT

Расширения создаются с помощью различных источников и не разрабатываются в рамках проекта Entity Framework Core. Выбирая стороннее расширение, обязательно оцените качество, лицензирование, совместимость, поддержку и другие показатели на соответствие вашим требованиям.

Инструменты

LLBLGen Pro

LLBLGen Pro — решение для моделирования сущностей с поддержкой Entity Framework и Entity Framework Core. Оно позволяет легко определить модель сущности и сопоставить ее с базой данных с помощью подходов Database-First (сначала база данных) или Model-First (сначала модель), таким образом, вы сможете сразу приступить к написанию запросов.

[веб-сайт](#)

Devart Entity Developer

Entity Developer — мощный конструктор ORM для ADO.NET Entity Framework, NHibernate, LinqConnect, Telerik Data Access и LINQ to SQL. Используйте подходы Model-First (сначала модель) и Database-First (сначала база данных) для разработки модели ORM и создания для нее кода C# или Visual Basic .NET. В нем используются новые подходы для разработки моделей ORM, повышения производительности и упрощения разработки приложений баз данных.

[веб-сайт](#)

EF Core Power Tools

Расширение Visual Studio 2017+. Вы можете реконструировать классы DbContext и POCO из существующей базы данных или проекта базы данных SQL Server и визуализировать и проверить DbContext различными способами.

[Вики-сайт GitHub](#)

Редактор Visual Entity Framework

Расширение Visual Studio 2017, которое добавляет конструктор ORM для проектирования визуальных объектов на платформе Entity Framework 6, Core 2.0 и Core 2.1. Код создается с помощью шаблонов T4, поэтому его можно адаптировать к любым потребностям. Наследование, однонаправленные и двунаправленные ассоциации поддерживаются, как и перечисления и возможность цветового выделения классов и добавления текстовых блоков для объяснения потенциально сложных частей проекта.

[Marketplace](#)

Расширения

Microsoft.EntityFrameworkCore.AutoHistory

Подключаемый модуль Microsoft.EntityFrameworkCore для поддержки автоматической записи истории

изменения данных.

[Репозиторий GitHub](#)

Microsoft.EntityFrameworkCore.DynamicLinq

Расширения динамического Linq для Microsoft.EntityFrameworkCore, которые добавляют поддержку асинхронных запросов

[Репозиторий GitHub](#)

EFCore.Practices

Попытка объединить некоторые хорошие примеры использования в один API, поддерживающий тестирование, включая небольшую платформу для проверки на наличие запросов N+1.

[Репозиторий GitHub](#)

EFSecondLevelCache.Core

Кэширующая библиотека второго уровня. Кэширование второго уровня — это кэширование запросов. Результаты команд EF будут храниться в кэше, чтобы такие же команды EF получали данные из кэша, а не выполнялись в базе данных еще раз.

[Репозиторий GitHub](#)

Detached.EntityFrameworkCore

Загружает и сохраняет полные диаграммы отсекенных сущностей (сущность со своими дочерними сущностями и списками). Создан под впечатлением от расширения [GraphDiff](#). Идея также заключается в добавлении подключаемых модулей для упрощения некоторых повторяющихся задач, таких как аудит и разбиение на страницы.

[Репозиторий GitHub](#)

EntityFrameworkCore.PrimaryKey

Получает из любой сущности первичный ключ (включая составные ключи) как словарь.

[Репозиторий GitHub](#)

EntityFrameworkCore.Rx

Оболочки реактивных расширений для критически важных наблюдаемых сущностей Entity Framework.

[Репозиторий GitHub](#)

EntityFrameworkCore.Triggers

Добавление триггеров для сущностей с реагированием на события insert, update и delete. Существует три события для каждой операции: до, после и при сбое.

[Репозиторий GitHub](#)

EntityFrameworkCore.TypedOriginalValues

Получите типизированный доступ к OriginalValue свойств сущности. Поддерживаются простые и сложные свойства, навигации и коллекции не поддерживаются.

[Репозиторий GitHub](#)

Geco

Geco предоставляет генератор реконструированной модели с поддержкой преобразования во множественную или единичную форму и настраиваемых шаблонов на основе интерполированных строк C# 6.0, выполняющихся в .NET Core. Он также предоставляет генератор сценариев начальных значений со сценариями слияния SQL и запускатель сценариев.

[Репозиторий GitHub](#)

LinqKit.Microsoft.EntityFrameworkCore

LinqKit.Microsoft.EntityFrameworkCore — бесплатный набор расширений для опытных пользователей LINQ to SQL и EntityFrameworkCore. С поддержкой Include(...) и IDbAsync.

[Репозиторий GitHub](#)

NeinLinq.EntityFrameworkCore

NeinLinq.EntityFrameworkCore предоставляет полезные расширения для использования поставщиков LINQ, таких как Entity Framework, поддерживающие только небольшое подмножество функций .NET, повторно используемые функции, перезапись запросов (даже переделку их в безопасные для обработки NULL) и построение динамических запросов с помощью транслируемых предикатов и селекторов.

[Репозиторий GitHub](#)

Microsoft.EntityFrameworkCore.UnitOfWork

Подключаемый модуль Microsoft.EntityFrameworkCore для поддержки репозитория, шаблонов Unit of Work и нескольких баз данных с поддержкой распределенных транзакций.

[Репозиторий GitHub](#)

EntityFramework.LazyLoading

Отложенная загрузка для EF Core 1.1

[Репозиторий GitHub](#)

EFCore.BulkExtensions

Расширения EntityFrameworkCore для массовых операций (Insert, Update, Delete).

[Репозиторий GitHub](#)

Bricelam.EntityFrameworkCore.Pluralizer

Добавляет в EF Core преобразование во множественную форму для времени разработки.

[Репозиторий GitHub](#)

Справочник по инструментам Entity Framework Core

12.10.2018 • 2 minutes to read • [Edit Online](#)

Инструменты Entity Framework Core используются для задач разработки. Эти инструменты используются в основном для управления миграциями и формирования `DbContext` и типов сущностей путем реконструирования схемы базы данных.

- [Инструменты консоли диспетчера пакетов EF Core](#) запускаются в [консоли диспетчера пакетов](#) в Visual Studio. Эти инструменты совместимы с проектами .NET Framework и .NET Core.
- [Инструменты интерфейса командной строки EF Core .NET](#) представляют собой расширение для кроссплатформенных [инструментов .NET Core CLI](#). Этим инструментам нужен проект пакета SDK .NET Core (с `Sdk="Microsoft.NET.Sdk"` или аналогичным объектом в файле проекта).

Оба варианта инструментов предоставляют одинаковую функциональность. При разработке в Visual Studio мы рекомендуем использовать инструменты **консоли диспетчера пакетов**, так как они лучше интегрированы.

Следующие шаги

- [Справочник по инструментам консоли диспетчера пакетов EF Core](#)
- [Справочник по инструментам EF Core — .NET CLI](#)

Справочник по - консоль диспетчера пакетов в Visual Studio средствам Entity Framework Core

06.10.2018 • 14 minutes to read • [Edit Online](#)

Инструменты консоли диспетчера пакетов (PMC) для Entity Framework Core выполнения задач разработки во время разработки. Например, они создают [миграций](#), применить миграции и создать код для модели, основанные на существующей базы данных. Команды выполняются внутри Visual Studio с помощью [консоль диспетчера пакетов](#). Эти инструменты совместимы с проектами .NET Framework и .NET Core.

Если вы не используете Visual Studio, мы рекомендуем [инструменты командной строки EF Core](#) вместо этого. Средства CLI кросс платформенные и выполнения в командной строке.

Установка инструментов

Процедуры по установке и средства обновления отличаются между ASP.NET Core 2.1 или более поздней и более ранних версий или других типов проектов.

ASP.NET Core версии 2.1 и более поздние версии

Средства автоматически включаются в проекте ASP.NET Core 2.1 или более поздней, так как `Microsoft.EntityFrameworkCore.Tools` пакет включен в [метапакет Microsoft.AspNetCore.App](#).

Поэтому не нужно ничего делать, чтобы установить средства, но вам нужно:

- Восстановите пакеты перед использованием средства в новый проект.
- Установите пакет для обновления до более новой версии инструментов.

Чтобы убедиться в том, что вы получаете последнюю версию средства, рекомендуется также выполнить следующие действия:

- Изменить ваш `.csproj` файл и добавьте строку, указав последнюю версию `Microsoft.EntityFrameworkCore.Tools` пакета. Например `.csproj` файл может содержать `ItemGroup` , выглядящий следующим образом:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.1.3" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1" />
</ItemGroup>
```

Обновите инструменты при получении сообщения, как в следующем примере:

Версия средства EF Core «2.1.1-rtm-30846» старше, среды выполнения «2.1.3-rtm-32065». Обновите инструменты для новейших функций и исправления ошибок.

Чтобы обновить средства:

- Установите последний пакет SDK .NET Core.
- Обновление Visual Studio до последней версии.
- Изменить `.csproj` файл, теперь она содержит ссылки на последнюю версию пакета средств, пакет, как показано выше.

Другие версии и типы проектов

Установить средства консоль диспетчера пакетов, выполнив следующую команду в **консоль диспетчера пакетов**:

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Обновить инструменты, выполнив следующую команду в **консоль диспетчера пакетов**.

```
Update-Package Microsoft.EntityFrameworkCore.Tools
```

Проверка установки

Убедитесь, что установлены средства, выполнив следующую команду:

```
Get-Help about_EntityFrameworkCore
```

Результат выглядит следующим образом (она не сообщает, какие версии инструментов, вы используете):

```
          _/\_
         -==/   \
         |.    \|\
         |||||  || )  \\\
         |||||  \_ / //|\\
         |||||      / \\\|\

TOPIC
about_EntityFrameworkCore

SHORT DESCRIPTION
Provides information about the Entity Framework Core Package Manager Console Tools.

<A list of available commands follows, omitted here.>
```

С помощью средств

Перед использованием средства:

- Понимать разницу между целевой и запуска проекта.
- Узнайте, как использовать средства с помощью библиотеки классов .NET Standard.
- Для проектов ASP.NET Core настройте среду.

Целевой объект и запуска проекта

См. команды проекта и запускаемым проектом.

- *Проекта* также известен как *целевой проект* так как это где команды добавления или удаления файлов. По умолчанию **проект по умолчанию** на **консоль диспетчера пакетов** является целевого проекта. Можно указать другой проект в качестве целевого проекта с помощью `--project` параметр.
- *Запускаемым проектом* является тот, который построение и запуск средств. Средства должны выполнить код приложения во время разработки для получения сведений о проекте, например строку подключения базы данных и конфигурации модели. По умолчанию **запускаемым проектом в обозревателе решений** является запускаемым проектом. Можно указать другой проект запускаемым проектом с помощью `--startup-project` параметр.

Зачастую того же проекта запускаемого проекта и целевой проект. Типичный сценарий, где они находятся отдельные проекты — когда:

- В библиотеке классов .NET Core, EF Core контекст и классы сущностей.
- Консольное приложение .NET Core или веб-приложение ссылается на библиотеку классов.

Существует также возможность [поместите код миграции в библиотеку классов, отдельно от контекста EF Core](#).

Другие целевые платформы

Консоль диспетчера пакетов средств работы с проектами .NET Core или .NET Framework. Приложения, которые имеют модель EF Core в библиотеку классов .NET Standard может отсутствовать, .NET Core или .NET Framework проекта. Например это верно для приложений Xamarin и универсальной платформы Windows. В таком случае можно создать .NET Core или .NET Framework проекта консольного приложения, единственным назначением которого является в качестве запускаемого проекта для средств. Проект может быть пустым проектом с не содержащим реального кода — этот компонент необходим только для предоставления целевой объект для выполнения средства.

Почему является пустым проектом требуется? Как упоминалось ранее, средства нужно выполнять код приложения во время разработки. Чтобы сделать это, они должны использовать среду выполнения .NET Core или .NET Framework. Когда модель EF Core находится в проекте, ориентированном на .NET Core или .NET Framework, средства EF Core позаимствовать среды выполнения из проекта. Они не могут делать, если модель EF Core в библиотеку классов .NET Standard. .NET Standard не фактическую реализацию .NET; это спецификация набора API-интерфейсов, реализации .NET должны поддерживать. Поэтому .NET Standard недостаточно для инструментов EF Core для выполнения кода приложения. Фиктивный проект, созданный для использования в качестве запускаемого проекта предоставляет конкретную целевую платформу, в которой средства можно загружать библиотеки классов .NET Standard.

Среда ASP.NET Core

Чтобы указать среду для проектов ASP.NET Core, задайте **env:ASPNETCORE_ENVIRONMENT** перед выполнением команд.

Общие параметры

В следующей таблице показаны параметры, которые являются общими для всех команд EF Core:

ПАРАМЕТР	ОПИСАНИЕ:
-Контексте <строка >	<code>DbContext</code> Класс для использования. Имя класса, только из цифр или полное с пространствами имен. Если этот параметр опущен, EF Core находит класс контекста. Если имеется несколько классов контекста, этот параметр является обязательным.
— Проект <строка >	Целевой проект. Если этот параметр опущен, проект по умолчанию для консоль диспетчера пакетов используется в качестве целевого проекта.
-StartupProject <строка >	Запускаемый проект. Если этот параметр опущен, запускаемым проектом в свойства решения используется в качестве целевого проекта.
-Verbose	Показать подробные выходные данные.

Чтобы отобразить справочные сведения о команде, используйте PowerShell `Get-Help` команды.

TIP

Параметры контекста, проекта и StartupProject поддерживает расширение функций клавиши tab.

Add-Migration

Добавление новой миграции.

Параметры:

ПАРАМЕТР	ОПИСАНИЕ:
-Name <строка >	Имя миграции. Это является позиционным параметром и является обязательным.
-OutputDir <строка >	Каталог (и sub-namespace) для использования. Пути задаются относительно каталога проекта целевой объект. По умолчанию используется «Миграция».

DROP Database

Удаляет базу данных.

Параметры:

ПАРАМЕТР	ОПИСАНИЕ:
-WhatIf	Показать базу данных, которая будет удалена, но не удаляйте ее.

Get-DbContext

Выводит список доступных `DbContext` типов.

Remove-Migration

Удаляет последнюю миграцию (выполняется откат изменений кода, которые были выполнены для миграции).

Параметры:

ПАРАМЕТР	ОПИСАНИЕ:
-Force	Миграции отменить (откатить изменения, которые были применены к базе данных).

Scaffold-DbContext

Создает код для `DbContext` и типы сущностей для базы данных. Чтобы `Scaffold-DbContext` для создания типа сущности, в таблице базы данных должна иметь первичный ключ.

Параметры:

ПАРАМЕТР	ОПИСАНИЕ:
-Connection <строка >	Строка подключения к базе данных. Для проектов ASP.NET Core 2.x, значение может быть <i>имя =<имя строки подключения ></i> . В этом случае имя берется из источников конфигурации, которые были настроены для проекта. Это является позиционным параметром и является обязательным.
-Поставщик <строка >	Используемый поставщик. Обычно это имя пакета NuGet, например: <code>Microsoft.EntityFrameworkCore.SqlServer</code> . Это является позиционным параметром и является обязательным.
-OutputDir <строка >	Чтобы поместить файлы каталог. Пути задаются относительно каталога проекта.
-ContextDir <строка >	Каталог для размещения <code>DbContext</code> в файл. Пути задаются относительно каталога проекта.
-Контексте <строка >	Имя <code>DbContext</code> класса, необходимо создать.
-Схемы <String [] >	Схемы таблиц для создания типов сущности для. Если этот параметр опущен, включаются все схемы.
-Таблицы <String [] >	Для создания типов сущности для таблиц. Если этот параметр опущен, включаются все таблицы.
-DataAnnotations	Атрибуты можно использовать для настройки модели (где это возможно). Если этот параметр опущен, используется только текущего API.
-UseDatabaseNames	Используйте имена таблиц и столбцов, так же, как в базе данных. Если этот параметр опущен, имена баз данных меняется на более точно соответствовать соглашениям C# имени стиля.
-Force	Перезаписывайте существующие файлы.

Пример

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Пример, который формирует шаблоны только указанных таблиц и создает контекст в отдельную папку с указанным именем.

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables "Blog","Post" -ContextDir Context -Context  
BlogContext
```

Миграция скриптов

Формирует скрипт SQL, который применяет все изменения из одной выбранной миграции к другой выбранной миграции.

Параметры:

ПАРАМЕТР	ОПИСАНИЕ:
-Из <строка >	Начала миграции. Миграция может быть идентифицирован по имени или по идентификатору. Число 0 является особым случаем, это значит, что <i>перед первой миграции</i> . По умолчанию равно 0.
- <Строка >	Окончания миграции. По умолчанию к последней миграции.
-Идемпотентными	Создайте скрипт, который может использоваться в любой базе данных с любой миграции.
-Выходной <строка >	Файл для записи результата. Если этот параметр указан, файл создается с созданным именем в той же папке при создании файлов среди выполнения приложения, например: /obj/Debug/netcoreapp2.1/ghbkztfz.sql.

TIP

To, From, и выходные параметры поддерживают расширение функций клавиши tab.

В следующем примере создается скрипт для миграции это InitialCreate, используя имя миграции.

```
Script-Migration -To InitialCreate
```

В следующем примере создается скрипт для всех операций миграции после миграции это InitialCreate, используя идентификатор миграции.

```
Script-Migration -From 20180904195021_InitialCreate
```

Обновления базы данных

Обновляет базу данных, последнюю миграцию или указанного миграции.

ПАРАМЕТР	ОПИСАНИЕ:
-Миграции <строка >	Целевой объект миграции. Миграция может быть идентифицирован по имени или по идентификатору. Число 0 является особым случаем, это значит, что <i>перед первой миграции</i> и вызывает всех операций переноса для отмены изменений. Если перенос не указан, команда по умолчанию к последней миграции.

TIP

Параметр миграции поддерживает расширение функций клавиши tab.

Следующий пример возвращает все миграции.

```
Update-Database -Migration 0
```

Следующие примеры обновления базы данных для указанного миграции. Первый использует имя миграции, а второй использует идентификатор миграции:

```
Update-Database -Migration InitialCreate  
Update-Database -Migration 20180904195021_InitialCreate
```

Справочник по — .NET CLI средствам Entity Framework Core

06.10.2018 • 15 minutes to read • [Edit Online](#)

Средства интерфейса командной строки (CLI) для Entity Framework Core выполнения задач разработки во время разработки. Например, они создают [миграций](#), применить миграции и создать код для модели, основанные на существующей базы данных. Команды представляют собой расширение для кроссплатформенного [dotnet](#) команду, которая входит в состав из [пакет SDK для .NET Core](#). Эти средства работы с проектами .NET Core.

Если вы используете Visual Studio, мы рекомендуем [инструменты консоли диспетчера пакетов](#) вместо:

- Они автоматически взаимодействуют с текущего проекта, выбранного в **консоль диспетчера пакетов** без необходимости вручную переключить каталоги.
- Они автоматически открывают файлы, созданные с помощью команды после выполнения команды.

Установка инструментов

Процедура установки зависит от типа проекта и версии:

- ASP.NET Core версии 2.1 и более поздние версии
- EF Core 2.x
- EF Core 1.x

ASP.NET Core 2.1 или более поздней

- Установка текущего [пакет SDK для .NET Core](#). Это нужно сделать, даже если у вас установлена последняя версия Visual Studio 2017.

Это все, что необходимо для ASP.NET Core 2.1 или более поздней, поскольку

`Microsoft.EntityFrameworkCore.Design` пакет включен в [метапакет Microsoft.AspNetCore.App](#).

EF Core 2.x (не в ASP.NET Core)

`dotnet ef` Они включены в пакет SDK для .NET Core, но для включения команд необходимо установить `Microsoft.EntityFrameworkCore.Design` пакета.

- Установка текущего [пакет SDK для .NET Core](#). Это нужно сделать, даже если у вас установлена последняя версия Visual Studio 2017.
- Установите последнюю стабильную `Microsoft.EntityFrameworkCore.Design` пакета.

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

EF Core 1.x

- Установка пакета SDK .NET Core версии 2.1.200. Более поздние версии не совместимы с помощью средств CLI для EF Core 1.0 и 1.1.
- Настройка приложения для использования 2.1.200 версию пакета SDK, изменив его `global.json` файла. Этот файл обычно включаются в каталоге решения (по одному выше проекта).
- Измените файл проекта и добавьте `Microsoft.EntityFrameworkCore.Tools.DotNet` как

`DotNetCliToolReference` элемента. Укажите последнюю версию 1.x, например: 1.1.6. См. в примере файла проекта в конце этого раздела.

- Установка последней версии 1.x `Microsoft.EntityFrameworkCore.Design` пакета, например:

```
dotnet add package Microsoft.EntityFrameworkCore.Design -v 1.1.6
```

С помощью ссылки на пакеты оба добавлены файле проекта выглядит примерно следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="1.1.6"
      PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="1.1.6" />
  </ItemGroup>
</Project>
```

Ссылку на пакет с `PrivateAssets="All"` не предоставляется для проектов, которые ссылаются на этот проект. Это ограничение особенно полезно для пакетов, которые обычно используются только во время разработки.

Проверка установки

Выполните следующие команды, чтобы проверить, правильно установлены средства интерфейса командной строки EF Core:

```
dotnet restore
dotnet ef
```

Результатом выполнения команды определяет версию средства в используйте:

```

 _/\_
---=/ \ \
 __ __ |. \ \
|_||_|_|_) \ \
|_|_|_| \/_ //|\ \
|__||__| / \\\/\ \

```

```
Entity Framework Core .NET Command-line Tools 2.1.3-rtm-32065
<Usage documentation follows, not shown.>
```

С помощью средств

Перед использованием средств, может потребоваться создать запускаемый проект, или установите среду.

Целевой и запускаемый проект

См. команды проекта и запускаемым проектом.

- Проекта также известен как *целевой проект* так как это где команды добавления или удаления файлов. По умолчанию проект в текущем каталоге является целевого проекта. Можно указать другой проект в качестве целевого проекта с помощью `--project` параметр.
- *Запускаемым проектом* является тот, который построение и запуск средств. Средства должны выполнить код приложения во время разработки для получения сведений о проекте, например строку подключения базы данных и конфигурации модели. По умолчанию проект в текущем каталоге является запускаемым проектом. Можно указать другой проект запускаемым проектом с помощью `--startup-project` параметр.

Зачастую того же проекта запускаемого проекта и целевой проект. Типичный сценарий, где они находятся отдельные проекты — когда:

- В библиотеке классов .NET Core, EF Core контекст и классы сущностей.
- Консольное приложение .NET Core или веб-приложение ссылается на библиотеку классов.

Существует также возможность [поместите код миграции в библиотеку классов, отдельно от контекста EF Core](#).

Другие целевые платформы

Средства CLI работать с проектам .NET Core и .NET Framework. Приложения, которые имеют модель EF Core в библиотеку классов .NET Standard может отсутствовать, .NET Core или .NET Framework проекта. Например это верно для приложений Xamarin и универсальной платформы Windows. В таком случае можно создать проект консольного приложения .NET Core, единственным назначением которого является в качестве запускаемого проекта для средств. Проект может быть пустым проектом с не содержащим реального кода — этот компонент необходим только для предоставления целевой объект для выполнения средства.

Почему является пустым проектом требуется? Как упоминалось ранее, средства нужно выполнять код приложения во время разработки. Для этого им необходимо использовать среду выполнения .NET Core. Когда модель EF Core находится в проекте, ориентированном на .NET Core или .NET Framework, средства EF Core позаимствовать среды выполнения из проекта. Они не могут делать, если модель EF Core в библиотеку классов .NET Standard. .NET Standard не фактическую реализацию .NET; это спецификация набора API-интерфейсов, реализации .NET должны поддерживать. Поэтому .NET Standard недостаточно для инструментов EF Core для выполнения кода приложения. Фиктивный проект, созданный для использования в качестве запускаемого проекта предоставляет конкретную целевую платформу, в которой средства можно загружать библиотеки классов .NET Standard.

Среда ASP.NET Core

Чтобы указать среду для проектов ASP.NET Core, задайте **ASPNETCORE_ENVIRONMENT** переменной среды перед запуском команды.

Общие параметры

	ПАРАМЕТР	ОПИСАНИЕ:
	<code>--json</code>	Показать выходные данные JSON.
<code>-c</code>	<code>--context <DBCONTEXT></code>	<code>DbContext</code> Класс для использования. Имя класса, только из цифр или полное с пространствами имен. Если этот параметр опущен, найти класс контекста EF Core. Если имеется несколько классов контекста, этот параметр является обязательным.

	ПАРАМЕТР	ОПИСАНИЕ:
<code>-p</code>	<code>--project <PROJECT></code>	Относительный путь к папке проекта целевого проекта. Значение по умолчанию является текущая папка.
<code>-s</code>	<code>--startup-project <PROJECT></code>	Относительный путь к папке проекта, запускаемого проекта. Значение по умолчанию является текущая папка.
	<code>--framework <FRAMEWORK></code>	Моникера целевой платформы для требуемой версии .NET framework. Используется, если в файле проекта указываются нескольких целевых платформ, и вы хотите выбрать один из них.
	<code>--configuration <CONFIGURATION></code>	Конфигурацию сборки, например: <code>Debug</code> или <code>Release</code> .
	<code>--runtime <IDENTIFIER></code>	Идентификатор целевой среды выполнения для восстановления пакетов. Список идентификаторов сред выполнения (RID) см. в каталоге RID .
<code>-h</code>	<code>--help</code>	Показать справочные сведения.
<code>-v</code>	<code>--verbose</code>	Показать подробные выходные данные.
	<code>--no-color</code>	Не необходимо выделить цветом выходные данные.
	<code>--prefix-output</code>	Префикс выходных данных с уровнем.

Удаление базы данных ef DotNet

Удаляет базу данных.

Параметры:

	ПАРАМЕТР	ОПИСАНИЕ:
<code>-f</code>	<code>--force</code>	Не подтверждено.
	<code>--dry-run</code>	Показать базу данных, которая будет удалена, но не удаляйте ее.

Обновление базы данных ef DotNet

Обновляет базу данных, последнюю миграцию или указанного миграции.

Аргументы:

АРГУМЕНТ	ОПИСАНИЕ:
<MIGRATION>	Целевой объект миграции. Миграция может быть идентифицирован по имени или по идентификатору. Число 0 является особым случаем, это значит, что <i>перед первой миграции</i> и вызывает всех операций переноса для отмены изменений. Если перенос не указан, команда по умолчанию к последней миграции.

Следующие примеры обновления базы данных для указанного миграции. Первый использует имя миграции, а второй использует идентификатор миграции:

```
dotnet ef database update InitialCreate
dotnet ef database update 20180904195021_InitialCreate
```

сведения о dbcontext ef DotNet

Получает сведения о `DbContext` типа.

DotNet ef dbcontext списка

Выводит список доступных `DbContext` типов.

DotNet ef dbcontext каркаса

Создает код для `DbContext` и типы сущностей для базы данных. Чтобы эту команду, чтобы создать тип сущности в таблице базы данных должен иметь первичный ключ.

Аргументы:

АРГУМЕНТ	ОПИСАНИЕ:
<CONNECTION>	Строка подключения к базе данных. Для проектов ASP.NET Core 2.x, значение может быть <code>имя =<имя строки подключения></code> . В этом случае имя берется из источников конфигурации, которые были настроены для проекта.
<PROVIDER>	Используемый поставщик. Обычно это имя пакета NuGet, например: <code>Microsoft.EntityFrameworkCore.SqlServer</code> .

Параметры:

	ПАРАМЕТР	ОПИСАНИЕ:
-d	<code>--data-annotations</code>	Атрибуты можно использовать для настройки модели (где это возможно). Если этот параметр опущен, используется только текущего API.
-c	<code>--context <NAME></code>	Имя <code>DbContext</code> класса, необходимо создать.

ПАРАМЕТР	ОПИСАНИЕ:
	--context-dir <PATH> Каталог для размещения <code>DbContext</code> файла класса. Пути задаются относительно каталога проекта. Пространства имен являются производными от ее имени.
-f	--force Перезаписывайте существующие файлы.
-o	--output-dir <PATH> Чтобы поместить файлы классов сущностей в каталог. Пути задаются относительно каталога проекта.
	--schema <SCHEMA_NAME>... Схемы таблиц для создания типов сущности для. Чтобы указать несколько схем, повторите <code>--schema</code> для каждого из них. Если этот параметр указан, включаются все схемы.
-t	--table <TABLE_NAME> ... Для создания типов сущности для таблиц. Чтобы указать несколько таблиц, повторите <code>-t</code> или <code>--table</code> для каждого из них. Если этот параметр указан, включаются все таблицы.
	--use-database-names Используйте имена таблиц и столбцов, так же, как в базе данных. Если этот параметр задан, имена баз данных меняются на более точно соответствовать соглашениям C# имён стиля.

В следующем примере, формирует шаблоны все схемы и таблицы и помещает новые файлы в *моделей* папки.

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -o Models
```

В следующем примере формирует шаблоны только указанных таблиц и создает контекст в отдельную папку с указанным именем:

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -o Models -t Blog Post --context-dir Context -c BlogContext
```

Добавление DotNet ef migrations

Добавление новой миграции.

Аргументы:

АРГУМЕНТ	ОПИСАНИЕ:
<NAME>	Имя миграции.

Параметры:

	ПАРАМЕТР	ОПИСАНИЕ:
-o	--output-dir <PATH>	Каталог (и sub-namespace) для использования. Пути задаются относительно каталога проекта. По умолчанию используется «Миграция».

DotNet ef migrations списка

Список доступных миграций.

удалить DotNet ef migrations

Удаляет последнюю миграцию (выполняется откат изменений кода, которые были выполнены для миграции).

Параметры:

	ПАРАМЕТР	ОПИСАНИЕ:
-f	--force	Миграции отменить (откатить изменения, которые были применены к базе данных).

DotNet ef migrations сценария

Формирует скрипт SQL из миграции.

Аргументы:

АРГУМЕНТ	ОПИСАНИЕ:
<FROM>	Начала миграции. Миграция может быть идентифицирован по имени или по идентификатору. Число 0 является особым случаем, это значит, что <i>перед первой миграции</i> . По умолчанию равно 0.
<TO>	Окончания миграции. По умолчанию к последней миграции.

Параметры:

	ПАРАМЕТР	ОПИСАНИЕ:
-o	--output <FILE>	Файл, чтобы написать сценарий для.

ПАРАМЕТР	ОПИСАНИЕ:
-i	--idempotent Создайте скрипт, который может использоваться в любой базе данных с любой миграции.

В следующем примере создается скрипт для миграции это InitialCreate:

```
dotnet ef migrations script 0 InitialCreate
```

В следующем примере создается скрипт для всех операций миграции после миграции это InitialCreate.

```
dotnet ef migrations script 20180904195021_InitialCreate
```

Создание DbContext во время разработки

27.09.2018 • 4 minutes to read • [Edit Online](#)

Некоторые команды Инструменты EF Core (например, [миграций](#) команды) требуют производный `DbContext` экземпляра во время разработки для сбора сведений о приложении типы сущностей и их сопоставлении схемы базы данных. В большинстве случаев желательно, `DbContext` тем самым созданный настраивается в том, как было бы так же, как [настроен во время выполнения](#).

Существует несколько способов, попробуйте создать средства `DbContext`:

Из служб приложений

Если ваш запускаемый проект приложения ASP.NET Core, инструменты попытаются получить объект `DbContext` у поставщика услуг приложения.

Средства сначала пытаются получить доступ к службе, вызвав `Program.BuildWebHost()` и доступ к `IWebHost.Services` свойство.

NOTE

При создании нового приложения ASP.NET Core 2.0, этот обработчик включается по умолчанию. В предыдущих версиях EF Core и ASP.NET Core, инструменты попытке вызвать `Startup.ConfigureServices` напрямую, чтобы получить поставщик службы приложения, но этот шаблон больше не работает правильно в приложениях ASP.NET Core 2.0. Если вы обновляете приложение ASP.NET Core 1.x на 2.0, вы можете [изменить ваш Program класс стоит следовать этому шаблону новый](#).

`DbContext` Сам и все зависимости в своем конструкторе должны быть зарегистрированы как службы в поставщике служб приложения. Это легко достигается за счет [конструктор в DbContext, принимающий экземпляр DbContextOptions<TContext> как аргумент](#) и с помощью `AddDbContext<TContext>` метод.

С помощью конструктора без параметров

`DbContext` не может быть получен от поставщика службы приложения, средства поиска производные `DbContext` тип внутри проекта. Затем они пытаются создать экземпляр с помощью конструктора без параметров. Это может быть конструктор по умолчанию, если `DbContext` настраивается с помощью `OnConfiguring` метод.

Из фабрики во время разработки

Вы также можете перенаправить средства Создание `DbContext`, реализовав `IDesignTimeDbContextFactory<TContext>` интерфейса: Если класс, реализующий этот интерфейс находится в любом проекте, производные `DbContext` или в проекте запуска приложения, средства обхода другие способы вместо создания `DbContext` и использование разработки фабрики.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace MyProject
{
    public class BloggingContextFactory : IDesignTimeDbContextFactory<BloggingContext>
    {
        public BloggingContext CreateDbContext(string[] args)
        {
            var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
            optionsBuilder.UseSqlite("Data Source=blog.db");

            return new BloggingContext(optionsBuilder.Options);
        }
    }
}
```

NOTE

`args` Параметра, используемого в данный момент. Существует [проблема](#) отслеживания возможность указывать аргументы во время разработки с помощью средств.

Во время разработки фабрика может быть особенно полезно в том случае, если вам нужно по-разному настроить `DbContext` для времени разработки, чем во время выполнения, если `DbContext` конструктор принимает дополнительные параметры не зарегистрированы в функции внедрения Зависимостей, если вы не используете DI вообще или, если для некоторых -либо причине вы не хотите иметь `BuildWebHost` метод в приложении ASP.NET Core `Main` класса.

Службы времени разработки

28.08.2018 • 2 minutes to read • [Edit Online](#)

Некоторые службы, используемые средствами используются только во время разработки. Эти службы управляются отдельно от служб среды выполнения EF Core, чтобы предотвратить их развертывание вместе с вашим приложением. Чтобы переопределить одну из этих служб (например службы, чтобы создать файлы миграции), добавьте реализацию `IDesignTimeServices` для запускаемого проекта.

```
class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
        => services.AddSingleton<IMigrationsCodeGenerator, MyMigrationsCodeGenerator>()
}
```

Строки подключения

28.08.2018 • 3 minutes to read • [Edit Online](#)

Большинство поставщиков баз данных требуется какая-либо разновидность строки подключения для подключения к базе данных. Иногда эта строка подключения содержит конфиденциальные сведения, которые необходимо защитить. Кроме того, может потребоваться изменить строку подключения, при перемещении приложения между средами, например, разработки, тестирования и эксплуатации.

Приложения .NET framework

Приложения .NET framework, такие как WinForms, WPF, консоли и ASP.NET 4, имеют строковый шаблон с помощью проверенная подключения. Стока подключения должны добавляться в файл App.config приложения (Web.config при использовании ASP.NET). Если строка подключения содержит конфиденциальные сведения, такие как имя пользователя и пароль, чтобы защитить содержимое файла конфигурации с помощью [защищенной конфигурации](#).

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

    <connectionStrings>
        <add name="BloggingDatabase"
            connectionString="Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;" />
    </connectionStrings>
</configuration>
```

TIP

`providerName` Параметр не является обязательным для строки подключения EF Core, хранятся в файле App.config, так как поставщик базы данных настраивается с помощью кода.

Вы сможете прочесть строку соединения с помощью `ConfigurationManager` API в вашем контексте `OnConfiguring` метод. Может потребоваться добавить ссылку на `System.Configuration` framework сборки, чтобы иметь возможность использовать этот API.

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {

        optionsBuilder.UseSqlServer(ConfigurationManager.ConnectionStrings["BloggingDatabase"].ConnectionString);
    }
}
```

Универсальная платформа Windows (UWP)

Строки подключения в приложении универсальной платформы Windows обычно являются SQLite подключение, которое просто задает имя локального файла. Обычно они не содержат конфиденциальной информации и не нужно изменить, так как приложение развертывается. Таким образом эти строки

подключения являются обычно нормально, которое должно остаться в коде, как показано ниже. Если вы хотите переместить их за пределы кода универсальной платформы Windows поддерживает концепцию параметров, см. в разделе [разделе параметров приложения UWP](#) сведения.

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blogging.db");
    }
}
```

ASP.NET Core

В ASP.NET Core в системе конфигурации является очень гибкой, а строка подключения, которые могут храниться в `appsettings.json`, переменную среды, хранилище секретов пользователя или другого источника конфигурации. См. в разделе [раздел конфигурации из документации по ASP.NET Core](#) для получения дополнительных сведений. В следующем примере показано строку подключения, сохраненную в `appsettings.json`.

```
{
  "ConnectionStrings": {
    "BloggingDatabase": "Server=
(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
  },
}
```

Контекст обычно настраивается в `Startup.cs` строкой подключения, считываемого из конфигурации.

Примечание `GetConnectionString()` метод ищет значение конфигурации, ключ которого

`ConnectionStrings:<connection string name>`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

Ведение журнала

28.08.2018 • 2 minutes to read • [Edit Online](#)

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

Приложения ASP.NET Core

EF Core автоматически интегрируется с механизмы ведения журнала ASP.NET Core всякий раз, когда `AddDbContext` или `AddDbContextPool` используется. Таким образом, при использовании ASP.NET Core, ведение журнала следует настроить как описано в разделе [документации по ASP.NET Core](#).

Другие приложения

Ведение журнала в данный момент EF Core требуется `ILoggerFactory`, который сам настроены `ILoggerProvider` один или несколько. Общих поставщиков поставляются в следующие пакеты:

- [Microsoft.Extensions.Logging.Console](#): простое консольное средство ведения журнала.
- [Microsoft.Extensions.Logging.AzureAppServices](#): службы приложений Azure поддерживает «Журналы диагностики» и «Вход потока» компоненты.
- [Microsoft.Extensions.Logging.Debug](#): журналы, чтобы отладчик монитор, используя `System.Diagnostics.Debug.WriteLine()`.
- [Microsoft.Extensions.Logging.EventLog](#): записывает журнал событий Windows.
- [Microsoft.Extensions.Logging.EventSource](#): поддерживает `EventSource/EventListener`.
- [Microsoft.Extensions.Logging.TraceSource](#): журналы, чтобы прослушиватель трассировки с помощью `System.Diagnostics.TraceSource.TraceEvent()`.

После установки соответствующих пакетов, в приложении необходимо создать одноэлементный/глобальный экземпляр `LoggerFactory`. Например используя средство ведения журнала консоли:

```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[] {new ConsoleLoggerProvider(_,_ => true, true)});
```

Этот одноэлементный или глобальный экземпляр затем должен быть зарегистрирован в EF Core на `DbContextOptionsBuilder`. Пример:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLoggerFactory(MyLoggerFactory) // Warning: Do not create a new ILoggerFactory instance each time
        .UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=EFLogging;Trusted_Connection=True;ConnectRetryCount=0");
```

WARNING

Очень важно, приложения не создают новый экземпляр `ILoggerFactory` для каждого экземпляра контекста. Это приведет к утечке памяти и снижению производительности.

Фильтрация, регистрируемых

Для фильтрации, регистрируемых проще всего настроить его при регистрации `ILoggerProvider`. Пример:

```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[]
    {
        new ConsoleLoggerProvider((category, level)
            => category == DbLoggerCategory.Database.Command.Name
            && level == LogLevel.Information, true)
    });
}
```

В этом примере для возвращения только сообщений фильтруется журнала:

- в категории «`Microsoft.EntityFrameworkCore.Database.Command`»
- на уровне «Сведения»

Категории средства ведения журнала для EF Core, определенных в `DbLoggerCategory` класса, чтобы его легко найти в категории, но эти разрешения в простых строк.

Дополнительные сведения о базовой инфраструктуре ведения журнала можно найти в [документации ведения журнала ASP.NET Core](#).

Устойчивость подключений

27.09.2018 • 8 minutes to read • [Edit Online](#)

Устойчивость подключения автоматически осуществляет новую попытку команды поврежденной базы данных. Эта функция может использоваться с любой базой данных, указав «стратегия выполнения», который инкапсулирует логику, необходимую для обнаружения сбоев и повторите команды. Поставщики EF Core может предоставлять адаптированные для их условия сбоя конкретной базы данных и политики повтора оптимальной стратегии выполнения.

Например поставщик SQL Server включает в себя стратегии выполнения, специально предназначенную для SQL Server (включая SQL Azure). Он учитывает типы исключений, которые могут быть повторены и содержит допустимые значения по умолчанию максимальное число повторных попыток, задержку между повторными попытками и т. д.

Стратегия выполнения указывается при настройке параметров для текущего контекста. Это обычно находится в `OnConfiguring` метода производного контекста, или в `Startup.cs` для приложения ASP.NET Core.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=
(localdb)\mssqllocaldb;Database=EFMiscellaneous.ConnectionResiliency;Trusted_Connection=True;ConnectRetryCount=0",
            options => options.EnableRetryOnFailure());
}
```

Стратегия выполнения пользовательских

Нет механизма регистрации стратегию выполнения пользовательских своим собственным в том случае, если вы хотите изменить любые параметры по умолчанию.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseMyProvider(
            "<connection string>",
            options => options.ExecutionStrategy(...));
}
```

Стратегии выполнения и транзакции

Стратегия выполнения, которая автоматически осуществляет новую попытку в случае сбоев необходимо иметь возможность воспроизвести каждой операции в блоке повторных попыток, которое не удается. После включения повторных попыток, каждой операции, выполняемые с помощью EF Core становится отдельной повторяемые операции. То есть каждый запрос и каждый вызов `SaveChanges()` будет повторена как единое целое, при возникновении временного сбоя.

Тем не менее если код запускает транзакцию с помощью `BeginTransaction()` вы определяете собственную группу операций, которые должны рассматриваться как единое целое, и все содержимое транзакции необходимо воспроизвести должен произойти сбой. Если попытаться сделать это, при использовании стратегии выполнения, вы получите исключение, как показано ниже:

InvalidOperationException: Настроенная стратегия выполнения 'SqlServerRetryingExecutionStrategy' не поддерживает запуск транзакций пользователем. Используйте стратегию выполнения, возвращенную 'DbContext.Database.CreateExecutionStrategy()', чтобы выполнить все операции в транзакции как повторяемую единицу.

Решение заключается в том, чтобы вызвать стратегию выполнения с помощью делегата, который представляет все, нужно выполнить вручную. В случае временного сбоя стратегия выполнения будет снова вызывать делегат.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var context = new BloggingContext())
        {
            using (var transaction = context.Database.BeginTransaction())
            {
                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/dotnet"});
                context.SaveChanges();

                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/visualstudio"});
                context.SaveChanges();

                transaction.Commit();
            }
        });
    });
}
```

Ошибка фиксации транзакции и проблема идемпотентности

В общем случае при сбое подключения текущая транзакция откатывается. Тем не менее, если соединение разорвано во время транзакции благосостояние зафиксирована полученный в результате состояние транзакции неизвестно. См. в разделе, это [блога](#) для получения дополнительных сведений.

По умолчанию, стратегия выполнения будет повторять эту операцию, так как в том случае, если откат транзакции, но если это не так это приведет к исключение, если новое состояние базы данных несовместим или может привести к **повреждения данных**. Если Операция не полагаться на определенном состоянии, например при вставке новой строки с помощью автоматически сформированных значений ключа.

Существует несколько способов решения этой проблемы.

Вариант 1. nothing (почти)

Вероятность сбоя подключения во время фиксации транзакции недостаточно, поэтому оно может быть приемлемым для вашего приложения, просто ошибкой, если данное состояние возникает, фактически.

Тем не менее необходимо избегать использования сформированные хранилищем ключи, чтобы гарантировать, что исключение вместо добавления повторяющуюся строку. Рекомендуется использовать значение идентификатора GUID, сформированное клиентом или генератор значение на стороне клиента.

Вариант 2 - состояния приложения перестроения

1. Отменить текущий `DbContext`.
2. Создайте новый `DbContext` и восстановить состояние приложения из базы данных.
3. Информировать пользователей о том, что последняя операция не выполнена успешно.

Вариант 3: Добавление проверки состояния

Для большинства операций, которые изменяют состояние базы данных можно добавить код, который проверяет, успешно ли оно выполнено. EF предоставляет метод расширения, чтобы облегчить эту задачу - `IExecutionStrategy.ExecuteInTransaction`.

Этот метод начинает и фиксирует транзакцию, а также принимает функцию в `verifySucceeded` параметр, который вызывается, когда возникает временная ошибка во время фиксации транзакции.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    var blogToAdd = new Blog {Url = "http://blogs.msdn.com/dotnet"};
    db.Blogs.Add(blogToAdd);

    strategy.ExecuteInTransaction(db,
        operation: context =>
    {
        context.SaveChanges(acceptAllChangesOnSuccess: false);
    },
    verifySucceeded: context => context.Blogs.AsNoTracking().Any(b => b.BlogId == blogToAdd.BlogId));

    db.ChangeTracker.AcceptAllChanges();
}
```

NOTE

Здесь `SaveChanges` вызывается с `acceptAllChangesOnSuccess` присвоено `false` во избежание изменения состояния `Blog` сущность `Unchanged`. Если `SaveChanges` завершается успешно. Это позволяет повторите ту же операцию, если фиксация завершается сбоем и выполняется откат транзакции.

Вариант 4: вручную отслеживания транзакции

Если требуется использовать сформированные хранилищем ключи или вам требуется универсальный способ обработки сбоев, не зависящая от операции, выполняемой каждой транзакции можно было назначать идентификатор, который проверяется при фиксация завершается сбоем.

1. Добавление таблицы в базу данных, используемый для отслеживания состояния транзакций.
2. Вставьте строку в таблице в начале каждой транзакции.
3. В случае сбоя подключения во время фиксации, проверьте наличие соответствующей строки в базе данных.
4. Если фиксация выполнена успешно, удалите соответствующей строки во избежание увеличение размера таблицы.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });

    var transaction = new TransactionRow { Id = Guid.NewGuid() };
    db.Transactions.Add(transaction);

    strategy.ExecuteInTransaction(db,
        operation: context =>
    {
        context.SaveChanges(acceptAllChangesOnSuccess: false);
    },
        verifySucceeded: context => context.Transactions.AsNoTracking().Any(t => t.Id == transaction.Id));

    db.ChangeTracker.AcceptAllChanges();
    db.Transactions.Remove(transaction);
    db.SaveChanges();
}
```

NOTE

Убедитесь, что контекст, используемый для проверки стратегии выполнения определяются как соединение может привести к отказу во время проверки, если не удалось выполнить во время фиксации транзакции.

Тестирование

28.08.2018 • 2 minutes to read • [Edit Online](#)

Вам может потребоваться протестировать компоненты, используя средство, имитирующее подключение к реальной базе данных, без дополнительных затрат, создаваемых операциями ввода-вывода базы данных.

Для этого существует два основных способа.

- [Режим выполнения в памяти SQLite](#) позволяет создавать эффективные тесты для поставщика, который ведет себя как реляционная база данных.
- [Поставщик InMemory](#) является упрощенным поставщиком, который имеет минимальные зависимости, но не всегда ведет себя как реляционная база данных.

Тестирование с помощью SQLite

28.08.2018 • 4 minutes to read • [Edit Online](#)

SQLite имеет режим в памяти, можно использовать для написания тестов с реляционной базой данных, без издержек на фактических операций базы данных SQLite.

TIP

Можно просмотреть в этой статье [пример](#) на GitHub

Пример сценария тестирования

Предположим, что благодаря которой код приложения для выполнения некоторых операций, связанных с блоги. Внутри она использует `DbContext`, соединяется с базой данных SQL Server. Оно полезно для обмена данном контексте для подключения к базе данных SQLite в памяти, таким образом, мы может создавать эффективные тесты для этой службы без необходимости изменять код или выполнить большой объем работы по созданию теста double контекста.

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

namespace BusinessLogic
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public void Add(string url)
        {
            var blog = new Blog { Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();
        }

        public IEnumerable<Blog> Find(string term)
        {
            return _context.Blogs
                .Where(b => b.Url.Contains(term))
                .OrderBy(b => b.Url)
                .ToList();
        }
    }
}
```

Подготовка к контексту

Не настраивать два поставщика базы данных

В тестах вы собираетесь извне настроить контекст, используемый поставщик `InMemory`. При настройке поставщика базы данных путем переопределения `OnConfiguring` в контексте, затем необходимо добавить

некоторые условный код, чтобы убедиться, только настроить поставщик базы данных, если уже не был настроен.

TIP

Если вы используете ASP.NET Core, то нет необходимости этот код с момента настройки поставщика базы данных вне контекста (в файле Startup.cs).

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}
```

Добавьте конструктор для тестирования

Чтобы обеспечить возможность тестирования в другой базе данных проще всего изменить текущий контекст предоставляют конструктор, принимающий `DbContextOptions<TContext>`.

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {}

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    {}
}
```

TIP

`DbContextOptions<TContext>` Указывает контексту на все его параметры, например базу данных для подключения к. Это тот же объект, в которой создано, выполнив метод `OnConfiguring` в вашем контексте.

Написание тестов

Ключ для тестирования с этим поставщиком является возможность определить контекст SQLite и управлять областью базы данных в памяти. Область базы данных управляется Открытие и закрытие соединения. Базы данных, ограничиваются длительность, при открытом соединении. Обычно требуется очистить базу данных для каждого метода теста.

```
using BusinessLogic;
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestProject.SQLite
{
    [TestClass]
    public class BlogServiceTests
    {
        [TestMethod]
        public void Add_writes_to_database()
        {
            // In-memory database only exists while the connection is open
        }
    }
}
```

```

// In-memory database only exists while the connection is open
var connection = new SqliteConnection("DataSource=:memory:");
connection.Open();

try
{
    var options = new DbContextOptionsBuilder<BloggingContext>()
        .UseSqlite(connection)
        .Options;

    // Create the schema in the database
    using (var context = new BloggingContext(options))
    {
        context.Database.EnsureCreated();
    }

    // Run the test against one instance of the context
    using (var context = new BloggingContext(options))
    {
        var service = new BlogService(context);
        service.Add("http://sample.com");
    }

    // Use a separate instance of the context to verify correct data was saved to database
    using (var context = new BloggingContext(options))
    {
        Assert.AreEqual(1, context.Blogs.Count());
        Assert.AreEqual("http://sample.com", context.Blogs.Single().Url);
    }
}
finally
{
    connection.Close();
}

[TestMethod]
public void Find_searches_url()
{
    // In-memory database only exists while the connection is open
    var connection = new SqliteConnection("DataSource=:memory:");
    connection.Open();

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlite(connection)
            .Options;

        // Create the schema in the database
        using (var context = new BloggingContext(options))
        {
            context.Database.EnsureCreated();
        }

        // Insert seed data into the database using one instance of the context
        using (var context = new BloggingContext(options))
        {
            context.Blogs.Add(new Blog { Url = "http://sample.com/cats" });
            context.Blogs.Add(new Blog { Url = "http://sample.com/catfish" });
            context.Blogs.Add(new Blog { Url = "http://sample.com/dogs" });
            context.SaveChanges();
        }

        // Use a clean instance of the context to run the test
        using (var context = new BloggingContext(options))
        {
            var service = new BlogService(context);
            var result = service.Find("cat");
        }
    }
}

```

```
        Assert.AreEqual(2, result.Count());
    }
}
finally
{
    connection.Close();
}
}
}
```

Тестирование с помощью InMemory

28.08.2018 • 5 minutes to read • [Edit Online](#)

Поставщик InMemory полезно в тех случаях, когда вы хотите протестировать компоненты, используя средство, имитирующее подключение к реальной базе данных, без издержек на фактических операций базы данных.

TIP

Для этой статьи вы можете скачать [пример](#) из репозитория GitHub.

InMemory не является реляционной базы данных

Поставщики базы данных EF Core не быть реляционных баз данных. InMemory предназначен для базы данных общего назначения для тестирования и не предназначен для имитации реляционной базы данных.

Некоторые примеры включают:

- InMemory позволит вам сохранить данные, которые нарушают ограничения ссылочной целостности в реляционной базе данных.
- Если вы используете `DefaultValueSql(string)` для свойства в модели, это — это реляционная база данных API и не имеет смысла при работе с InMemory.
- **Параллелизм с помощью метки времени и строк версии** (`[Timestamp]` или `IsRowVersion`) не поддерживается. Не `DbUpdateConcurrencyException` возникает, если обновление выполняется с помощью старого маркера параллелизма.

TIP

Для многих тестовых целей эти различия будут иметь значения. Тем не менее, если вы хотите проверить то, что поведение во многом напоминает true реляционной базы данных, рассмотрите возможность использования [режиме in-memory SQLite](#).

Пример сценария тестирования

Предположим, что благодаря которой код приложения для выполнения некоторых операций, связанных с блоги. Внутри она использует `DbContext`, соединяется с базой данных SQL Server. Оно полезно для обмена данном контексте для подключения к базе данных InMemory, таким образом, мы может создавать эффективные тесты для этой службы без необходимости изменять код или выполнить большой объем работы по созданию теста `double` контекста.

```

using System.Collections.Generic;
using System.Linq;

namespace BusinessLogic
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public void Add(string url)
        {
            var blog = new Blog { Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();
        }

        public IEnumerable<Blog> Find(string term)
        {
            return _context.Blogs
                .Where(b => b.Url.Contains(term))
                .OrderBy(b => b.Url)
                .ToList();
        }
    }
}

```

Подготовка к контексту

Не настраивать два поставщика базы данных

В тестах вы собираетесь извне настроить контекст, используемый поставщик InMemory. При настройке поставщика базы данных путем переопределения `OnConfiguring` в контексте, затем необходимо добавить некоторые условный код, чтобы убедиться, только настроить поставщик базы данных, если уже не был настроен.

```

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}

```

TIP

Если вы используете ASP.NET Core, то нет необходимости этот код так как поставщик базы данных уже настроена вне контекста (в файле Startup.cs).

Добавьте конструктор для тестирования

Чтобы обеспечить возможность тестирования в другой базе данных проще всего изменить текущий контекст предоставляют конструктор, принимающий `DbContextOptions<TContext>`.

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    { }

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }
```

TIP

`DbContextOptions<TContext>` Указывает контексту на все его параметры, например базу данных для подключения к. Это тот же объект, в которой создано, выполнив метод `OnConfiguring` в вашем контексте.

Написание тестов

Ключ для тестирования с этим поставщиком является возможность определить контекст для использования поставщика `InMemory` и управлять областью базы данных в памяти. Обычно требуется очистить базу данных для каждого метода теста.

Ниже приведен пример тестового класса, использующего базу данных `InMemory`. Каждый метод теста указывает уникальное имя базы данных, это означает, что каждый метод имеет свою собственную базу данных `InMemory`.

TIP

Чтобы использовать `.UseInMemoryDatabase()` метод расширения, ссылка на пакет NuGet `Microsoft.EntityFrameworkCore.InMemory`.

```

using BusinessLogic;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestProject.InMemory
{
    [TestClass]
    public class BlogServiceTests
    {
        [TestMethod]
        public void Add_writes_to_database()
        {
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseInMemoryDatabase(databaseName: "Add_writes_to_database")
                .Options;

            // Run the test against one instance of the context
            using (var context = new BloggingContext(options))
            {
                var service = new BlogService(context);
                service.Add("http://sample.com");
            }

            // Use a separate instance of the context to verify correct data was saved to database
            using (var context = new BloggingContext(options))
            {
                Assert.AreEqual(1, context.Blogs.Count());
                Assert.AreEqual("http://sample.com", context.Blogs.Single().Url);
            }
        }

        [TestMethod]
        public void Find_searches_url()
        {
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseInMemoryDatabase(databaseName: "Find_searches_url")
                .Options;

            // Insert seed data into the database using one instance of the context
            using (var context = new BloggingContext(options))
            {
                context.Blogs.Add(new Blog { Url = "http://sample.com/cats" });
                context.Blogs.Add(new Blog { Url = "http://sample.com/catfish" });
                context.Blogs.Add(new Blog { Url = "http://sample.com/dogs" });
                context.SaveChanges();
            }

            // Use a clean instance of the context to run the test
            using (var context = new BloggingContext(options))
            {
                var service = new BlogService(context);
                var result = service.Find("cat");
                Assert.AreEqual(2, result.Count());
            }
        }
    }
}

```

Настройка DbContext

28.08.2018 • 5 minutes to read • [Edit Online](#)

В этой статье показаны базовые шаблоны для настройки `DbContext` через `DbContextOptions` для подключения к базе данных с помощью определенного поставщика EF Core и необязательного поведения.

Конфигурация DbContext во время разработки

Средства EF Core во время разработки, такие как [миграции](#) должны иметь возможность находить и создавать экземпляр рабочего `DbContext` типа для сбора сведений о типах сущностей и их сопоставлении схеме базы данных приложения. Этот процесс может выполняться автоматически, до тех пор, пока средство может легко создавать `DbContext` таким образом, что он будет назначен точно так же как будет настраиваться во время выполнения.

Хотя любой шаблон, который предоставляет сведения о конфигурации, необходимые для `DbContext` можно работать во время выполнения, средства, которые требуют использования `DbContext` во время разработки можно работать только с ограниченный набор шаблонов. Эти файлы рассматриваются более подробно в [создание контекста разработки](#) раздел.

Настройка DbContextOptions

`DbContext` необходимо иметь экземпляр `DbContextOptions` для какой-либо работы. `DbContextOptions`

Экземпляр содержит сведения о конфигурации, такие как:

- Поставщик базы данных для использования, обычно выбирается путем вызова метода, такие как `UseSqlServer` или `UseSqlite`
- Любой необходимую строку подключения или идентификатор экземпляра базы данных, обычно передается в качестве аргумента в метод выбора поставщика, упомянутых выше
- Любое необязательное поведение уровня поставщика селекторы, также цепочку внутри вызова метода выбора поставщика
- Все общие селекторы поведение EF Core, обычно связанных после или до метода выбора поставщика

В следующем примере настраивается `DbContextOptions` использование поставщика SQL Server, подключение содержащихся в `connectionString` переменную, время ожидания команды уровня поставщика и селектор поведения EF Core, который делает все запросы, выполняемые в `DbContext` [Отключение отслеживания](#) по умолчанию:

```
optionsBuilder
    .UseSqlServer(connectionString, providerOptions=>providerOptions.CommandTimeout(60))
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
```

NOTE

Методы выбора поставщика и другие методы выбора поведение, упомянутых выше являются методами расширений в `DbContextOptions` или классы поставщика параметров. Чтобы получить доступ к эти методы расширения, может потребоваться пространство имен (обычно `Microsoft.EntityFrameworkCore`) в области видимости и включить зависимости дополнительных пакетов в проект.

`DbContextOptions` Может быть задано для `DbContext` путем переопределения `OnConfiguring` метод или извне

через аргумент конструктора.

Если указаны оба, `OnConfiguring` применяется последним и перезаписывают параметры, передаваемое аргументом конструктора.

Аргумент конструктора

Контекст кода с помощью конструктора:

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

TIP

Базовый конструктор `DbContext` также принимает нестандартную версию `DbContextOptions`, но неуниверсальная версия не рекомендуется использовать для приложений с несколькими типами контекста.

Код приложения для инициализации из аргумента конструктора:

```
var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))
{
    // do stuff
}
```

OnConfiguring

Контекст кода с помощью `OnConfiguring`:

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }
}
```

Код приложения для инициализации `DbContext`, использующий `OnConfiguring`:

```
using (var context = new BloggingContext())
{
    // do stuff
}
```

TIP

Этот подход не применяться для тестирования, если тесты предназначенных для всей базы данных.

С помощью DbContext с помощью внедрения зависимостей

EF Core поддерживает использование `DbContext` с контейнером внедрения зависимостей. Ваш тип `DbContext` можно добавить в контейнер службы с помощью `AddDbContext<TContext>` метод.

`AddDbContext<TContext>` сделает оба ваша типа `DbContext` `TContext` и соответствующих `DbContextOptions<TContext>` для внедрения из контейнера служб.

См. в разделе [дополнительные чтения](#) ниже дополнительные сведения о внедрении зависимостей.

Добавление `DbContext` для внедрения зависимостей:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data Source=blog.db"));
}
```

Это требует добавления [аргумент конструктора](#) для вашего типа `DbContext`, который принимает `DbContextOptions<TContext>`.

Контекст кода:

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        :base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

Код приложения (в ASP.NET Core):

```
public class MyController
{
    private readonly BloggingContext _context;

    public MyController(BloggingContext context)
    {
        _context = context;
    }

    ...
}
```

Код приложения (с использованием ServiceProvider напрямую, менее распространено):

```
using (var context = serviceProvider.GetService<BloggingContext>())
{
    // do stuff
}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>();
```

Дополнительные материалы

- Чтение [начало работы в ASP.NET Core](#) Дополнительные сведения об использовании EF с ASP.NET Core.
- Чтение [внедрения зависимостей](#) Дополнительные сведения об использовании внедрения Зависимостей.

- Чтение [тестирования](#) Дополнительные сведения.

Обновление с версии-кандидата 1 EF Core 1.0 до 1.0 RC2

28.08.2018 • 8 minutes to read • [Edit Online](#)

Эта статья содержит руководство по переносу приложения, созданного с помощью версии-кандидата 1 пакеты для версии-кандидата 2.

Имена пакетов и версий

Между RC1 и RC2 изменено с «Entity Framework 7» на «Entity Framework Core». Дополнительные сведения о причинах возникновения изменения [блога, Скотт Хансельман](#). Благодаря этому наши имена пакетов изменились с `EntityFramework.*` для `Microsoft.EntityFrameworkCore.*` и наши версии от `7.0.0-rc1-final` для `1.0.0-rc2-final` (или `1.0.0-preview1-final` инструментов).

Необходимо полностью удалить пакеты версии-кандидата 1, а затем установить RC2 объектам. Вот некоторые распространенные пакеты сопоставление.

ВЕРСИЯ-КАНДИДАТ 1 ПАКЕТА	ЭКВИВАЛЕНТ RC2
<code>EntityFramework.MicrosoftSqlServer 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.SqlServer 1.0.0-rc2-final</code>
<code>EntityFramework.SQLite 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.SQLite 1.0.0-rc2-final</code>
<code>EntityFramework7.Npgsql 3.1.0-rc1-3</code>	<code>Npgsql.EntityFrameworkCore.Postgres</code>
<code>EntityFramework.SqlServerCompact35 7.0.0-rc1-final</code>	<code>EntityFrameworkCore.SqlServerCompact35 1.0.0-rc2-final</code>
<code>EntityFramework.SqlServerCompact40 7.0.0-rc1-final</code>	<code>EntityFrameworkCore.SqlServerCompact40 1.0.0-rc2-final</code>
<code>EntityFramework.InMemory 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.InMemory 1.0.0-rc2-final</code>
<code>EntityFramework.IBMDataServer 7.0.0-beta1</code>	Еще не доступна для версии-кандидате 2
<code>EntityFramework.Commands 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.Tools 1.0.0-preview1-final</code>
<code>EntityFramework.MicrosoftSqlServer.Design 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.SqlServer.Design 1.0.0-rc2-final</code>

Пространства имен

Вместе с именами пакетов, пространства имен изменилось с `Microsoft.Data.Entity.*` для `Microsoft.EntityFrameworkCore.*`. Можно обработать это изменение с помощью поиска и замены `using Microsoft.Data.Entity` с `using Microsoft.EntityFrameworkCore`.

Таблицы изменений в соглашениях именования

Значительное изменение функциональных мы воспользовались в версии-кандидата 2 мы хотели использовать имя `DbSet< TEntity >` свойство для заданной сущности в качестве имени таблицы оно сопоставлено, а не только имя класса. Дополнительные сведения об этом изменении в [проблему связанные](#)

[объявления](#).

Для существующих приложений RC1, рекомендуется добавить следующий код в начало вашей `OnModelCreating` методом обновления стратегия именования RC1:

```
foreach (var entity in modelBuilder.Model.GetEntityTypes())
{
    entity.Relational().TableName = entity.DisplayName();
}
```

Если вы хотите внедрить новую стратегию именования, рекомендуется успешно, выполнив остальные этапы обновления и затем удаления кода и создания перехода на применение таблицы переименовывает.

AddDbContext / Startup.cs изменяет (только для проектов ASP.NET Core)

В RC1, приходилось добавлять Entity Framework службы поставщику услуг приложения -

`Startup.ConfigureServices(...)` :

```
services.AddEntityFramework()
    .AddSqlServer()
    .AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

В версии-кандидата 2, можно удалить вызовы `AddEntityFramework()`, `AddSqlServer()` и т. д.:

```
services.AddDbContext<ApplicationContext>(options =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

Необходимо также добавить конструктор, для вашего производного контекста, который принимает параметры контекста и передает их в базовый конструктор. Это необходимо, поскольку мы удалили некоторую часть scary волшебство, использовали их в фоновом режиме:

```
public ApplicationContext(DbContextOptions<ApplicationContext> options)
    : base(options)
{}
```

Передача IServiceProvider

Если у вас есть код версии-кандидата 1, который передает `IServiceProvider` к контексту, это теперь перемещена в `DbContextOptions`, а не параметра отдельных конструктора. Используйте `DbContextOptionsBuilder.UseInternalServiceProvider(...)` для задания поставщика услуг.

Тестирование

Наиболее распространенным сценарием для этого было управлять областью InMemory базы данных, при тестировании. См. в разделе обновленный [тестирования](#) статье пример делать это с помощью версии-кандидата 2.

Разрешение внутренних служб от поставщика службы приложений (только для проектов ASP.NET Core)

Если у вас есть приложение ASP.NET Core и хотите, чтобы EF для разрешения внутренних служб из поставщика служб приложений, является перегрузка метода `AddDbContext`, можно настроить это:

```
services.AddEntityFrameworkSqlServer()
    .AddDbContext<ApplicationContext>((serviceProvider, options) =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"])
        .UseInternalServiceProvider(serviceProvider)); );
```

WARNING

Мы рекомендуем, позволяя EF для внутренних целей управления собственных служб, если нет причин для объединения внутренних служб EF в поставщику службы приложения. Основная причина, что вы можете сделать это является использование поставщику службы приложений для замены службы, которые внутренне использует EF

Команды DNX = > .NET CLI (только для проектов ASP.NET Core)

Если вы ранее использовали `dnx ef` команд для проектов ASP.NET 5, они уже перешли `dotnet ef` команды. По-прежнему применяется один и тот же синтаксис команды. Можно использовать `dotnet ef --help` сведения о синтаксисе.

Способ регистрации команд изменилось в версии-кандидата 2, из-за DNX, заменяемое интерфейса командной строки .NET. Команды зарегистрированы в `tools` статьи `project.json`:

```
"tools": {
  "Microsoft.EntityFrameworkCore.Tools": {
    "version": "1.0.0-preview1-final",
    "imports": [
      "portable-net45+win8+dnxcore50",
      "portable-net45+win8"
    ]
  }
}
```

TIP

Если вы используете Visual Studio, теперь можно использовать консоль диспетчера пакетов для выполнения команд EF для проектов ASP.NET Core (это не поддерживается в версии-кандидата 1). По-прежнему необходимо зарегистрировать команды в `tools` раздел `project.json` это сделать.

Диспетчер пакетов команд требуется PowerShell 5

Если вы используете команды Entity Framework в консоли диспетчера пакетов в Visual Studio, затем необходимо будет убедиться, что у вас установлен PowerShell 5. Это временный требование, которое будет удален в следующем выпуске (см. в разделе [выдавать #5327](#) подробности).

С помощью «imports» в файле project.json

Некоторые из зависимостей EF Core не поддерживают .NET Standard еще. EF Core в проектах .NET Standard и .NET Core может потребоваться добавление «imports» в файл `project.json` в качестве временного решения.

При добавлении EF, восстановление NuGet будет отображаться это сообщение об ошибке:

```
Package Ix-Async 1.2.5 is not compatible with netcoreapp1.0 (.NETCoreApp,Version=v1.0). Package Ix-Async 1.2.5
supports:
- net40 (.NETFramework,Version=v4.0)
- net45 (.NETFramework,Version=v4.5)
- portable-net45+win8+wp8 (.NETPortable,Version=v0.0,Profile=Profile78)
Package Remotion.Linq 2.0.2 is not compatible with netcoreapp1.0 (.NETCoreApp,Version=v1.0). Package
Remotion.Linq 2.0.2 supports:
- net35 (.NETFramework,Version=v3.5)
- net40 (.NETFramework,Version=v4.0)
- net45 (.NETFramework,Version=v4.5)
- portable-net45+win8+wp8+wpa81 (.NETPortable,Version=v0.0,Profile=Profile259)
```

Обойти это можно вручную импортировать на переносимый профиль «portable net451 + win8». Это заставляет NuGet обрабатывать это двоичные файлы, которые соответствуют это предоставить в качестве платформа, совместимая с .NET Standard, несмотря на то, что они не являются. Несмотря на то, что «portable net451 + win8» не равно 100% совместим с .NET Standard, он совместим достаточно для перехода из библиотеки PCL .NET Standard. Импорты могут быть удалены, когда зависимости платформы EF в конечном счете обновление до .NET Standard.

Несколько платформ могут добавляться к «imports» в синтаксис массива. Других команд импорта может возникнуть необходимость в том случае, если добавить дополнительные библиотеки в проект.

```
{
  "frameworks": {
    "netcoreapp1.0": {
      "imports": [ "dnxcore50", "portable-net451+win8" ]
    }
  }
}
```

См. в разделе [выдавать #5176](#).

Обновление с EF Core 1.0 RC2 до RTM

28.08.2018 • 4 minutes to read • [Edit Online](#)

Эта статья содержит руководство по переносу приложения, созданного с помощью пакетов 1.0.0, версия-кандидат 2 RTM.

Версии пакетов

Имена пакетов верхнего уровня, которые требуется установить в приложение обычно не изменился между RC2 и RTM.

Вам потребуется обновить установленные пакеты до RTM-версии.

- Пакеты среды выполнения (например, `Microsoft.EntityFrameworkCore.SqlServer`) изменилось с `1.0.0-rc2-final` для `1.0.0`.
- `Microsoft.EntityFrameworkCore.Tools` Пакета изменен с `1.0.0-preview1-final` для `1.0.0-preview2-final`.
Обратите внимание, что оборудование по-прежнему предварительной версии.

Существующие миграции может потребоваться добавить `maxLength`

В версии-кандидата 2, определению столбца в миграции выглядело как `table.Column<string>(nullable: true)` и длину столбца было искать в некоторые метаданные, мы сохраняем в коде для миграции. В версии RTM, длина теперь входит в шаблонном коде `table.Column<string>(maxLength: 450, nullable: true)`.

Все существующие миграции, которые были сконфолдинга перед использованием RTM не будет иметь `maxLength` указан аргумент. Это означает, что будет использоваться максимальную длину, поддерживаемую в базе данных (`nvarchar(max)` на сервере SQL Server). Это может быть нормально для некоторых столбцов, но столбцы, являющиеся частью ключа, внешний ключ или индекс должен обновляться по максимальной длиной. По соглашению 450 — Максимальная длина для ключи, внешние ключи и индексированных столбцов. Если вы явным образом настроили длиной в модели, затем следует использовать этой длины, вместо этого.

ASP.NET Identity

Это изменение влияет на проекты, использующие ASP.NET Identity и были созданы из подготовительной-RTM шаблона проекта. Шаблон проекта включает миграции, используемый для создания базы данных. Этот вид миграции необходимо отредактировать для указания максимальной длиной `256` для следующих столбцов.

• **AspNetRoles**

- `name`
- `NormalizedName`

• **AspNetUsers**

- Адрес эл. почты
- `NormalizedEmail`
- `NormalizedUserName`

- UserName

Не удалось внести это изменение приведет к следующее исключение при первоначальной миграции применяется к базе данных.

```
System.Data.SqlClient.SqlException (0x80131904): Column 'NormalizedNames' in table 'AspNetRoles' is of a type  
that is invalid for use as a key column in an index.
```

.NET core: Удаление «import» в project.json

Если вы для .NET Core с помощью версии-кандидата 2, необходимо добавить `imports` в файл `project.json` в качестве временного решения для некоторых зависимостей EF Core не поддерживает .NET Standard. Они могут быть удален.

```
{  
  "frameworks": {  
    "netcoreapp1.0": {  
      "imports": ["dnxcore50", "portable-net451+win8"]  
    }  
  }  
}
```

NOTE

Начиная с версии 1.0 RTM, [пакет SDK для .NET Core](#) больше не поддерживает `project.json` или разработки приложений .NET Core с помощью Visual Studio 2015. Корпорация Майкрософт рекомендует [выполнить миграцию project.json в формат csproj](#). Если вы используете Visual Studio, мы рекомендуем обновить до [Visual Studio 2017](#).

UWP: Добавить переадресации привязок

При попытке выполнения команд EF на результатов проектов универсальной платформы Windows (UWP) следующая ошибка:

```
System.IO.FileLoadException: Could not load file or assembly 'System.IO.FileSystem.Primitives, Version=4.0.0.0,  
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' or one of its dependencies. The located assembly's manifest  
definition does not match the assembly reference.
```

Необходимо вручную добавить перенаправления привязки в проект универсальной платформы Windows. Создайте файл с именем `App.config` в проекте корневую папку и добавьте перенаправления правильные версии сборок.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="System.IO.FileSystem.Primitives"
                          publicKeyToken="b03f5f7f11d50a3a"
                          culture="neutral" />
        <bindingRedirect oldVersion="4.0.0.0"
                        newVersion="4.0.1.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Threading.Overlapped"
                          publicKeyToken="b03f5f7f11d50a3a"
                          culture="neutral" />
        <bindingRedirect oldVersion="4.0.0.0"
                        newVersion="4.0.1.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Обновление приложений из предыдущих версий до EF Core 2.0

27.09.2018 • 13 minutes to read • [Edit Online](#)

Мы предприняли возможность значительно улучшать наши существующие API-интерфейсы и поведения в версии 2.0. Существует несколько улучшений, что может потребовать изменения в существующий программный код, несмотря на то, что мы считаем, что для большинства приложений влияние будет низкой, в большинстве случаев, требующих просто перекомпиляции и интерактивная незначительно замените устаревшие API.

Может потребоваться обновление существующего приложения до EF Core 2.0:

1. Обновление реализации .NET целевого приложения, которая поддерживает .NET Standard 2.0. См. в разделе [поддерживаемые реализации .NET](#) для получения дополнительных сведений.
2. Определения поставщика для целевой базы данных, которая совместима с EF Core 2.0. См. в разделе [EF Core 2.0 не требуется поставщик базы данных 2.0](#) ниже.
3. Обновление всех пакетов EF Core (среды выполнения и инструментов) 2.0. Ссылаться на [Установка EF Core](#) для получения дополнительных сведений.
4. Внесите изменения необходимый код для компенсации критические изменения, описанные в оставшейся части этого документа.

ASP.NET Core теперь включает EF Core

Приложения, предназначенные для ASP.NET Core 2.0, могут использовать EF Core 2.0 без дополнительных зависимостей, кроме сторонних поставщиков базы данных. Тем не менее приложения, предназначенные для предыдущих версий ASP.NET Core должны выполнить обновление до ASP.NET Core 2.0, чтобы использовать EF Core 2.0. Дополнительные сведения об обновлении приложений ASP.NET Core 2.0 см. [документации по ASP.NET Core по этой теме](#).

Новый способ получения службы приложений в ASP.NET Core

Рекомендуемый шаблон для веб-приложений ASP.NET Core была обновлена для 2.0, в результате которого было передано логику во время разработки, используемые EF Core в версии 1.x. Ранее во время разработки, EF Core будет выполнена попытка вызвать `Startup.ConfigureServices` напрямую, чтобы получить доступ к поставщику службы приложения. В ASP.NET Core 2.0 инициализации конфигурации используется за пределами `Startup` класса. Приложения, использующие EF Core обычно доступ к своей строке подключения из конфигурации, поэтому `Startup` сама по себе больше не достаточно. Если вы обновляете приложение ASP.NET Core 1.x, при использовании средства EF Core может появиться следующая ошибка.

Конструктор без параметров не найден на «`ApplicationContext`». Добавьте конструктор без параметров «`ApplicationContext`» или добавьте реализацию "`IDesignTimeDbContextFactory<ApplicationContext>`" в сборке «`ApplicationContext`»

В шаблоне по умолчанию ASP.NET Core 2.0 был добавлен новый обработчик во время разработки.

Статический `Program.BuildWebHost` метод, который позволяет EF Core для доступа к поставщику службы приложения во время разработки. Если вы обновляете приложение ASP.NET Core 1.x, необходимо обновить `Program` класс следующим образом::

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2._0App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

Внедрение этого нового шаблона, при обновлении приложений 2.0 настоятельно рекомендуется и необходимо для таких функций, как миграция Entity Framework Core для работы. Типичная альтернатива — реализовать `IDesignTimeDbContextFactory<TContext>`.

IDbContextFactory переименован

Чтобы поддерживать шаблоны для различных приложений и предоставление пользователям больший контроль над тем, как их `DbContext` используется во время разработки, у нас есть, в прошлом, предоставляемых `IDbContextFactory<TContext>` интерфейс. Во время разработки, средства EF Core найдет реализации этого интерфейса в проекте и использовать его для создания `DbContext` объектов.

Этот интерфейс был очень общее имя, которое ввести в заблуждение нескольких пользователей, чтобы попробовать повторно использовать его для других `DbContext`-Создание сценариев. Они были застигнутым врасплох средствам EF попытался использовать свою реализацию во время разработки и вызвало команд, таких как `Update-Database` ИЛИ `dotnet ef database update` переход на другой.

Для связи семантика строгого во время разработки этот интерфейс, мы переименовали его `IDesignTimeDbContextFactory<TContext>`.

2.0 release `IDbContextFactory<TContext>` по-прежнему существует, но помечен как устаревший.

Удалить DbContextFactoryOptions

Из-за изменения ASP.NET Core 2.0, описанные выше, мы обнаружили, что `DbContextFactoryOptions` был больше не нужны на новом `IDesignTimeDbContextFactory<TContext>` интерфейс. Ниже приведены альтернативные варианты, которые следует использовать вместо этого.

DBCONTEXTFACTORYOPTIONS	АЛЬТЕРНАТИВА
ApplicationBasePath	ApplicationContext.BaseDirectory
Атрибут ContentRootPath	Directory.GetCurrentDirectory()
EnvironmentName	Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")

Во время разработки рабочий каталог изменен

ASP.NET Core 2.0 изменений также требуется рабочий каталог, используемый `dotnet ef` выравнивается рабочий каталог, используемый в Visual Studio при работе с приложением. Один наблюдаемый побочный эффект этого заключается в SQLite, что имена файлов теперь являются относительно каталога проекта, а не в выходной каталог, как они привыкли быть.

EF Core 2.0 не требуется поставщик 2.0 базы данных

Для EF Core 2.0 мы внесли множество упрощения и улучшениях в поставщики баз данных способ работы. Это означает, что поставщики 1.0.x и 1.1.x, не будут работать с EF Core 2.0.

Поставщики SQL Server и SQLite поставляются группой EF и 2.0 версии будут доступны как часть 2.0 release. Открытый сторонних поставщиков для [SQL Compact](#), [PostgreSQL](#), и [MySQL](#) обновляются 2.0. Для всех остальных поставщиков обратитесь в службу модуля записи поставщика.

События диагностики и ведения журнала были изменены

Примечание: эти изменения не должны повлиять большая часть кода приложения.

ИД событий для сообщений, отправляемых `ILogger` были изменены в версии 2.0. Теперь эти идентификаторы событий являются уникальными в рамках всего кода EF Core. Эти сообщения теперь соответствуют стандартному шаблону для структурированного ведения журнала, используемому, например, MVC.

Кроме того, были изменены категории средства ведения журнала. Теперь используется известный набор категорий, доступных через `DbLoggerCategory`.

`DiagnosticSource` события теперь используют те же имена идентификатор события как соответствующие `ILogger` сообщений. Полезные данные события являются все номинальные типы, производные от `EventData`.

Идентификаторы событий, категории и типы полезных данных описаны в `CoreEventId` и `RelationalEventId` классы.

Идентификаторы также перенесены из `Microsoft.EntityFrameworkCore.Infrastructure` `Microsoft.EntityFrameworkCore.Diagnostics` пространство имен.

EF Core реляционных метаданных API изменения

EF Core 2.0 теперь будет создавать отдельный объект `IModel` для каждого из используемых поставщиков. Обычно это является прозрачным для приложения. Это позволило упростить работу с API метаданных более низкого уровня, например, добиться того, что любое обращение к основным концепциям реляционных метаданных всегда осуществляется путем вызова `.Relational()` вместо `.SqlServer()`, `.Sqlite()` и т. д. Например код 1.1.x следующим образом:

```
var tableName = context.Model.FindEntityType(typeof(User)).SqlServer().TableName;
```

Теперь должен быть записан следующим образом:

```
var tableName = context.Model.FindEntityType(typeof(User)).Relational().TableName;
```

Вместо того чтобы использовать методы, такие как `ForSqlServerToTable`, методы расширения теперь доступны для записи на основе текущего поставщика используется условный код. Пример:

```
modelBuilder.Entity<User>().ToTable(
    Database.IsSqlServer() ? "SqlServerName" : "OtherName");
```

Обратите внимание, что это изменение применяется только к API-интерфейсы и метаданных, который определен для *все* реляционных поставщиков. API и метаданные остается неизменным, если он определяется только один поставщик. Например, кластеризованные индексы относятся к SQL Server, поэтому `ForSqlServerIsClustered` и `.SqlServer().IsClustered()` по-прежнему должны использоваться.

Не принимают управления поставщика услуг EF

EF Core использует внутренний `IServiceProvider` (контейнер внедрения зависимостей) для его внутренней реализации. Приложения должны поддерживать EF Core для создания и управления этот поставщик, за исключением в особых случаях. Настоятельно рекомендуется удалить все вызовы `UseInternalServiceProvider`. Если в приложении нужно вызвать `UseInternalServiceProvider`, затем попробуйте [отправьте проблему](#) чтобы мы могли изучить другие способы обработки вашего сценария.

Вызов `AddEntityFramework`, `AddEntityFrameworkSqlServer`, и т.д. требуется только в коде приложения `UseInternalServiceProvider` также называется. Удалить все существующие вызовы `AddEntityFramework` или `AddEntityFrameworkSqlServer` т. д. `AddDbContext` следует по-прежнему использовать так же как и раньше.

Должен иметь имя базы данных в памяти

Глобальные без имени базы данных в памяти была удалена, и вместо этого необходимо присвоить имя всех баз данных в памяти. Пример:

```
optionsBuilder.UseInMemoryDatabase("MyDatabase");
```

Это создает и использует базу данных с именем «`MyDatabase`». Если `UseInMemoryDatabase` вызывается снова с тем же именем, то будет использоваться одну и ту же базу данных в памяти, что позволяет совместно использоваться несколькими экземплярами контекста.

Изменения API только для чтения

`IsReadOnlyBeforeSave`, `IsReadOnlyAfterSave`, и `IsStoreGeneratedAlways` устарел и заменены `BeforeSaveBehavior` и `AfterSaveBehavior`. Такое поведение, применяются к любому свойству (не только свойства, сформированное хранилищем) и определить, каким образом следует использовать значение свойства при вставке в строку базы данных (`BeforeSaveBehavior`) или при обновлении существующей базы данных строки (`AfterSaveBehavior`).

Свойства помечены как `ValueGenerated.OnAddOrUpdate` (например, для вычисляемых столбцов) по умолчанию игнорирует любые текущее значение свойства. Это означает, что сформированное хранилищем значение будет получаться независимо от того задать или изменить на отслеживаемой сущности любое значение. Это можно изменить, задав другой `Before\AfterSaveBehavior`.

Новое поведение удаления ClientSetNull

В предыдущих выпусках `DeleteBehavior.Restrict` бы поведение для сущностей, отслеживаемые по контексту нескольких закрытых сопоставленная `SetNull` семантику. В EF Core 2.0 новый `ClientSetNull` представлена поведение по умолчанию для связи необязательно. Такое поведение влечет `SetNull` семантику для отслеживаемых сущностей и `Restrict` поведение для баз данных, созданных с помощью EF Core. Наш опыт показывает это ожидалось/важнейших поведений отслеживаемых сущностей и базой данных.

`DeleteBehavior.Restrict` Теперь учитывается для отслеживаемых сущностей, при установке для связи необязательно.

Удалить пакеты разработки поставщиков

`Microsoft.EntityFrameworkCore.Relational.Design` Пакет был удален. Ее содержимое были объединены в `Microsoft.EntityFrameworkCore.Relational` И `Microsoft.EntityFrameworkCore.Design`.

Это распространяется на поставщика во время разработки пакетов. Эти пакеты (`Microsoft.EntityFrameworkCore.Sqlite.Design`, `Microsoft.EntityFrameworkCore.SqlServer.Design` т. д.) были удалены и их содержимое объединяется в пакеты основным поставщиком.

Чтобы включить `Scaffold-DbContext` или `dotnet ef dbcontext scaffold` в EF Core 2.0, требуется только для ссылки на пакет одного поставщика:

```
<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
    Version="2.0.0" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
    Version="2.0.0"
    PrivateAssets="All" />
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
    Version="2.0.0" />
```

Entity Framework 6

13.09.2018 • 3 minutes to read • [Edit Online](#)

Entity Framework 6 (EF6) — это проверенное средство объектно-реляционного сопоставления (O/RM) для .NET, которое разрабатывалось и совершенствовалось в течение нескольких лет.

В качестве O/RM EF6 уменьшает несогласованность между реляционным и объектно ориентированным мирами, позволяя разработчикам создавать приложения, которые взаимодействуют с данными, хранящимися в реляционных базах данных, с помощью строго типизированных объектов .NET, представляющих прикладную область, и устранив необходимость писать большой объем инфраструктурного кода для доступа к данным.

В EF6 реализованы многие популярные возможности O/RM:

- Сопоставление классов сущностей **POCO** не зависит ни от каких типов EF
- Автоматическое отслеживание изменений.
- Разрешение идентификаторов и единицы работы.
- Безотложная, отложенная и явная загрузка.
- Преобразование строго типизированных запросов с помощью LINQ (Language INtegrated Query).
- Широкие возможности сопоставления, включая поддержку следующих элементов:
 - Отношения "один к одному", "один ко многим" и "многие ко многим"
 - Наследование (одна таблица на иерархию, одна таблица на тип и одна таблица на конкретный класс).
 - Сложные типы
 - Хранимые процедуры
- Визуальный конструктор для создания моделей сущностей.
- Интерфейс Code First для создания моделей сущностей путем написания кода.
- Модели можно создавать на основе существующих баз данных и затем редактировать вручную или же создавать с нуля и затем использовать для создания новых баз данных.
- Интеграция с моделями приложений .NET Framework, включая ASP.NET, и через привязку данных с помощью WPF и WinForms.
- Возможность подключения к базам данных SQL Server, Oracle, MySQL, SQLite, PostgreSQL, DB2 и т. д. на основе ADO.NET и различных поставщиков.

Что выбрать — EF6 или EF Core?

EF Core — это более современная, упрощенная и расширяемая версия Entity Framework, почти аналогичная EF6 по возможностям и преимуществам. EF Core была создана с нуля и содержит новые возможности, недоступные в EF6, хотя в ней пока отсутствуют некоторые расширенные возможности сопоставления из EF6. Мы рекомендуем использовать EF Core в новых приложениях, если набор функций соответствует вашим требованиям. В разделе [Сравнение EF Core и EF6](#) подробно описывается, как сделать выбор.

Начало работы

Добавьте в проект пакет NuGet EntityFramework или установите Entity Framework Tools для Visual Studio. А затем смотрите видео, читайте руководства и знакомьтесь с расширенной документацией, чтобы научиться использовать все возможности Entity Framework 6.

Предыдущие версии Entity Framework

Это документация по последней версии Entity Framework 6, но по большей части она применима и к предыдущим выпускам. Прочтите разделы [Новые возможности](#) и [Прошлые выпуски](#), чтобы узнать обо всех выпусках EF и их возможностях.

Новые возможности EF6

13.09.2018 • 3 minutes to read • [Edit Online](#)

Корпорация Майкрософт настоятельно рекомендует использовать последнюю выпущенную версию Entity Framework, чтобы вы могли использовать новые функции с высочайшим уровнем надежности. Тем не менее мы понимаем, что вам может потребоваться предыдущая версия или вы захотите поэкспериментировать с новыми усовершенствованиями в последнем предварительном выпуске. Чтобы установить определенные версии EF, см. сведения в разделе о [получении Entity Framework](#).

На этой странице приводятся функции, которые включены в каждый новый выпуск.

Последние выпуски

Обновления инструментов для Entity Framework в Visual Studio 2017 версии 15.7

В мае 2018 г. мы выпустили обновленную версию EF Tools в составе Visual Studio 2017 версии 15.7. Она содержит ряд улучшений по частым проблемным вопросам.

- Исправлено несколько ошибок в специальных возможностях пользовательского интерфейса.
- Создано обходное решение для проблемы с медленной работой SQL Server при создании моделей из существующих баз данных [#4](#).
- Улучшена поддержка обновления моделей большого размера из SQL Server [#185](#).

Другим улучшением новой версии EF Tools является автоматическая установка среды выполнения EF 6.2 при создании модели в новом проекте. В предыдущих версиях Visual Studio среду выполнения EF 6.2 (а также любую более раннюю версию EF) можно использовать, установив соответствующую версию пакета NuGet.

Среда выполнения EF 6.2

Среда выполнения EF 6.2 была выпущена в NuGet в октябре 2017 г. Благодаря значительным усилиям нашего сообщества участников, вносящих вклад в разработку ресурсов с открытым исходным кодом, EF 6.2 содержит многочисленные [исправления ошибок](#) и [усовершенствования продуктов](#).

Далее приведен краткий перечень наиболее важных изменений, затрагивающих среду выполнения EF 6.2.

- Сокращено время запуска за счет загрузки готовых моделей Code First из постоянного кэша [#275](#).
- Доступен текущий API для определения индексов [#274](#).
- DbFunctions.Like() для написания запросов LINQ, преобразующих LIKE в SQL [#241](#).
- Migrate.exe теперь поддерживает параметр -script [#240](#).
- EF6 теперь может работать со значениями ключей, созданными с помощью последовательности в SQL Server [#165](#).
- Обновленный список временных ошибок для стратегии выполнения SQL Azure [#83](#).
- Ошибка. Повторное выполнение запросов или команд SQL завершается ошибкой The SqlParameter is already contained by another SqlParameterCollection ("Параметр SqlParameter уже содержится в другом объекте SqlParameterCollection") [#81](#).
- Ошибка. Часто истекает время ожидания оценки DbQuery.ToString() в отладчике [#73](#).

Будущие выпуски

Сведения о следующей версии EF6, см. в нашем [плане выпусков](#).

Прошлые выпуски

На странице [прошлых выпусков](#) содержится архив всех предыдущих версий EF и основных функций, которые были представлены в каждом выпуске.

Будущие версии Entity Framework

13.09.2018 • 2 minutes to read • [Edit Online](#)

Здесь можно найти сведения о будущих версий платформы Entity Framework. Хотя основные усилия команды EF в настоящее время находится на добавление новых функций и улучшений для [EF Core](#), мы планируем по-прежнему исправления важных ошибок, реализовать усовершенствований и использует предложения участников сообщества в базе кода с EF6.

Освобождает после EF 6.2

План после EF 6.2 выпусков находятся в разработке. Дополнительные сведения будут публиковаться здесь, когда она станет доступна.

Оставаясь в актуальном состоянии

Помимо этой странице новых выпусков сообщается обычно на [блог команды разработчиков .NET](#) и нашей учетной записи Twitter, [@efmagicunicorns](#) .

Прошлые выпуски платформы Entity Framework

27.09.2018 • 24 minutes to read • [Edit Online](#)

Первая версия Entity Framework была выпущена в 2008 г., как часть .NET Framework 3.5 SP1 и Visual Studio 2008 SP1.

Начиная с выпуска EF4.1, он поставлялся в виде [пакета NuGet EntityFramework](#) -сейчас один из наиболее популярных пакетов на сайте NuGet.org.

Версии 4.1 и 5.0 пакета EntityFramework NuGet расширенных библиотек EF, поставляемые как часть платформы .NET Framework.

Начиная с версии 6, EF стал проект с открытым кодом и также полностью перемещены из формы аппаратного контроллера управления .NET Framework. Это означает, что при добавлении пакета NuGet EntityFramework версии 6 к приложению, вы получаете полную копию EF библиотеки, в которой не зависит от EF биты, которые поставляются в составе .NET Framework. Это позволило немного ускорить темп разработки и внедрять новые функции.

В июне 2016 г. Корпорация Майкрософт выпустила EF Core 1.0. EF Core основан на новой базы кода и предназначен как более легким и расширяемый версия EF. Сейчас EF Core является основной темой разработки для Entity Framework корпорации Microsoft. Это означает, что нет новых основных функций, запланированных для EF6. Тем не менее EF6 по-прежнему поддерживаются как проект с открытым кодом и поддерживаемых продуктов корпорации Майкрософт.

Ниже приведен список прошлых выпусках, в обратном хронологическом порядке, с информацией о новых функциях, которые были введены в каждом выпуске.

EF 6.1.3

EF 6.1.3 среда выполнения была выпущена в NuGet в октябре 2015 г. Этот выпуск содержит только исправления дефектов с высоким приоритетом, а регрессий, указанное в отчетах для 6.1.2 выпуска. К важным исправлениям относятся:

- Запрос: Регрессии в EF 6.1.2: OUTER APPLY появились и более сложные запросы для отношения 1:1 и «let» предложения
- Неполадки TPT скрытие свойства базового класса в унаследованного класса
- DbMigration.Sql завершается ошибкой, если в тексте содержится слово «go»
- Создание флаг совместимости для UnionAll и Intersect, поддержка плоских структур
- Запрос с несколькими включает не работает в 6.1.2 (Работа в 6.1.1)
- Исключение «Имеется ошибка синтаксиса SQL» после обновления с EF 6.1.1 для 6.1.2

EF 6.1.2

EF 6.1.2 среда выполнения была выпущена в NuGet в декабре 2014 года. Эта версия предназначена главным образом о исправления ошибок. Мы также принято несколько значимых изменений от членов сообщества:

- **Можно настроить параметры кэша запроса из файла app/web.configuration**

```
<entityFramework>
  <queryCache size='1000' cleaningIntervalInSeconds=' -1' />
</entityFramework>
```

- Методы **SqFile** и **SqResource** на **DbMigration** позволяют запускать SQL скрипта сохраняется в файле или внедренный ресурс.

EF 6.1.1

EF 6.1.1 среда выполнения была выпущена в NuGet в июне 2014 года. Эта версия содержит исправления проблем, которые столкнулись с нескольких человек. Среди прочего:

- Конструктор: Ошибка при открытии EF5 edmx с точностью в конструкторе EF6
- По умолчанию логика определения нахождения экземпляра LocalDB не работает с SQL Server 2014

EF 6.1.0

EF 6.1.0 среда выполнения была выпущена в NuGet в марте 2014 г. Это незначительное обновление содержит значительное число новых функций:

- **Средства консолидации** предоставляет согласованный способ создания новой модели EF. Эта функция [расширяет мастер модели EDM ADO.NET для поддержки создания моделей Code First](#), включая реконструирование из существующей базы данных. Эти функции ранее были доступны в бета-версии качества в EF Power Tools.
- **Обработка сбоев фиксации транзакции** предоставляет CommitFailureHandler, использующий новых способность перехватывать операции транзакции. CommitFailureHandler обеспечивает автоматическое восстановление после сбоев подключения во время фиксации транзакции.
- **IndexAttribute** позволяет индексы, чтобы указать, поместив `[Index]` атрибут свойство (или свойства) в модели Code First. Код сначала создаст соответствующий индекс в базе данных.
- **Сопоставление общедоступного API** предоставляет доступ к сведениям, имеет EF в сопоставлении свойств и типов, со столбцами и таблицами в базе данных. В предыдущих версиях этот API был внутренней.
- **Возможность настройки перехватчики посредством файла App/Web.config** позволяет перехватчики должен быть добавлен без повторной компиляции приложения.
- **System.Data.Entity.Infrastructure.Interception.DatabaseLogger** — это новый interceptor, который упрощает процесс в журнал всех операций базы данных в файл. В сочетании с предыдущей функции, это позволяет легко [включить ведение журнала операций базы данных для развернутого приложения](#), без необходимости повторной компиляции.
- **Обнаружение изменений модели миграций** была улучшена так, чтобы точнее; также улучшена производительность процесса обнаружения изменений шаблонный миграций.
- **Повышение производительности** включая операции ограниченной базы данных во время инициализации, оптимизации для сравнения на равенство null в запросах LINQ, быстрее Просмотр поколения (Создание модели) в различных сценариях и эффективнее Материализация отслеживаемые сущности с несколькими связями.

EF 6.0.2

EF 6.0.2 среда выполнения была выпущена в NuGet в декабря 2013 г. Этот выпуск исправления ограничено решение проблем, которые были введены в версии EF6 (регрессии в производительности или поведения с момента EF5).

EF 6.0.1

EF 6.0.1 среда выполнения была выпущена в NuGet в октябре 2013 г. одновременно с EF 6.0.0, так как последний внедрен в версиях Visual Studio, которая заблокирована несколько месяцев до. Этот выпуск исправления ограничено решение проблем, которые были введены в версии EF6 (регрессии в производительности или поведения с момента EF5). Чтобы устранить некоторые проблемы с

производительностью во время прогрева для моделей EF были наиболее существенные изменения. Это было важно, так как производительность прогрева был область фокуса в EF6 и эти проблемы Инверсия некоторые из других средств повышения производительности в EF6.

EF 6.0

EF 6.0.0 среда выполнения была выпущена в NuGet в октябре 2013 г. Это первая версия, в котором полные исполняющую среду EF включается в [пакета NuGet EntityFramework](#) которой не зависит от EF биты, которые являются частью платформы .NET Framework. Перемещение остальных компонентов среды выполнения на пакет NuGet требуется ряд критическое изменение для существующего кода. См. в разделе [обновление до платформы Entity Framework 6](#) узнать больше о ручного действия, необходимые для обновления.

Этот выпуск содержит множество новых функций. Для моделей, созданных с помощью Code First или конструкторе EF работают следующие функции:

- **Асинхронный запрос и сохранить** добавлена поддержка основанные на задачах асинхронные шаблоны, которые были введены в .NET 4.5.
- **Устойчивость подключения** обеспечивает автоматическое восстановление после сбоев временных соединений.
- **Конфигурация на основе кода** дает возможность выполнять конфигурации, — обычно в файле конфигурации — в коде.
- **Разрешение зависимостей** появилась поддержка для поиска служб шаблон и мы выделен некоторые части функциональных возможностей, который можно заменить на пользовательские реализации.
- **Ведение журнала перехвата/SQL** предоставляет низкоуровневые строительные блоки для перехвата операций EF с простой ведения журнала SQL, созданных в верхней части.
- **Улучшение возможностей тестирования** облегчают создание тестовых дублеров для DbContext и DbSet при [использование платформы имитированной](#) или [написав свой собственный тестовых дублеров](#).
- **Теперь можно создать DbContext с DbConnection, который уже открыт** которого позволяет реализовать сценарии, где было бы полезно, если соединение может быть открыт, при создании контекста (таких как совместное использование соединения между компонентами где Вы можете нельзя гарантировать состояние подключения).
- **Улучшенная поддержка транзакций** обеспечивает поддержку транзакций, внешним по отношению к framework, а также улучшенные средства создания транзакции, в структуре.
- **Перечисления, пространственный и повышения производительности в .NET 4.0** - переместив основные компоненты, которые ранее были в .NET Framework в пакете EF NuGet, теперь мы можем предложить поддержке нумерации, Пространственные типы данных и улучшения производительности из EF5 на платформе .NET 4.0.
- **Улучшена производительность Enumerable.Contains в запросах LINQ.**
- **Улучшенная прогрева, время (создание представлений)**, особенно для больших моделей.
- **Преобразование во множественную форму подключаемые & службы единичную форму и настраиваемых.**
- **Пользовательские реализации Equals и GetHashCode** сущности теперь поддерживаются классы.
- **DbSet.AddRange/RemoveRange** предоставляет оптимальный способ для добавления или удаления нескольких сущностей из набора.
- **DbChangeTracker.HasChanges** обеспечивает простой и эффективный способ узнать, есть все ожидающие изменения, сохраняется в базе данных.
- **Sq|CeFunctions** предоставляет SQL Compact эквивалентно SqlFunctions.

Следующие функции применяются только к Code First.

- **Пользовательские соглашения о коде первый** позволяют писать собственные правила, чтобы

избежать повторяющихся конфигураций. Мы предоставляем простой API для упрощенного соглашения, а также некоторые более сложные стандартных блоков позволяет вам создавать более сложные правила.

- **Код первого сопоставления хранимые процедуры Insert/Update/Delete** теперь поддерживается.
- **Идемпотентные сценарии миграции** позволяют создавать скрипт SQL, который можно обновить базу данных в любой версии до последней версии.
- **Можно настроить таблицу журнала миграции** позволяет настроить определение таблицы журнала миграции. Это особенно полезно для поставщиков базы данных, требующих соответствующие типы данных и т.д. для таблицы журнала миграции для правильной работы.
- **Несколько контекстов на одну базу данных** сняты предыдущие ограничения для одной модели Code First на базу данных при использовании миграции, или когда Code First автоматически создает базу данных для вас.
- **DbModelBuilder.HasDefaultSchema** — это новый API первый код, который позволяет схему базы данных по умолчанию для модели Code First настраиваться в одном месте. Ранее схема по умолчанию Code First была жестко "dbo" и единственным способом настроить схему, которой принадлежала таблицы было через totable-API.
- **Метод DbModelBuilder.Configurations.AddFromAssembly** позволяет легко добавлять все классы конфигурации, определенные в сборке, при использовании классов конфигурации с Fluent API для Code First.
- **Пользовательские операции миграции** можно было добавить дополнительные операции для использования в вашей миграции на базе кода.
- **Уровень изоляции транзакций по умолчанию меняется на READ_COMMITTED_SNAPSHOT** для баз данных, созданных с помощью Code First, что обеспечивает большую масштабируемость и меньшее количество взаимоблокировок.
- **Сущности и сложные типы теперь могут быть nested inside классы.** |

EF 5.0

EF 5.0.0 среда выполнения была выпущена в NuGet в августе 2012. Этой версии появился ряд новых возможностей, включая поддержку нумераций, возвращающие табличные значения функции, Пространственные типы данных и различные улучшения производительности.

Entity Framework Designer в Visual Studio 2012 также добавлена поддержка нескольких схем, на модель, выделение цветом фигур на поверхности и пакетной службы импорта проектирования хранимых процедур.

Ниже приведен список содержимого, которые мы собрали специально для выпуска EF 5.

- [Post EF 5 выпуск](#)
- Новые возможности в EF5
 - [Поддержку перечисления в коде в первую очередь](#)
 - [Поддержку перечисления в конструкторе EF](#)
 - [Типы пространственных данных в коде сначала](#)
 - [Типы пространственных данных в конструкторе EF](#)
 - [Поддержка пространственных типов](#)
 - [Функции с табличным значением](#)
 - [Несколько диаграмм на модель](#)
- Настройка модели
 - [Создание модели](#)
 - [Подключения и моделей](#)
 - [Особенности производительности](#)
 - [Работа с Microsoft SQL Azure](#)
 - [Параметры файла конфигурации](#)

- Глоссарий
- Code First
 - Code First в новой базе данных (Пошаговое руководство и видео)
 - Code First для существующей базы данных (Пошаговое руководство и видео)
 - Соглашения
 - Заметки к данным
 - Fluent API — Настройка и сопоставления типов & свойств
 - Fluent API — Настройка связей
 - Fluent API с использованием VB.NET
 - Code First Migrations
 - Автоматическое Code First Migrations
 - Migrate.exe
 - Определение DbSets
- Конструктор EF
 - Первый модели (Пошаговое руководство и видео)
 - Database First (Пошаговое руководство и видео)
 - Сложные типы
 - Связи и связи
 - Шаблон наследования ТРТ
 - Шаблон наследование ТРН
 - Запрос с помощью хранимых процедур
 - Хранимые процедуры с несколькими результирующими наборами
 - Вставить, обновить и удалить с помощью хранимых процедур
 - Сопоставить сущности с несколькими таблицами (разделение сущностей)
 - Сопоставление нескольких сущностей в одной таблице (разделение таблицы)
 - Определение запросов
 - Шаблоны создания кода
 - Возврат к ObjectContext
- С помощью вашей модели
 - Работа с DbContext
 - Запросы и поиск сущности
 - Работа со связями
 - Загрузка связанных сущностей
 - Работа с локальными данными
 - N-уровневых приложениях
 - Необработанные SQL-запросы
 - Шаблоны оптимистичного параллелизма
 - Работа с прокси
 - Автоматическое обнаружение изменений
 - Отключение отслеживания запросов
 - Метод Load
 - Добавить или присоединить и состояния сущностей
 - Работа со значениями свойств
 - Привязка данных с помощью WPF (Windows Presentation Foundation)
 - Привязка данных с помощью WinForms (Windows Forms)

EF 4.3.1

EF 4.3.1 среда выполнения была выпущена в NuGet в февраль 2012 вскоре после EF 4.3.0. Этот выпуск исправления включены некоторые исправления ошибок, до версии EF 4.3 и появилась Улучшенная поддержка LocalDB клиентам, использующим EF 4.3 с Visual Studio 2012.

Ниже приведен список содержимого, которые мы собрали специально для выпуска EF 4.3.1, большая часть содержимого для EF 4.1 по-прежнему применяется к EF 4.3 также.

- [EF 4.3.1 выпуска записи блога](#)

EF 4.3

EF 4.3.0 среда выполнения была выпущена в NuGet в февраль 2012 г. Этот выпуск включены новой функции Code First Migrations, позволяет базе данных, созданной механизмом Code First, постепенно изменять по мере развития модели Code First.

Ниже приведен список содержимого, которые мы собрали специально для выпуска EF 4.3, большая часть содержимого для EF 4.1 по-прежнему применяется к EF 4.3 также:

- [Post 4.3 выпуска EF](#)
- [Пошаговое руководство по EF 4.3 миграция на базе кода](#)
- [Пошаговое руководство 4.3 автоматический перенос EF](#)

EF 4.2

EF 4.2.0 среда выполнения была выпущена в NuGet в ноябре 2011 г. Этот выпуск включает исправления ошибок для EF 4.1.1 выпуска. Так как этот выпуск включены только исправления ошибок, могло бы быть EF 4.1.2 patch выпуска, но мы выбрали переместиться 4.2 позволит нам отойти от даты, на основе номера версии исправлений, мы использовали в 4.1.x освобождает и внедрять [семантической Versionsing](#) standard для семантического управления версиями.

Вот список содержимого, которые мы собрали специально для выпуска EF 4.2, содержимое, указанное для EF 4.1 по-прежнему применяется к EF 4.2 также.

- [Post 4.2 выпуска EF](#)
- [Код первого пошагового руководства](#)
- [Модель & базы данных первого пошагового руководства](#)

EF 4.1.1

EF 4.1.10715 среда выполнения была выпущена в NuGet в июле 2011 г. Помимо исправления ошибок этот выпуск исправления появились некоторые компоненты, чтобы упростить для времени разработки, средства для работы с моделью Code First. Эти компоненты используются Code First Migrations (входит в EF 4.3) и EF Power Tools.

Вы заметите, что версия странное число 4.1.10715 пакета. Мы использовали для использования версии исправлений на основе даты, прежде чем мы решили внедрить [семантического управления версиями](#). Можно рассматривать этой версии, как EF 4.1 исправление 1 (или EF 4.1.1).

Ниже приведен список содержимого, мы собрали для 4.1.1 выпуска:

- [EF 4.1.1 выпуска Post](#)

EF 4.1

EF 4.1.10331 среди выполнения был первым опубликованы в NuGet, а в апреле 2011 г. Этот выпуск включен упрощенный API DbContext и Code First рабочего процесса.

Обратите внимание, номер версии странной, 4.1.10331, в которых действительно должен был 4.1. Кроме того, имеется 4.1.10311 версии, в которых должен был 4.1.0-rc ('rc' означает «версия-кандидат»). Мы использовали для использования версии исправлений на основе даты, прежде чем мы решили внедрить [семантического управления версиями](#).

Ниже приведен список содержимого, которые мы собрали для версии 4.1. Большая часть будет по-прежнему относится к более поздних версиях Entity Framework:

- [Post выпуск 4.1 EF](#)
- [Код первого пошагового руководства](#)
- [Модель & базы данных первого пошагового руководства](#)
- [SQL Azure федераций и Entity Framework](#)

EF 4.0

Этот выпуск был включен в .NET Framework 4 и Visual Studio 2010 в апреле 2010. Важных новых функций в этом выпуске включены РОСО поддержки, внешнего сопоставления ключей, отложенная загрузка, улучшение возможностей тестирования, настраиваемую генерацию кода и Model First рабочего процесса.

Несмотря на то, что он был второй выпуск платформы Entity Framework, он именуется EF 4 в соответствии с версией .NET Framework, поставляемой вместе с. После этого выпуска мы запуска размещения на NuGet Entity Framework и применяет семантического управления версиями, так как мы больше не были привязаны к версии платформы .NET Framework.

Обратите внимание, что некоторые последующие версии платформы .NET Framework отправки со значительными обновлениями на биты, включенные в EF. На самом деле многие из новых функций EF 5.0 были реализованы в виде улучшений на эти биты. Тем не менее, чтобы рационализировать истории управления версиями для Entity FRAMEWORK, мы продолжаем для ссылки на биты EF, которые являются частью платформы .NET Framework как EF 4.0 среда выполнения, а все более новые версии состоят из [пакета NuGet EntityFramework](#).

EF 3.5

Исходная версия Entity Framework был включен в пакет обновления 1 для .NET 3.5 и Visual Studio 2008 SP1, выпущенные в августе 2008 г. Этот выпуск предоставляет базовой поддержки O/RM, с помощью базы данных первого рабочего процесса.

Обновление до платформы Entity Framework 6

13.09.2018 • 7 minutes to read • [Edit Online](#)

В предыдущих версиях EF код был разделить между основными библиотеками (в основном System.Data.Entity.dll), в состав платформы .NET Framework и библиотеки (OOB)-каналу (в основном EntityFramework.dll) поставляется в виде пакета NuGet. EF6 принимает код из библиотеки ядра и включает его в библиотеках OOB. Это потребовалось, чтобы разрешить EF сделать открытым исходным кодом и для того, чтобы иметь возможность развиваться в разных темпе с .NET Framework. Вследствие этого является то, что приложения должны быть перестроено на перемещенные типы.

Это должно быть простым для приложений, которые используют DbContext как поставляемый в EF 4.1 и более поздних версий. Немного больше работы требуется для приложения, использующие класса ObjectContext, но он по-прежнему не трудно.

Ниже приведен контрольный список задач, которые необходимо выполнить для обновления существующего приложения в EF6.

1. Установите пакет EF6 NuGet

Необходимо обновить до новой среды выполнения Entity Framework 6.

1. Щелкните правой кнопкой мыши проект и выберите **управление пакетами NuGet...**
2. В разделе **Online** вкладке выберите **EntityFramework** и нажмите кнопку **установки**

NOTE

Если предыдущая версия EntityFramework NuGet, пакет установлен это обновит ее к EF6.

Также можно запустить следующую команду из консоли диспетчера пакетов:

```
Install-Package EntityFramework
```

2. Убедитесь, что удалены ссылки на сборки для System.Data.Entity.dll

Установка пакета EF6 NuGet автоматически следует удалить все ссылки на System.Data.Entity из проекта для вас.

3. Замените любые модели EF конструктора (EDMX), создания кода EF 6.x

При наличии модели, созданные в конструкторе EF, будет необходимо обновить шаблоны создания кода, чтобы создать совместимый код EF6.

NOTE

Доступны в настоящее время только EF 6.x DbContext генератор шаблонов для Visual Studio 2012 и 2013.

1. Удалите существующие шаблоны создания кода. Эти файлы обычно называется

`<edmx_file_name>.tt` и `<edmx_file_name>.Context.tt` и быть вложен в узел файла edmx в обозревателе решений. Можно выбрать шаблоны, в обозревателе решений и нажмите клавишу **Del** ключа для их удаления.

NOTE

В проектах веб-сайт шаблоны будут не вложен в файле edmx, но в списке с его в обозревателе решений.

NOTE

В проектах VB.NET необходимо включить «Показать все файлы» иметь возможность см. в файлах вложенный шаблон.

2. Добавьте соответствующий шаблон генерации кода EF 6.x. Откройте модель в конструкторе EF, щелкните правой кнопкой мыши область конструктора и выберите **добавить элемент формирования кода...**

- Если вы используете API DbContext (рекомендуется), затем **EF 6.x генератор DbContext** будут доступны в разделе **данных** вкладки.

NOTE

Если вы используете Visual Studio 2012, необходимо установить инструменты EF 6, чтобы этот шаблон. См. в разделе [получение Entity Framework](#) подробные сведения.

- Если вы используете ObjectContext API, а затем будет необходимо выбрать **Online** вкладку и выполните поиск **EF 6.x EntityObject Generator**.

3. Если вы применили все настройки, шаблоны создания кода, вам потребуется повторно применить их к в обновленных шаблонах.

4. Обновление пространства имен для использования типов EF core

Пространства имен для типов DbContext и Code First не изменились. Это означает, что для многих приложений, использующих EF 4.1 или более поздней версии не нужно ничего менять.

Типы как ObjectContext, которые ранее находились в System.Data.Entity.dll были перемещены в новые пространства имен. Это означает, что может потребоваться обновить ваш *с помощью или импорта* директивы для сборки с EF6.

Общее правило для изменения пространства имен является то, что любой тип в System.Data.* перемещается в System.Data.Entity.Core.*. Другими словами, просто вставьте **Entity.Core**. После System.Data. Пример:

- System.Data.EntityException = > System.Data. **Entity.Core**. EntityException
- System.Data.Objects.ObjectContext = > System.Data. **Entity.Core**. Objects.ObjectContext
- System.Data.Objects.DataClasses.RelationshipManager = > System.Data. **Entity.Core**. Objects.DataClasses.RelationshipManager

Эти типы находятся в Core пространства имен, так как они не используются непосредственно для большинства приложений на основе DbContext. Некоторые типы, которые были частью System.Data.Entity.dll по-прежнему используются часто, так и непосредственно для приложений на основе DbContext и поэтому не были перемещены в Core пространства имен. Эти особые значения приведены ниже.

- System.Data.EntityState = > System.Data. **Сущности.** EntityState
- System.Data.Objects.DataClasses.EdmFunctionAttribute = > System.Data. **Entity.DbFunctionAttribute**

NOTE

Этот класс был переименован; класс со старым именем все еще существует и работает, но он теперь помечен как устаревший.

- System.Data.Objects.EntityFunctions = > System.Data. **Entity.DbFunctions**

NOTE

Этот класс был переименован; класс со старым именем все еще существует и работает, но теперь помечены как устаревшие.)

- Пространственные классы (например, DbGeography, DbGeometry) перешли из System.Data.Spatial = > System.Data. **Сущности.** Пространственных

NOTE

Некоторые типы в пространстве имен System.Data находятся в System.Data.dll, которая не является сборкой EF. Эти типы не были перемещены и поэтому их пространства имен остаются неизменными.

Выпуски Visual Studio

13.09.2018 • 7 minutes to read • [Edit Online](#)

Мы рекомендуем всегда использовать последнюю версию Visual Studio, так как она содержит новейшие средства для .NET, NuGet и Entity Framework. На самом деле различные примеры и пошаговые руководства по документации по Entity Framework предполагается, что вы используете последнюю версию Visual Studio.

Это возможно, однако для использования с различными версиями платформы Entity Framework более старых версиях Visual Studio до тех пор, пока вы учетной записи некоторые различия:

Visual Studio 2017 15.7 и более поздних версий

- Эта версия Visual Studio включает в себя последнюю версию средства платформы Entity Framework и среды выполнения EF 6.2 и не требует дополнительных шагов настройки. См. в разделе [новые](#) Дополнительные сведения об этих выпусках.
- Добавление платформы Entity Framework к новым проектам, со средствами EF автоматически добавляет пакет EF 6.2 NuGet. Можно вручную установить или обновить до любой пакет EF NuGet, доступный через Интернет.
- По умолчанию экземпляр SQL Server, доступные в этой версии Visual Studio — это экземпляр LocalDB, вызывается MSSQLLocalDB. В разделе сервера следует использовать строки подключения «(localdb)\MSSQLLocalDB». Не забывайте использовать буквальная строка с префиксом `@` или двойной обратной косой черты "\\\" при указании строку подключения в коде C#.

Visual Studio 2015 до Visual Studio 2017 версии 15.6

- Эти версии Visual Studio включают средства платформы Entity Framework и среды выполнения 6.1.3. См. в разделе [последние выпуски](#) Дополнительные сведения об этих выпусках.
- Добавление платформы Entity Framework к новым проектам, со средствами EF автоматически добавит EF 6.1.3 пакет NuGet. Можно вручную установить или обновить до любой пакет EF NuGet, доступный через Интернет.
- По умолчанию экземпляр SQL Server, доступные в этой версии Visual Studio — это экземпляр LocalDB, вызывается MSSQLLocalDB. В разделе сервера следует использовать строки подключения «(localdb)\MSSQLLocalDB». Не забывайте использовать буквальная строка с префиксом `@` или двойной обратной косой черты "\\\" при указании строку подключения в коде C#.

Visual Studio 2013

- Эта версия Visual Studio включает и более старую версию средства платформы Entity Framework и среды выполнения. Рекомендуется обновить средства платформы Entity Framework 6.1.3, с помощью [установщик](#) доступны в центре загрузки Майкрософт. См. в разделе [последние выпуски](#) Дополнительные сведения об этих выпусках.
- Добавление новых проектов с помощью обновленных средств EF Entity Framework автоматически добавит EF 6.1.3 пакет NuGet. Можно вручную установить или обновить до любой пакет EF NuGet, доступный через Интернет.
- По умолчанию экземпляр SQL Server, доступные в этой версии Visual Studio — это экземпляр LocalDB, вызывается MSSQLLocalDB. В разделе сервера следует использовать строки подключения «(localdb)\MSSQLLocalDB». Не забывайте использовать буквальная строка с префиксом `@` или двойной обратной косой черты "\\\" при указании строку подключения в коде C#.

Visual Studio 2012

- Эта версия Visual Studio включает и более старую версию средства платформы Entity Framework и среды выполнения. Рекомендуется обновить средства платформы Entity Framework 6.1.3, с помощью [установщик](#) доступны в центре загрузки Майкрософт. См. в разделе [последние выпуски](#) Дополнительные сведения об этих выпусках.
- Добавление новых проектов с помощью обновленных средств EF Entity Framework автоматически добавит EF 6.1.3 пакет NuGet. Можно вручную установить или обновить до любой пакет EF NuGet, доступный через Интернет.
- По умолчанию экземпляр SQL Server, доступные в этой версии Visual Studio — это экземпляр LocalDB, вызывается v11.0. В разделе сервера следует использовать строки подключения «(localdb)\v11.0». Не забывайте использовать буквальная строка с префиксом `@` или двойной обратной косой черты «\\» при указании строку подключения в коде C#.

Visual Studio 2010

- Версии доступны средства платформы Entity Framework с данной версией Visual Studio несовместим со средой выполнения Entity Framework 6 и не могут быть обновлены.
- По умолчанию средства платформы Entity Framework будет добавлять в проекты Entity Framework 4.0. Чтобы создать приложения, использующие более новые версии EF, необходимо сначала установить [расширение диспетчера пакетов NuGet](#).
- По умолчанию все создание кода в версии инструментов EF основан на EntityObject и Entity Framework 4. Мы рекомендуем переключиться формирования кода должен быть основан на DbContext и Entity Framework 5, установив шаблоны создания кода DbContext для [C#](#) или [Visual Basic](#).
- После установки расширения диспетчера пакетов NuGet, можно вручную установить или обновить до любой пакет EF NuGet, доступный через Интернет и использовать EF6 в режиме Code First, которая не требует конструктора.
- По умолчанию экземпляр SQL Server, доступные в этой версии Visual Studio — SQL Server Express с именем SQLEXPRESS. В разделе сервера следует использовать строки подключения «. \SQLEXPRESS». Не забывайте использовать буквальная строка с префиксом `@` или двойной обратной косой черты «\\» при указании строку подключения в коде C#.

Начало работы с платформой Entity Framework 6

12.10.2018 • 3 minutes to read • [Edit Online](#)

Это руководство содержит набор ссылок на отдельные статьи с документацией, пошаговые руководства и видео, которые помогут быстро начать работу.

Основы

- [Установка Entity Framework](#)

Здесь вы узнаете, как добавить Entity Framework в свои приложения. Если вы хотите использовать конструктор EF, установите его в Visual Studio.

- [Создание модели: Code First, конструктор EF и рабочие процессы EF](#)

Вы предпочтете задать модель EF, написав код или нарисовав поля и линии? Вы собираетесь использовать EF для сопоставления объектов с существующей базой данных или хотите создать в EF базу данных, настроенную для ваших объектов? Здесь вы узнаете о двух разных подходах к использованию EF6: конструктор EF и Code First. Обязательно подпишитесь на обсуждение и посмотрите видео о различиях между ними.

- [Работа с DbContext](#)

DbContext — это первый и самый важный тип EF, который вам нужно научиться использовать. Он служит источником запросов базы данных и отслеживает изменения, внесенные в объекты, чтобы их можно было сохранить в базе данных.

- [Вопросы](#)

Вы узнаете, как получить справку от экспертов и сможете сами отвечать на вопросы сообщества.

- [Участие](#)

В Entity Framework 6 используется модель разработки с открытым исходным кодом. Посетите репозиторий GitHub, чтобы узнать, что поможет улучшить EF.

Ресурсы по Code First

- [Code First для рабочего процесса существующей базы данных](#)
- [Code First для рабочего процесса новой базы данных](#)
- [Сопоставление перечислений с помощью Code First](#)
- [Сопоставление пространственных типов с помощью Code First](#)
- [Запись пользовательских соглашений Code First](#)
- [Настройка Code First для использования текущего API в Visual Basic](#)
- [Code First Migrations](#)
- [Code First Migrations в командных средах](#)
- [Автоматические миграции Code First Migrations \(больше не рекомендуется\)](#)

Ресурсы по конструктору EF

- [Рабочий процесс Database First](#)
- [Рабочий процесс Model First](#)

- Сопоставление перечислений
- Сопоставление пространственных типов
- Сопоставление наследования "одна таблица на иерархию"
- Сопоставление наследования "одна таблица на тип"
- Сопоставление хранимых процедур для обновлений
- Сопоставление хранимых процедур для запроса
- Разделение сущности
- Разделение таблицы
- Определение запроса (дополнительно)
- Функции с табличными значениями (дополнительно)

Другие источники

- Асинхронный запрос и сохранение
- Привязка данных с помощью WinForms
- Привязка данных с помощью WPF
- Отключенные сценарии с самоотслеживающимися сущностями (больше не рекомендуется)

Основы Entity Framework 6

13.09.2018 • 2 minutes to read • [Edit Online](#)

В этом разделе описываются различные базовые аспекты работы с моделью EF6.

Получение платформы Entity Framework

13.09.2018 • 3 minutes to read • [Edit Online](#)

Платформа Entity Framework состоит из средства EF для Visual Studio и среды выполнения EF.

Средства EF для Visual Studio

Инструменты Entity Framework для Visual Studio включают конструктор EF и мастера моделей EF и являются обязательными для базы данных сначала модели первый рабочих процессов. Средства EF включены во всех последних версиях Visual Studio. При выполнении настраиваемой установки Visual Studio, вам потребуется убедиться, что элемент «Инструменты Entity Framework 6» выбирается путем либо рабочая нагрузка, которая включает его или выбрав его в качестве отдельного компонента.

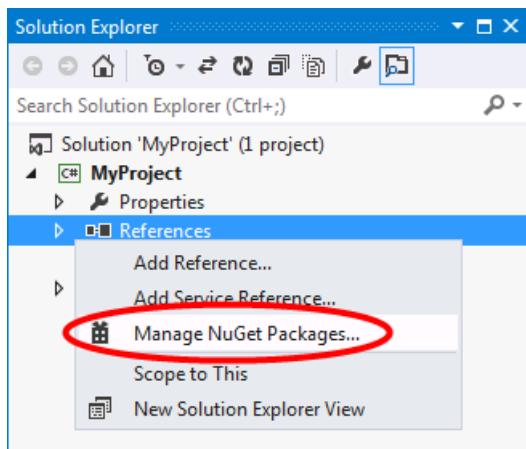
Для некоторых предыдущих версий Visual Studio обновленные средства EF доступны для загрузки. См. в разделе [версий Visual Studio](#) рекомендации о том, как получить последнюю версию средства EF, доступные для вашей версии Visual Studio.

Исполняющая среда EF

Последнюю версию Entity Framework доступен в виде [пакета EntityFramework NuGet](#). Если вы не знакомы с помощью диспетчера пакетов NuGet, мы рекомендуем вам ознакомиться [Обзор NuGet](#).

Установка пакета NuGet EF

Можно установить пакет EntityFramework щелкните правой кнопкой мыши **ссылки** папку проекта и выбрав **управление пакетами NuGet...**



Установка консоли диспетчера пакетов

Кроме того, можно установить, выполнив следующую команду в EntityFramework консоль диспетчера пакетов.

```
Install-Package EntityFramework
```

Установка определенной версии EF

Новые версии среды выполнения EF из EF версии 4.1 и более поздних версий были выпущены как [пакета NuGet EntityFramework](#). Любой из этих версий можно добавить в проект на основе .NET Framework, выполнив следующую команду в Visual Studio [консоль диспетчера пакетов](#):

```
Install-Package EntityFramework -Version <number>
```

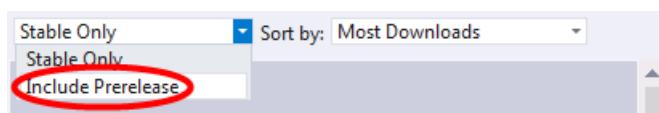
Обратите внимание, что `<number>` представляет определенную версию EF для установки. Например 6.2.0 — версия номер для EF 6.2.

EF сред выполнения, прежде чем 4.1 были частью .NET Framework и не может устанавливаться отдельно.

Установка последней предварительной версии

Указанные выше методы позволит получить последнюю версию полностью поддерживается версия Entity Framework. Часто возникают предварительных версиях Entity Framework, что мы будем рады вас опробовать и Присылайте ваши отзывы о доступных.

Чтобы установить последнюю предварительную версию можно выбрать EntityFramework **включить предварительный выпуск** в окне «Управление пакетами NuGet». Если доступны без предварительной версии вы автоматически получите последнюю версию полностью поддерживаемую версию Entity Framework.



Также можно запустить следующую команду [КОНСОЛЬ диспетчера пакетов](#).

```
Install-Package EntityFramework -Pre
```

Работа с DbContext

13.09.2018 • 6 minutes to read • [Edit Online](#)

Чтобы использовать Entity Framework для запроса, вставки, обновления и удаления данных, используя объекты .NET, необходимо сначала [создать модель](#) который сопоставляет сущности и связи, которые определены в модели с таблицами в базе данных.

При наличии модели является основным классом, ваше приложение взаимодействует с `System.Data.Entity.DbContext` (часто обозначается как класс контекста). Можно использовать связанные модели для DbContext:

- Запись и выполнение запросов
- Материализовать результаты запроса в форме объектов сущности
- Отслеживать изменения, внесенные в эти объекты
- Сохранить изменения объектов обратно в базе данных
- Привязка объектов в памяти к элементам управления пользовательского интерфейса

На этой странице приведены некоторые рекомендации по управлению класс контекста.

Определение класса, производного DbContext

Для работы с контекстом рекомендуется определить класс, производный от DbContext и обеспечивает доступ к свойствам DbSet, представляющие коллекцию сущностей, указанный в контексте. При работе с конструктором, EF, контекст будет создаваться автоматически. Если вы работаете в режиме Code First, будут обычно контексте самостоятельном написании.

```
public class ProductContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

Получив контекст, нужно запросить, добавить (с помощью `Add` или `Attach` методы) или удалить (с помощью `Remove`) сущностей в контексте через эти свойства. Доступ к `DbSet` свойство в объекте контекста представляет начальный запрос, возвращающий все сущности заданного типа. Обратите внимание на то, что только доступ к свойству не будет выполняться запрос. Запрос выполняется при:

- Запрос перечисляется с помощью инструкции `foreach` (C#) или `For Each` (Visual Basic).
- Запрос перечисляется операцией коллекции, такие как `ToArray`, `ToDictionary`, ИЛИ `ToList`.
- Операторы LINQ, такие как `First` или `Any` указываются в внешней части запроса.
- Вызван один из следующих методов: `Load` метод расширения, `DbEntityEntry.Reload`,
`Database.ExecuteSqlCommand`, И `DbSet<T>.Find`, если сущность с указанным ключом отсутствует уже загружены в контекст.

Время существования

Время существования контекста начинается, когда создается экземпляр и заканчивается, когда экземпляр удаляется или сборщиком мусора. Используйте **с помощью** Если необходимо, чтобы все ресурсы, контекст управляет непосредственно перед удалением в конце блока. При использовании **с помощью**, компилятор автоматически создает блок `try/finally` и вызывает удаление в **наконец** блока.

```
public void UseProducts()
{
    using (var context = new ProductContext())
    {
        // Perform data access using the context
    }
}
```

Ниже приведены некоторые общие рекомендации при принятии решения о времени существования контекста.

- При работе с веб-приложений, используйте экземпляр контекста каждого запроса.
- При работе с Windows Presentation Foundation (WPF) или Windows Forms, используйте экземпляр контекста на каждую форму. Это позволяет использовать функции отслеживания изменений предоставляет этот контекст.
- Если создан экземпляр контекста с контейнер внедрения зависимостей, он отвечает обычно контейнера — высвобождение контекста.
- Если контекст создается в коде приложения, не забывайте удалять контекст, когда он больше не требуется.
- При работе с длительно существующем контексте учитывайте следующее:
 - Несколько объектов и их ссылок при загрузке в память, может быстро увеличить потребление памяти контекста. Это может вызвать снижение производительности.
 - Контекст не является потокобезопасным, поэтому его не следует использовать совместно в нескольких потоках одновременно выполняют его работу.
 - Если исключение вызывает контекст быть в состоянии неустранимых, всего приложения может завершиться.
 - Вероятность возникновения проблем с параллелизмом возрастает по мере увеличения разрыва между временем запроса и временем обновления данных.

Подключения

По умолчанию контекст управляет подключениями к базе данных. Контекст открывает и закрывает соединения по мере необходимости. Например контекст открывает соединение для выполнения запроса и затем закрывает соединение, когда будут обработаны все результирующие наборы.

В некоторых случаях необходимо больше контроля над открытием и закрытием соединения. Например при работе с SQL Server Compact, часто рекомендуется поддерживать отдельные открытия подключения к базе данных в течение времени существования приложения для повышения производительности. Процессом можно управлять вручную с помощью свойства `Connection`.

Связи, свойства навигации и внешние ключи

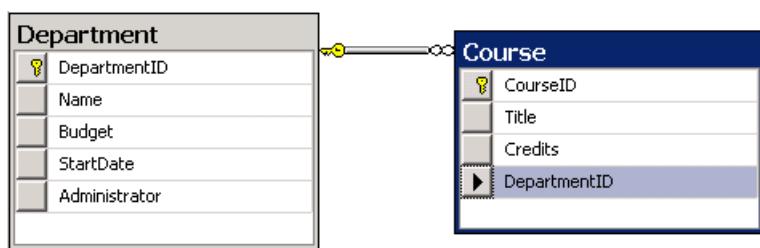
14.10.2018 • 15 minutes to read • [Edit Online](#)

Этот раздел содержит общие сведения о том, как платформа Entity Framework управляет связями между сущностями. Она также предоставляет некоторые рекомендации о том, как сопоставить и управлять связями.

Связи в EF

В реляционных базах данных связи (также называемый связи) между таблицами определяются через внешние ключи. Внешний ключ (FK) — столбец или сочетание столбцов, который используется для принудительного связи между данными в двух таблицах. Обычно существует три типа связей: один к одному, один ко многим и многие ко многим. В отношении "один ко многим" внешний ключ определен в таблице много окончания отношения. Связь многие ко многим включает в себя определением третьей таблицы (другое название таблицу соединения или соединения), первичный ключ состоит из внешних ключей из обоих связанных таблиц. В взаимно-однозначной связи Кроме того, как внешний ключ действует первичный ключ и отсутствует отдельный столбец внешнего ключа, для любой из таблиц.

На следующем рисунке две таблицы, участвующие в связи один ко многим. **Курс** таблица является зависимой таблицы, так как она содержит **DepartmentID** столбец, который связывает его с **отдел** таблицы.



В Entity Framework сущности могут быть связаны с другими сущностями через связь или связь. Каждое отношение содержит две конечные точки, описывающих тип сущности и кратность типа (один, ноль или один или множество) для двух сущностей в этой связи. Связи могут регулироваться ссылочное ограничение, описывающий, какие конечных элементов отношения относится к основной роли, а какой зависимой роли.

Свойства навигации предоставляют способ навигации по ассоциации между двумя типами сущностей. Каждый объект может обладать свойством навигации для каждого отношения, в котором участвует. Свойства навигации позволяют перейти связей и управление ими в обоих направлениях, возвращая объект ссылки (если атрибут кратности имеет один или ноль или один) или коллекции (Если кратность равна много). Также можно иметь односторонней навигации в этом случае только на одном из типов, участвует в связи, а не на оба определения свойства навигации.

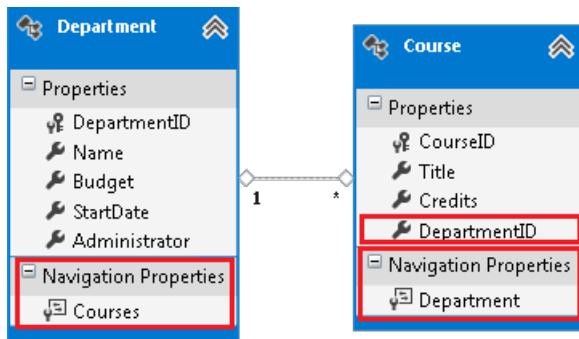
Рекомендуется, чтобы включить свойства в модели, которые сопоставляются с внешними ключами в базе данных. Включение свойств внешних ключей позволяет создавать или изменять отношение, изменяя значение внешнего ключа для зависимого объекта. Сопоставление такого типа называется сопоставлением на основе внешнего ключа. С помощью внешних ключей еще более важно при работе с отсоединенными сущностями. Обратите внимание, что при работе с 1-к-1 или 1-0... 1 связи отсутствует отдельный столбец внешнего ключа, со свойством первичного ключа выступает в качестве внешнего ключа и всегда включается в модель.

Если внешние ключевые столбцы не включены в модель, данные сопоставления управляется как независимый объект. Отношения отслеживаются с помощью ссылки на объекты вместо свойств внешнего ключа. Этот тип связи называется *независимым сопоставлением*. Наиболее распространенный способ

изменить независимом сопоставлении является изменение свойства навигации, которые создаются для каждой сущности, участвующей в ассоциации.

В модели можно использовать один или оба типа сопоставлений. Тем не менее если у вас есть чистые связь многие ко многим, связанной с таблице соединения, которая содержит только внешние ключи, EF будет использовать независимом сопоставлении для управления таких связей многие ко многим.

Ниже показана концептуальная модель, созданную с помощью Entity Framework Designer. Модель содержит две сущности, участвующие в связи один ко многим. Обе эти сущности имеют свойства навигации. **Курс** depend сущность и имеет **DepartmentID** определено свойство внешнего ключа.



В следующем фрагменте кода та же модель, созданную с помощью Code First.

```
public class Course
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public int DepartmentID { get; set; }
    public virtual Department Department { get; set; }
}

public class Department
{
    public Department()
    {
        this.Course = new HashSet<Course>();
    }
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public DateTime StartDate { get; set; }
    public int? Administrator { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

Настройка или сопоставление связей

Оставшуюся часть этой страницы описывается, как получить доступ и использовать данные с помощью связей. Сведения о настройке связи в модели см. следующие страницы.

- Для настройки связей в Code First, см. в разделе [записку к данным и Fluent API — связи](#).
- Настройка связи с помощью Entity Framework Designer, см. в разделе [связи в конструкторе EF](#).

Создание и изменение отношений

В ассоциации внешнего ключа, при изменении отношения состояние зависимого объекта с `EntityState.Unchanged` состояние изменяется на `EntityState.Modified`. В отношении независимых изменение отношения не обновляет состояние зависимого объекта.

Следующие примеры демонстрируют использование свойств внешнего ключа и свойства навигации для сопоставления связанных объектов. С помощью ассоциации внешних ключей каждый из этих методов можно использовать для изменения, создания или изменения связей. Для независимых сопоставлений нельзя использовать свойство внешнего ключа.

- Присваивая новое значение свойству внешнего ключа, как показано в следующем примере.

```
course.DepartmentID = newCourse.DepartmentID;
```

- Следующий код удаляет связь, задавая внешний ключ **null**. Обратите внимание, что свойство внешнего ключа должен допускать значение NULL.

```
course.DepartmentID = null;
```

NOTE

Если ссылка находится в добавленном состоянии (в этом примере объект курса), свойство навигации ссылки не будут синхронизированы с значениями ключа нового объекта, пока не вызове метода `SaveChanges`.

Синхронизация не выполняется, поскольку контекст объекта не содержит постоянных ключей для добавленных объектов, пока они не будут сохранены. Если необходимо выполнить полную синхронизацию сразу же установите связь новых объектов, используйте один из следующих методов.*

- С помощью присваивания нового объекта свойству навигации. Следующий код создает связь между курс и `department`. Если объекты присоединены к контексту, `course` также добавляется `department.Courses` коллекции и соответствующим внешним ключевое свойство на `course` объекта присваивается значение ключевого свойства отдела.

```
course.Department = department;
```

- Чтобы удалить связь, присвойте свойству навигации `null`. Если вы работаете с платформой Entity Framework, который основан на .NET 4.0, связанный элемент должен быть загружен, прежде чем ему присвоено значение `null`. Пример:

```
context.Entry(course).Reference(c => c.Department).Load();
course.Department = null;
```

Начиная с Entity Framework 5.0, который основан на .NET 4.5, можно задать связь значение `null` без загрузки связанного окончания. Можно также задать текущее значение `NULL`, используя следующий метод.

```
context.Entry(course).Reference(c => c.Department).CurrentValue = null;
```

- Путем удаления или добавления объекта в коллекцию сущностей. Например, можно добавить объект типа `Course` для `department.Courses` коллекции. Эта операция создает отношение между конкретным **курс** и конкретным `department`. Если объекты присоединены к контексту, ссылку на подразделение и свойства внешнего ключа на **курс** будет установлен объект к соответствующему `department`.

```
department.Courses.Add(newCourse);
```

- С помощью `ChangeRelationshipState` метод, чтобы изменить состояние указанного отношения между

двумя объектами сущностей. Этот метод наиболее часто используется при работе с N-уровневые приложения и независимом сопоставлении (он не может использоваться с сопоставлением на основе внешнего ключа). Кроме того, для использования этого метода необходимо удалить вниз до `ObjectContext`, как показано в следующем примере.

В следующем примере устанавливается отношение "многие ко многим" между инструкторов и курсы. Вызов `ChangeRelationshipState` и передав `EntityState.Added` параметр, позволяет `SchoolContext` знать, что добавлен связь между двумя объектами:

```
((IObjectContextAdapter)context).ObjectContext.  
    ObjectStateManager.  
    ChangeRelationshipState(course, instructor, c => c.Instructor, EntityState.Added);
```

Обратите внимание, что при обновлении (не просто прибавляет) связи, необходимо удалить старые связь после добавления новой:

```
((IObjectContextAdapter)context).ObjectContext.  
    ObjectStateManager.  
    ChangeRelationshipState(course, oldInstructor, c => c.Instructor, EntityState.Deleted);
```

Синхронизация изменений между внешними ключами и свойствами навигации

При изменении отношения объектов, присоединенных к контексту с помощью одного из описанных выше методов Entity Framework необходимо синхронизировать внешние ключи, ссылки и коллекции. Платформа Entity Framework автоматически управляет Эта синхронизация (также известный как связь исправление up) для сущностей РОСО с прокси-серверами. Дополнительные сведения см. в разделе [работа с прокси](#).

При использовании сущностей РОСО без прокси-объектов, необходимо убедиться в том, **DetectChanges** метод вызывается для синхронизации связанных объектов в контексте. Обратите, внимание, что следующие API автоматически активирует **DetectChanges** вызова.

- `DbSet.Add`
- `DbSet.AddRange`
- `DbSet.Remove`
- `DbSet.RemoveRange`
- `DbSet.Find`
- `DbSet.Local`
- `DbContext.SaveChanges`
- `DbSet.Attach`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`
- Выполнение LINQ запросы к `DbSet`

Загрузка связанных объектов

В Entity Framework, наиболее часто используемых свойства навигации можно использовать для загрузки сущностей, которые связаны с возвращенной сущностью посредством определенной ассоциации. Дополнительные сведения см. в разделе [загрузка связанных объектов](#).

NOTE

В сопоставлении на основе внешнего ключа при загрузке связанного конечного элемента зависимого объекта связанный объект загружается на основе значения внешнего ключа зависимого объекта, находящегося на момент загрузки в памяти:

```
// Get the course where currently DepartmentID = 2.  
Course course2 = context.Courses.First(c=>c.DepartmentID == 2);  
  
// Use DepartmentID foreign key property  
// to change the association.  
course2.DepartmentID = 3;  
  
// Load the related Department where DepartmentID = 3  
context.Entry(course).Reference(c => c.Department).Load();
```

В независимом сопоставлении связанный конечный элемент зависимого объекта запрашивается на основе значения внешнего ключа зависимого объекта, находящегося на момент загрузки в базе данных. Тем не менее если отношение изменено и ссылочное свойство зависимого объекта указывает на разные основного объекта, который загружается в контексте объекта Entity Framework будет пытаться создать отношение, как он определен на клиенте.

Управление параллелизмом

Внешний ключ, и для независимых сопоставлений проверки параллелизма основаны на ключах сущностей и других свойствах сущностей, определенных в модели. При использовании EF конструктора для создания модели, задайте `ConcurrencyMode` атрибут **фиксированной** для указания, что свойство должно проверяться для параллелизма. При использовании Code First для определения модели используйте `ConcurrencyCheck` заметки на свойства, которые будут проверяться для параллелизма. При работе в режиме Code First можно также использовать `TimeStamp` заметки, чтобы указать, что свойство должно проверяться для параллелизма. В одном классе может иметь только одно свойство метки времени. Во-первых, это свойство карты кода к полю, не допускающие значения NULL в базе данных.

Мы рекомендуем всегда использовать ассоциации внешнего ключа при работе с сущностями, участвующие в проверке параллелизма и разрешения.

Дополнительные сведения см. в разделе [обработка конфликтов параллелизма](#).

Работа с перекрывающимися ключами

Перекрывающиеся ключи представляют собой составные ключи, некоторые из свойств в которых также являются частью другого ключа в сущности. Для независимых сопоставлений использовать перекрывающиеся ключи нельзя. Для изменения сопоставления на основе внешнего ключа, содержащей перекрывающиеся ключи, рекомендуется изменять значения внешнего ключа вместо использования ссылок на объекты.

Асинхронный запрос и сохраните

27.09.2018 • 10 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

EF6 появилась поддержка для асинхронного запроса и сохраните файл с [async и await ключевые слова](#), которые появились в .NET 4.5. Не все приложения могут выиграть от асинхронность, его можно использовать для повышения масштабируемости клиента времени отклика и сервера при обработке долго выполняющихся, сети или задачи ввода-вывода.

Когда действительно использовать асинхронный

В этом пошаговом руководстве предназначено для ознакомления с основными понятиями `async` способом, который позволяет легко определить разницу между выполнениями программы асинхронные и синхронные. В этом пошаговом руководстве не предназначен для иллюстрации этих основных сценариев где асинхронного программирования предоставляет преимущества.

Асинхронное программирование в основном ориентировано на освобождение текущему управляемому потоку (поток выполнение кода .NET) может выполнять другие задачи при ожидании операции, которая не требует любое время вычислений из управляемого потока. Например в процессе ядра СУБД обработки запроса нет ничего выполнять код .NET.

В клиентских приложениях (WinForms, WPF и т.д.) текущего потока используется для сохранения отклика пользовательского интерфейса во время выполнения асинхронной операции. В потоке позволяют обрабатывать другие входящие запросы - серверных приложений (ASP.NET и т.д.) это может уменьшить уровень использования памяти и/или увеличить пропускную способность сервера.

В большинстве приложений использование метода `async` даст без заметного преимущества и даже может привести к ухудшению результатов. Использование тестов, профилирования и здравый смысл, чтобы оценить влияние асинхронного программирования в конкретной ситуации, прежде чем зафиксировать его.

Ниже приведены некоторые дополнительные ресурсы, чтобы узнать о `async`:

- [Общие сведения о Brandon Bray async/await в .NET 4.5](#)
- [Асинхронное программирование](#) страниц в библиотеке MSDN
- [Как построить ASP.NET Web приложений с помощью Async](#) (включает в себя демонстрацию пропускной способности сервера)

Создание модели

Мы будем использовать [код первого рабочего процесса](#) для создания нашей модели и создания базы данных, однако асинхронные функции будет работать со всеми моделями EF, включая созданные в конструкторе EF.

- Создайте консольное приложение и присвойте ему **AsyncDemo**
- Добавление пакета EntityFramework NuGet
 - В обозревателе решений щелкните правой кнопкой мыши **AsyncDemo** проекта

- Выберите **управление пакетами NuGet...**
- В диалоговом окне «Управление пакетами NuGet» выберите **Online** вкладку и выберите **EntityFramework** пакета
- Нажмите кнопку **установки**
- Добавить **Model.cs** класс следующей реализацией

```
using System.Collections.Generic;
using System.Data.Entity;

namespace AsyncDemo
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}
```

Создание синхронная программа

Теперь, когда у нас есть модель EF, давайте напишем код, использует его для выполнения некоторых доступа к данным.

- Замените содержимое файла **Program.cs** следующим кодом

```

using System;
using System.Linq;

namespace AsyncDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            PerformDatabaseOperations();

            Console.WriteLine();
            Console.WriteLine("Quote of the day");
            Console.WriteLine(" Don't worry about the world coming to an end today... ");
            Console.WriteLine(" It's already tomorrow in Australia.");

            Console.WriteLine();
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        public static void PerformDatabaseOperations()
        {
            using (var db = new BloggingContext())
            {
                // Create a new blog and save it
                db.Blogs.Add(new Blog
                {
                    Name = "Test Blog #" + (db.Blogs.Count() + 1)
                });
                db.SaveChanges();

                // Query for all blogs ordered by name
                var blogs = (from b in db.Blogs
                            orderby b.Name
                            select b).ToList();

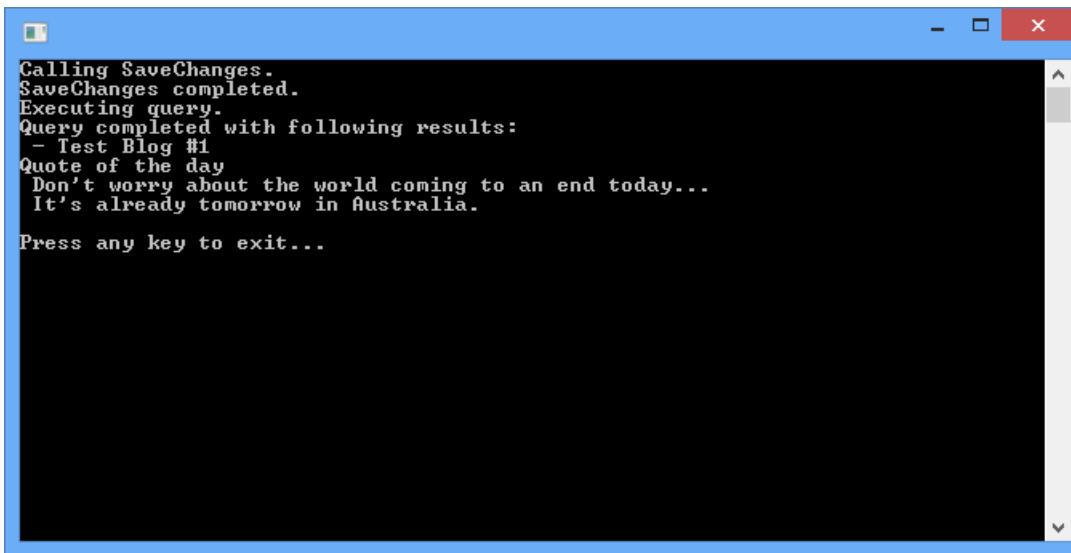
                // Write all blogs out to Console
                Console.WriteLine();
                Console.WriteLine("All blogs:");
                foreach (var blog in blogs)
                {
                    Console.WriteLine(" " + blog.Name);
                }
            }
        }
    }
}

```

Этот код вызывает **PerformDatabaseOperations** метод, который сохраняет новый **блог** в базу данных и затем извлекает все **блоги** из базы данных и выводит их на **Консоли**. После этого программа записывает знак кавычек дня для **консоли**.

Так как код является синхронным, мы заметили следующий поток выполнения, при запуске программы:

1. **SaveChanges** начинает отправлять новый **блог** к базе данных
2. **SaveChanges** завершения
3. Запрос для всех **блоги** отправляется в базу данных
4. Запрос возвращает и результаты записываются в **консоли**
5. Цитата дня записывается **консоли**



The screenshot shows a standard Windows application window with a title bar and a single text box containing the following output:

```
Calling SaveChanges.  
SaveChanges completed.  
Executing query.  
Query completed with following results:  
- Test Blog #1  
Quote of the day  
Don't worry about the world coming to an end today...  
It's already tomorrow in Australia.  
  
Press any key to exit...
```

Воплощение в асинхронный

Теперь, когда у нас есть нашей программы работу программ, мы можем начать использовать новый асинхронного программирования и ключевые слова ожидания. Мы внесли следующие изменения в файл Program.cs

- Строка 2: С помощью инструкции для **System.Data.Entity** пространство имен дает доступ к методам расширения `async EF`.
- Строки 4: С помощью инструкции для **System.Threading.Tasks** пространства имен позволяет использовать **задачи** типа.
- Строка 12 & 18: мы записи как задачу, которая отслеживает ход выполнения **PerformSomeDatabaseOperations** (строка 12) и затем блокирует выполнение программы для этой задачи до завершения одного раза всю работу для программы создается (строка 18).
- Строка 25: Мы включили обновления **PerformSomeDatabaseOperations** были помечены как **async** и вернуть **задачи**.
- Строка 35: Мы теперь вызова асинхронная версия метода `SaveChanges` и ожидает его завершения.
- Строки 42: Мы теперь вызова асинхронная версия `ToList` и ожидает от результата.

Полный список методов доступные расширения в пространстве имен `System.Data.Entity` ссылки на класс `QueryableExtensions`. *Также необходимо добавить «использование `System.Data.Entity`» с помощью инструкций.*

```

using System;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace AsyncDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var task = PerformDatabaseOperations();

            Console.WriteLine("Quote of the day");
            Console.WriteLine(" Don't worry about the world coming to an end today... ");
            Console.WriteLine(" It's already tomorrow in Australia.");

            task.Wait();

            Console.WriteLine();
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        public static async Task PerformDatabaseOperations()
        {
            using (var db = new BloggingContext())
            {
                // Create a new blog and save it
                db.Blogs.Add(new Blog
                {
                    Name = "Test Blog #" + (db.Blogs.Count() + 1)
                });
                Console.WriteLine("Calling SaveChanges.");
                await db.SaveChangesAsync();
                Console.WriteLine("SaveChanges completed.");

                // Query for all blogs ordered by name
                Console.WriteLine("Executing query.");
                var blogs = await (from b in db.Blogs
                                   orderby b.Name
                                   select b).ToListAsync();

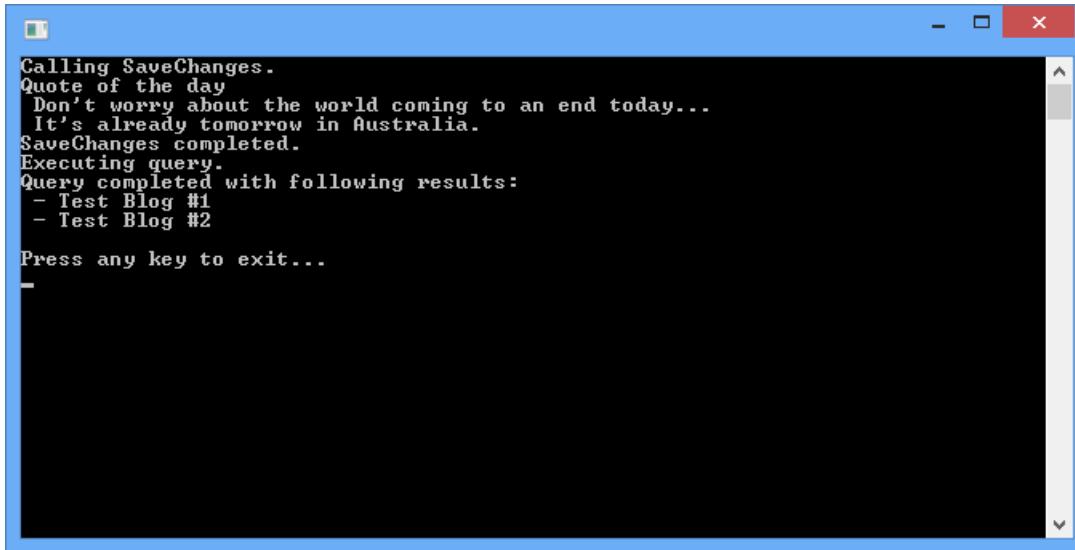
                // Write all blogs out to Console
                Console.WriteLine("Query completed with following results:");
                foreach (var blog in blogs)
                {
                    Console.WriteLine(" - " + blog.Name);
                }
            }
        }
    }
}

```

Теперь, когда код выполняется асинхронно, мы можем наблюдать поток выполнения при запуске программы:

1. **SaveChanges** начинает отправлять новый **блог** в базу данных *после отправки команды в базу данных больше нет вычислений, время, необходимое на текущего управляемого потока.*
PerformDatabaseOperations метод возвращает (даже если он еще не завершено выполнение) и продолжается выполнением программы в методе *Main*.
2. **Цитата дня записывается в консоль** *поскольку больше работы в методе *Main*, управляемый поток заблокирован на время ожидания вызова до завершения операции базы данных. После завершения оставшейся части нашей **PerformDatabaseOperations** * будет выполняться.

3. **SaveChanges** завершения
4. Запрос для всех **блоги** отправляется в базу данных. Опять же, управляемый поток может выполнять другую работу, пока запрос обрабатывается в базе данных. Так как все другие выполнения потока программы просто приостанавливает работу во время ожидания вызова `хотя`.
5. Запрос возвращает и результаты записываются в **консоли**



```
Calling SaveChanges.  
Quote of the day  
  Don't worry about the world coming to an end today...  
  It's already tomorrow in Australia.  
SaveChanges completed.  
Executing query.  
Query completed with following results:  
  - Test Blog #1  
  - Test Blog #2  
Press any key to exit...
```

Отсюда вывод:

Теперь мы увидели, насколько это просто чтобы сделать использование асинхронных методов платформы EF. Несмотря на то, что преимущества `async` не может быть ясно, с помощью простого консольного приложения, эти же стратегии можно применять в ситуациях, где долго выполняющиеся или привязкой сети действий может в противном случае блокировать приложение или привести к большим числом потоков. Увеличьте объем памяти.

Конфигурация на основе кода

13.09.2018 • 7 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Конфигурация приложения Entity Framework можно указать в файле конфигурации (app.config/web.config) или с помощью кода. Последний известен как конфигурация на основе кода.

Конфигурации в файле конфигурации описан в [отдельной статье](#). Файл конфигурации имеет приоритет над конфигурацией на основе кода. Другими словами Если параметр конфигурации в коде и в файле конфигурации, используется параметр в файле конфигурации.

С помощью DbConfiguration

Конфигурация на основе кода в EF6 и выше достигается путем создания подкласса `System.Data.Entity.Config.DbConfiguration`. Следующие рекомендации необходимо соблюдать при подклассификации `DbConfiguration`:

- Создайте только один класс `DbConfiguration` для вашего приложения. Этот класс задает параметры на уровне домена приложения.
- Поместите класс `DbConfiguration` в ту же сборку класса `DbContext`. (См. в разделе *перемещение DbConfiguration* Если вы хотите изменить этот раздел.)
- Присвойте классу `DbConfiguration` открытый конструктор без параметров.
- Задать параметры конфигурации, вызвав защищенных методов `DbConfiguration` в этот конструктор.

Следуя этим рекомендациям позволяет EF для обнаружения и автоматически использовать конфигурацию, оба средства, требуется доступ к модели и при запуске приложения.

Пример

Класс, производный от `DbConfiguration` может выглядеть следующим образом:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;

namespace MyNamespace
{
    public class MyConfiguration : DbConfiguration
    {
        public MyConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
            SetDefaultConnectionFactory(new LocalDBConnectionFactory("mssqllocaldb"));
        }
    }
}
```

Этот класс задает EF для использования стратегии выполнения SQL Azure — автоматический повтор операций поврежденной базы данных - и локальная база данных для базы данных, которые создаются в соответствии с соглашением с Code First.

Перемещение DbConfiguration

Бывают случаи, когда не может поместить класс DbConfiguration в ту же сборку класса DbContext. Например в разных сборках имеется два класса DbContext. Существует два варианта обработки такой.

Первый вариант — использовать файл конфигурации можно указать экземпляр DbConfiguration для использования. Чтобы сделать это, установите атрибут codeConfigurationType разделе entityFramework. Пример:

```
<entityFramework codeConfigurationType="MyNamespace.MyDbConfiguration, MyAssembly">
    ...
</entityFramework>
```

Значение codeConfigurationType должен быть сборки и полное имя пространства имен класса DbConfiguration.

Второй вариант — разместить DbConfigurationTypeAttribute на класс контекста. Пример:

```
[DbConfigurationType(typeof(MyDbConfiguration))]
public class MyContextContext : DbContext
{
}
```

Значение, передаваемое в атрибут может быть ваш тип DbConfiguration — как показано выше - или сборки и пространства имен полное имя типа string. Пример:

```
[DbConfigurationType("MyNamespace.MyDbConfiguration, MyAssembly")]
public class MyContextContext : DbContext
{
}
```

Явно настроить параметр DbConfiguration

Существуют ситуации, где конфигурации могут быть необходимы перед использует любого типа DbContext. Примеры включают:

- С помощью DbModelBuilder для построения модели без контекста
- С помощью другой служебная программа framework код, использующий DbContext, где используется этот контекст, прежде чем использовать контексте приложения

В таких ситуациях не удается автоматически обнаружить конфигурации EF и вместо этого необходимо выполнить одно из следующих:

- Задайте для типа DbConfiguration в файле конфигурации, как описано в разделе *перемещение DbConfiguration* выше
- Вызовите статический метод DbConfiguration.SetConfiguration во время запуска приложения

Переопределение DbConfiguration

Существуют ситуации, когда необходимо переопределяют конфигурацию, заданную в DbConfiguration. Обычно этого не сделать, разработчиками приложений, а скорее сторонних поставщиков и подключаемые

модули, которые нельзя использовать в производном классе DbConfiguration.

Для этого EntityFramework позволяет обработчик событий для регистрации, можно изменить текущую конфигурацию, непосредственно перед ее блокировкой. Он также предоставляет метод sugar специально для замены любая служба, возвращенный указатель служб EF. Это, как он предназначен для использования:

- При запуске приложения (до использования EF) подключаемого модуля или поставщик должен зарегистрировать метод обработчика событий для данного события. (Обратите внимание, что это должно произойти, прежде чем приложение использует EF).
- Обработчик событий вызывает ReplaceService для каждой службы, необходимо заменить.

Например replace IDbConnectionFactory и DbProviderService зарегистрировать обработчик примерно так:

```
DbConfiguration.Loaded += (_, a) =>
{
    a.ReplaceService<DbProviderServices>((s, k) => new MyProviderServices(s));
    a.ReplaceService<IDbConnectionFactory>((s, k) => new MyConnectionFactory(s));
};
```

В коде выше MyProviderServices и MyConnectionFactory представляют реализации службы.

Можно также добавить обработчики дополнительную зависимость, чтобы получить тот же эффект.

Обратите внимание, что можно также поместить DbProviderFactory таким образом, но это таким образом влияют только на EF и не использования DbProviderFactory вне EF. По этой причине, вероятно, стоит продолжить перенос DbProviderFactory, у вас есть до.

Следует также помнить службы, выполняемых извне приложения. Например, при выполнении миграции из консоли диспетчера пакетов. При выполнении миграции из консоли, предпринимается попытка найти ваш DbConfiguration. Тем не менее, зависит от того ли он получит в оболочку службу о том, где он зарегистрирован обработчик событий. Если он зарегистрирован как часть конструкции вашего DbConfiguration код должен выполняться, и служба должны получить оболочку. Обычно это работать не будет, и это означает, что инструменты не получить службу оболочку.

Параметры файла конфигурации

29.09.2018 • 11 minutes to read • [Edit Online](#)

Entity Framework дает ряд параметров, чтобы указать в файле конфигурации. В целом EF следует принципу «соглашение относительно настройки»: все параметры, описываемые в этой записи имеют поведение по умолчанию, необходимо только волноваться об изменении параметра, когда значение по умолчанию больше не удовлетворяет вашим требованиям.

Это альтернативное решение на уровне кода

Все эти параметры можно также применять, используя код. Начиная с версии EF6, мы представили [конфигурация на основе кода](#), который предоставляет основные способы применения конфигурации из кода. До EF6 конфигурация может применяться по-прежнему из кода, но вам нужно использовать различные API-интерфейсы для настройки различных областей. Параметр файла конфигурации позволяет легко изменять во время развертывания без обновления кода эти параметры.

Раздел конфигурации Entity Framework

Начиная с EF4.1 удалось задать инициализации базы данных для контекста с помощью **appSettings** раздел файла конфигурации. В EF 4.3 мы представили пользовательский **entityFramework** разделе для обработки новых параметров. Платформа Entity Framework по-прежнему будет распознавать инициализаторы базы данных, заданные с помощью старого формата, но мы рекомендуем перейти в новый формат, где это возможно.

EntityFramework раздел был автоматически добавлен в файл конфигурации проекта при установке пакета EntityFramework NuGet.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.3.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </configSections>
</configuration>
```

Строки подключения

[Эта страница](#) Дополнительные сведения о как Entity Framework определяет базу данных для использования, включая строки подключения в файле конфигурации.

Строки подключения перейдите в стандарте **connectionStrings** элемента и не требуют **entityFramework** раздел.

Модели кода, в первую очередь основано используйте обычные строки подключения ADO.NET. Пример:

```
<connectionStrings>
  <add name="BlogContext"
    providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Blogging;Integrated Security=True;" />
</connectionStrings>
```

Конструктор EF на основе строки подключения специальные EF моделей использования. Пример:

```
<connectionStrings>
  <add name="BlogContext"
    connectionString=
      "metadata=
        res://*/BloggingModel.csdl|
        res://*/BloggingModel.ssdl|
        res://*/BloggingModel.msl;
      provider=System.Data.SqlClient
      provider connection string=
        "data source=(localdb)\mssqllocaldb;
        initial catalog=Blogging;
        integrated security=True;
        multipleactiveresultsets=True;"
      providerName="System.Data.EntityClient" />
</connectionStrings>
```

Тип конфигурации на основе кода (для EF6 и выше)

Начиная с EF6, можно указать DbConfiguration для Entity FRAMEWORK для [конфигурация на основе кода](#) в приложении. В большинстве случаев не нужно указать этот параметр, как EF автоматически обнаружит ваш DbConfiguration. Подробные сведения о при необходимо указать в файле config DbConfiguration **перемещение DbConfiguration** раздел [конфигурация на основе кода](#).

Чтобы задать тип DbConfiguration, укажите имя типа с указанием сборки в **codeConfigurationType** элемент.

NOTE

Полное имя сборки — это полное имя пространства имен, используя запятую, затем сборку, тип находится в. При необходимости вы можете также указать версию сборки, язык и региональные параметры и маркер открытого ключа.

```
<entityFramework codeConfigurationType="MyNamespace.MyConfiguration, MyAssembly">
</entityFramework>
```

Поставщики EF баз данных (для EF6 и выше)

До EF6 приходилось включаемое как часть основной поставщик ADO.NET Entity Framework части поставщика базы данных. Начиная с EF6, конкретных частей EF теперь управляемых и зарегистрировать отдельно.

Обычно не нужно регистрировать поставщики самостоятельно. Это обычно выполняется поставщиком при его установке.

Поставщики регистрируются, включив **поставщика** элемента под **поставщики** дочернего **entityFramework** раздел. Существует два обязательных атрибута для записи поставщика:

- **invariantName** определяет основной поставщик ADO.NET, целевые объекты поставщика EF

- **Тип** — это имя типа с указанием сборки для реализации поставщика EF

NOTE

Полное имя сборки — это полное имя пространства имен, используя запятую, затем сборку, тип находится в. При необходимости вы можете также указать версию сборки, язык и региональные параметры и маркер открытого ключа.

В качестве примера ниже приведен записи, созданной для регистрации поставщика SQL Server по умолчанию при установке платформы Entity Framework.

```
<providers>
  <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices,
    EntityFramework.SqlServer" />
</providers>
```

Перехватчики (EF6.1 и более поздние версии)

Начиная с EF6.1 перехватчики можно зарегистрировать в файле конфигурации. Перехватчики позволяют запускать дополнительной логики, когда EF выполняет некоторые операции, такие как выполнение запросов к базе данных, открытия подключений и т. д.

Перехватчики зарегистрированы, включая **перехватчик** элемента под **перехватчики** дочернего **entityFramework** раздел. Например, следующая конфигурация регистрирует встроенной **DatabaseLogger** перехватчика, которые регистрируются все операции с базой данных в консоль.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework"/>
</interceptors>
```

Ведение журнала операций базы данных в файл (EF6.1 и более поздние версии)

Регистрация перехватчики через файл конфигурации особенно полезна при вы хотите добавить в существующее приложение для отладки проблемы ведения журнала. **DatabaseLogger** поддерживает ведение журнала в файл, указав имя файла в качестве параметра конструктора.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Temp\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

По умолчанию в результате файл журнала перезапись файла каждый раз, когда приложение запускается. Для вместо добавления в журнал файла, если он уже существует рекомендуется использовать примерно следующим образом:

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Temp\LogOutput.txt"/>
      <parameter value="true" type="System.Boolean"/>
    </parameters>
  </interceptor>
</interceptors>
```

Дополнительные сведения о **DatabaseLogger** и регистрации перехватчики, см. в записи блога [EF 6.1: Включение ведения журнала без повторной компиляции](#).

Фабрика соединений по умолчанию первый код

Раздел конфигурации позволяет указать фабрику соединений по умолчанию, который Code First следует использовать для поиска в базе данных для контекста. Фабрика соединений по умолчанию используется только в том случае, если строка подключения не был добавлен в файл конфигурации для контекста.

При установке пакета EF NuGet фабрика соединений по умолчанию был зарегистрирован, указывающий на SQL Express или LocalDB, в зависимости от того, какой из них установки.

Чтобы задать фабрику соединений, укажите имя типа с указанием сборки в **defaultConnectionFactory** элемент.

NOTE

Полное имя сборки — это полное имя пространства имен, используя запятую, затем сборку, тип находится в. При необходимости вы можете также указать версию сборки, язык и региональные параметры и маркер открытого ключа.

Ниже приведен пример настройки собственной фабрики подключения по умолчанию:

```
<entityFramework>
  <defaultConnectionFactory type="MyNamespace.MyCustomFactory, MyAssembly"/>
</entityFramework>
```

Приведенный выше пример требует собственного производства, чтобы иметь конструктор без параметров. При необходимости можно указать с помощью параметров конструктора **параметры** элемент.

Например SqlCeConnectionFactory, который включен в Entity Framework, необходимо предоставить неизменяемое имя поставщика в конструктор. Неизменяемое имя поставщика определяет версию SQL Compact, вы хотите использовать. Следующая конфигурация вызовет контекстов для использования SQL Compact версии 4.0 по умолчанию.

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="System.Data.SqlClient" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

Если фабрика соединений по умолчанию не задан, Code First использует SqlConnectionFactory, указывающий на `.\SQLEXPRESS`. SqlConnectionFactory также имеет конструктор, который позволяет переопределить части строки подключения. Если вы хотите использовать экземпляр SQL Server, отличное

от .\SQLEXPRESS этот конструктор можно использовать для настройки сервера.

Следующая конфигурация приведет к Code First для использования **Сервер_базы_данных** для контекстов, не имеющие явную строку подключения значение.

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="Data Source=MyDatabaseServer; Integrated Security=True;
MultipleActiveResultSets=True" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

По умолчанию предполагается, что аргументы конструктора являются типом string. Чтобы изменить этот параметр можно использовать атрибут type.

```
<parameter value="2" type="System.Int32" />
```

Инициализаторы базы данных

Инициализаторы базы данных настраиваются на основе каждого контекста. Их можно задать в файле конфигурации с помощью **контекст** элемент. Этот элемент используется полное имя сборки для определения контекста выполняется настройка.

По умолчанию Code First контекстов настроены на использование инициализатора CreateDatabaseIfNotExists. Существует **disableDatabaseInitialization** атрибут **контекст** элемент, который может использоваться для отключения инициализации базы данных.

Например следующая конфигурация отключает инициализацию базы данных для Blogging.BlogContext контексте, определенном в файле MyAssembly.dll.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly" disableDatabaseInitialization="true" />
</contexts>
```

Можно использовать **databaseInitializer** элемент для задания настраиваемого инициализатора.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly" />
  </context>
</contexts>
```

Параметры конструктора используется тот же синтаксис, как фабрики подключения по умолчанию.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly">
      <parameters>
        <parameter value="MyConstructorParameter" />
      </parameters>
    </databaseInitializer>
  </context>
</contexts>
```

Можно настроить один инициализаторов универсальную базу данных, включенных в Entity Framework. **Тип** формата .NET Framework использует атрибут для универсальных типов.

Например, при использовании Code First Migrations, можно настроить базу данных для миграции автоматически с помощью `MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>` инициализатора.

```
<contexts>
  <context type="Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[[Blogging.BlogContext,
MyAssembly], [Blogging.Migrations.Configuration, MyAssembly]], EntityFramework" />
  </context>
</contexts>
```

Строки подключения и моделей

13.09.2018 • 9 minutes to read • [Edit Online](#)

В этом разделе рассматриваются как Entity Framework обнаруживает какие подключения к базе данных, и как его можно изменить. Обе модели, созданные с помощью Code First и конструктором EF рассматриваются в этом разделе.

Обычно в приложении Entity Framework использует класс, производный от DbContext. Этот производный класс будет вызовите один из конструкторов в базовом классе DbContext для элемента управления.

- Как контекст будет подключаться к базе данных, то есть как строка подключения — обнаружении используется
- Будет использовать контекст вычисления модели с помощью Code First или загрузить модели, созданной с помощью конструктора EF
- Дополнительные параметры

Следующие фрагменты показать некоторые из способов конструкторы DbContext может использоваться.

Использовать Code First с подключением по соглашению

Если вы не сделали любую другую конфигурацию в приложении, затем вызов конструктора без параметров для DbContext приведет к DbContext для запуска в режиме Code First с помощью подключения к базе данных, созданные по соглашению. Пример:

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
        // C# will call base class parameterless constructor by default
        {
        }
    }
}
```

В этом примере DbContext использует полное имя пространства имен вашего производного контекста class —Demo.EF.BloggingContext—as имя базы данных и создает строку подключения для этой базы данных с помощью SQL Express или LocalDB. Если установлены оба, SQL Express будут использоваться.

Visual Studio 2010 включает в себя SQL Express по умолчанию и Visual Studio 2012 и более поздних версий LocalDB. Во время установки пакета EntityFramework NuGet проверяет, какой сервер базы данных доступна. Пакет NuGet затем обновит файл конфигурации, задав сервер базы данных по умолчанию, использующий Code First, при создании подключения по соглашению. Если выполняется SQL Express, он будет использоваться. Если SQL Express не доступен затем LocalDB будет зарегистрировано по умолчанию вместо этого. Нет изменения вносятся в файл конфигурации, если он уже содержит параметр для фабрики подключения по умолчанию.

Использовать Code First с подключением, соглашение о вызовах и указанное имя базы данных

Если вы не сделали любую другую конфигурацию в приложении, последующим вызовом конструктор строк на DbContext с именем базы данных, которую вы хотите использовать приведет к DbContext для запуска в

режиме Code First с помощью подключения к базе данных, созданные по соглашению к базе данных Это имя. Пример:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingDatabase")
    {
    }
}
```

В этом примере DbContext использует «BloggingDatabase» в качестве имени базы данных и создает строку подключения для этой базы данных с помощью SQL Express (устанавливается с Visual Studio 2010) или LocalDB (устанавливается с Visual Studio 2012). Если установлены оба, SQL Express будут использоваться.

Использование Code First, строкой подключения в файле app.config/web.config

Вы можете поместить строку подключения в файле app.config или web.config. Пример:

```
<configuration>
  <connectionStrings>
    <add name="BloggingCompactDatabase"
        providerName="System.Data.SqlServerCe.4.0"
        connectionString="Data Source=Blogging.sdf"/>
  </connectionStrings>
</configuration>
```

Это простой способ сообщить DbContext, чтобы использовать сервер базы данных, отличных от SQL Express или LocalDB, приведенном выше примере указывает базу данных SQL Server Compact Edition.

Если имя строки подключения совпадает с именем контекста (с или без квалификации пространства имен) затем он будет найден DbContext при использовании конструктора без параметров. Если имя строки подключения отличается от имени контекста можно сказать DbContext, чтобы использовать это подключение в режиме Code First, передавая имя строки подключения для конструктора DbContext.

Пример:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingCompactDatabase")
    {
    }
}
```

Кроме того, можно использовать форму «имя = <имя строки подключения>» для строки, переданных конструктору DbContext. Пример:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingCompactDatabase")
    {
    }
}
```

Эта форма позволяет явные, что предполагается, что строка подключения должна находиться в файле конфигурации. Исключение возникает, если строку подключения с заданным именем не найден.

Database First или Model First со строкой подключения в файле app.config/web.config

Модели, созданные в конструкторе EF, отличаются от Code First, в том, что модель уже существует и не создается из кода при запуске приложения. Модель обычно существует как файл EDMX в проекте.

Конструктор добавляет строки подключения к EF в файл app.config или web.config. Эта строка подключения отличается тем, что он содержит сведения о том, как найти сведения в файле EDMX. Пример:

```
<configuration>
  <connectionStrings>
    <add name="Northwind_Entities"
      connectionString="metadata=res://*/Northwind.csdl|
                        res://*/Northwind.ssdl|
                        res://*/Northwind.msl;
      provider=System.Data.SqlClient;
      provider connection string=
        "Data Source=.\sqlexpress;
          Initial Catalog=Northwind;
          Integrated Security=True;
          MultipleActiveResultSets=True";
      providerName="System.Data.EntityClient"/>
  </connectionStrings>
</configuration>
```

Конструктор EF также создаст код, который сообщает DbContext, чтобы использовать это подключение, передав имя строки подключения конструктору DbContext. Пример:

```
public class NorthwindContext : DbContext
{
    public NorthwindContext()
        : base("name=Northwind_Entities")
    {
    }
}
```

DbContext знал о необходимости загрузить существующей модели (а не с помощью Code First для вычисления его из кода), так как строка подключения имеет строки подключения к EF, содержащий подробные сведения о модели для использования.

Другие параметры конструктора DbContext

Класс DbContext содержит другие конструкторы и шаблоны использования, которые обеспечивают некоторые более расширенные сценарии. Ниже приведены некоторые из них.

- Класс DbModelBuilder можно использовать для построения модели Code First без создания экземпляра DbContext. В результате является объектом DbModel. Затем можно передать этот объект DbModel для одного или конструкторов DbContext, когда будете готовы для создания экземпляра DbContext.
- Стока подключения целиком можно передать DbContext вместо базы данных или соединения строки имени. По умолчанию эта строка подключения используется с поставщиком System.Data.SqlClient; Это можно изменить, задав другой реализации IConnectionFactory в контекст Database.DefaultConnectionFactory.
- Можно использовать существующий объект DbConnection, передавая его конструктора DbContext. Если

объект подключения — это экземпляр EntityConnection, модели, указанное в соединении будет использоваться, а не вычисление модель с помощью Code First. Если объект является экземпляром другого типа — например, SqlConnection, затем контекст будет использовать для режиме Code First.

- Можно передать существующий ObjectContext DbContext конструктора для создания DbContext упаковки существующего контекста. Это можно использовать для существующих приложений, использующие ObjectContext, но которым нужно воспользоваться преимуществами DbContext в некоторых частях приложения.

Разрешение зависимостей

13.09.2018 • 13 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Начиная с EF6, Entity Framework содержит механизм общего назначения для получения реализаций служб, которые необходимы. То есть при EF используется экземпляр некоторые интерфейсы или базовые классы вам будет предложено конкретную реализацию интерфейса или базового класса для использования. Это достигается путем использования интерфейса `IDbDependencyResolver`:

```
public interface IDbDependencyResolver
{
    object GetService(Type type, object key);
}
```

Метод `GetService` обычно вызывается EF и обрабатывается реализацией `IDbDependencyResolver`, предоставленное платформой EF или приложением. При вызове аргумент типа является типом интерфейса или базового класса запрашиваемой службы, а объект ключа равно `null` или объект, предоставляющий контекстную информацию о запрошенную службу.

Если не указано иное, любой объект, возвращаемый должен быть поточно ориентированной, так как он может использоваться как единственный экземпляр. Во многих случаях объекты, возвращаемые в этом случае является фабрикой самой фабрики должен быть поточно ориентированной, но не нужно быть потокобезопасными, поскольку у фабрики для каждого использования запрашивается новый экземпляр объекта, возвращаемого из фабрики.

Эта статья содержит подробные сведения о том, как реализовать `IDbDependencyResolver`, но вместо выступает в качестве ссылки для типов служб (то есть, интерфейсов и базовых типах классов), для которых EF вызывает `GetService` и семантика объекта ключа для каждого из них вызовы.

`System.Data.Entity.IDatabaseInitializer < TContext >`

Представленные версии: EF6.0.0

Объект, возвращаемый: инициализатор базы данных для заданного контекста типа

Ключ: не использовать, будет иметь значение `null`

`Func < System.Data.Entity.Migrations.Sql.SqlGenerator >`

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику для создания генератора SQL, который может использоваться для миграции и другие действия,зывающие базы данных, создаваемых, таких как создание базы данных с инициализаторами базы данных.

Ключ: строка, содержащая неизменяемое имя поставщика ADO.NET, указав тип базы данных, для которого

будет создаваться SQL. Например генератор SQL Server SQL возвращается для ключа «System.Data.SqlClient»».

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Entity.Core.Common.DbProviderServices

Представленные версии: EF6.0.0

Объект, возвращаемый: поставщик EF для использования данного неизменяемое имя поставщика

Ключ: строка, содержащая неизменяемое имя поставщика ADO.NET, указав тип базы данных, для которого требуется поставщик. Например поставщик SQL Server возвращается для ключа «System.Data.SqlClient»».

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Entity.Infrastructure.IDbConnectionFactory

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику соединений, который будет использоваться при EF создает подключение к базе данных по соглашению. То есть при без подключения или строка подключения передается EF, а не строку подключения можно найти в файле app.config или web.config, эта служба затем используется для создания подключения, по соглашению. Изменение Фабрика соединений можно разрешить EF использовать другой тип базы данных (например, SQL Server Compact Edition) по умолчанию.

Ключ: не использовать, будет иметь значение null

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Entity.Infrastructure.IManifestTokenService

Представленные версии: EF6.0.0

Объект, возвращаемый: служба, можно создать маркер манифеста поставщика из соединения. Эта служба обычно используется двумя способами. Во-первых он может использоваться для предотвращения Code First, соединения с базой данных при построении модели. Во-вторых его можно использовать для принудительного Code First для создания модели для версии конкретной базы данных — например, чтобы принудительно модели для SQL Server 2005, даже если иногда используется SQL Server 2008.

Время жизни объекта: одноэлементный--тот же объект может быть используется несколько раз и одновременно в разных потоков

Ключ: не использовать, будет иметь значение null

System.Data.Entity.Infrastructure.IDbProviderFactoryService

Представленные версии: EF6.0.0

Объект, возвращаемый: служба, можно получить фабрику поставщика из данного соединения. В .NET 4.5 поставщик является общедоступным из соединения. В .NET 4 Реализация по умолчанию эта служба использует часть эвристических правил, чтобы найти соответствующий поставщик. Если они не затем новую реализацию этой службы можно зарегистрировать для обеспечения соответствующего разрешения.

Ключ: не использовать, будет иметь значение null

Func < DbContext, System.Data.Entity.Infrastructure.IDbModelCacheKey>

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику, которая создаст ключ кэша модели в данном контексте. По умолчанию EF кэширует одной модели каждого типа DbContext поставщика. Чтобы добавить другие сведения, такие как имя схемы, ключ кэша можно использовать другую реализацию этой службы.

Ключ: не использовать, будет иметь значение null

System.Data.Entity.Spatial.DbSpatialServices

Представленные версии: EF6.0.0

Объект, возвращаемый: EF пространственных поставщика, который добавляет в базовый поставщик EF для geography и geometry пространственных типов.

Ключ: DbSpatialServices запрашивается двумя способами. Во-первых, поставщика пространственной службы запрашиваются с помощью объекта DbProviderInfo (который содержит инвариант имя и маркер манифеста) как ключ. Во-вторых DbSpatialServices могут запрашиваться без ключа. Это используется для разрешения «глобальные пространственных поставщик» используемый при создании автономного DbGeography или DbGeometry типов.

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

Func < System.Data.Entity.Infrastructure.IDbExecutionStrategy>

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику для создания службы, которая позволяет поставщику для реализации повторных попыток или другого поведения, при выполнении запросов и команд в базе данных. Если реализации не указаны, затем EF будет просто выполните команды и распространить все исключения. Для SQL Server эта служба используется для предоставления политику повтора, что особенно полезно при запуске для серверов базы данных на основе облака, таких как SQL Azure.

Ключ: ExecutionStrategyKey объект, содержащий неизменяемое имя поставщика и, при необходимости, имя сервера, для которого будет использоваться стратегии выполнения.

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

Func < DbConnection, строка, System.Data.Entity.Migrations.History.HistoryContext>

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику, которая позволяет поставщику настроить сопоставление HistoryContext для `_MigrationHistory` таблицу, используемую миграции EF. HistoryContext является первый DbContext кода и могут быть настроены в обычном fluent API для изменения действия, такие как имя таблицы и спецификации сопоставления столбцов.

Ключ: не использовать, будет иметь значение null

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Common.DbProviderFactory

Представленные версии: EF6.0.0

Объект, возвращаемый: поставщик ADO.NET для использования данного неизменяемое имя поставщика.

Ключ: строка, содержащая неизменяемое имя поставщика ADO.NET

NOTE

Эта служба не изменяется обычно непосредственно, так как в реализации по умолчанию используется обычный Регистрация поставщика ADO.NET. См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Entity.Infrastructure.IProviderInvariantName

Представленные версии: EF6.0.0

Объект, возвращаемый: это служба, которая используется для определения неизменяемое имя поставщика для данного типа DbProviderFactory. Реализация по умолчанию эта служба использует Регистрация поставщика ADO.NET. Это означает, что если поставщик ADO.NET не зарегистрирован обычным способом, так как объект DbProviderFactory разрешается путем EF, затем он также будет необходимо для устранения этой службы.

Ключ: The DbProviderFactory экземпляра, для которого необходим является инвариантным именем.

NOTE

См. Дополнительные сведения о службах, связанные с поставщиком, в EF6 [модель поставщика EF6](#) раздел.

System.Data.Entity.Core.Mapping.ViewGeneration.IViewAssemblyCache

Представленные версии: EF6.0.0

Объект, возвращаемый: кэш сборок, содержащих заранее сформированные представления. Замена обычно используется позволяет узнать, какие сборки содержит заранее сформированные представления без выполнения любой обнаружения EF.

Ключ: не использовать, будет иметь значение null

`System.Data.Entity.Infrastructure.Pluralization.IPluralizationService`

Представленные версии: EF6.0.0

Объект, возвращаемый: служба, используемая EF pluralize и множественном числе имени. По умолчанию используется в службе преобразования во множественную форму на английском языке.

Ключ: не использовать, будет иметь значение null

`System.Data.Entity.Infrastructure.Interception.IDbInterceptor`

Представленные версии: EF6.0.0

Объекты, возвращаемые: любой перехватчики, которые должны быть зарегистрированы при запуске приложения. Обратите внимание, что эти объекты запрошены с помощью вызова GetServices и все перехватчики, возвращаемый любой Сопоставитель зависимостей будет зарегистрирована.

Ключ: не использовать, будет иметь значение null.

`Func < System.Data.Entity.DbContext, действие < строка>, System.Data.Entity.Infrastructure.Interception.DatabaseLogFormatter>`

Представленные версии: EF6.0.0

Объект, возвращаемый: фабрику, которая будет использоваться для создания модуля форматирования журнала базы данных, который будет использовать, когда контекст. Свойство Database.Log устанавливается в заданном контексте.

Ключ: не использовать, будет иметь значение null.

`Func < System.Data.Entity.DbContext>`

Представленные версии: EF6.1.0

Объект, возвращаемый: фабрику, которая будет использоваться для создания экземпляров контекста для миграций, если контекст не имеет доступный конструктор без параметров.

Ключ: объект типа для типа производном DbContext, для которого требуется фабрику.

`Func < System.Data.Entity.Core.Metadata.Edm.IMetadataAnnotationSerializer >`

Представленные версии: EF6.1.0

Объект, возвращаемый: фабрику, которая будет использоваться для создания сериализаторы для сериализации со строгой типизацией пользовательских заметок, таким образом, сериализации и deserialized в XML для использования в Code First Migrations.

Ключ: имя, для которого создается заметка сериализации или десериализации.

`Func < System.Data.Entity.Infrastructure.TransactionHandler>`

Представленные версии: EF6.1.0

Объект, возвращаемый: фабрику, которая будет использоваться для создания обработчиков для транзакций, таким образом, чтобы специальная обработка может применяться для ситуаций, таких как

обработка сбоев.

Ключ: ExecutionStrategyKey объект, содержащий неизменяемое имя поставщика и, при необходимости, имя сервера, для которого будет использоваться обработчик транзакции.

Управление подключениями

13.09.2018 • 7 minutes to read • [Edit Online](#)

Эта страница описывает поведение Entity Framework по отношению к передачи подключений в контексте и функциональные возможности **Database.Connection.Open()** API.

Передача подключений к контексту

Поведение для EF5 и более ранних версий

Существует два конструктора, которые принимать подключения.

```
public DbContext(DbConnection existingConnection, bool contextOwnsConnection)
public DbContext(DbConnection existingConnection, DbCompiledModel model, bool contextOwnsConnection)
```

Можно использовать их, но вам нужно решить ряд ограничений:

1. При передаче открытое подключение к одному из этих затем первый раз, платформа пытается использовать его, возникает исключение `InvalidOperationException` о том, что не удается повторно открыть уже открытое соединение.
2. Флаг `contextOwnsConnection` интерпретируется для обозначения ли Базовое соединение хранилища должен быть удален при освобождении контекста. Но, независимо от этого параметра соединения с хранилищем всегда закрывается при освобождении контекста. Поэтому при наличии нескольких `DbContext` того же соединения удаляется независимо от контекста сначала будет закрыть соединение (аналогично Если имеются смешанные существующее соединение ADO.NET с `DbContext`, `DbContext` будет всегда закрывать соединение после его удаления) .

Это можно обойти первое ограничение выше путем передачи закрытого соединения и только выполнение кода, который бы открыл его, когда будут созданы все контексты:

```

using System.Collections.Generic;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;
using System.Linq;

namespace ConnectionManagementExamples
{
    class ConnectionManagementExampleEF5
    {
        public static void TwoDbContextsOneConnection()
        {
            using (var context1 = new BloggingContext())
            {
                var conn =
                    ((EntityConnection)
                        ((IObjectContextAdapter)context1).ObjectContext.Connection)
                        .StoreConnection;

                using (var context2 = new BloggingContext(conn, contextOwnsConnection: false))
                {
                    context2.Database.ExecuteSqlCommand(
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'");

                    var query = context1.Posts.Where(p => p.Blog.Rating > 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }
                    context1.SaveChanges();
                }
            }
        }
    }
}

```

Вторым ограничением просто означает, что необходимо воздержаться от удаления любой из ваших объектов DbContext, пока не будете готовы для подключения будет закрыта.

Поведение в EF6 и будущих версий

В EF6 и будущих версий DbContext имеет же два конструктора, но больше не требуется, что передается конструктору подключение закрыто при его получении. Так что теперь это возможно:

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace ConnectionManagementExamples
{
    class ConnectionManagementExample
    {
        public static void PassingAnOpenConnection()
        {
            using (var conn = new SqlConnection("{connectionString}"))
            {
                conn.Open();

                var sqlCommand = new SqlCommand();
                sqlCommand.Connection = conn;
                sqlCommand.CommandText =
                    @"UPDATE Blogs SET Rating = 5" +
                    " WHERE Name LIKE '%Entity Framework%'";
                sqlCommand.ExecuteNonQuery();

                using (var context = new BloggingContext(conn, contextOwnsConnection: false))
                {
                    var query = context.Posts.Where(p => p.Blog.Rating > 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }
                    context.SaveChanges();
                }

                var sqlCommand2 = new SqlCommand();
                sqlCommand2.Connection = conn;
                sqlCommand2.CommandText =
                    @"UPDATE Blogs SET Rating = 7" +
                    " WHERE Name LIKE '%Entity Framework Rocks%'";
                sqlCommand2.ExecuteNonQuery();
            }
        }
    }
}

```

Также флаг `contextOwnsConnection` теперь управляет ли соединение и закрытия при освобождении `DbContext`. Поэтому в приведенном выше примере подключение не закрывается при контексте удален (строка 32), как это было бы в предыдущих версиях EF, но вместо этого в том случае, когда удаляется само соединение (строки 40).

Само собой, это по-прежнему возможно для `DbContext` контролировать подключения (только набор `contextOwnsConnection` значение `true`, или использовать один из других конструкторов) можно.

NOTE

Существуют некоторые дополнительные соображения при использовании транзакций с помощью этой новой модели. Дополнительные сведения см. [работа с транзакциями](#).

Database.Connection.Open()

Поведение для EF5 и более ранних версий

В EF5 и более ранних версиях есть ошибка таким образом, чтобы `ObjectContext.Connection.State` не был

обновлен для отражения состояния true базового соединения с хранилищем. Например, если выполняется следующий код вы могут возвращаться состояние **закрыто** несмотря на то, что на самом деле базового подключения к хранилищу **откройте**.

```
((IObjectContextAdapter)context).ObjectContext.Connection.State
```

Отдельно, при открытии подключения к базе данных, вызвав Database.Connection.Open() его будут открыты до следующего выполнения запроса, или вызывать никаких действий, которая требует подключения к базе данных (например, SaveChanges()), но после базового хранения, соединение будет закрыто. Контекст будет повторно открыть и повторно закрыть подключение каждый раз, когда необходима другая операция базы данных:

```
using System;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;

namespace ConnectionManagementExamples
{
    public class DatabaseOpenConnectionBehaviorEF5
    {
        public static void DatabaseOpenConnectionBehavior()
        {
            using (var context = new BloggingContext())
            {
                // At this point the underlying store connection is closed

                context.Database.Connection.Open();

                // Now the underlying store connection is open
                // (though ObjectContext.Connection.State will report closed)

                var blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);

                // The underlying store connection is still open

                context.SaveChanges();

                // After SaveChanges() the underlying store connection is closed
                // Each SaveChanges() / query etc now opens and immediately closes
                // the underlying store connection

                blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();
            }
        }
    }
}
```

Поведение в EF6 и будущих версий

Для EF6 и будущих версий мы предприняли подход, если вызывающий код выбирает для открытия соединения с вызывающим контекстом. Database.Connection.Open(), то оно имеет веские причины таким образом и framework предполагает, что он хочет контролировать Открытие и закрытие соединения и больше не закроет это подключение автоматически.

NOTE

Это потенциально может привести к подключениям, которые открыты для много времени, поэтому используйте с осторожностью.

Мы также обновили код таким образом, чтобы `ObjectContext.Connection.State` теперь следит за состоянием базового соединения правильно.

```
using System;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Core.EntityClient;
using System.Data.Entity.Infrastructure;

namespace ConnectionManagementExamples
{
    internal class DatabaseOpenConnectionBehaviorEF6
    {
        public static void DatabaseOpenConnectionBehavior()
        {
            using (var context = new BloggingContext())
            {
                // At this point the underlying store connection is closed

                context.Database.Connection.Open();

                // Now the underlying store connection is open and the
                // ObjectContext.Connection.State correctly reports open too

                var blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();

                // The underlying store connection remains open for the next operation

                blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();

                // The underlying store connection is still open
            } // The context is disposed - so now the underlying store connection is closed
        }
    }
}
```

Логику устойчивости и повторите попытку подключения

13.09.2018 • 11 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Приложений, подключающихся к серверу базы данных всегда были уязвимы для разрывов соединений из-за внутренних сбоев и нестабильностью сети. Тем не менее в среде локальной сети на основе работе для выделенной базы данных серверов эти ошибки можно достаточно редко, что дополнительную логику для обработки этих ошибок не часто требуется. С ростом облачных основаны менее надежной сети, которые теперь чаще разрывы подключения серверов базы данных, таких как база данных SQL Windows Azure и подключения. Это может быть вызвано защитных техник, облачной базы данных используют, чтобы обеспечить распределение ресурсов службы, например, регулирование подключения, либо к нестабильной работе в сети, приводит к времени ожидания периодических и других временных ошибок.

Устойчивость подключений относится к возможности для Entity FRAMEWORK, автоматический повтор любых команд, которые не отвечают из-за этих разрывов соединения.

Стратегии выполнения

Повторную попытку подключения возложите на реализацию интерфейса `IDbExecutionStrategy`. Реализации `IDbExecutionStrategy` будет отвечать за принятие операции и, при возникновении исключения, определение, если Повтор может быть выполнен и повторять их в случае. Существуют четыре стратегии выполнения, входящие в состав EF.

1. **DefaultExecutionStrategy:** Эта стратегия выполнения не выполняет повторную попытку запуска любых операций, это значение по умолчанию для баз данных, отличных от sql server.
2. **DefaultSqlExecutionStrategy:** это стратегия внутреннего выполнения, которая используется по умолчанию. Эта стратегия не пытается повторно вообще, тем не менее, она будет разбита любые исключения, которые может быть временной для информирования пользователей, которые они хотели включить устойчивость подключения.
3. **DbExecutionStrategy:** этот класс подходит как базовый класс для других стратегий выполнения, включая собственные пользовательские. Она реализует экспоненциальную политику повторов, где начальной повторная попытка происходит с нулевой задержки и задержка увеличивается экспоненциально пока не достигается максимальное число повторных попыток. Этот класс содержит абстрактный метод `ShouldRetryOn`, который может быть реализован в производном выполнения стратегии для управления, какие исключения следует повторить.
4. **SqlAzureExecutionStrategy:** Эта стратегия выполнения наследует от `DbExecutionStrategy` и повторит попытку для исключений, которые заведомо возможно временных при работе с базой данных SQL Azure.

NOTE

Стратегии выполнения, 2 и 4, включаются в поставщике Sql Server, поставляемый с EF, которая находится в сборке EntityFramework.SqlServer и предназначены для работы с SQL Server.

Включение стратегия выполнения

Самый простой способ сообщить EF для использования стратегии выполнения — с помощью метода SetExecutionStrategy [DbConfiguration](#) класса:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
    }
}
```

Этот код указывает EF для использования SqlAzureExecutionStrategy при подключении к SQL Server.

Настройка стратегии выполнения

Конструктор SqlAzureExecutionStrategy может принимать два параметра, значение параметра MaxRetryCount и MaxDelay. Число MaxRetry — это максимальное число попыток повтора стратегии. MaxDelay — это интервал времени, представляющий максимальную задержку между повторными попытками, которые будут использовать стратегии выполнения.

Чтобы задать максимальное число повторных попыток 1 и Максимальная задержка до 30 секунд можно execute следующее:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy(
            "System.Data.SqlClient",
            () => new SqlAzureExecutionStrategy(1, TimeSpan.FromSeconds(30)));
    }
}
```

SqlAzureExecutionStrategy повторит попытку мгновенной превышено в первый раз, временная ошибка возникает, но будет больше задержка между попытками, пока либо максимальное предельное число повторов или общее время достигает максимальный интервал.

Стратегии выполнения повторит только ограниченное число исключений, которые обычно являются transient, по-прежнему необходимо будет обрабатывать другие ошибки, а также перехват исключения RetryLimitExceeded в случае когда ошибка не временна, или занимает слишком много времени для решения сам.

Существуют некоторые известные ограничения при использовании стратегии выполнения повторной попытки:

Потоковой передачи запросы не поддерживаются

По умолчанию EF6 или более поздней версии будет буфера, результаты запроса, а не их streaming. Если требуется, чтобы результаты потоковую передачу вы можно использовать метод AsStreaming изменение

LINQ to Entities в запрос для потоковой передачи.

```
using (var db = new BloggingContext())
{
    var query = (from b in db.Blogs
                 orderby b.Url
                 select b).AsStreaming();
}
```

Не поддерживается потоковая передача, при регистрации стратегии выполнения повторной попытки. Это ограничение существует, так как подключение удалось удалить части через возвращаемых результатов. В этом случае EF, потребуется повторно выполнить весь запрос, но имеет надежного способа узнать, какие результаты уже были возвращены (данные могут измениться после начального запроса было отправлено, результаты могут вернуться в другом порядке, результаты могут не иметь уникальный идентификатор и т.д.).

Инициированной пользователем транзакции не поддерживаются

Когда вы настроили стратегия выполнения, которая приводит к повторных попыток, существуют некоторые ограничения в использовании транзакций.

По умолчанию EF будет выполнять все обновления базы данных в рамках транзакции. Не нужно ничего делать, чтобы включить эту, EF всегда выполняет это автоматически.

Например в следующем коде SaveChanges выполняется автоматически в рамках транзакции. Если SaveChanges ошибкой после вставки одного нового узла, а затем был бы выполнен откат транзакции и никакие изменения не применены к базе данных. Контекст также остается в состоянии, допускающем SaveChanges вызываться снова, чтобы повторить попытку применения изменений.

```
using (var db = new BloggingContext())
{
    db.Blogs.Add(new Site { Url = "http://msdn.com/data/ef" });
    db.Blogs.Add(new Site { Url = "http://blogs.msdn.com/adonet" });
    db.SaveChanges();
}
```

Если не используется Повтор стратегию выполнения, можно включить несколько операций в рамках одной транзакции. Например следующий код создает оболочку двух вызовов SaveChanges в одной транзакции. Если любой части либо операция завершается ошибкой затем изменения применяются.

```
using (var db = new BloggingContext())
{
    using (var trn = db.Database.BeginTransaction())
    {
        db.Blogs.Add(new Site { Url = "http://msdn.com/data/ef" });
        db.Blogs.Add(new Site { Url = "http://blogs.msdn.com/adonet" });
        db.SaveChanges();

        db.Blogs.Add(new Site { Url = "http://twitter.com/efmagicunicorns" });
        db.SaveChanges();

        trn.Commit();
    }
}
```

Это не поддерживается при использовании стратегии выполнения повторной попытки, поскольку EF не будет учитывать любых предыдущих операций и как повторить их. Например если второй SaveChanges неудачей, EF больше не имеет сведения, необходимые для повторите первый вызов метода SaveChanges.

Инструкции по решению: Приостановка выполнения стратегии

Возможным решением является Приостановка Повтор стратегия выполнения для фрагмент программного кода, который должен использовать пользователь инициировал транзакции. Самый простой способ сделать это является добавление флага SuspendExecutionStrategy в код на основе класса конфигурации и изменить лямбда-выражения стратегия выполнения для возврата стратегия выполнения по умолчанию (не retrying), если флаг установлен.

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;
using System.Runtime.Remoting.Messaging;

namespace Demo
{
    public class MyConfiguration : DbConfiguration
    {
        public MyConfiguration()
        {
            this.SetExecutionStrategy("System.Data.SqlClient", () => SuspendExecutionStrategy
                ? ( IDbExecutionStrategy )new DefaultExecutionStrategy()
                : new SqlAzureExecutionStrategy());
        }

        public static bool SuspendExecutionStrategy
        {
            get
            {
                return (bool?)CallContext.LogicalGetData("SuspendExecutionStrategy") == false;
            }
            set
            {
                CallContext.LogicalSetData("SuspendExecutionStrategy", value);
            }
        }
    }
}
```

Обратите внимание на то, что мы используем CallContext для хранения значения флага. Это предоставляет аналогичную функциональность в локальное хранилище потока, но безопасен для использования асинхронного кода — включая асинхронный запрос и сохранить с Entity Framework.

Теперь мы может приостановить стратегия выполнения для раздела кода, использующего инициированной пользователем транзакции.

```

using (var db = new BloggingContext())
{
    MyConfiguration.SuspendExecutionStrategy = true;

    using (var trn = db.Database.BeginTransaction())
    {
        db.Blogs.Add(new Blog { Url = "http://msdn.com/data/ef" });
        db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
        db.SaveChanges();

        db.Blogs.Add(new Blog { Url = "http://twitter.com/efmagicunicorns" });
        db.SaveChanges();

        trn.Commit();
    }

    MyConfiguration.SuspendExecutionStrategy = false;
}

```

Инструкции по решению: Вручную вызвать стратегию выполнения

Другой вариант — вручную используйте стратегию выполнения и присвойте ему всего набора логики для выполнения, таким образом, чтобы его можно повторить все, что при сбое одной из операций. Нам еще нужно приостановить стратегия выполнения — с применением технологии показанный выше -, так что любой контекст, используемые внутри блока кода повторной попытки не пытайтесь повторить попытку.

Обратите внимание на то, что все контексты должно быть создано в блоке кода, для повтора. Это гарантирует, что мы начинаем с чистого состояния для каждой повторной попытки.

```

var executionStrategy = new SqlAzureExecutionStrategy();

MyConfiguration.SuspendExecutionStrategy = true;

executionStrategy.Execute(
    () =>
{
    using (var db = new BloggingContext())
    {
        using (var trn = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "http://msdn.com/data/ef" });
            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "http://twitter.com/efmagicunicorns" });
            db.SaveChanges();

            trn.Commit();
        }
    }
});

```

MyConfiguration.SuspendExecutionStrategy = false;

Обработка сбоев фиксации транзакции

27.09.2018 • 5 minutes to read • [Edit Online](#)

NOTE

EF6.1 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в версии 6.1 Entity Framework. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Как часть 6.1 мы представляем новый функция устойчивости подключений для Entity FRAMEWORK: возможность обнаружить и восстановить автоматически, когда к временному сбою подключения на подтверждения фиксации транзакций. Подробные сведения о сценарий лучше всего описывается в записи блога [подключение к базе данных SQL и проблема идемпотентности](#). Таким образом сценарий: при возникновении исключения во время фиксации транзакции существуют две возможные причины:

1. Не удалось выполнить фиксацию транзакции на сервере
2. Успешность фиксации транзакции на сервере, но проблема с подключением, препятствующих переходу уведомление о выполнении из доставляется клиенту

Когда происходит первая ситуация или пользователя приложения можно повторить операцию, но когда вторая ситуация происходит следует избегать повторных попыток, и приложение может восстановиться автоматически. Проблема в том, если отсутствует возможность определить, какой параметр имел фактическая причина исключения была обнаружена во время фиксации, приложение не может выбрать правильный способ действий. Новая функция в EF 6.1 позволяет EF еще раз проверьте с базой данных, если транзакция выполнена успешно и прозрачно использовать правильный способ действий.

С помощью функции

Чтобы включить функцию требуется включить вызов `SetTransactionHandler` в конструкторе вашего `DbConfiguration`. Если вы не знакомы с `DbConfiguration`, см. в разделе [код конфигурации](#). Эта функция может использоваться в сочетании с автоматическими повторными попытками, предложенная нами в EF6, которые помогают в ситуации, в котором фактически не удалось зафиксировать транзакцию на сервере из-за временного сбоя:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;

public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetTransactionHandler(SqlProviderServices.ProviderInvariantName, () => new CommitFailureHandler());
        SetExecutionStrategy(SqlProviderServices.ProviderInvariantName, () => new SqlAzureExecutionStrategy());
    }
}
```

Способ отслеживания операций

Когда эта функция включена, EF автоматически добавит новую таблицу в базе данных с именем `__Transactions`. Новая строка вставляется в этой таблице, каждый раз, транзакция создается с EF и этой

строки проверены на существование, если происходит сбой транзакции во время фиксации.

Несмотря на то, что EF делает максимум усилий, чтобы очистить строки из таблицы, когда они больше не нужны, таблицы могут увеличиваться, если приложение завершает работу преждевременно и по этой причине, которые могут потребоваться для очистки таблицы вручную в некоторых случаях.

Способ обработки сбоев с предыдущими версиями

Прежде чем EF 6.1 не механизм обработки сбоев в продукте EF. Существует несколько способов работы с этой ситуацией, которые могут применяться в предыдущих версиях EF6:

- Вариант 1: не выполнять никаких действий

Вероятность сбоя подключения во время фиксации транзакции недостаточно, поэтому оно может быть приемлемым для вашего приложения, просто ошибкой, если данное состояние возникает, фактически.

- Вариант 2: использование базы данных для сброса состояния

1. Отменить текущее DbContext
2. Создать новый DbContext и восстановить состояние приложения из базы данных
3. Информировать пользователей о том, что последняя операция не выполнена успешно

- Вариант 3: вручную отслеживания транзакции

1. Добавление таблицы не отслеживаются для базы данных, используемой для отслеживания состояния транзакций.
2. Вставьте строку в таблице в начале каждой транзакции.
3. В случае сбоя подключения во время фиксации, проверьте наличие соответствующей строки в базе данных.
 - о При наличии строки выполняться как обычно, так как транзакция была успешно зафиксирована
 - о Если строки отсутствует, используйте стратегию выполнения повторить попытку текущей операции.
4. Если фиксация выполнена успешно, удалите соответствующую строку во избежание увеличение размера таблицы.

В [этой записи блога](#) содержит пример кода для решения этой проблемы на SQL Azure.

Привязка данных с помощью WinForms

13.09.2018 • 22 minutes to read • [Edit Online](#)

Это пошаговое руководство показывает, как выполнить привязку к элементам управления Windows Forms (WinForms) в форме «основной / подробности» типов POCO. Приложение использует Entity Framework для заполнения объектов с данными из базы данных, отслеживать изменения и сохранения данных в базе данных.

Модель определяет два типа, участвующие в связи один ко многим: категория (основной\master) и продукта (зависимые\подробно). Средства Visual Studio, затем используются для привязки типов, определенных в модели для элементов управления WinForms. Платформа привязки данных WinForms позволяет осуществлять переходы между связанными объектами: Выбор строк в главное представление приводит представлении «Подробности» для обновления с соответствующими дочерними данными.

Снимки экрана и примеры кода в этом пошаговом руководстве, взяты из Visual Studio 2013, но можно выполнить в этом пошаговом руководстве с помощью Visual Studio 2012 или Visual Studio 2010.

Предварительные требования

Вам потребуется Visual Studio 2013, Visual Studio 2012 или Visual Studio 2010 установлен для выполнения этого пошагового руководства.

Если вы используете Visual Studio 2010, необходимо также установить NuGet. Дополнительные сведения см. в разделе [Установка NuGet](#).

Создание приложения

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в левой области и **Windows FormsApplication** в области справа
- Введите **WinFormswithEFSample** как имя
- Нажмите кнопку **OK**

Установите пакет Entity Framework NuGet

- В обозревателе решений щелкните правой кнопкой мыши **WinFormswithEFSample** проекта
- Выберите **управление пакетами NuGet...**
- В диалоговом окне «Управление пакетами NuGet» выберите **Online** вкладку и выберите **EntityFramework** пакета
- Нажмите кнопку **установки**

NOTE

Помимо сборки EntityFramework также добавляется ссылка System.ComponentModel.DataAnnotations. Если проект содержит ссылку на System.Data.Entity, затем он будет удален при установке EntityFramework. Сборка System.Data.Entity больше не используется для приложений Entity Framework 6.

Реализация **IListSource** для коллекций

Свойства коллекции необходимо реализовать интерфейс **IListSource**, чтобы обеспечить двухстороннюю

привязку данных с сортировкой, при использовании Windows Forms. Для этого мы хотим расширить ObservableCollection Добавление IListSource функциональных возможностей.

- Добавить **ObservableListSource** класс в проект:
 - Щелкните правой кнопкой мыши имя проекта
 - Выберите **Add -> новый элемент**
 - Выберите **класс** и введите **ObservableListSource** имени класса
- Замените код, созданный по умолчанию следующим кодом:

Этот класс позволяет двухсторонняя привязка, а также для сортировки. Класс является производным от ObservableCollection<T> и добавляет явной реализации IListSource. Метод встречает IListSource реализуется для возврата IBindingList реализация, которая остается синхронизированной с ObservableCollection. Реализация IBindingList, создаваемые ToBindingList поддерживает сортировку. Метод расширения ToBindingList определен в сборке EntityFramework.

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Diagnostics.CodeAnalysis;
using System.Data.Entity;

namespace WinFormswithEFSample
{
    public class ObservableListSource<T> : ObservableCollection<T>, IListSource
        where T : class
    {
        private IBindingList _bindingList;

        bool IListSource.ContainsListCollection { get { return false; } }

        IList IListSource.GetList()
        {
            return _bindingList ?? (_bindingList = this.ToBindingList());
        }
    }
}
```

Определение модели

В этом пошаговом руководстве, вы можете решить реализовать модель с помощью Code First или конструкторе EF. Выполните одно из двух следующих разделах.

Вариант 1: Определение модели с помощью Code First

В этом разделе показано, как создать модель и его связанные базы данных, с помощью Code First. Перейдите к следующему разделу (**вариант 2: определение модели с помощью Database First**) Если вы предпочитаете использовать Database First обратить реконструирование модели из базы данных в конструкторе EF

При использовании шаблона разработки Code First обычно начинается с написания классов .NET Framework, которые определяют модель концептуальный (домен).

- Добавьте новый **продукта** класс в проект
- Замените код, созданный по умолчанию следующим кодом:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }

        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}

```

- Добавить **категории** класс в проект.
- Замените код, созданный по умолчанию следующим кодом:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Category
    {
        private readonly ObservableListSource<Product> _products =
            new ObservableListSource<Product>();

        public int CategoryId { get; set; }
        public string Name { get; set; }
        public virtual ObservableListSource<Product> Products { get { return _products; } }
    }
}

```

Кроме определения сущностей, необходимо определить класс, производный от **DbContext** и предоставляет **DbSet< TEntity >** свойства. **DbSet** свойства let контекст знать, какие типы, которые вы хотите включить в модель. **DbContext** и **DbSet** определенные типы в сборке EntityFramework.

Экземпляр типа **DbContext** производным управляет объектов сущности во время выполнения, который включает заполнение объектов с данными из базы данных, изменение отслеживания и сохранения данных в базу данных.

- Добавьте новый **ProductContext** класс в проект.
- Замените код, созданный по умолчанию следующим кодом:

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;

namespace WinFormswithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}

```

Скомпилируйте проект.

Вариант 2: Определение модели с помощью Database First

В этом разделе демонстрируется использование Database First чтобы реконструировать модель из базы данных в конструкторе EF. Если действия в предыдущем разделе (**вариант 1: определение модели с помощью Code First**), пропустите этот раздел и перейти непосредственно к **отложенной загрузки** раздел.

Создание базы данных

Обычно при ориентировании существующей базы данных, он будет уже создан, но для этого пошагового руководства необходимо создать базу данных для доступа к.

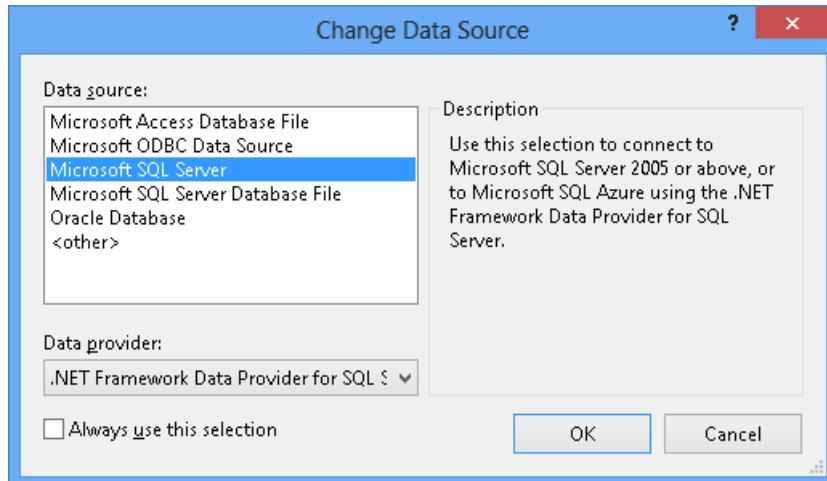
Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, вы установили:

- Если вы используете Visual Studio 2010 вы создадите базу данных SQL Express.
- Если вы используете Visual Studio 2012, а затем вы создадите [LocalDB](#) базы данных.

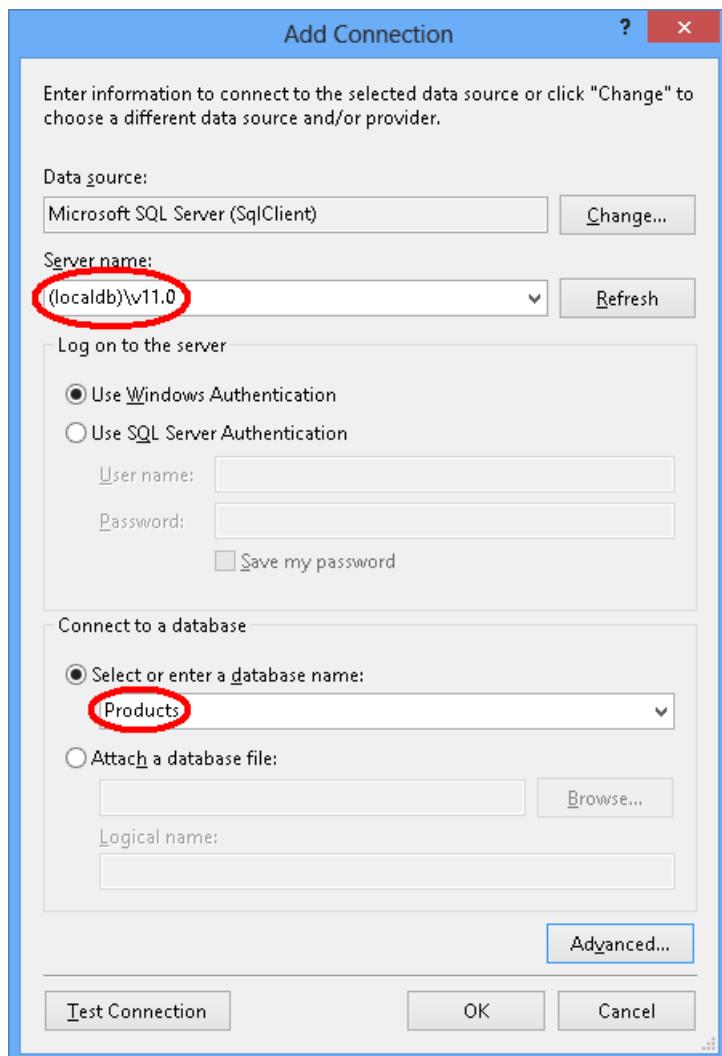
Перейдем дальше и создать базу данных.

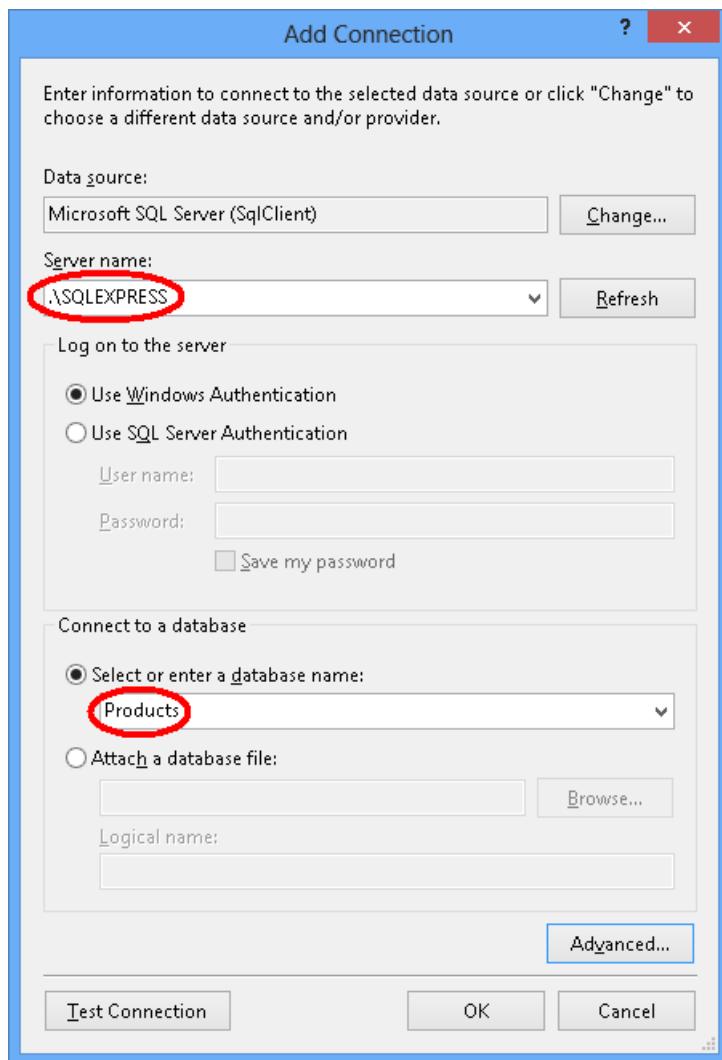
- **Представление —> обозревателя серверов**

- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**
- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам потребуется выбрать в качестве источника данных Microsoft SQL Server



- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них вы установили и введите **продуктов** имя базы данных





- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**



- Новой базы данных будут отображаться в обозревателе сервера щелкните его правой кнопкой мыши и выберите **новый запрос**
- Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**

```

CREATE TABLE [dbo].[Categories] (
    [CategoryId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    CONSTRAINT [PK_dbo.Categories] PRIMARY KEY ([CategoryId])
)

CREATE TABLE [dbo].[Products] (
    [ProductId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    [CategoryId] [int] NOT NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY ([ProductId])
)

CREATE INDEX [IX_CategoryId] ON [dbo].[Products]([CategoryId])

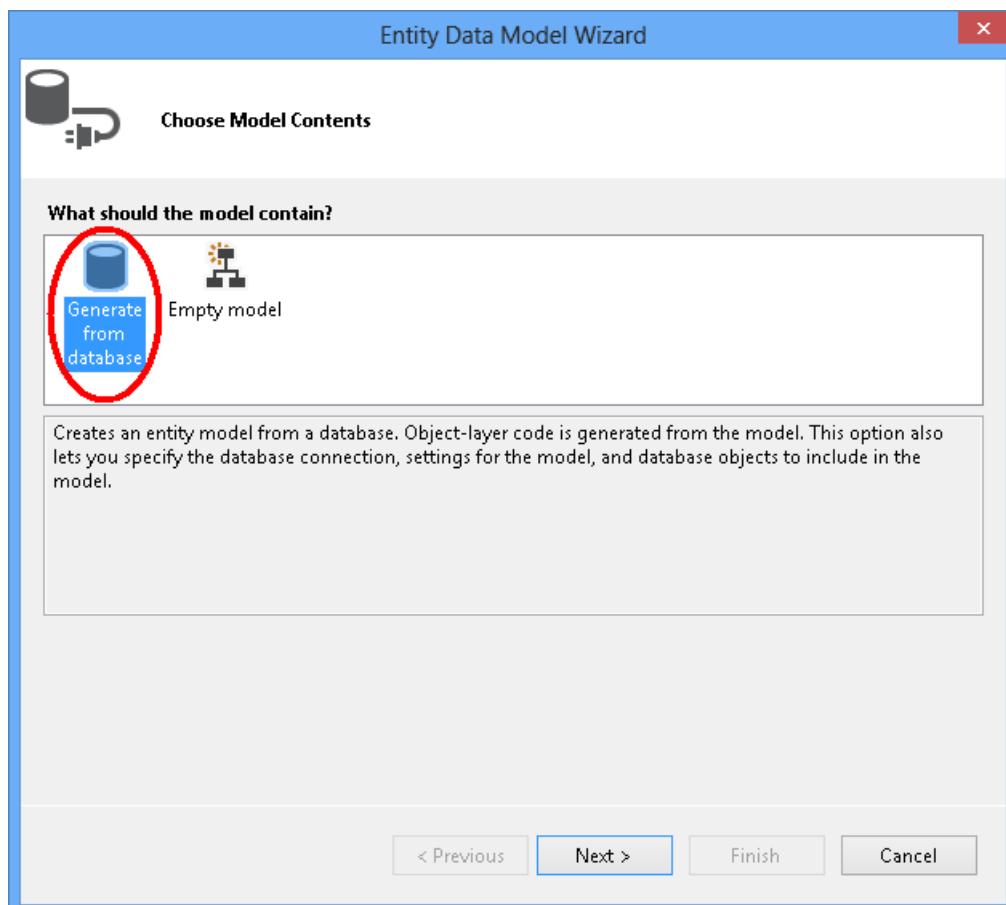
ALTER TABLE [dbo].[Products] ADD CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY
([CategoryId]) REFERENCES [dbo].[Categories] ([CategoryId]) ON DELETE CASCADE

```

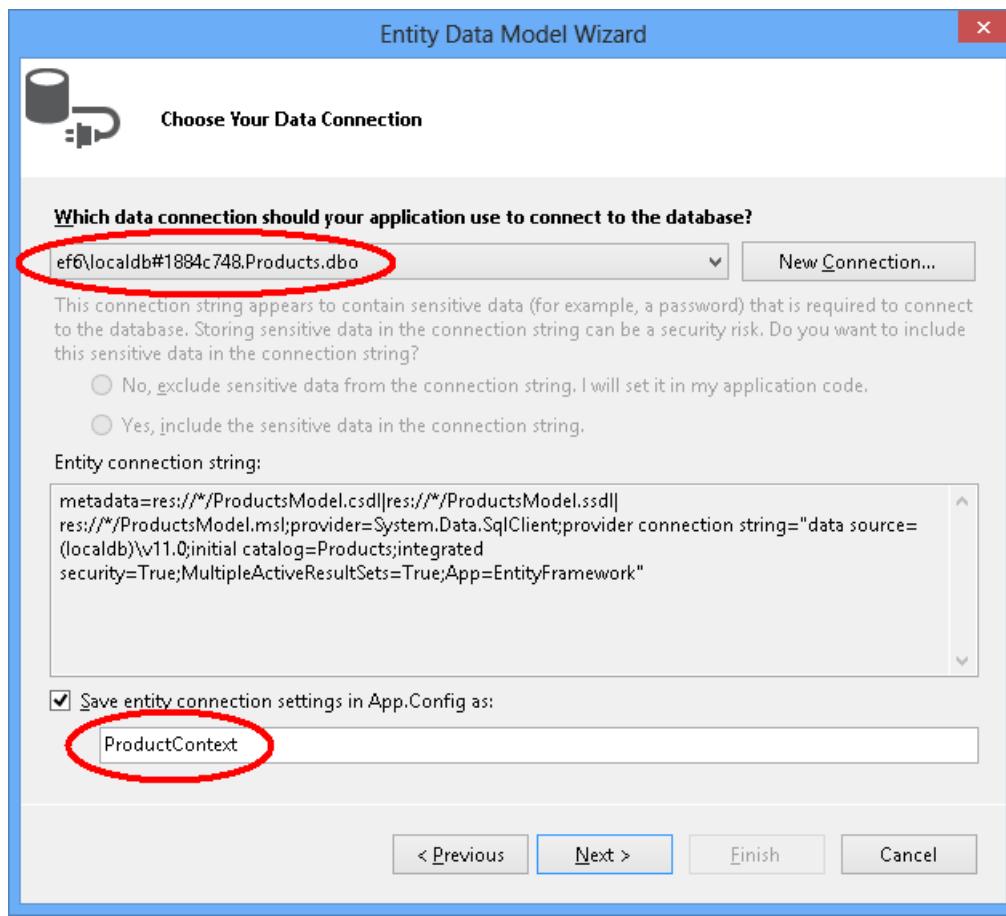
Реконструирование модели

Мы собираемся использовать Entity Framework Designer, который входит в состав Visual Studio, для создания нашей модели.

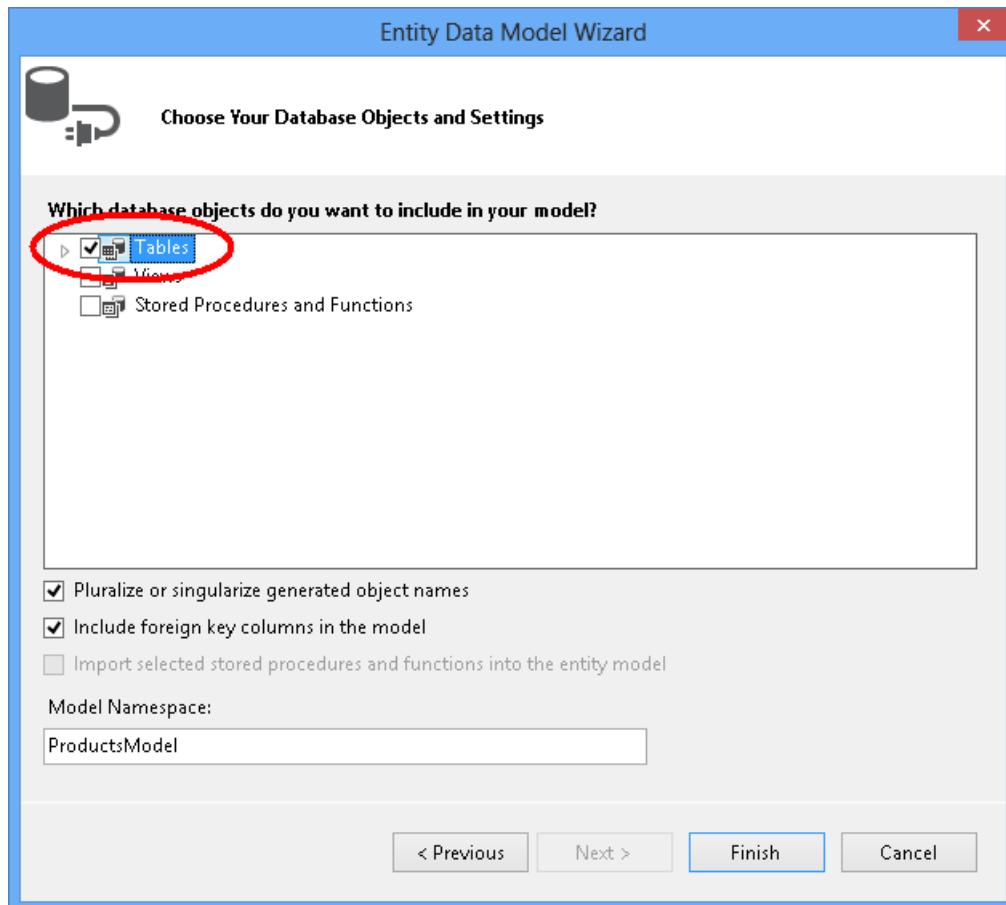
- Проект -> добавить новый элемент...
- Выберите **данных** в меню слева и затем **модель EDM ADO.NET**
- Введите **ProductModel** имя и нажмите кнопку **OK**
- Это откроет **мастер моделей EDM**
- Выберите **создать из базы данных** и нажмите кнопку **Далее**



- Выберите соединение с базой данных, созданной в первом разделе, введите **ProductContext** как имя строки подключения и нажмите кнопку **Далее**



- Установите флажок рядом с «Таблицы», чтобы импортировать все таблицы и нажмите кнопку «Готово»



После завершения процесса реконструирования новой модели добавлен в проект и открывается для просмотра в конструкторе Entity Framework. Файл App.config также был добавлен в проект со сведениями о

подключении для базы данных.

Дополнительные действия в Visual Studio 2010

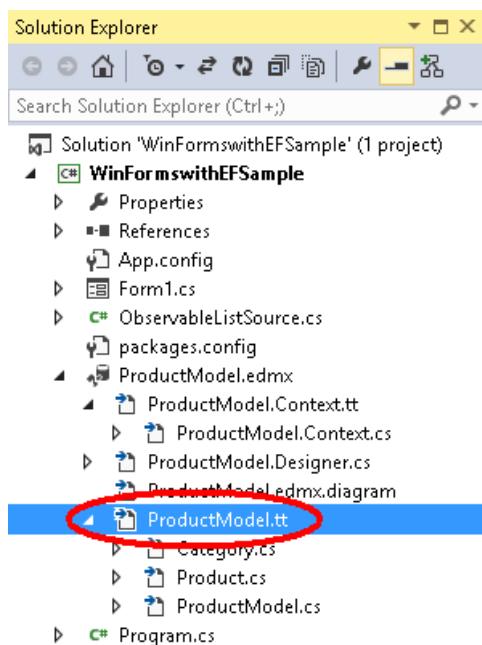
Если вы работаете в Visual Studio 2010 будет необходимо обновить конструктор EF будет использовать создание кода EF6.

- Щелкните правой кнопкой мыши пустое место модели в конструкторе EF и выберите **добавить элемент формирования кода...**
- Выберите **шаблоны в Интернете** из меню слева и выполните поиск **DbContext**
- Выберите **EF 6.x генератор DbContext для C#**, введите **ProductsModel** как имя и щелкните "Добавить"

Обновление создания кода для привязки данных

EF создает код из модели с помощью шаблонов T4. Шаблоны, поставляемых с Visual Studio или скачать из коллекции Visual Studio предназначены для использования общего назначения. Это означает, что сущности, созданные на основе этих шаблонов имеют простой `ICollection<T>` свойства. Тем не менее при выполнении привязки данных желательно иметь свойства коллекции, которые реализуют `IListSource`. Именно поэтому мы создали класс `ObservableListSource` выше, и теперь мы собираемся изменить шаблоны, чтобы сделать использование этого класса.

- Откройте **обозреватель решений** и найти **ProductModel.edmx** файла
- Найти **ProductModel.tt** файл, который будет вложен в файле ProductModel.edmx



- Дважды щелкните файл `ProductModel.tt`, чтобы открыть его в редакторе Visual Studio
- Найти и заменить два вхождения "`ICollection<with>ObservableListSource`". Они находятся в около строки 296 и 484.
- Найти и заменить первое вхождение "`HashSet<with>ObservableListSource`". Это событие находится примерно в строке 50. Не замените второе вхождение `HashSet` Найти далее в коде.
- Сохраните файл `ProductModel.tt`. Это может вызвать код для сущностей, быть создан повторно. Если код не создается автоматически, затем щелкнуть правой кнопкой мыши `ProductModel.tt` и выберите «Запустить пользовательский инструмент».

Если вы теперь откроите файл `Category.cs` (который вложен в `ProductModel.tt`), то вы увидите, что продукты коллекция имеет тип **ObservableListSource<продукта>**.

Скомпилируйте проект.

Отложенная загрузка

Продуктов свойство **категории** класс и **категории** свойство **продукта** класс — это свойства навигации. В Entity Framework свойства навигации позволяют для перехода по связи между двумя типами сущностей.

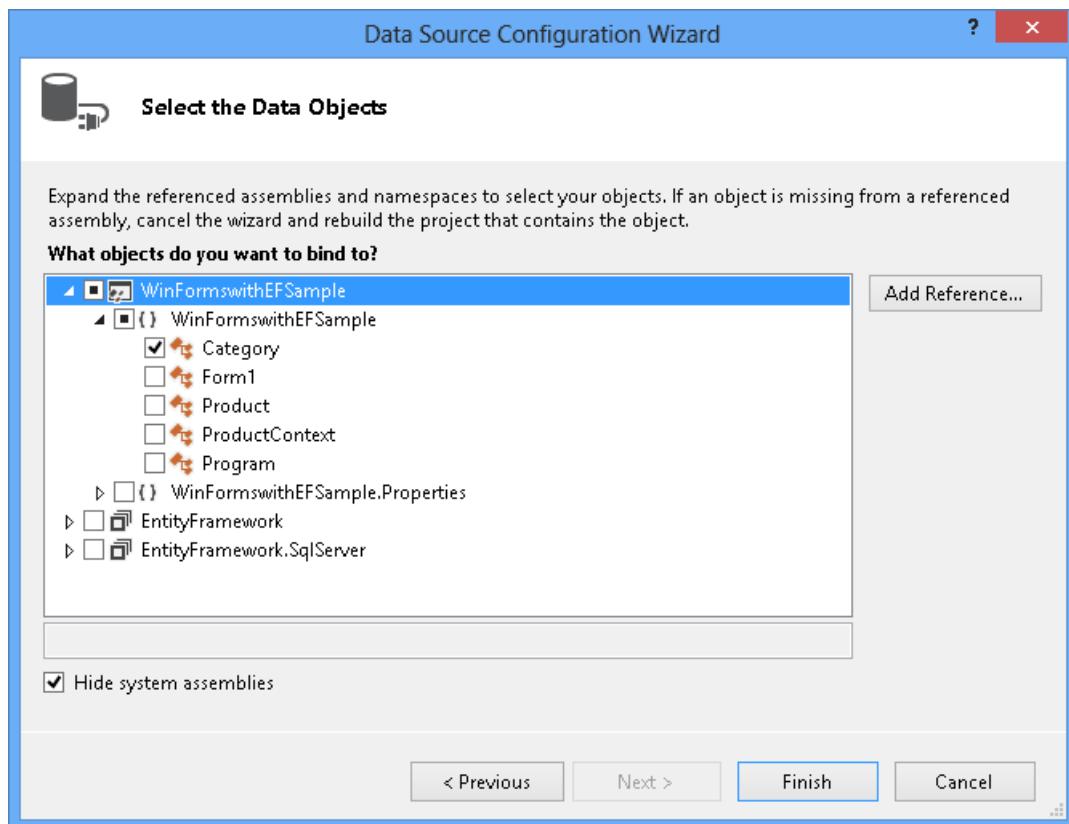
EF дает возможность загрузки связанных сущностей из базы данных автоматически при первом обращении к свойству навигации. С помощью этого типа загрузки (называется отложенной загрузки) Имейте в виду, что при первом доступе к каждому свойству навигации отдельный запрос будет выполнен в базе данных, если содержимое еще не в контексте.

При использовании типов сущностей POCO, EF обеспечивает отложенную загрузку, создании экземпляров производного прокси-типов во время выполнения и затем переопределение виртуальных свойств в классах для добавления обработчика загрузки. Чтобы получить отложенная загрузка связанных объектов, необходимо объявить методы получения свойств как навигации **открытый и виртуального (Overridable** в Visual Basic), и вы класс не должен быть **запечатанный (NotOverridable** в Visual Basic). Когда базы данных с помощью первого свойства навигации, автоматически становятся виртуальные, чтобы включить отложенную загрузку. В разделе Code First, мы решили сделать виртуальные свойства навигации по той же причине

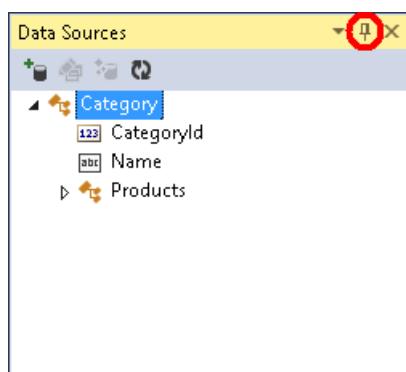
Привязка объектов к элементам управления

Добавьте классы, которые определены в модели в качестве источников данных для этого приложения WinForms.

- В главном меню выберите **проект -> добавить новый источник данных...** (в Visual Studio 2010, необходимо выбрать **тарифный план —> добавить новый источник данных...**)
- В списке выберите тип источника данных окна выберите **объект** и нажмите кнопку **Далее**
- В диалоговом окне объекты данных выберите unfold **WinFormswithEFSample** двух раз и выберите **категории** нет необходимости для выбора источника данных Product, так как мы вернемся к нему через продукта свойство в источнике данных категории.



- Нажмите кнопку **Готово**. Если окно "Источники данных", не отображается, выберите *** представление —> Other Windows -> источников данных*
- Скрыть значок ПИН-код, поэтому окна источников данных автоматически. Может потребоваться нажмите кнопку "Обновить", если окно уже было открыто.

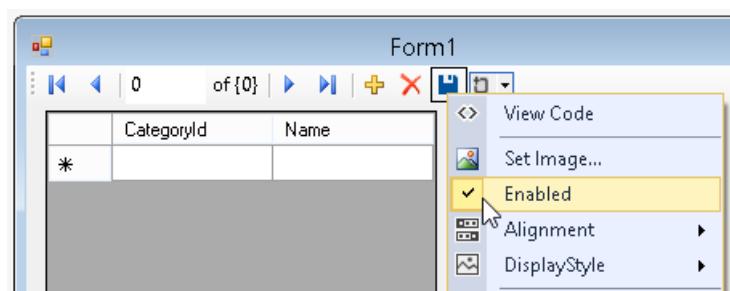


- В обозревателе решений дважды щелкните **Form1.cs** файл, чтобы открыть главную форму в конструкторе.
- Выберите **категории** источника данных и перетащите его на форме. По умолчанию новый DataGridView (**categoryDataGridView**) и элементы управления панели инструментов навигации добавляются в конструктор. Эти элементы управления привязаны к BindingSource (**categoryBindingSource**) и привязка Навигатор (**categoryBindingNavigator**) компонентов, которые также будут созданы.
- Изменить столбцы, на **categoryDataGridView**. Нам нужно выбрать **CategoryId** столбец только для чтения. Значение для **CategoryId** после сохранения данных в базе данных создается свойство.
 - Щелкните правой кнопкой мыши элемента управления DataGridView и выберите столбцы, изменить...
 - Выберите столбец CategoryId и присвоено значение True только для чтения
 - Нажмите кнопку OK

- Выберите продукты из категории источника данных и перетащите его на форме. ProductDataGridView и productBindingSource добавляются в форму.
- Измените столбцы, на productDataGridView. Нам нужно скрыть столбцы CategoryId и категории и значение ProductId только для чтения. Значение свойства ProductId создается в базе данных после сохранения данных.
 - Щелкните правой кнопкой мыши элемента управления DataGridView и выберите **Правка столбцов...**.
 - Выберите **ProductId** столбцы и наборы **ReadOnly** для **True**.
 - Выберите **CategoryId** столбец и нажмите клавишу **удалить** кнопки. Сделайте то же самое с **категории** столбца.
 - Нажмите клавишу **OK**.

На данный момент мы связанный элемент управления DataGridView с компонентами BindingSource в конструкторе. В следующем разделе будет добавлен код для кода программной части присвоено categoryBindingSource.DataSource коллекцию сущностей, которые в настоящее время отслеживаются с DbContext. При мы перетащить и удалении продукты из категории, WinForms занималась Настройка свойство productsBindingSource.DataSource свойству categoryBindingSource и productsBindingSource.DataMember к продуктам. Из-за этой привязкой, только продукты, принадлежащие к текущей выбранной категории будет отображаться в productDataGridView.

- Включить **Сохранить** кнопку на панели навигации, щелкнув правой кнопкой мыши и выбрав **включено**.



- Добавьте обработчик событий для сохранения кнопки, дважды щелкнув кнопку. Это будет добавьте обработчик событий и вы хотели кода программной части формы. Код для **categoryBindingNavigatorSaveItem_Click** обработчик событий будет добавлена в следующем разделе.

Добавьте код, который обрабатывает взаимодействие с данными

Теперь мы добавим код, чтобы использовать ProductContext производить доступ к данным. Обновите код для окна главной формы, как показано ниже.

Код объявляет экземпляр ProductContext выполняющейся длительное время. Объект ProductContext позволяет запрашивать и сохранять данные в базу данных. Метод Dispose() в экземпляре ProductContext затем вызывается из переопределенного метода OnClosing. В комментариях к коду содержат сведения о что делает код.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
```

```

using System.Data.Entity;

namespace WinFormswithEFSample
{
    public partial class Form1 : Form
    {
        ProductContext _context;
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnLoad(EventArgs e)
        {
            base.OnLoad(e);
            _context = new ProductContext();

            // Call the Load method to get the data for the given DbSet
            // from the database.
            // The data is materialized as entities. The entities are managed by
            // the DbContext instance.
            _context.Categories.Load();

            // Bind the categoryBindingSource.DataSource to
            // all the Unchanged, Modified and Added Category objects that
            // are currently tracked by the DbContext.
            // Note that we need to call ToBindingList() on the
            // ObservableCollection<TEntity> returned by
            // the DbSet.Local property to get the BindingList<T>
            // in order to facilitate two-way binding in WinForms.
            this.categoryBindingSource.DataSource =
                _context.Categories.Local.ToBindingList();
        }

        private void categoryBindingNavigatorSaveItem_Click(object sender, EventArgs e)
        {
            this.Validate();

            // Currently, the Entity Framework doesn't mark the entities
            // that are removed from a navigation property (in our example the Products)
            // as deleted in the context.
            // The following code uses LINQ to Objects against the Local collection
            // to find all products and marks any that do not have
            // a Category reference as deleted.
            // The ToList call is required because otherwise
            // the collection will be modified
            // by the Remove call while it is being enumerated.
            // In most other situations you can do LINQ to Objects directly
            // against the Local property without using ToList first.
            foreach (var product in _context.Products.Local.ToList())
            {
                if (product.Category == null)
                {
                    _context.Products.Remove(product);
                }
            }

            // Save the changes to the database.
            this._context.SaveChanges();

            // Refresh the controls to show the values
            // that were generated by the database.
            this.categoryDataGridView.Refresh();
            this.productsDataGridView.Refresh();
        }

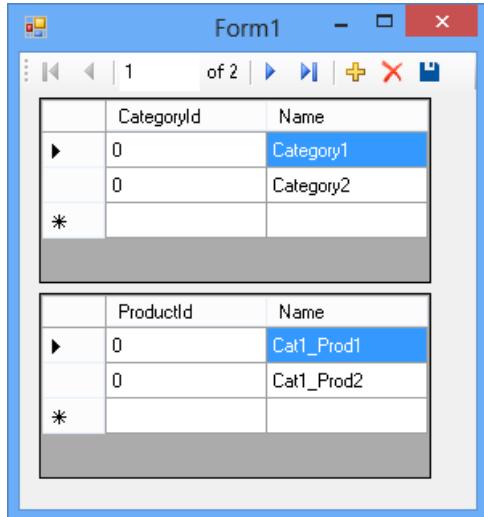
        protected override void OnClosing(CancelEventArgs e)
        {
            base.OnClosing(e);
        }
    }
}

```

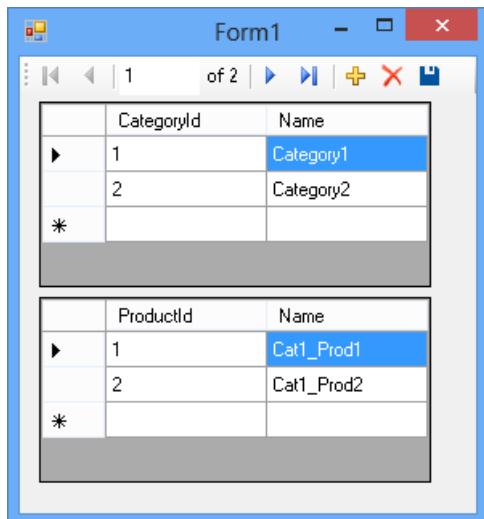
```
        this._context.Dispose();
    }
}
```

Тестирование приложения Windows Forms

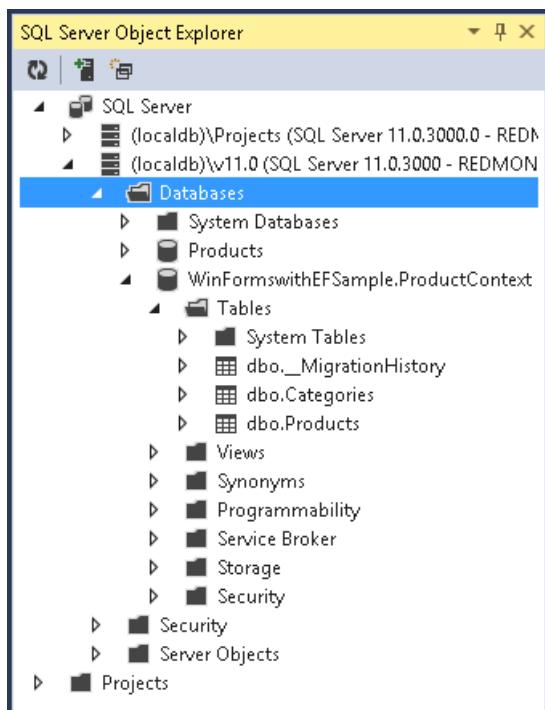
- Компиляции и выполнения приложения и вы можете проверить работу функции.



- После сохранения хранения созданных ключей отображаются на экране.



- Если вы использовали Code First, то вы также увидите, что **WinFormswithEFSample.ProductContext** база данных создается автоматически.



Привязка данных с помощью WPF

04.10.2018 • 22 minutes to read • [Edit Online](#)

Это пошаговое руководство показывает, как выполнить привязку к элементам управления WPF в форме «основной / подробности» типов РОСО. Приложение использует API-интерфейсов Entity Framework для заполнения объектов с данными из базы данных, отслеживать изменения и сохранения данных в базе данных.

Модель определяет два типа, участвующие в связи один ко многим: **категории** (основной\master) и **продукта** (зависимые\подробно). Затем в Visual Studio tools используются для привязки типов, определенных в модели, чтобы элементы управления WPF. Инфраструктура привязки данных WPF позволяет осуществлять навигацию между связанными объектами: Выбор строк в главное представление приводит представлении «Подробности» для обновления с соответствующими дочерними данными.

Снимки экрана и примеры кода в этом пошаговом руководстве, взяты из Visual Studio 2013, но можно выполнить в этом пошаговом руководстве с помощью Visual Studio 2012 или Visual Studio 2010.

Используйте параметр «Object» для создания источников данных в WPF

С помощью предыдущей версии Entity Framework мы использовали рекомендуем использовать **базы данных** параметр при создании нового источника данных на основе модели создан с помощью конструктора EF. Это было, так как конструктор создаст контекст, производный от ObjectContext и классов сущностей, производные от класса EntityObject. С помощью параметра базы данных поможет вам создавать лучший код для взаимодействия с этой областью API.

Конструкторы для Visual Studio 2012 и Visual Studio 2013 EF создать контекст, который является производным от DbContext вместе с простых классов сущностей РОСО. С помощью Visual Studio 2010, мы рекомендуем обращений шаблон генерации кода, использующего DbContext, как описано далее в этом пошаговом руководстве.

При использовании рабочей области DbContext API следует использовать **объект** параметр при создании нового источника данных, как показано в этом пошаговом руководстве.

При необходимости вы можете [вернуться к созданию кода на основе ObjectContext](#) для моделей, созданных с помощью конструктора EF.

Предварительные требования

Вам потребуется Visual Studio 2013, Visual Studio 2012 или Visual Studio 2010 установлен для выполнения этого пошагового руководства.

Если вы используете Visual Studio 2010, необходимо также установить NuGet. Дополнительные сведения см. в разделе [Установка NuGet](#).

Создание приложения

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в левой области и **WPFApplication** в области справа
- Введите **WPFwithEFSample** как имя

- Нажмите кнопку **OK**

Установите пакет Entity Framework NuGet

- В обозревателе решений щелкните правой кнопкой мыши **WinFormswithEFSample** проекта
- Выберите **управление пакетами NuGet...**
- В диалоговом окне «Управление пакетами NuGet» выберите **Online** вкладку и выберите **EntityFramework** пакета
- Нажмите кнопку **установки**

NOTE

Помимо сборки EntityFramework также добавляется ссылка System.ComponentModel.DataAnnotations. Если проект содержит ссылку на System.Data.Entity, затем он будет удален при установке EntityFramework. Сборка System.Data.Entity больше не используется для приложений Entity Framework 6.

Определение модели

В этом пошаговом руководстве, вы можете решить реализовать модель с помощью Code First или конструкторе EF. Выполните одно из двух следующих разделах.

Вариант 1: Определение модели с помощью Code First

В этом разделе показано, как создать модель и его связанные базы данных, с помощью Code First. Перейдите к следующему разделу (**вариант 2: определение модели с помощью Database First**) Если вы предпочитаете использовать Database First обратить реконструирование модели из базы данных в конструкторе EF

При использовании шаблона разработки Code First обычно начинается с написания классов .NET Framework, которые определяют модель концептуальный (домен).

- Добавьте новый класс к **WPFwithEFSample**:
 - Щелкните правой кнопкой мыши имя проекта
 - Выберите **добавить**, затем **новый элемент**
 - Выберите **класс** и введите **продукта** имени класса
- Замените **продукта** класса определения следующим кодом:

```

namespace WPFwithEFSample
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }

        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}

```

- Add a ****Category**** class with the following definition:

```

using System.Collections.ObjectModel;

namespace WPFwithEFSample
{
    public class Category
    {
        public Category()
        {
            this.Products = new ObservableCollection<Product>();
        }

        public int CategoryId { get; set; }
        public string Name { get; set; }

        public virtual ObservableCollection<Product> Products { get; private set; }
    }
}

```

Продуктов свойство **категории** класс и **категории** свойство **продукта** класс — это свойства навигации. В Entity Framework свойства навигации позволяют для перехода по связи между двумя типами сущностей.

Кроме определения сущностей, необходимо определить класс, который является производным от DbContext и DbSet предоставляет < TEntity > свойства. DbSet < TEntity > свойства let контекст знать, какие типы, которые вы хотите включить в модель.

Экземпляр типа DbContext производным управляет объектов сущности во время выполнения, который включает заполнение объектов с данными из базы данных, изменение отслеживания и сохранения данных в базу данных.

- Добавьте новый **ProductContext** класс в проект со следующим определением:

```

using System.Data.Entity;

namespace WPFwithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}

```

Скомпилируйте проект.

Вариант 2: Определение модели с помощью Database First

В этом разделе демонстрируется использование Database First чтобы реконструировать модель из базы данных в конструкторе EF. Если действия в предыдущем разделе (**вариант 1: определение модели с помощью Code First**), пропустите этот раздел и перейти непосредственно к **отложенной загрузки** раздел.

Создание базы данных

Обычно при ориентировании существующей базы данных, он будет уже создан, но для этого пошагового руководства необходимо создать базу данных для доступа к.

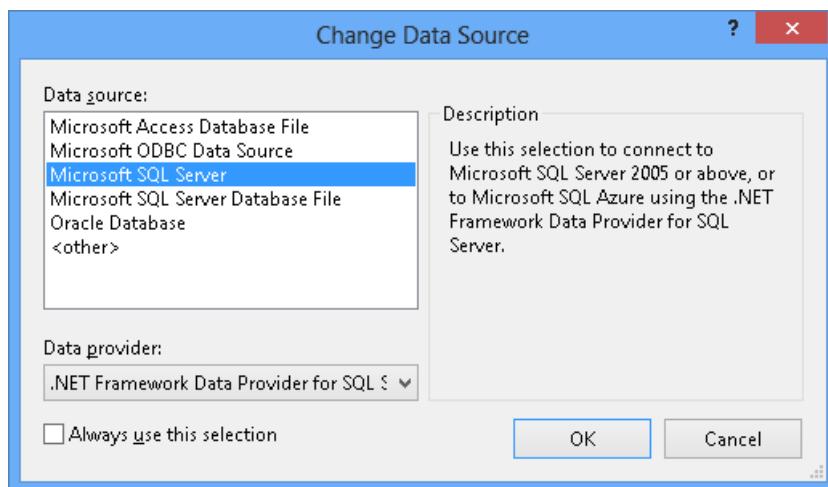
Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, вы установили:

- Если вы используете Visual Studio 2010 вы создадите базу данных SQL Express.
- Если вы используете Visual Studio 2012, а затем вы создадите [LocalDB](#) базы данных.

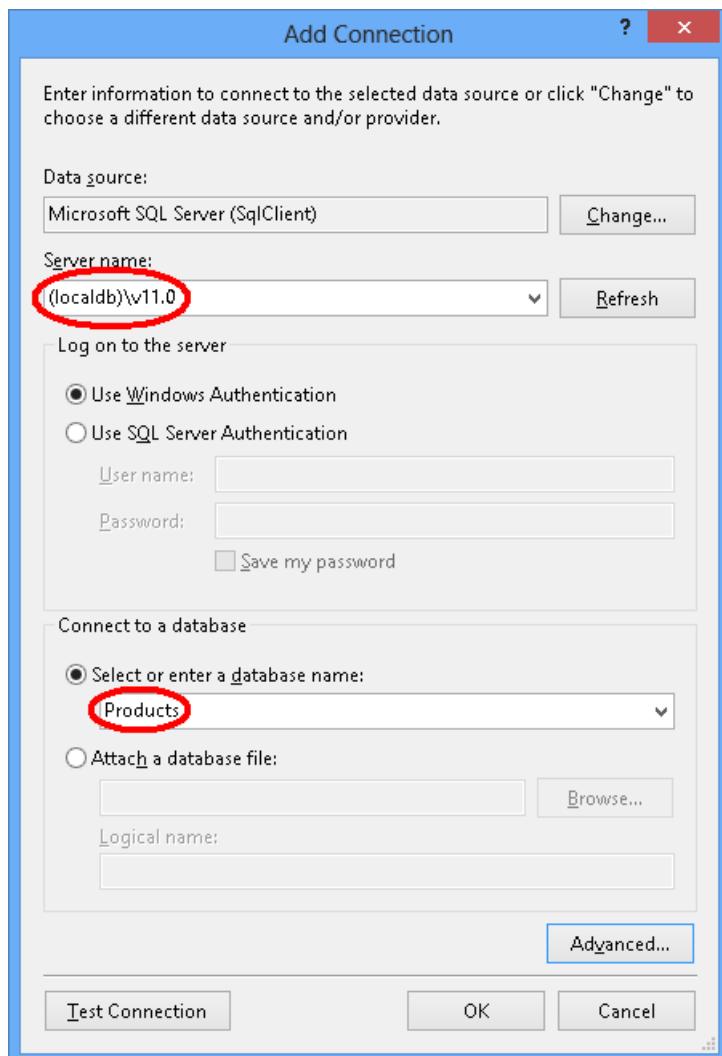
Перейдем дальше и создать базу данных.

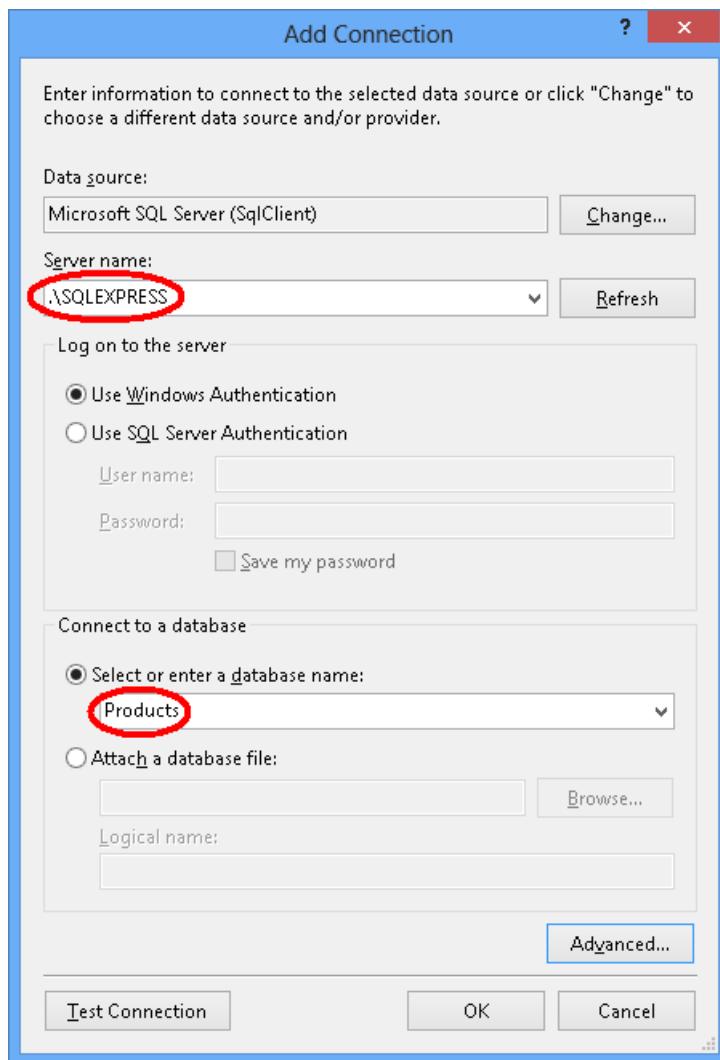
- **Представление —> обозревателя серверов**

- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**
- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам потребуется выбрать в качестве источника данных Microsoft SQL Server



- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них вы установили и введите **продуктов** имя базы данных





- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**



- Новой базы данных будут отображаться в обозревателе сервера щелкните его правой кнопкой мыши и выберите **новый запрос**
- Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**

```

CREATE TABLE [dbo].[Categories] (
    [CategoryId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    CONSTRAINT [PK_dbo.Categories] PRIMARY KEY ([CategoryId])
)

CREATE TABLE [dbo].[Products] (
    [ProductId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    [CategoryId] [int] NOT NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY ([ProductId])
)

CREATE INDEX [IX_CategoryId] ON [dbo].[Products]([CategoryId])

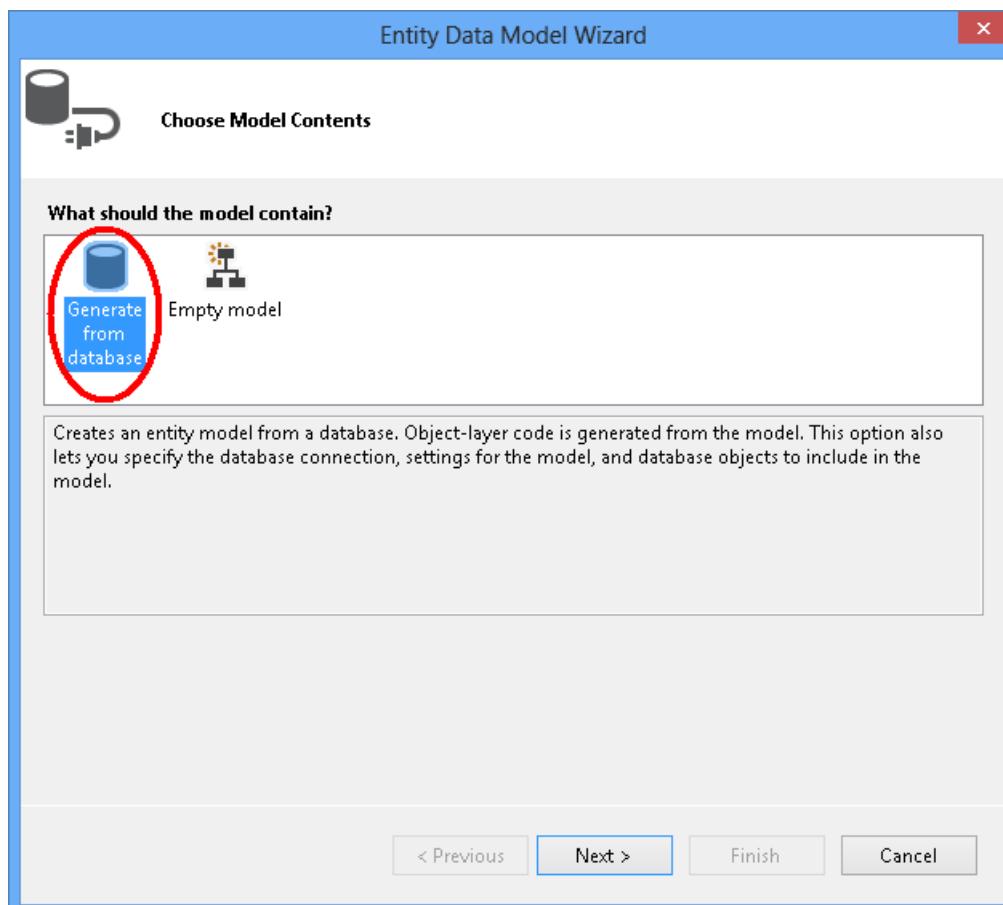
ALTER TABLE [dbo].[Products] ADD CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY
([CategoryId]) REFERENCES [dbo].[Categories] ([CategoryId]) ON DELETE CASCADE

```

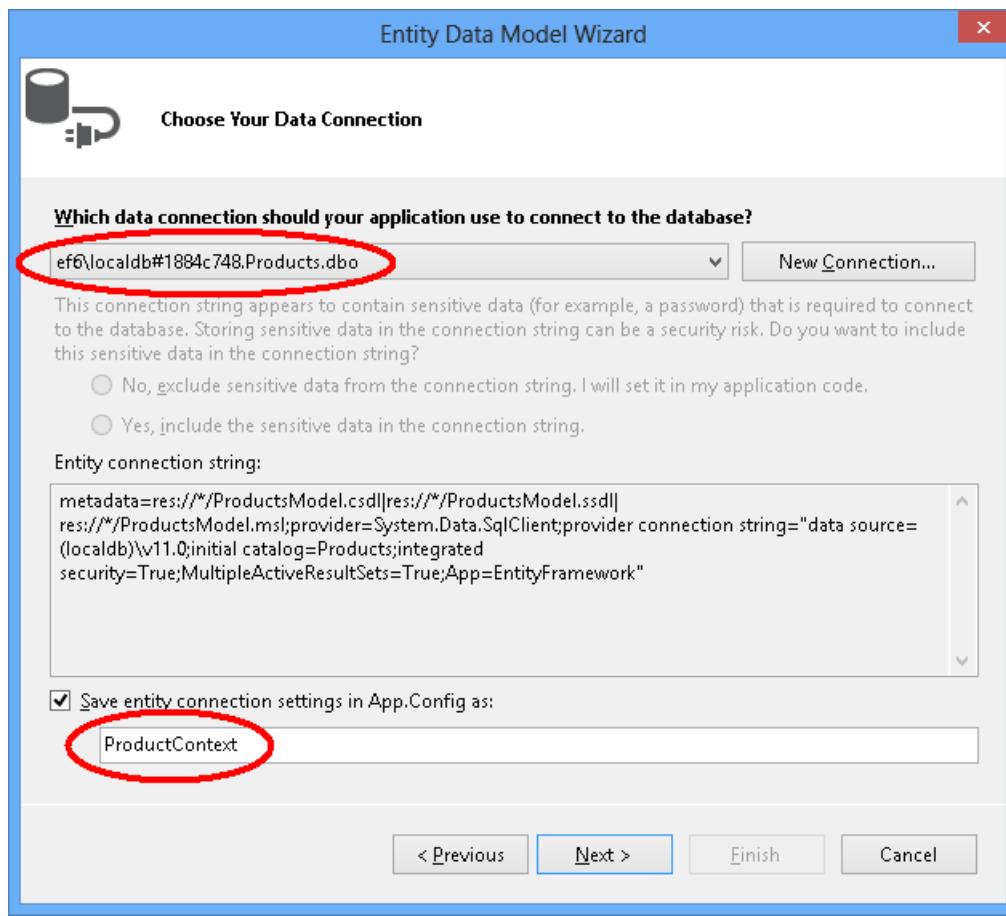
Реконструирование модели

Мы собираемся использовать Entity Framework Designer, который входит в состав Visual Studio, для создания нашей модели.

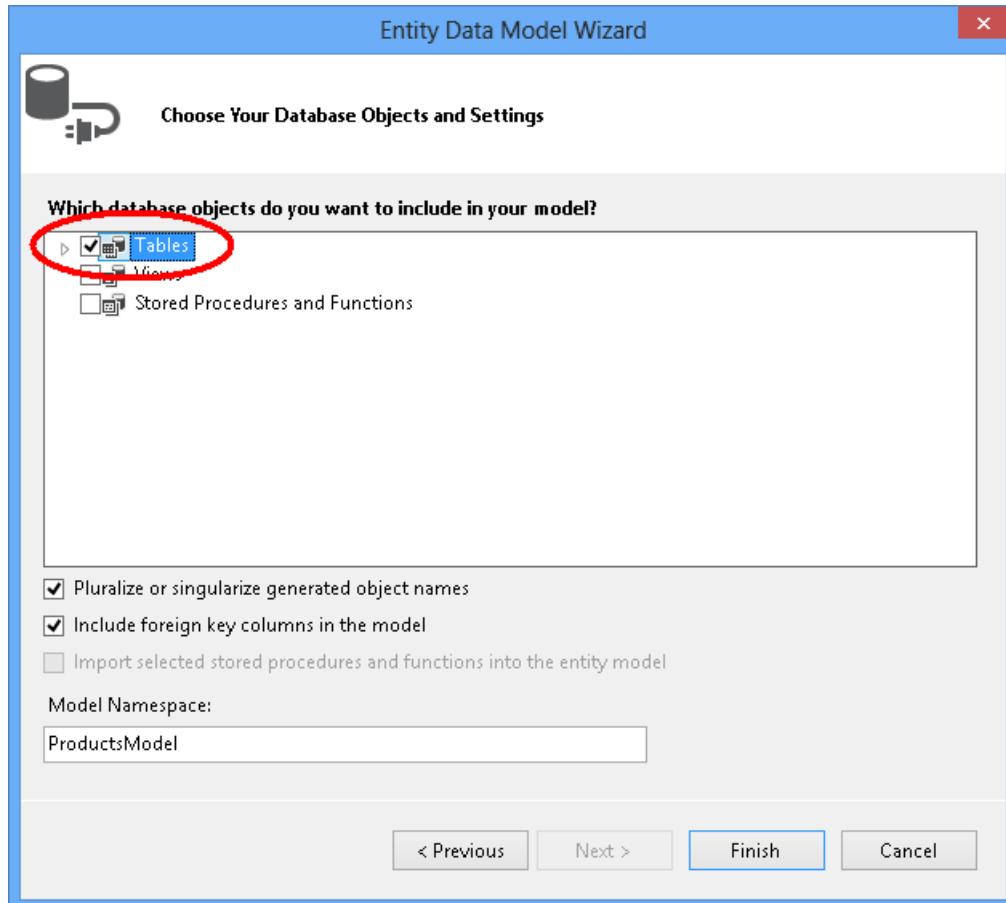
- Проект -> добавить новый элемент...
- Выберите **данных** в меню слева и затем **модель EDM ADO.NET**
- Введите **ProductModel** имя и нажмите кнопку **OK**
- Это откроет **мастер моделей EDM**
- Выберите **создать из базы данных** и нажмите кнопку **Далее**



- Выберите соединение с базой данных, созданной в первом разделе, введите **ProductContext** как имя строки подключения и нажмите кнопку **Далее**



- Установите флажок рядом с «Таблицы», чтобы импортировать все таблицы и нажмите кнопку «Готово»



После завершения процесса реконструирования новой модели добавлен в проект и открывается для просмотра в конструкторе Entity Framework. Файл App.config также был добавлен в проект со сведениями о

подключении для базы данных.

Дополнительные действия в Visual Studio 2010

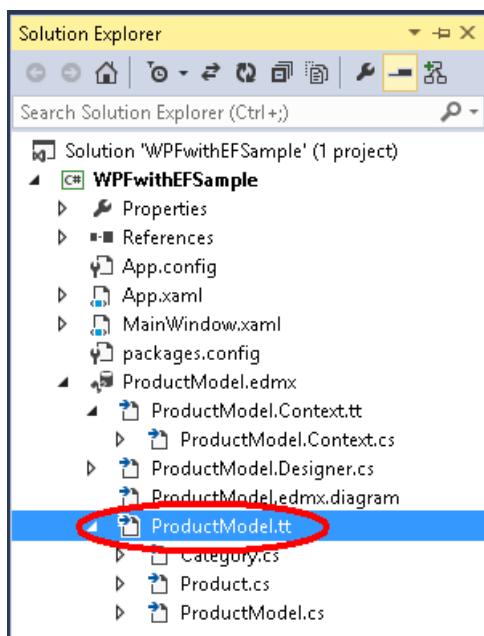
Если вы работаете в Visual Studio 2010 будет необходимо обновить конструктор EF будет использовать создание кода EF6.

- Щелкните правой кнопкой мыши пустое место модели в конструкторе EF и выберите **добавить элемент формирования кода...**
- Выберите **шаблоны в Интернете** из меню слева и выполните поиск **DbContext**
- Выберите **EF 6.x генератор DbContext для C#**, введите **ProductsModel** как имя и щелкните "Добавить"

Обновление создания кода для привязки данных

EF создает код из модели с помощью шаблонов T4. Шаблоны, поставляемых с Visual Studio или скачать из коллекции Visual Studio предназначены для использования общего назначения. Это означает, что сущности, созданные на основе этих шаблонов имеют простой `ICollection<T>` свойства. Тем не менее, при этом данные привязки с использованием WPF желательно использовать **ObservableCollection** для свойств коллекции, так что WPF можно отслеживать, изменения, внесенные в коллекции. Для этого нужно будет изменить шаблоны для использования ObservableCollection.

- Откройте **обозреватель решений** и найти **ProductModel.edmx** файла
- Найти **ProductModel.tt** файл, который будет вложен в файле ProductModel.edmx



- Дважды щелкните файл ProductModel.tt, чтобы открыть его в редакторе Visual Studio
- Найти и заменить два вхождения "`ICollection`" с "ObservableCollection". Они находятся приблизительно в строках 296 и 484.
- Найти и заменить первое вхождение "`HashSet`" с "ObservableCollection". Это событие находится примерно в строке 50. Не замените второе вхождение HashSet Найти далее в коде.
- Поиск и замена только вхождения "`System.Collections.Generic`" с "System.Collections.ObjectModel". Это примерно в строке 424.
- Сохраните файл ProductModel.tt. Это может вызвать код для сущностей, быть создан повторно. Если код не создается автоматически, затем щелкнуть правой кнопкой мыши ProductModel.tt и выберите «Запустить пользовательский инструмент».

Если вы теперь откроите файл Category.cs (который вложен в ProductModel.tt), то вы увидите, что продукты коллекция имеет тип **ObservableCollection<продукта>**.

Скомпилируйте проект.

Отложенная загрузка

Продуктов свойство **категории** класс и **категории** свойство **продукта** класс — это свойства навигации. В Entity Framework свойства навигации позволяют для перехода по связи между двумя типами сущностей.

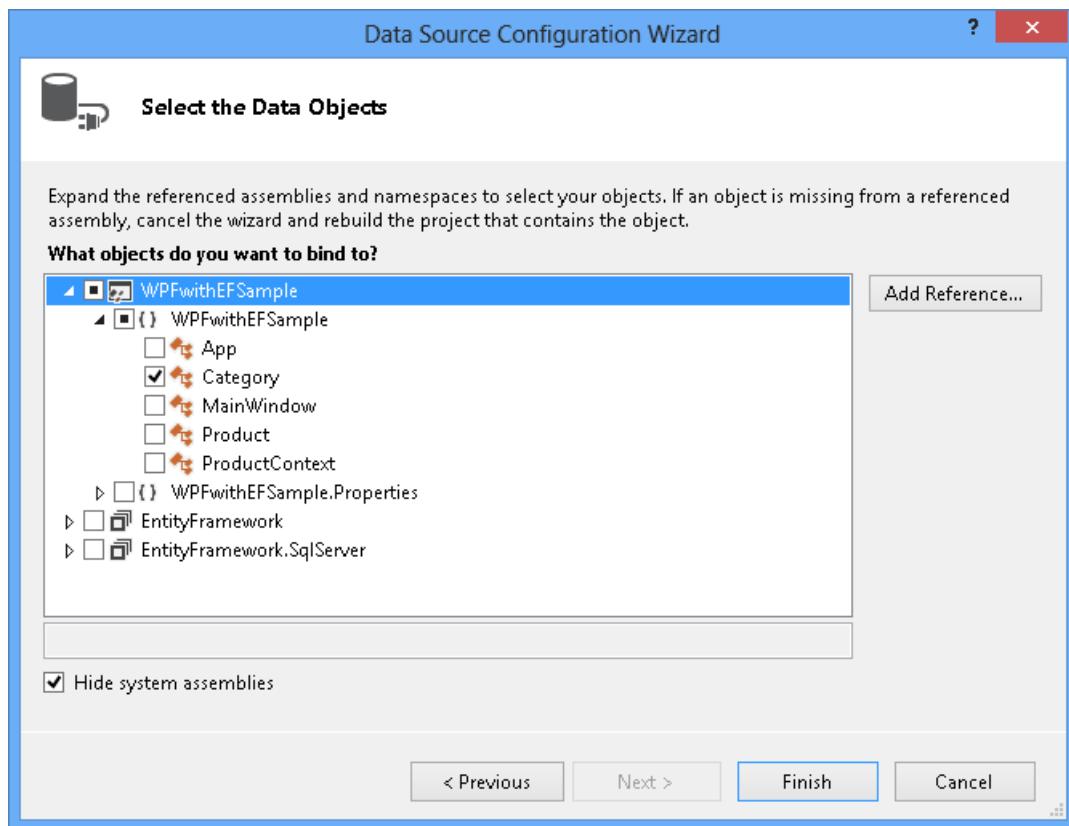
EF дает возможность загрузки связанных сущностей из базы данных автоматически при первом обращении к свойству навигации. С помощью этого типа загрузки (называется отложенной загрузки) Имейте в виду, что при первом доступе к каждому свойству навигации отдельный запрос будет выполнен в базе данных, если содержимое еще не в контексте.

При использовании типов сущностей POCO, EF обеспечивает отложенную загрузку, создании экземпляров производного прокси-типов во время выполнения и затем переопределение виртуальных свойств в классах для добавления обработчика загрузки. Чтобы получить отложенная загрузка связанных объектов, необходимо объявить методы получения свойств как навигации **открытый и виртуального (Overridable** в Visual Basic), и вы класс не должен быть **запечатанный (NotOverridable** в Visual Basic). Когда базы данных с помощью первого свойства навигации, автоматически становятся виртуальные, чтобы включить отложенную загрузку. В разделе Code First, мы решили сделать виртуальные свойства навигации по той же причине

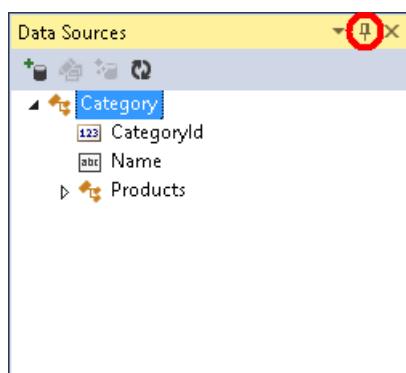
Привязка объектов к элементам управления

Добавьте классы, которые определены в модели в качестве источников данных для этого приложения WPF.

- Дважды щелкните **MainWindow.xaml** в обозревателе решений, чтобы открыть главную форму
- В главном меню выберите **проект -> добавить новый источник данных...** (в Visual Studio 2010, необходимо выбрать **тарифный план —> добавить новый источник данных...**)
- В выберите Typewindow источника данных, выберите **объект** и нажмите кнопку **Далее**
- В диалоговом окне объекты данных выберите unfold **WPFwithEFSample** двух раз и выберите **категории**
*Нет необходимости, чтобы выбрать **продукта** источника данных, так как мы вернемся к нему через **продукт** свойство **категории** источника данных*



- Нажмите кнопку **Готово**.
- Открывается окно "Источники данных" рядом с окном MainWindow.xaml. Если окно "Источники данных", не отображается, выберите **представление** —> **Other Windows -> источников данных**
- Скрыть значок ПИН-код, поэтому окна источников данных автоматически. Может потребоваться нажмите кнопку "Обновить", если окно уже было открыто.



- Выберите **категории** источника данных и перетащите его на форме.

Когда мы перетащили этот источник произошло следующее:

- **CategoryViewSource** ресурсов и **categoryDataGrid** управления были добавлены в XAML
- Свойство **DataContext** для родительского элемента сетки установлено в «{StaticResource categoryViewSource}». **CategoryViewSource** ресурсов выступает в качестве источника привязки для внешнего\родительского элемента сетки. Внутренние элементы сетки, наследуют значение **DataContext** от родительского элемента сетки (свойство **ItemsSource** **categoryDataGrid** имеет значение «{Binding}»)

```

<Window.Resources>
    <CollectionViewSource x:Key="categoryViewSource"
        d:DesignSource="{d:DesignInstance {x:Type local:Category}, CreateList=True}"/>
</Window.Resources>
<Grid DataContext="{StaticResource categoryViewSource}">
    <DataGrid x:Name="categoryDataGrid" AutoGenerateColumns="False" EnableRowVirtualization="True"
        ItemsSource="{Binding}" Margin="13,13,43,191"
        RowDetailsVisibilityMode="VisibleWhenSelected">
        <DataGrid.Columns>
            <DataGridTextColumn x:Name="categoryIdColumn" Binding="{Binding CategoryId}"
                Header="Category Id" Width="SizeToHeader"/>
            <DataGridTextColumn x:Name="nameColumn" Binding="{Binding Name}"
                Header="Name" Width="SizeToHeader"/>
        </DataGrid.Columns>
    </DataGrid>
</Grid>

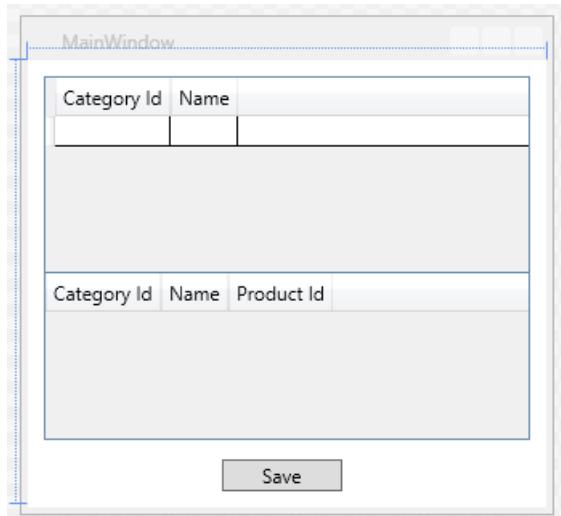
```

Добавление сетки сведений

Теперь, когда у нас есть сетки для отображения категорий, давайте добавьте сведения о сетке для отображения соответствующих продуктов.

- Выберите **продуктов** свойства в разделе **категории** источника данных и перетащите его на форму.
 - **CategoryProductsViewSource** ресурсов и **productDataGrid** сетки добавляются в XAML
 - Путь привязки для этого ресурса будет присвоено продуктов
 - Инфраструктура привязки данных WPF гарантирует, что только продукты, относящиеся к выбранной категории будут отображаться в **productDataGrid**
- Из панели элементов перетащите **кнопку** в форму. Задайте **имя** свойства **buttonSave** и **содержимого** свойства **Сохранить**.

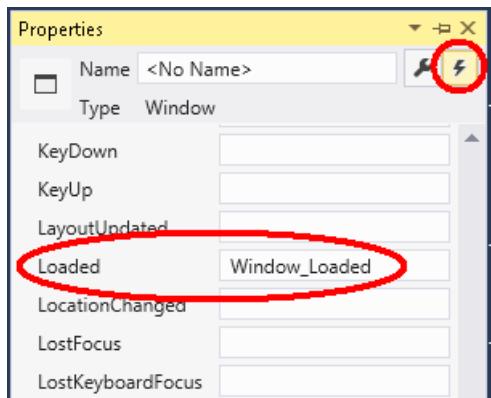
Форма должна выглядеть примерно так:



Добавьте код, который обрабатывает взаимодействие с данными

Пришло время добавить некоторые обработчики событий в главное окно.

- В окне XAML, щелкните **<окно** элемент, Выбор главного окна
- В **свойства** выберите окно **события** в правом верхнем углу, затем дважды щелкните текстовое поле справа от **Loaded** метки



- Также добавьте **щелкните** событие для **Сохранить** кнопки, дважды щелкнув "Сохранить" в конструкторе.

Откроется кода программной части формы, теперь мы отредактируем код, чтобы использовать ProductContext производить доступ к данным. Обновите код для MainWindow, как показано ниже.

Код объявляет экземпляр длительных **ProductContext**. **ProductContext** объект используется для запроса и сохранения данных в базу данных. **Dispose()** на **ProductContext** экземпляра вызывается из переопределенного **OnClosing** метод. В комментариях к коду содержат сведения о что делает код.

```

using System.Data.Entity;
using System.Linq;
using System.Windows;

namespace WPFwithEFSample
{
    public partial class MainWindow : Window
    {
        private ProductContext _context = new ProductContext();
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            System.Windows.Data.CollectionViewSource categoryViewSource =
                ((System.Windows.Data.CollectionViewSource)(this.FindResource("categoryViewSource")));

            // Load is an extension method on IQueryable,
            // defined in the System.Data.Entity namespace.
            // This method enumerates the results of the query,
            // similar to ToList but without creating a list.
            // When used with Linq to Entities this method
            // creates entity objects and adds them to the context.
            _context.Categories.Load();

            // After the data is loaded call the DbSet<T>.Local property
            // to use the DbSet<T> as a binding source.
            categoryViewSource.Source = _context.Categories.Local;
        }

        private void buttonSave_Click(object sender, RoutedEventArgs e)
        {
            // When you delete an object from the related entities collection
            // (in this case Products), the Entity Framework doesn't mark
            // these child entities as deleted.
            // Instead, it removes the relationship between the parent and the child
            // by setting the parent reference to null.
            // So we manually have to delete the products
            // that have a Category reference set to null.
        }
    }
}

```

```

// The following code uses LINQ to Objects
// against the Local collection of Products.
// The ToList call is required because otherwise the collection will be modified
// by the Remove call while it is being enumerated.
// In most other situations you can use LINQ to Objects directly
// against the Local property without using ToList first.
foreach (var product in _context.Products.Local.ToList())
{
    if (product.Category == null)
    {
        _context.Products.Remove(product);
    }
}

_context.SaveChanges();
// Refresh the grids so the database generated values show up.
this.categoryDataGrid.Items.Refresh();
this.productsDataGrid.Items.Refresh();
}

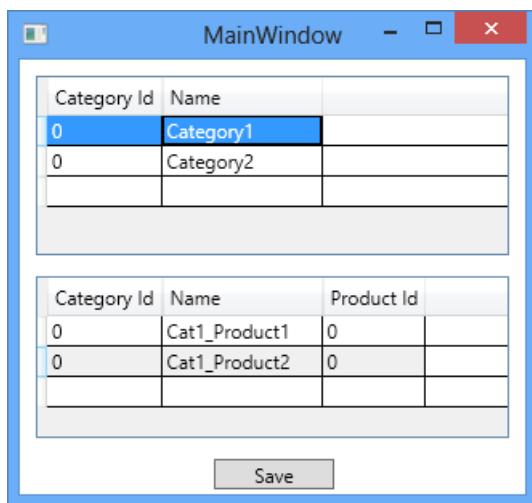
protected override void OnClosing(System.ComponentModel.CancelEventArgs e)
{
    base.OnClosing(e);
    this._context.Dispose();
}
}

}

```

Тестирование приложения WPF

- Скомпилируйте и запустите приложение. Если вы использовали Code First, то вы увидите, что **WPFwithEFSample.ProductContext** база данных создается автоматически.
- Ведите имя категории в верхней сетке и продукта имена в нижней таблице *ничего не вводится в столбцов ID, так как первичный ключ создается в базе данных*



- Нажмите клавишу **Сохранить** кнопку, чтобы сохранить данные в базе данных

После вызова **DbContext SaveChanges()**, идентификаторы заполняются значениями сформированный базой данных. Так как мы называли **Refresh()** после **SaveChanges()** **DataGrid** элементы управления обновляются с использованием новых значений.

The screenshot shows a Windows application window titled "MainWindow". Inside the window, there are two data grids and a "Save" button.

The first data grid has columns "Category Id" and "Name". It contains three rows:

Category Id	Name
3	Category1
4	Category2

The second data grid has columns "Category Id", "Name", and "Product Id". It contains four rows:

Category Id	Name	Product Id
4	Cat2_Product1	5
4	Cat2_Product2	6

Below the data grids is a "Save" button.

Дополнительные ресурсы

Дополнительные сведения о привязке данных в коллекции с помощью WPF, см. в разделе [в этом разделе](#) в документации по WPF.

Работа с отключенными сущностями

13.09.2018 • 3 minutes to read • [Edit Online](#)

В приложении на основе Entity Framework класс контекста несет ответственность за обнаружение изменений, применяемых к отслеживаемым сущностям. Вызов метода `SaveChanges` сохраняет изменения, отслеживаемые по контексту, в базу данных. При работе с н-уровневыми приложениями объекты сущностей, как правило, изменяются при отключении от контекста, и необходимо решить, как отслеживать изменения и передавать эти изменения обратно в контекст. В этом разделе рассматриваются различные параметры, доступные при использовании Entity Framework с отключенными сущностями.

Платформы веб-службы

Как правило, технологии веб-служб поддерживают шаблоны, которые могут использоваться для сохранения изменений в отдельных отключенных объектах. Например, веб-API ASP.NET позволяет написать код для действий контроллера, которые могут включать вызовы к EF для сохранения изменений, внесенных в объект в базе данных. По сути, средства веб-API в Visual Studio упрощают автоматическую генерацию контроллера веб-API из модели Entity Framework 6. Дополнительные сведения см. в разделе [Использование веб-API с Entity Framework 6](#).

Существовали и другие технологии веб-служб, которые обеспечивали интеграцию с Entity Framework, например [WCF Data Services](#) и [RIA Services](#).

Низкоуровневые интерфейсы API EF

Если вы не хотите использовать имеющееся н-уровневое решение или хотите настроить, что происходит внутри действия контроллера в службах веб-API, платформа Entity Framework предоставляет интерфейсы API, в которых можно вносить изменения на отключенном уровне. Дополнительные сведения см. в разделе [Добавление, вложение и состояние сущности](#).

Сущности с самостоятельным отслеживанием

Отслеживать изменения на произвольных графах сущностей при отсутствии подключения к контексту EF — это задача непростая. Одной из попыток решить ее был шаблон генерации кода сущностей с самостоятельным отслеживанием. Этот шаблон создает классы сущностей, которые содержат логику для отслеживания изменений на отключенном уровне как состояния самих сущностей. Также создается набор методов расширения для применения этих изменений к контексту.

Этот шаблон можно использовать с моделями, созданными с помощью конструктора EF, но не с моделями Code First. Дополнительные сведения см. в разделе [Сущности с самостоятельным отслеживанием](#).

IMPORTANT

Больше не рекомендуется использовать шаблон сущностей с самостоятельным отслеживанием. Он по-прежнему будет доступен только для поддержки существующих приложений. Если приложению необходимо работать с отключенными графиками сущностей, рассмотрите другие варианты, такие как [отслеживаемые сущности](#), которые технологически эквивалентны сущностям с самостоятельным отслеживанием, активно разрабатываемым сообществом. Или остановитесь на написании пользовательского кода с помощью API-интерфейсов отслеживания изменений низкого уровня.

Сущности с самостоятельным отслеживанием

18.09.2018 • 9 minutes to read • [Edit Online](#)

IMPORTANT

Больше не рекомендуется использовать шаблон сущностей с самостоятельным отслеживанием. Он по-прежнему будет доступен только для поддержки существующих приложений. Если приложению необходимо работать с отключенными графиками сущностей, рассмотрите другие варианты, такие как [отслеживаемые сущности](#), которые технологически эквивалентны сущностям с самостоятельным отслеживанием, активно разрабатываемым сообществом. Или остановитесь на написании пользовательского кода с помощью API-интерфейсов отслеживания изменений низкого уровня.

В приложении на базе Entity Framework контекст отвечает за отслеживание изменений в объектах. Затем изменения сохраняются в базе данных с помощью метода `SaveChanges`. При работе с многоуровневыми приложениями объекты сущностей, как правило, отключаются от контекста и необходимо решить, как отслеживать изменения и передавать эти изменения обратно в контекст. Сущности с самостоятельным отслеживанием (STE) могут помочь отслеживать изменения на любом уровне и затем воспроизводить их в контексте для сохранения.

Используйте сущности с самостоятельным отслеживанием, только если контекст недоступен на том уровне, где вносятся изменения в граф объекта. Если контекст доступен, использовать эти сущности нет необходимости, так как отслеживать изменения будет контекст.

Этот элемент шаблона создает два ТТ-файла (text template).

- Файл `<имя_модели>.tt` создает типы сущности и вспомогательный класс, содержащий логику отслеживания изменений, используемую в сущностях с самостоятельным отслеживанием, и методы расширения, позволяющие задавать состояние этих сущностей.
- Файл `<имя_модели>.Context.tt` создает производный контекст и класс расширения, который содержит методы `ApplyChanges` для классов `ObjectContext` и `ObjectSet`. Эти методы проверяют сведения об отслеживании изменений, которые содержатся в графе сущностей с самостоятельным отслеживанием, чтобы вывести набор операций, которые следует выполнить для сохранения изменений в базе данных.

Приступая к работе

Чтобы приступить к работе, посетите страницу с [пошаговым руководством по использованию сущностей с самостоятельным отслеживанием](#).

Рекомендации по функциям при работе с сущностями с автоматическим отслеживанием

IMPORTANT

Больше не рекомендуется использовать шаблон сущностей с самостоятельным отслеживанием. Он по-прежнему будет доступен только для поддержки существующих приложений. Если приложению необходимо работать с отключенными графиками сущностей, рассмотрите другие варианты, такие как [отслеживаемые сущности](#), которые технологически эквивалентны сущностям с самостоятельным отслеживанием, активно разрабатываемым сообществом. Или остановитесь на написании пользовательского кода с помощью API-интерфейсов отслеживания изменений низкого уровня.

При работе с сущностями с самостоятельным отслеживанием нужно принимать во внимание следующие соображения.

- Убедитесь в том, что в клиентском проекте имеется ссылка на сборку, содержащую типы сущностей. Если в клиентский проект будет добавлена только ссылка на службу, то в клиентском проекте будут использоваться прокси-типы WCF, а не действительные типы сущностей с самостоятельным отслеживанием. Это означает, что не будут получены средства автоматизированного уведомления, которые управляют отслеживанием сущностей на клиенте. Если сознательно решено не включать типы сущностей, то придется вручную задавать сведения отслеживания на клиенте, чтобы изменения передавались обратно в службу.
- Вызовы этой операции службы должны быть не поддерживающими состояние и создавать новый экземпляр контекста объекта. Также рекомендуется создавать контекст объекта в блоке **using**.
- Если после отправки измененного на клиенте графа службе вы намерены продолжать работу с тем же графиком на клиенте, то вам придется вручную провести итерацию по графу на клиенте, вызывая для каждого объекта метод **AcceptChanges** для сброса состояния объекта отслеживания изменений.

Если объекты в графе содержат свойства со значениями, формируемыми базой данных (например, значения удостоверения или параллелизма), то после вызова метода **SaveChanges** Entity Framework заменит значения этих свойств значениями, сформированными базой данных. Предусмотрена возможность реализовать собственную операцию службы для возврата клиенту сохраненных объектов или списка сформированных значений свойств для объекта. Затем клиенту потребуется заменить экземпляры объектов или значения свойств объектов теми объектами или значениями свойств, которые были возвращены из операции службы.

- Объединение графов из нескольких запросов к службе может привести к появлению в итоговом графике повторяющихся значений ключей. Entity Framework не удаляет объекты с повторяющимися ключами при вызове метода **ApplyChanges**. Вместо этого формируется исключение. Для предотвращения появления графов с повторяющимися значениями ключей придерживайтесь одного из шаблонов, описанных в следующем блоге: [Сущности с самостоятельным отслеживанием. Сущности ApplyChanges и повторяющиеся сущности](#).
- При изменении связи между объектами путем задания свойства внешнего ключа ссылка на свойство навигации задается равной NULL и не синхронизируется с основной сущностью на клиенте. После присоединения графа к контексту объекта (например, после вызова метода **ApplyChanges**) свойства внешнего ключа и свойства навигации синхронизируются.

Отсутствие синхронизации свойства навигации ссылки с соответствующим основным объектом может стать причиной проблем, если задано каскадное удаление по связи по внешнему ключу. Если будет удален основной объект, операция удаления не распространится на зависимые объекты. Если заданы каскадные удаления, то для изменения связей следует использовать свойства навигации, а не задавать свойство внешнего ключа.

- Сущностям с самостоятельным отслеживанием не разрешается использовать отложенную загрузку.
- Сущности с самостоятельным отслеживанием не поддерживают двоичную сериализацию и сериализацию в объекты управления состоянием ASP.NET. Тем не менее этот шаблон можно изменить в целях добавления поддержки двоичной сериализации. Дополнительные сведения см. в статье [Using Binary Serialization and ViewState with Self-Tracking Entities](#) (Использование двоичной сериализации и ViewState с сущностями с самостоятельным отслеживанием).

Вопросы безопасности

При работе с сущностями с самостоятельным отслеживанием следует учитывать следующие рекомендации по безопасности.

- Служба не должна доверять запросам на получение или обновление данных, исходящим от клиентов и каналов, не являющихся доверенными. Клиент должен пройти проверку подлинности; следует пользоваться безопасным каналом или конвертом сообщения. Запросы клиентов на обновление или получение данных следует проверять на соответствие ожидаемым и разрешенным изменениям для данного сценария.
- Не следует использовать в качестве ключей сущностей конфиденциальных сведений (например, номера карточек социального страхования). Тогда конфиденциальные сведения не смогут случайно подвергнуться сериализации в графах сущностей с самостоятельным отслеживанием и попасть к клиенту, который не является полностью доверенным. При использовании независимых сопоставлений исходный ключ сущности, сопоставляемый с сериализуемым, также может быть отправлен клиенту.
- Чтобы избежать передачи сообщений об исключениях, содержащих конфиденциальные данные, на клиентский уровень, вызовы методов **ApplyChanges** и **SaveChanges** на уровне сервера следует упаковывать в код обработки исключений.

Самоотслеживающиеся сущности Пошаговое руководство

13.09.2018 • 19 minutes to read • [Edit Online](#)

IMPORTANT

Больше не рекомендуется использовать шаблон сущностей с самостоятельным отслеживанием. Он по-прежнему будет доступен только для поддержки существующих приложений. Если приложению необходимо работать с отключенными графиками сущностей, рассмотрите другие варианты, такие как [отслеживаемые сущности](#), которые технологически эквивалентны сущностям с самостоятельным отслеживанием, активно разрабатываемым сообществом. Или остановитесь на написании пользовательского кода с помощью API-интерфейсов отслеживания изменений низкого уровня.

В этом пошаговом руководстве описывается сценарий, в котором службы Windows Communication Foundation (WCF) предоставляет операцию, которая возвращает граф объекта. Затем клиентское приложение обрабатывает этот граф и передает изменения операции службы, которая проверяет и сохраняет их в базу данных с помощью Entity Framework.

Перед завершением этого пошагового руководства обязательно прочтите статью [сущностей с самостоятельным отслеживанием](#) страницы.

В этом пошаговом руководстве выполняются следующие действия.

- Создает базу данных для доступа к.
- Создает библиотеку классов, которая содержит модель.
- Переключений в шаблон генератора сущностей с самостоятельным отслеживанием.
- Перемещает классы сущностей в отдельном проекте.
- Создает службу WCF, которая предоставляет операции для запроса и сохранение объектов.
- Создает клиентские приложения (консольное и WPF), которые используют службу.

Мы будем использовать первой базы данных в этом пошаговом руководстве, но те же методы также применяются Model First.

Предварительные требования

Для выполнения этого пошагового руководства требуется последнюю версию Visual Studio.

Создание базы данных

Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, вы установили:

- Если вы используете Visual Studio 2012, а затем вы создадите базу данных LocalDB.
- Если вы используете Visual Studio 2010 вы создадите базу данных SQL Express.

Перейдем дальше и создать базу данных.

- Открытие Visual Studio
- **Представление —> обозревателя серверов**
- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**

- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам нужно будет выбрать **Microsoft SQL Server** как источник данных
- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них установки
- Введите **STESample** имя базы данных
- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**
- Новая база данных появится в обозревателе серверов
- Если вы используете Visual Studio 2012
 - Щелкните правой кнопкой мыши, в базе данных в обозревателе серверов и выберите **новый запрос**
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**
- Если вы используете Visual Studio 2010
 - Выберите **тарифный план** —> **Transact редактор SQL** -> **новое соединение запроса...**
 - Введите **.\
SQLEXPRESS** имя сервера и нажмите кнопку **OK**
 - Выберите **STESample** базы данных в раскрывающемся вниз в верхней части редактора запросов
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Выполнение SQL**

```

CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId])
ON DELETE CASCADE
);

SET IDENTITY_INSERT [dbo].[Blogs] ON
INSERT INTO [dbo].[Blogs] ([BlogId], [Name], [Url]) VALUES (1, N'ADO.NET Blog', N'blogs.msdn.com/adonet')
SET IDENTITY_INSERT [dbo].[Blogs] OFF
INSERT INTO [dbo].[Posts] ([Title], [Content], [BlogId]) VALUES (N'Intro to EF', N'Interesting stuff...', 1)
INSERT INTO [dbo].[Posts] ([Title], [Content], [BlogId]) VALUES (N'What is New', N'More interesting stuff...', 1)

```

Создание модели

Мы сначала поместить модели в проект.

- **Файл** —> **Новинка** —> **проекта...**
- Выберите **Visual C#** в области слева и затем **библиотеки классов**
- Введите **STESample** имя и нажмите кнопку **OK**

Теперь мы создадим простую модель в конструкторе EF для доступа к базе данных:

- **Проект** -> **добавить новый элемент...**
- Выберите **данных** в области слева и затем **модель EDM ADO.NET**
- Введите **BloggingModel** имя и нажмите кнопку **OK**

- Выберите **создать из базы данных** и нажмите кнопку **Далее**
- Введите сведения о подключении для базы данных, созданный в предыдущем разделе
- Введите **BloggingContext** как имя строки подключения и нажмите кнопку **Далее**
- Установите флажок рядом с полем **таблиц** и нажмите кнопку **Готово**

Вы сможете перейти в ШАГЕ создания кода

Теперь нам нужно отключить создание кода по умолчанию и замены для сущностей с самостоятельным отслеживанием.

Если вы используете Visual Studio 2012

- Разверните **BloggingModel.edmx** в **обозревателе решений** и удалить **BloggingModel.tt** и **BloggingModel.Context.tt** *Это отключит формирование кода по умолчанию*
- Щелкните правой кнопкой мыши пустую область в конструкторе EF и выберите пункт **добавить элемент формирования кода...**
- Выберите **Online** из левой области и выполните поиск **ТАВИТЬ генератор**
- Выберите **ТАВИТЬ генератор для C#** шаблона, введите **STETemplate** имя и нажмите кнопку **добавить**
- **STETemplate.tt** и **STETemplate.Context.tt** файлы добавляются как вложенные в файле **BloggingModel.edmx**

Если вы используете Visual Studio 2010

- Щелкните правой кнопкой мыши пустую область в конструкторе EF и выберите пункт **добавить элемент формирования кода...**
- Выберите **кода** в области слева и затем **генератора сущностей с самостоятельным отслеживанием ADO.NET**
- Введите **STETemplate** имя и нажмите кнопку **добавить**
- **STETemplate.tt** и **STETemplate.Context.tt** файлы добавляются непосредственно в проект

Переместите типы сущностей в отдельном проекте

Для использования сущностей с самостоятельным отслеживанием в клиентском приложении требуется доступ к классам сущностей, созданные из нашей модели. Так как мы не хотим предоставлять всю модель в клиентское приложение мы собираемся переместить классы сущностей в отдельном проекте.

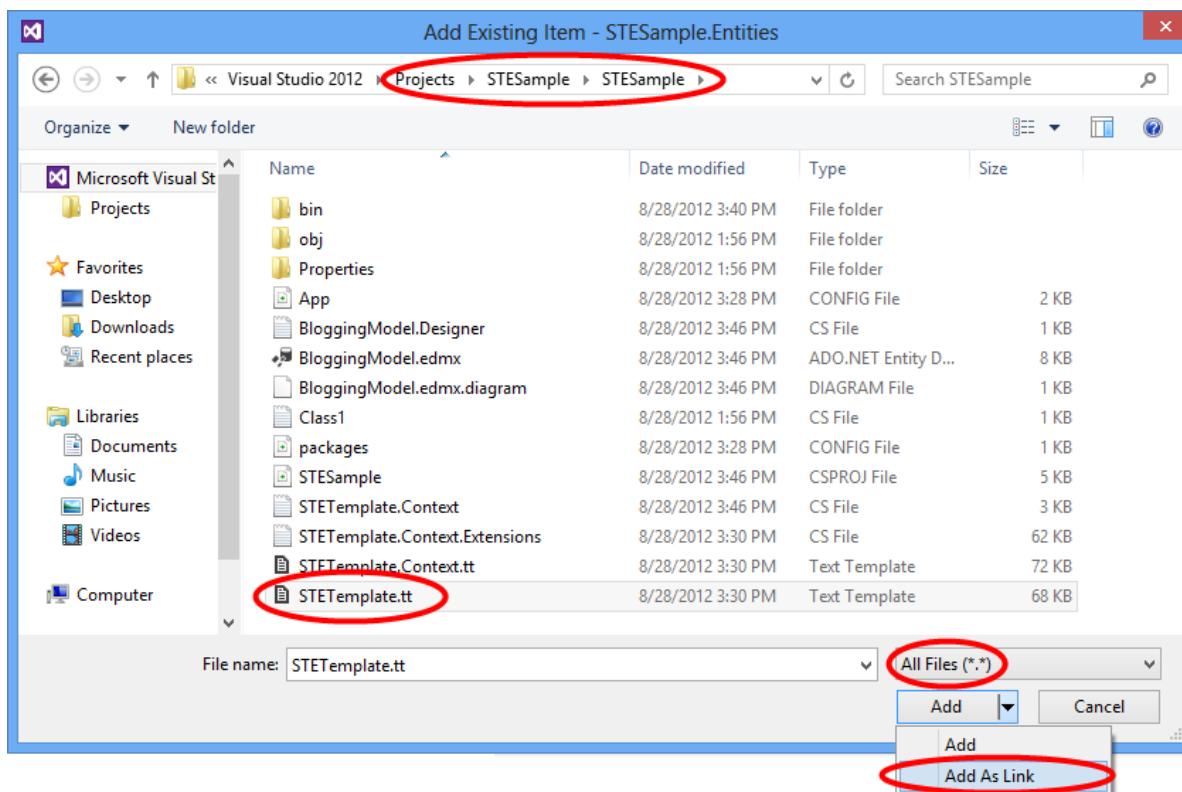
Первым шагом является прекращается создание классов сущностей в существующий проект:

- Щелкните правой кнопкой мыши **STETemplate.tt** в **обозревателе решений** и выберите **свойства**
- В **свойства** очистить окно **TextTemplatingFileGenerator** из **CustomTool** свойство
- Разверните **STETemplate.tt** в **обозревателе решений** и удалить все файлы, вложенным в него

Затем мы собираемся добавить новый проект и создания классов сущностей в нем

- **Файл —> Add -> проекта...**
- Выберите **Visual C#** в области слева и затем **библиотеки классов**
- Введите **STESample.Entities** имя и нажмите кнопку **OK**
- **Проект -> добавить существующий элемент...**
- Перейдите к **STESample** папки проекта
- Выберите для просмотра **все файлы (*.*)**
- Выберите **STETemplate.tt** файла

- Щелкните стрелку раскрывающегося списка рядом с **добавить** и выберите **добавить как ссылку**



Кроме того, мы собираемся убедиться, что классы сущностей создаются в том же пространстве имен, как контекст. Это просто уменьшает число с помощью инструкций, которые необходимо добавить на протяжении нашего приложения.

- Щелкните правой кнопкой мыши на связанном **STETemplate.tt** в **обозревателе решений** и выберите **свойства**
- В **свойства** задайте **пространство имен CustomTool** для **STESample**

Код, созданный с помощью шаблона ТАВИТЬ будет нужна ссылка на **System.Runtime.Serialization** для компиляции. Эта библиотека нужна для WCF **DataContract** и **DataMember** атрибутов, используемых для сериализуемых типов сущности.

- Щелкните правой кнопкой мыши **STESample.Entities** в проекте **обозревателе решений** и выберите **добавить ссылку...**
 - В Visual Studio 2012, установите флажок рядом с полем **System.Runtime.Serialization** и нажмите кнопку **OK**
 - В Visual Studio 2010 — выберите **System.Runtime.Serialization** и нажмите кнопку **OK**

Наконец проект с нашим контекстом, в нем будет нужна ссылка на типы сущностей.

- Щелкните правой кнопкой мыши **STESample** в проекте **обозревателе решений** и выберите **добавить ссылку...**
 - В Visual Studio 2012 — выберите **решение** в левой области, установите флажок рядом с полем **STESample.Entities** и нажмите кнопку **OK**
 - В Visual Studio 2010 — выберите **проекты** выберите **STESample.Entities** и нажмите кнопку **OK**

NOTE

Другой вариант перемещения типы сущностей в отдельном проекте — перемещение файла шаблона, а не связывая его из расположения по умолчанию. После этого необходимо обновить **inputFile** переменных в шаблоне, чтобы указать относительный путь в EDMX-файл (в этом примере, которая была бы .. \BloggingModel.edmx).

Создание службы WCF

Пришло время добавить службу WCF для предоставления наши данные, мы начнем с создания проекта.

- **Файл —> Add -> проекта...**
- Выберите **Visual C#** в области слева и затем **приложение службы WCF**
- Введите **STESample.Service** имя и нажмите кнопку **OK**
- Добавьте ссылку на **System.Data.Entity** сборки
- Добавьте ссылку на **STESample** и **STESample.Entities** проектов

Нам нужно скопировать строку подключения EF для этого проекта, таким образом, найденных во время выполнения.

- Откройте **App.Config** файл ** STESample ** проект и скопировать **connectionStrings** элемент
- Вставить **connectionStrings** элемент как дочерний элемент элемента **конфигурации** элемент **Web.Config** файла **STESample.Service** проекта

Пришло время для реализации фактической службы.

- Откройте **IService1.cs** и замените его содержимое следующим кодом

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

namespace STESample.Service
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        List<Blog> GetBlogs();

        [OperationContract]
        void UpdateBlog(Blog blog);
    }
}
```

- Откройте **Service1.svc** и замените его содержимое следующим кодом

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;

namespace STESample.Service
{
    public class Service1 : IService1
    {
        /// <summary>
        /// Gets all the Blogs and related Posts.
        /// </summary>
        public List<Blog> GetBlogs()
        {
            using (BloggingContext context = new BloggingContext())
            {
                return context.Blogs.Include("Posts").ToList();
            }
        }

        /// <summary>
        /// Updates Blog and its related Posts.
        /// </summary>
        public void UpdateBlog(Blog blog)
        {
            using (BloggingContext context = new BloggingContext())
            {
                try
                {
                    // TODO: Perform validation on the updated order before applying the changes.

                    // The ApplyChanges method examines the change tracking information
                    // contained in the graph of self-tracking entities to infer the set of operations
                    // that need to be performed to reflect the changes in the database.
                    context.Blogs.ApplyChanges(blog);
                    context.SaveChanges();

                }
                catch (UpdateException)
                {
                    // To avoid propagating exception messages that contain sensitive data to the client
                    tier
                    // calls to ApplyChanges and SaveChanges should be wrapped in exception handling code.
                    throw new InvalidOperationException("Failed to update. Try your request again.");
                }
            }
        }
    }
}

```

Использовать службу в консольном приложении

Давайте создадим консольное приложение, которое использует нашей службы.

- **Файл —> Новинка —> проекта...**
- Выберите **Visual C#** в области слева и затем **консольного приложения**
- Введите **STESample.ConsoleTest** имя и нажмите кнопку **OK**
- Добавьте ссылку на **STESample.Entities** проекта

Нам нужно ссылку на службу в нашу службу WCF

- Щелкните правой кнопкой мыши **STESample.ConsoleTest** в проекте **обозревателе решений** и выберите **добавить ссылку на службу...**

- Нажмите кнопку **обнаружение**
- Введите **BloggingService** пространства имен и нажмите кнопку **OK**

Теперь можно написать код для использования службы.

- Откройте **Program.cs** и замените его содержимое следующим кодом.

```

using STESample.ConsoleTest.BloggingService;
using System;
using System.Linq;

namespace STESample.ConsoleTest
{
    class Program
    {
        static void Main(string[] args)
        {
            // Print out the data before we change anything
            Console.WriteLine("Initial Data:");
            DisplayBlogsAndPosts();

            // Add a new Blog and some Posts
            AddBlogAndPost();
            Console.WriteLine("After Adding:");
            DisplayBlogsAndPosts();

            // Modify the Blog and one of its Posts
            UpdateBlogAndPost();
            Console.WriteLine("After Update:");
            DisplayBlogsAndPosts();

            // Delete the Blog and its Posts
            DeleteBlogAndPost();
            Console.WriteLine("After Delete:");
            DisplayBlogsAndPosts();

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        static void DisplayBlogsAndPosts()
        {
            using (var service = new Service1Client())
            {
                // Get all Blogs (and Posts) from the service
                // and print them to the console
                var blogs = service.GetBlogs();
                foreach (var blog in blogs)
                {
                    Console.WriteLine(blog.Name);
                    foreach (var post in blog.Posts)
                    {
                        Console.WriteLine(" - {0}", post.Title);
                    }
                }
            }

            Console.WriteLine();
            Console.WriteLine();
        }

        static void AddBlogAndPost()
        {
            using (var service = new Service1Client())
            {
                // Create a new Blog with a couple of Posts
                var newBlog = new Blog

```

```

    {
        Name = "The New Blog",
        Posts =
        {
            new Post { Title = "Welcome to the new blog"},
            new Post { Title = "What's new on the new blog"}
        }
    };

    // Save the changes using the service
    service.UpdateBlog(newBlog);
}

static void UpdateBlogAndPost()
{
    using (var service = new Service1Client())
    {
        // Get all the Blogs
        var blogs = service.GetBlogs();

        // Use LINQ to Objects to find The New Blog
        var blog = blogs.First(b => b.Name == "The New Blog");

        // Update the Blogs name
        blog.Name = "The Not-So-New Blog";

        // Update one of the related posts
        blog.Posts.First().Content = "Some interesting content...";

        // Save the changes using the service
        service.UpdateBlog(blog);
    }
}

static void DeleteBlogAndPost()
{
    using (var service = new Service1Client())
    {
        // Get all the Blogs
        var blogs = service.GetBlogs();

        // Use LINQ to Objects to find The Not-So-New Blog
        var blog = blogs.First(b => b.Name == "The Not-So-New Blog");

        // Mark all related Posts for deletion
        // We need to call ToList because each Post will be removed from the
        // Posts collection when we call MarkAsDeleted
        foreach (var post in blog.Posts.ToList())
        {
            post.MarkAsDeleted();
        }

        // Mark the Blog for deletion
        blog.MarkAsDeleted();

        // Save the changes using the service
        service.UpdateBlog(blog);
    }
}
}

```

Теперь вы можете запустить приложение и увидеть, как оно работает:

- Щелкните правой кнопкой мыши **STESample.ConsoleTest** в проекте **обозревателе решений** и выберите «**Отладка** —> **запустить новый экземпляр**

При запуске приложения вы увидите следующие выходные данные.

```
Initial Data:  
ADO.NET Blog  
- Intro to EF  
- What is New  
  
After Adding:  
ADO.NET Blog  
- Intro to EF  
- What is New  
The New Blog  
- Welcome to the new blog  
- What's new on the new blog  
  
After Update:  
ADO.NET Blog  
- Intro to EF  
- What is New  
The Not-So-New Blog  
- Welcome to the new blog  
- What's new on the new blog  
  
After Delete:  
ADO.NET Blog  
- Intro to EF  
- What is New  
  
Press any key to exit...
```

Использование службы в приложении WPF

Давайте создадим приложение WPF, использующего нашей службы.

- **Файл —> Новинка —> проекта...**
- Выберите **Visual C#** в области слева и затем **приложения WPF**
- Введите **STESample.WPFTest** имя и нажмите кнопку **OK**
- Добавьте ссылку на **STESample.Entities** проекта

Нам нужно ссылку на службу в нашу службу WCF

- Щелкните правой кнопкой мыши **STESample.WPFTest** в проекте **обозревателе решений** и выберите **добавить ссылку на службу...**
- Нажмите кнопку **обнаружение**
- Введите **BloggingService** пространства имен и нажмите кнопку **OK**

Теперь можно написать код для использования службы.

- Откройте **MainWindow.xaml** и замените его содержимое следующим кодом.

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:STESample="clr-namespace:STESample;assembly=STESample.Entities"
    mc:Ignorable="d" x:Class="STESample.WPFTest.MainWindow"
    Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded">

    <Window.Resources>
        <CollectionViewSource
            x:Key="blogViewSource"
            d:DesignSource="{d:DesignInstance {x:Type STESample:Blog}, CreateList=True}"/>
        <CollectionViewSource
            x:Key="blogPostsViewSource"
            Source="{Binding Posts, Source={StaticResource blogViewSource}}"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource blogViewSource}">
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding}" Margin="10,10,10,179">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding BlogId}" Header="Id" Width="Auto" IsReadOnly="True"
/>
                <DataGridTextColumn Binding="{Binding Name}" Header="Name" Width="Auto"/>
                <DataGridTextColumn Binding="{Binding Url}" Header="Url" Width="Auto"/>
            </DataGrid.Columns>
        </DataGrid>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding Source={StaticResource blogPostsViewSource}}"
Margin="10,145,10,38">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding PostId}" Header="Id" Width="Auto"
IsReadOnly="True"/>
                <DataGridTextColumn Binding="{Binding Title}" Header="Title" Width="Auto"/>
                <DataGridTextColumn Binding="{Binding Content}" Header="Content" Width="Auto"/>
            </DataGrid.Columns>
        </DataGrid>
        <Button Width="68" Height="23" HorizontalAlignment="Right" VerticalAlignment="Bottom"
Margin="0,0,10,10" Click="buttonSave_Click">Save</Button>
    </Grid>
</Window>

```

- Откройте код программной части для MainWindow (**MainWindow.xaml.cs**) и замените его содержимое следующим кодом

```

using STESample.WPFTest.BloggingService;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Data;

namespace STESample.WPFTest
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            using (var service = new Service1Client())
            {
                // Find the view source for Blogs and populate it with all Blogs (and related Posts)
                // from the Service. The default editing functionality of WPF will allow the objects
                // to be manipulated on the screen.
                var blogsViewSource = (CollectionViewSource)this.FindResource("blogViewSource");
                blogsViewSource.Source = service.GetBlogs().ToList();
            }
        }

        private void buttonSave_Click(object sender, RoutedEventArgs e)
        {
            using (var service = new Service1Client())
            {
                // Get the blogs that are bound to the screen
                var blogsViewSource = (CollectionViewSource)this.FindResource("blogViewSource");
                var blogs = (List<Blog>)blogsViewSource.Source;

                // Save all Blogs and related Posts
                foreach (var blog in blogs)
                {
                    service.UpdateBlog(blog);
                }

                // Re-query for data to get database-generated keys etc.
                blogsViewSource.Source = service.GetBlogs().ToList();
            }
        }
    }
}

```

Теперь вы можете запустить приложение и увидеть, как оно работает:

- Щелкните правой кнопкой мыши **STESample.WPFTest** в проекте **обозревателе решений** и выберите «**Отладка**» —> **запустить новый экземпляр**
- Можно обрабатывать данные, используя экрана и сохраните его с помощью службы, используя **Сохранить** кнопки

Blog Id	Name	Url
1	ADO.NET Blog	blogs.msdn.com/adonet
3	My Blog	

Post Id	Title	Content
5	Welcome	This is my first post...

Ведение журнала и перехвата операций базы данных

13.09.2018 • 22 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Начиная с Entity Framework 6, каждый раз, когда Entity Framework отправляет команду базе данных этой команды могут быть перехвачены кодом приложения. Это чаще всего используется для ведения журнала SQL, но можно также использовать для изменения или прервите команду.

В частности платформа EF предусматривает:

- Свойство Log для контекста, аналогичную DataContext.Log в LINQ to SQL
- Механизм для настройки содержимого и форматирования выходных данных, отправляются в журнал
- Низкоуровневые строительные блоки для перехвата, обеспечивая гибкость управления /

Свойство контекста журнала

Свойство DbContext.Database.Log можно задать делегат для любого метода, который принимает строку. Чаще всего он используется с любой TextWriter при установке для него метод «Запись», TextWriter. Все SQL, созданных в текущем контексте будет регистрироваться для этого средства записи. Например следующий код, регистрируются SQL в консоли:

```
using (var context = new BlogContext())
{
    context.Database.Log = Console.WriteLine;

    // Your code here...
}
```

Обратите внимание, что этот контекст. Database.Log имеет значение "Console.WriteLine". Это все, что требуется для входа SQL на консоль.

Давайте добавим немного простого кода запроса и вставки или обновления, чтобы мы могли видеть некоторые выходные данные:

```

using (var context = new BlogContext())
{
    context.Database.Log = Console.WriteLine;

    var blog = context.Blogs.First(b => b.Title == "One Unicorn");

    blog.Posts.First().Title = "Green Eggs and Ham";

    blog.Posts.Add(new Post { Title = "I do not like them!" });

    context.SaveChangesAsync().Wait();
}

```

Это создаст следующие выходные данные:

```

SELECT TOP (1)
    [Extent1].[Id] AS [Id],
    [Extent1].[Title] AS [Title]
FROM [dbo].[Blogs] AS [Extent1]
WHERE (N'One Unicorn' = [Extent1].[Title]) AND ([Extent1].[Title] IS NOT NULL)
-- Executing at 10/8/2013 10:55:41 AM -07:00
-- Completed in 4 ms with result: SqlDataReader

SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Title] AS [Title],
    [Extent1].[BlogId] AS [BlogId]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[BlogId] = @EntityKeyValue1
-- EntityKeyValue1: '1' (Type = Int32)
-- Executing at 10/8/2013 10:55:41 AM -07:00
-- Completed in 2 ms with result: SqlDataReader

UPDATE [dbo].[Posts]
SET [Title] = @0
WHERE ([Id] = @1)
-- @0: 'Green Eggs and Ham' (Type = String, Size = -1)
-- @1: '1' (Type = Int32)
-- Executing asynchronously at 10/8/2013 10:55:41 AM -07:00
-- Completed in 12 ms with result: 1

INSERT [dbo].[Posts]([Title], [BlogId])
VALUES (@0, @1)
SELECT [Id]
FROM [dbo].[Posts]
WHERE @@ROWCOUNT > 0 AND [Id] = scope_identity()
-- @0: 'I do not like them!' (Type = String, Size = -1)
-- @1: '1' (Type = Int32)
-- Executing asynchronously at 10/8/2013 10:55:41 AM -07:00
-- Completed in 2 ms with result: SqlDataReader

```

(Обратите внимание, что это выходные данные, при условии, что уже произошло инициализацию базы данных. Если инициализация базы данных не уже была применена, то будет гораздо больше результат, показаны всю работу миграции не на самом деле поиск или создать новую базу данных.)

Новые журналы?

Когда все указанные ниже имеет значение свойства Log будет записан в журнал:

- SQL для всех различных типов команд. Пример:
 - Запросы, включая обычные запросы LINQ, запросов eSQL и необработанные запросы из методов, таких как SQL-запрос

- Операции вставки, обновления и удаления, созданным как часть SaveChanges
- Отношение, загрузке запросов, которые созданы при отложенной загрузки
- Параметры
- Ли команда выполняется асинхронно
- Меткой начала выполнения команды
- Независимо от того, имеется ли команда выполнена успешно, сбой, создается исключение, или для асинхронных, была отменена
- Некоторые указания результирующего значения
- Приблизительное количество времени, которое потребовалось для выполнения команды. Обратите внимание, что это время с отправкой команды для получения объекта результат обратно. Не включает время считать результаты.

Глядя в примере выходных данных выше, каждый из четырех команд в журнал:

- Запрос, полученный из вызова к контексту. Blogs.First
 - Обратите внимание, что метод ToString получения SQL не были бы выполнены для этого запроса, так как "First" не поддерживает IQueryable, для которого может быть вызван ToString
- Запрос, полученный в результате отложенной загрузки блога. Сообщения
 - Обратите внимание, что сведения о параметрах для значения ключа для какой отложенная загрузка происходит
 - Регистрируются только свойства параметра, которые присваиваются значения по умолчанию. Например свойство размера отображается только если оно не равно нулю.
- Две команды, полученный в результате SaveChangesAsync; для обновления изменить заголовок post, другой для инструкции insert для добавления новой записи
 - Обратите внимание, что сведения о параметрах для свойства внешнего ключа и заголовок
 - Обратите внимание на то, что эти команды выполняются асинхронно

Ведение журнала для различных мест

Как показано выше для ведения журнала консоли — очень легко. Это средство легко входить в памяти, файл, и т.д., с помощью различных типов из [TextWriter](#).

Если вы знакомы с помощью LINQ to SQL, следует отметить, что в LINQ to SQL, имеет значение свойства Log на фактический TextWriter объект (например, Console.Out) во время в EF журнала задано значение метод, который принимает строку (например Console.WriteLine или Console.Out.WriteLine). Причина этого — отделить EF из TextWriter, принимая любому делегату, может выступать в качестве приемника для строк. Например, представьте, что у вас уже есть некоторые платформы ведения журналов, и определяет метод ведения журнала следующим образом:

```
public class MyLogger
{
    public void Log(string component, string message)
    {
        Console.WriteLine("Component: {0} Message: {1} ", component, message);
    }
}
```

Это может привязать к свойству EF журнала следующим образом:

```
var logger = new MyLogger();
context.Database.Log = s => logger.Log("EFApp", s);
```

Записи результатов

Средство ведения журнала по умолчанию текст команды (SQL), параметры и журналов в строке «Executing» с меткой времени перед отправкой команды в базу данных. «Завершено» строки, содержащей затраченное время — журнал выполнения следующие команды.

Обратите внимание на то, что для асинхронных команд «завершено» строки не регистрируются до асинхронной задачи фактически завершении, сбое или отмене.

Строка «завершено» содержит различные сведения в зависимости от типа команды и ли выполнение прошло успешно.

Успешное выполнение

Для команд, которые успешно завершили выходные данные — «выполнено в x ms с результатом: "следуют некоторые указания результаты. Для команд, которые возвращают результат модуля чтения данных — это значение, указывающее тип [DbDataReader](#) возвращается. Для команд, которые возвращают значение типа integer, таких как обновление команде, показанной выше показан результат является целого числа.

Сбой при выполнении

Для команд, которые не удается, создается исключение выходные данные содержат сообщение из исключения. Например с помощью SQL-запрос для запроса к таблице, которая существует будет результат в журнале выходные данные примерно следующее:

```
SELECT * from ThisTableIsMissing
-- Executing at 5/13/2013 10:19:05 AM
-- Failed in 1 ms with error: Invalid object name 'ThisTableIsMissing'.
```

Выполнение отменено

Асинхронные команды, где задача отменяется, результатом может стать сбой с исключением, так как это и базового поставщика ADO.NET часто делает при попытке отменить. Если этого не происходит, если задача отменена аккуратно выходных данных будет выглядеть примерно следующим образом:

```
update Blogs set Title = 'No' where Id = -1
-- Executing asynchronously at 5/13/2013 10:21:10 AM
-- Canceled in 1 ms
```

Изменение содержимого журнала и форматирование

На самом деле Database.Log, делает свойство использовать DatabaseLogFormatter объекта. Этот объект фактически привязывает делегат, который принимает строки и DbContext реализацию IDbCommandInterceptor (см. ниже). Это означает, что методы DatabaseLogFormatter вызываются до и после выполнения команды EF. Эти методы DatabaseLogFormatter сбора и отформатировать выходные данные журнала и отправить его на делегат.

Настройка DatabaseLogFormatter

Изменение регистрируемых и форматирования можно сделать, создав новый класс, производный от DatabaseLogFormatter и переопределяет соответствующие методы. Ниже приведены наиболее распространенных методов для переопределения.

- LogCommand — переопределить изменение процессом регистрации команд до их выполнения. По умолчанию LogCommand вызывает LogParameter для каждого параметра; Вы можете сделать то же самое в переопределении или обработки параметров вместо этого по-разному.
- LogResult — переопределить изменение регистрируется как результат выполнения команды.

- LogParameter — переопределить это, чтобы изменить форматирование и содержимое параметра ведения журнала.

Например предположим, что мы хотели бы журнала лишь одна строка, перед отправкой каждой команды в базу данных. Это можно сделать с помощью переопределения:

- Переопределить LogCommand для форматирования и записи одной строки SQL
- Переопределите LogResult, чтобы не выполнять никаких действий.

Код будет выглядеть примерно так:

```
public class OneLineFormatter : DatabaseLogFormatter
{
    public OneLineFormatter(DbContext context, Action<string> writeAction)
        : base(context, writeAction)
    {
    }

    public override void LogCommand<TResult>(
        DbCommand command, DbCommandInterceptionContext<TResult> interceptionContext)
    {
        Write(string.Format(
            "Context '{0}' is executing command '{1}'{2}",
            Context.GetType().Name,
            command.CommandText.Replace(Environment.NewLine, ""),
            Environment.NewLine));
    }

    public override void LogResult<TResult>(
        DbCommand command, DbCommandInterceptionContext<TResult> interceptionContext)
    {
    }
}
```

В журнал выходные простым вызовом метод Write, которая будет отправлять выходные данные к делегату настроенные записи.

(Обратите внимание, что этот код выполняет упрощенную удаления разрывы строк так же, как пример. Он скорее всего не будет работать для просмотра сложных SQL.)

Параметр DatabaseLogFormatter

Создав новый класс DatabaseLogFormatter он должен быть зарегистрирован с EF. Это делается с помощью конфигурации на основе кода. По сути это означает, создается новый класс, производный от DbConfiguration в ту же сборку класса DbContext и последующего вызова SetDatabaseLogFormatter в конструкторе этого нового класса. Пример:

```
public class MyDbConfiguration : DbConfiguration
{
    public MyDbConfiguration()
    {
        SetDatabaseLogFormatter(
            (context, writeAction) => new OneLineFormatter(context, writeAction));
    }
}
```

С помощью нового DatabaseLogFormatter

Теперь этот новый DatabaseLogFormatter будет использоваться каждый раз, когда Database.Log имеет значение. Таким образом выполнение кода из части 1 теперь приведет следующие выходные данные:

```
Context 'BlogContext' is executing command 'SELECT TOP (1) [Extent1].[Id] AS [Id], [Extent1].[Title] AS [Title]FROM [dbo].[Blogs] AS [Extent1]WHERE (N'One Unicorn' = [Extent1].[Title]) AND ([Extent1].[Title] IS NOT NULL)'
Context 'BlogContext' is executing command 'SELECT [Extent1].[Id] AS [Id], [Extent1].[Title] AS [Title], [Extent1].[BlogId] AS [BlogId]FROM [dbo].[Posts] AS [Extent1]WHERE [Extent1].[BlogId] = @EntityKeyValue1'
Context 'BlogContext' is executing command 'update [dbo].[Posts]set [Title] = @0where ([Id] = @1)'
Context 'BlogContext' is executing command 'insert [dbo].[Posts]([Title], [BlogId])values (@0, @1)select [Id]from [dbo].[Posts]where @@rowcount > 0 and [Id] = scope_identity()'
```

Стандартные блоки перехвата

Пока мы рассматривали как `DbContext.Database.Log` для ведения журнала SQL, созданного EF. Но этот код предоставляются фактически относительно тонкой через некоторые низкоуровневые строительные блоки для более общих перехвата.

Перехват интерфейсы

Перехват кода построена вокруг концепция перехвата интерфейсов. Этих интерфейсов наследуют от `IDbInterceptor` и определять методы, которые вызываются, когда EF выполняет определенное действие. Целью является один интерфейс по типам перехватываемым объектом. Например `IDbCommandInterceptor` интерфейс определяет методы, которые вызываются до EF вызывает метод `ExecuteNonQuery`, `ExecuteScalar`, `ExecuteReader` и связанные с ним методы. Аналогичным образом интерфейс определяет методы, вызываемые при завершении каждой из этих операций. Класс `DatabaseLogFormatter`, который мы рассмотрели выше реализует этот интерфейс, чтобы зарегистрировать в журнале команды.

Контекст перехвата

Просмотрев методы, определенные на любой из этих интерфейсов перехватчик его очевидным, что при каждом вызове является заданный объект типа `DbInterceptionContext` или какого-либо типа производным от этого например `DbCommandInterceptionContext<>`. Этот объект содержит контекстные сведения о действии, занимает EF. Например если это действие выполняется от имени `DbContext`, `DbContext` включается в `DbInterceptionContext`. Аналогичным образом для команд, которые выполняются асинхронно, `IsAsync` установлен флаг `DbCommandInterceptionContext`.

Обработка результатов

`DbCommandInterceptionContext<>` класс содержит свойства, результат, `OriginalResult`, исключения и первоначальное исключение. Эти свойства устанавливаются в `null/zero` для перехвата методов, которые вызываются до выполнения операции, то есть для... Выполнение методов. Если операция выполняется и завершается успешно, результат и `OriginalResult` задаются на результат операции. Затем эти значения можно наблюдать в перехвата методы, которые вызываются после выполнения операции, то есть на... Выполненных методов. Аналогично Если операция создает исключение, то свойства исключения и первоначальное исключение будет происходить.

Подавление выполнения

Если перехватчик свойства результата до выполнения команды (в одном из... Выполнение методов) затем EF не будет пытаться фактически выполнить команду, но вместо этого просто будет использовать результатирующий набор. Другими словами перехватчик можно подавить выполнение команды, но имеют EF по-прежнему так, как если была выполнена команда.

Команды пакетной обработки, традиционно было сделано с помощью упаковки поставщика — пример того, как это может использовать. Перехватчик будет храниться команды для последующего выполнения в пакетном режиме, но будет «» выступить в роли EF, что команда выполнена в обычном режиме. Обратите внимание, что больше, чем это требуется для реализации пакетной обработки, но ниже приведен пример того, как может использовать изменения перехвата результатов.

Выполнение можно также подавить, задав свойство `Exception` в одном из... Выполнение методов. В этом случае EF продолжить, как будто выполнение операции прервано возникновения данного исключения. Это

Конечно, приложение дает сбой, но также может временного исключения или других исключений, который обрабатывается EF. Например это может использоваться в тестовой среде для тестирования поведения приложения, когда происходит сбой выполнения команды.

Изменения результатов после выполнения

Если перехватчик свойства результат после выполнения команды (в одном из... Выполняется методы), а затем EF будет использовать измененные результат вместо результата, который фактически был возвращен из операции. Аналогично Если перехватчик задает свойство исключения после выполнения команды, затем EF будет исключение набор так, как если бы операция исключение.

Перехватчик может также присвоено свойство Exception значение null, указывающее, что исключение не должно вызываться. Это может быть полезно, если не удалось выполнить операцию, но перехватчик намерена EF, чтобы продолжить, как если бы операция выполнена успешно. Это обычно также включает в себя установку результат, чтобы EF другому значению результат для работы с текущего сеанса.

OriginalResult и первоначальное исключение

После выполнения операции EF метод установит результат и OriginalResult свойства, если выполнение не завершилось сбоем или свойства исключения и первоначальное исключение если выполнение завершилось сбоем с исключением.

Свойства OriginalResult и первоначальное исключение доступны только для чтения и только задаются EF после фактического выполнения операции. Эти свойства задаются не перехватчики. Это означает, что любой перехватчик может различать исключения или результат, который был задан некоторые перехватчик, в отличие от реальных исключение или результат, возникшей при выполнении операции.

Регистрация перехватчики

После создания класса, который реализует один или несколько интерфейсов перехват его можно зарегистрировать с помощью класса DbInterception EF. Пример:

```
DbInterception.Add(new NLogCommandInterceptor());
```

Перехватчики также может быть зарегистрирован на уровне домена приложения с помощью механизма DbConfiguration конфигурация на основе кода.

Пример: Ведение журнала для NLog

Давайте рассмотрим все это вместе в пример, при помощи IDbCommandInterceptor и [NLog](#) для:

- Журнал предупреждение для любой команды, который выполняется не в асинхронном режиме
- Регистрирует ошибку для любой команды, который создает исключение при выполнении

Ниже показан класс, который выполняет ведение журнала, который должен быть зарегистрирован как показано выше:

```

public class NLogCommandInterceptor : IDbCommandInterceptor
{
    private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

    public void NonQueryExecuting(
        SqlCommand command, SqlCommandInterceptionContext<int> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void NonQueryExecuted(
        SqlCommand command, SqlCommandInterceptionContext<int> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ReaderExecuting(
        SqlCommand command, SqlCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ReaderExecuted(
        SqlCommand command, SqlCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ScalarExecuting(
        SqlCommand command, SqlCommandInterceptionContext<object> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ScalarExecuted(
        SqlCommand command, SqlCommandInterceptionContext<object> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    private void LogIfNonAsync<TResult>(
        SqlCommand command, SqlCommandInterceptionContext<TResult> interceptionContext)
    {
        if (!interceptionContext.IsAsync)
        {
            Logger.Warn("Non-async command used: {0}", command.CommandText);
        }
    }

    private void LogIfError<TResult>(
        SqlCommand command, SqlCommandInterceptionContext<TResult> interceptionContext)
    {
        if (interceptionContext.Exception != null)
        {
            Logger.Error("Command {0} failed with exception {1}",
                command.CommandText, interceptionContext.Exception);
        }
    }
}

```

Обратите внимание на то, как этот код использует контекст перехвата для обнаружения при выполнении команды не асинхронно и обнаружения, когда произошла ошибка при выполнении команды.

Рекомендации по ускорению EF 4, 5 и 6

01.10.2018 • 121 minutes to read • [Edit Online](#)

Дэвид Obando, Эрик Dettinger и другими

Дата публикации: Апрель 2012 г.

Последнее обновление: Май 2014 г.

1. Вступление

Объектно-реляционное сопоставление платформ являются удобным способом для обеспечения абстрактного для доступа к данным в объектно ориентированного приложения. Для приложений .NET, что платформа Entity Framework является объектно-реляционное отображение Рекомендованное корпорацией Microsoft. С любой абстракции, производительность может стать проблемой.

Этот технический документ был написан для отображения вопросы производительности при разработке приложений с помощью Entity Framework, чтобы предоставить разработчикам представление о внутренних алгоритмов Entity Framework, которые могут повлиять на производительность и выполнять советы для исследования и Повышение производительности в своих приложениях с помощью Entity Framework. Хороший Бернса на производительность уже доступны в Интернете, и мы постарались также указывает на эти ресурсы, где это возможно.

Производительность — это сложная тема. Этот официальный документ предназначен в качестве ресурса для внесения производительности связанные решения для приложений, использующих Entity Framework. Мы добавили некоторые показатели теста для демонстрации производительности, но эти показатели не предназначены как абсолютный показатели производительности, которое появится в вашем приложении.

С практической точки зрения в этом документе предполагается, что Entity Framework 4 выполняется в .NET 4.0 и Entity Framework 5 и 6 выполняются в .NET 4.5. Многие из улучшений производительности для Entity Framework 5 находятся в пределах основные компоненты, входящие в состав .NET 4.5.

Entity Framework 6 является готовую выпуска аппаратного контроллера управления и не зависит от компонентов Entity Framework, поставляемых вместе с .NET. Entity Framework 6 работать на .NET 4.0 и .NET 4.5, а можно позволяют оптимизировать производительность больших тем, кто еще не обновлен с .NET 4.0, но хотите последние Entity Framework в своих приложениях. При этом документе упоминается Entity Framework 6, он ссылается на последнюю версию, доступную во время написания этой статьи: версия 6.1.0.

2. "Холодных" vs. Выполнение запроса «горячего» резервирования

Первый раз, когда любой запрос выполняется к данной модели, Entity Framework выполняет массу работы за кулисами для загрузки и проверки модели. Мы часто ссылаются на этот первый запрос как запрос на «холодный». Дальнейшие запросы к модели уже загруженный известны как «горячего» резервирования запросов и выполняется гораздо быстрее.

Давайте высокоуровневое представление расходуется время при выполнении запроса с помощью Entity Framework и см. в разделе, где постоянно улучшаем вещи в Entity Framework 6.

Выполнение первого запроса — "холодных" запрос

КОД ЗАПИСЫВАЕТ ПОЛЬЗОВАТЕЛЯ	ДЕЙСТВИЕ	EF4 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF5 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF6 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ
<pre>using(var db = new MyContext()) {</pre>	Создание контекста	Средняя	Средняя	Low
<pre>var q1 = from c in db.Customers where c.Id == id1 select c;</pre>	Создание выражения запроса	Low	Low	Low
<pre>var c1 = q1.First();</pre>	Выполнение запроса LINQ	<ul style="list-style-type: none"> -Загрузка Metadata: высокий уровень, но кэшированные — Просмотр поколения: потенциально очень высокой, но кэшированные -Параметр оценки: средний -Query перевода: средний — Поколение материализатора: среднего размера, но кэшированные -Database выполнения запроса: потенциально высокой + Connection.Open + Command.ExecuteReader + DataReader.Read <p>Объект материализации: средний</p> <p>-Поиск identity: средний</p>	<ul style="list-style-type: none"> -Загрузка Metadata: высокий уровень, но кэшированные — Просмотр поколения: потенциально очень высокой, но кэшированные -Параметр оценки: низкий -Query перевода: среднего размера, но кэшированные — Поколение материализатора: среднего размера, но кэшированные -Database выполнения запроса: потенциально высокой (запросы, в некоторых ситуациях лучше) + Connection.Open + Command.ExecuteReader + DataReader.Read <p>Объект материализации: средний</p> <p>-Поиск identity: средний</p>	<ul style="list-style-type: none"> -Загрузка Metadata: высокий уровень, но кэшированные — Просмотр поколения: среднего размера, но кэшированные -Параметр оценки: низкий -Query перевода: среднего размера, но кэшированные — Поколение материализатора: среднего размера, но кэшированные -Database выполнения запроса: потенциально высокой (запросы, в некоторых ситуациях лучше) + Connection.Open + Command.ExecuteReader + DataReader.Read <p>Объект материализации: средний (быстрее, чем EF5)</p> <p>-Поиск identity: средний</p>
}	Connection.Close	Low	Low	Low

Выполнение запроса, второй — «теплого» запроса

КОД ЗАПИСЫВАЕТ ПОЛЬЗОВАТЕЛЯ	ДЕЙСТВИЕ	EF4 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF5 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF6 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ
<pre>using(var db = new MyContext()) {</pre>	Создание контекста	Средняя	Средняя	Low

КОД ЗАПИСЫВАЕТ ПОЛЬЗОВАТЕЛЯ	ДЕЙСТВИЕ	EF4 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF5 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ	EF6 ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ
<pre>var q1 = from c in db.Customers where c.Id == id1 select c;</pre>	Создание выражения запроса	Low	Low	Low
<pre>var c1 = q1.First();</pre>	Выполнение запроса LINQ	<ul style="list-style-type: none"> -Metadata Загрузка подстановки: высокого уровня, но кэшированных низкий — Просмотр поколения подстановки: потенциально очень высокой, но кэшированных низкий -Параметр оценки: средний -Query перевода подстановки: средний -Материализатора поколения подстановки: среднего размера, но кэшированных низкий -Database выполнения запроса: потенциально высокой + Connection.Open + Command.ExecuteReader + DataReader.Read Объект материализации: средний -Поиск identity: средний 	<ul style="list-style-type: none"> -Metadata Загрузка подстановки: высокого уровня, но кэшированных низкий — Просмотр поколения подстановки: потенциально очень высокой, но кэшированных низкий -Параметр оценки: низкий -Query перевода подстановки: среднего размера, но кэшированных низкий -Материализатора поколения подстановки: среднего размера, но кэшированных низкий -Database выполнения запроса: потенциально высокой + Connection.Open + Command.ExecuteReader + DataReader.Read Объект материализации: средний 	<ul style="list-style-type: none"> -Metadata Загрузка подстановки: высокого уровня, но кэшированных низкий — Просмотр поколения подстановки: среднего размера, но кэшированных низкий -Параметр оценки: низкий -Query перевода подстановки: среднего размера, но кэшированных низкий -Материализатора поколения подстановки: среднего размера, но кэшированных низкий -Database выполнения запроса: потенциально высокой (запросы, в некоторых ситуациях лучше) + Connection.Open + Command.ExecuteReader + DataReader.Read Объект материализации: средний (быстрее, чем EF5) -Поиск identity: средний
}	Connection.Close	Low	Low	Low

Существует несколько способов, позволяющих сократить стоимость производительности запросов "холодных" и "горячего" резервирования, и мы рассмотрим в следующем разделе. В частности мы рассмотрим снижая затраты на загрузку в "холодных" запросы с помощью заранее созданные представления, которые должны помочь устранить трудности производительности, возникшие во время создания представления модели. Для "горячего" резервирования запросов мы обсудим кэширование планов запросов, запросы без отслеживания и параметры выполнения другой запрос.

2.1. что такое создание представлений

Чтобы понять, какое представление сформировано, нам необходимо сначала понять, каковы «Сопоставление представлений». Сопоставление представления — это исполняемый файл представления преобразований, указанных в сопоставлении для каждого набора сущностей и ассоциаций. На внутреннем уровне эти представления сопоставления принимают форму CQTs (деревья каноническую запроса). Существует два типа представлений сопоставления:

- Отправить запрос представлений: они представляют собой преобразования, необходимо вернуться в схеме базы данных в концептуальную модель.
- Обновление представлений: они представляют собой преобразований, необходимых для перехода из концептуальной модели со схемой базы данных.

Имейте в виду, что концептуальной модели может отличаться от схемы базы данных различными способами. Например один из одной таблицы может использоваться для хранения данных для двух разных типов сущностей. Наследование и без того нетривиальные сопоставления играют роль в сложности сопоставления представления.

Процесс вычисления этих представлений, на основе спецификации сопоставления так называемый средство создания представления. Создание представлений может либо выполняться динамически при загрузке модели или во время сборки с помощью «заранее сформированные представления»; Последнее, сериализуются в виде инструкций Entity SQL для C# или VB-файла.

При создании представления, они также проверяются. С точки зрения производительности большинство стоимость создания представления является фактически проверки представлений, что гарантирует смысла соединений между сущностями и иметь правильную кратность для всех поддерживаемых операций.

При выполнении запроса через набор сущностей, запрос объединяется с соответствующее представление запроса, а результат этой композиции выполняется через план компилятору создавать представление запроса, который может понять резервного хранилища. Для SQL Server конечным результатом компиляции будет инструкцию T-SQL SELECT. В первый раз выполняется обновление через набор сущностей, сходную процедуру для преобразования его в инструкции DML для целевой базы данных выполняется обновление представления.

2.2 факторы, влияющие на производительность создания представления

Производительность этапа формирования представления, не только, зависит от размер вашей модели, но и о том, как взаимосвязанных модели. Если две сущности подключены по цепочке наследования или ассоциации, они называются должен быть подключен. Аналогичным образом, если две таблицы связаны с помощью внешнего ключа, они подключены. Как увеличить количество подключенных сущностями и таблицами в схемах, средств создания представления платы увеличивается.

Алгоритм, который используется для создания и проверки представления экспоненциального в худшем случае, хотя мы используем некоторые оптимизации для улучшения качества. Ниже приведены крупнейших фактора, которые могут отрицательно повлиять на производительность.

- Размер модели, ссылка на число сущностей и объем ассоциации между этими сущностями.
- Модель сложности, в частности наследования, включающих большое число типов.
- С помощью независимых сопоставлений вместо внешнего ключа ассоциации.

Для моделей небольшие, простые стоимость может быть достаточно небольшим, чтобы не возиться с помощью заранее сформированные представления. Как увеличить размер модели и сложности, существует несколько вариантов, доступных для сокращения затрат на создание представлений и проверки.

Время загрузки 2.3 Pre-Generated представлениями для уменьшения модели

Подробные сведения о том, как использовать заранее созданные представления Entity Framework 6 [Pre-Generated сопоставление представлений](#)

2.3.1 заранее сформированные представления, с помощью Entity Framework Power Tools Community Edition

Можно использовать [Entity Framework 6 Power Tools Community Edition](#) для создания представлений моделей EDMX и Code First, щелкнув правой кнопкой мыши файл класса модели и выберите «Создание представления» с помощью меню Entity Framework. Entity Framework Power Tools Community Edition работают только с производным DbContext контекстов.

2.3.2 как использовать заранее созданные представления с помощью модели, созданные EDMGen

EDMGen — это служебная программа, которая поставляется с .NET и работает с Entity Framework 4 и 5, но не с Entity Framework 6. EDMGen позволяет создавать файл модели, на уровне объектов и представления из командной строки. Один из выходов появится файл представления на языке программирования, Visual Basic или C#. Это файл кода, содержащий фрагменты Entity SQL для каждого набора сущностей. Чтобы включить заранее созданные представления, просто включите файл в проекте.

Если вы вручную редактируете файлы схемы для модели, необходимо будет повторно создать файл представления. Это можно сделать, выполнив EDMGen с **/mode:ViewGeneration** флагом.

2.3.3 способы использования представления Pre-Generated с EDMX-файла

EDMGen также можно использовать для создания представлений для EDMX-файла — ранее указанной статьи MSDN описывается добавление события перед сборкой, чтобы сделать это — но ситуация осложняется и существуют случаи, где невозможно. Обычно проще использовать шаблон T4 для создания представления, когда модель находится в EDMX-файла.

Блог группы ADO.NET имеет пост, в котором описано, как использовать шаблон T4 для создания представлений (<http://blogs.msdn.com/b/adonet/archive/2008/06/20/how-to-use-a-t4-template-for-view-generation.aspx>). Эта запись включает в себя шаблон, который можно загрузить и добавлен в проект. Шаблон был написан для первой версии Entity Framework, поэтому они не обязательно работают с последними версиями платформы Entity Framework. Тем не менее можно загрузить набор быстрее получать новые шаблоны создания представления для Entity Framework 4 и 5 from коллекции Visual Studio:

- VB.NET: <<http://visualstudiogallery.msdn.microsoft.com/118b44f2-1b91-4de2-a584-7a680418941d>>
- C#: <<http://visualstudiogallery.msdn.microsoft.com/ae7730ce-ddab-470f-8456-1b313cd2c44d>>

Если вы используете Entity Framework 6 представления можно получить шаблоны создания T4 из коллекции Visual Studio в <<http://visualstudiogallery.msdn.microsoft.com/18a7db90-6705-4d19-9dd1-0a6c23d0751f>>.

2.4, снижая затраты средств создания представления

Используя заранее созданные представления перемещает затраты на создание представлений из модели загрузки (времени выполнения) время разработки. Хотя это улучшает производительность при запуске во время выполнения, будут по-прежнему возникать усилий, затрачиваемых на создание представлений при разработке. Существует несколько дополнительных приемов, которые могут помочь снизить затраты на создание представлений, как во время компиляции и выполнения.

2.4.1 с помощью внешнего ключа ассоциации, чтобы сократить затраты на создание представления

Мы видели несколько ситуаций, где переключение ассоциации в модели, из независимых сопоставлений для внешнего ключа ассоциации значительно улучшена время, затраченное на создание представлений.

Чтобы продемонстрировать это улучшение, мы создания двух версий модели Navision EDMGen.

Примечание: see appendix C for описание Navision модели. Модель Navision представляет интерес для этого упражнения из-за его очень большой объем сущностей и связей между ними.

Одна версия этой очень большой модели был создан с помощью ассоциации внешних ключей и другой был сформирован с использованием независимых сопоставлений. Мы затем истекло время ожидания как долго длилось ее для создания представлений для каждой модели. Тест Framework5 сущности используется метод GenerateViews() из класса EntityViewGenerator для создания представления, при работе с Entity Framework 6 тестовый метод GenerateViews() из класса StorageMappingItemCollection. Это из-за кода реструктуризации, произошедшего в базе кода Entity Framework 6.

С помощью Entity Framework 5, средств создания представления для модели с внешними ключами заняло

65 минут на машине лаборатории. Нет сколько времени потребовалось бы, чтобы создать представления для модели, используемые независимых сопоставлений. Мы оставили тест будет выполняться более одного месяца, прежде чем компьютер был перезагружен в нашей лаборатории, чтобы установить ежемесячных обновлений.

С помощью Entity Framework 6, средств создания представления для модели с внешними ключами занял 28 секунд в том же компьютере лаборатории. Создание представлений для модели, который использует независимых сопоставлений заняло 58 секунд. Повышение для Entity Framework 6 на его просмотр создания кода означает, что многие проекты не нужно заранее сформированные представления для получения более быстрое время запуска.

Очень важно замечание, что предварительное создание представлений в Entity Framework 4 и 5 можно сделать с помощью EDMGen или Entity Framework Power Tools. Для представления Entity Framework 6 создания можно с помощью Entity Framework Power Tools или программными средствами, как описано в разделе [Pre-Generated сопоставления представлений](#).

2.4.1.1 как использовать внешние ключи, вместо независимых сопоставлений

При использовании EDMGen или Entity Designer в Visual Studio, вы получаете внешних ключей по умолчанию, и только у одного флагка или командной строки флага для переключения между внешних ключей и IAs.

Если у вас есть большой модели Code First, с помощью независимых сопоставлений будет иметь такое же влияние на средств создания представления. Это влияние можно избежать, включая свойства внешнего ключа в классах для зависимых объектов, хотя некоторые разработчики рассматривает его как избавляют их объектная модель. Можно найти дополнительные сведения по этой теме в <http://blog.oneunicorn.com/2011/12/11/whats-the-deal-with-mapping-foreign-keys-using-the-entity-framework/>.

В ЯЗЫКЕ	СДЕЛАЙТЕ СЛЕДУЮЩЕЕ
Entity Designer	После добавления ассоциацию между двумя сущностями, убедитесь, что у вас есть справочного ограничения. Ссылочные ограничения сообщить Entity Framework для использования внешних ключей вместо независимых сопоставлений. Для получения дополнительных сведений посетите http://blogs.msdn.com/b/efdesign/archive/2009/03/16/foreign-keys-in-the-entity-framework.aspx .
EDMGen	При использовании EDMGen для формирования файлов из базы данных, внешние ключи, все доступные и добавляются в модель таким образом. Дополнительные сведения о различных параметрах, предоставляемыми EDMGen http://msdn.microsoft.com/library/bb387165.aspx .
Code First	См. в разделе «Соглашение о связи» первый соглашения о коде сведения о том, как включить свойства внешнего ключа для зависимых объектов при использовании Code First.

2.4.2 перемещение модели в отдельной сборке

Когда модели включается непосредственно в проект приложения и создать представления через событие перед построением или шаблон T4, средств создания представления и проверка будет выполняться каждый раз, когда проект будет перестроен, даже если модель не была изменена. Если вы перемещаете модель в отдельную сборку и сослаться на него из проекта приложения, внесения других изменений в приложение без необходимости перестройки проекта, содержащего модель.

Примечание: при перемещении модели в отдельные сборки не забудьте скопировать строки подключения

для модели в файле конфигурации клиентского проекта.

2.4.3 отключите проверку модели на основе edmx

Модели EDMX проверяются во время компиляции, даже если модель не содержит изменений. Если модель уже проверены, их можно отключить проверки во время компиляции, задав свойство «Проверка при сборке» значение `false` в окне «Свойства». При изменении сопоставления или модели, вы можете временно повторно включить проверки, чтобы проверить внесенные изменения.

Обратите внимание, что внесены улучшения производительности в конструкторе Entity Framework для Entity Framework 6 «проверка при сборке» обходится намного ниже, чем в предыдущих версиях конструктора.

3 кэширования на платформе Entity Framework

Платформа Entity Framework должна кэширования встроенных следующих форм:

1. Кэширование объектов — `ObjectStateManager`, встроенные в экземпляр `ObjectContext` отслеживает в памяти объектов, которые были получены с помощью этого экземпляра. Это также называется кэш первого уровня.
2. Запрос кэширование плана — повторное использование команды созданный, когда запрос выполняется более одного раза.
3. Метаданные, кэширование — совместное использование разных способа подключения к той же модели метаданных для модели.

Помимо кэши, предоставляет EF по умолчанию, это специальный поставщик данных ADO.NET, известных как поставщик упаковки также может использоваться для расширения Entity Framework с кэшем результатов, полученных из базы данных, также известный как кэширование второго уровня.

3.1 кэширование объектов

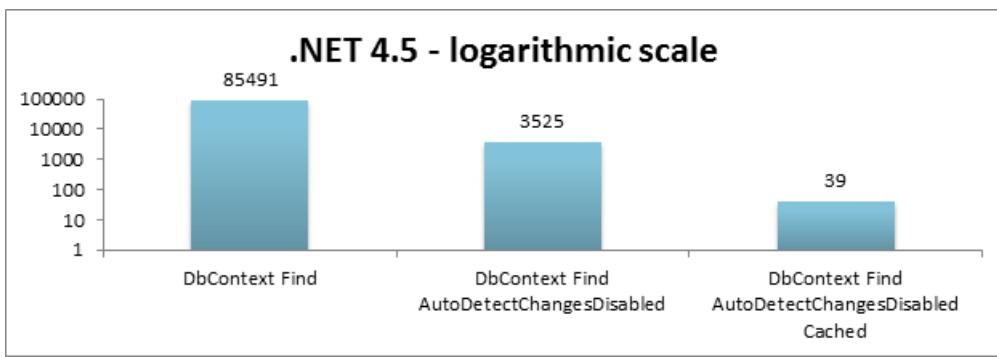
По умолчанию когда сущность возвращается в результатах запроса, непосредственно перед EF материализует, `ObjectContext` проверяет, если сущность с таким ключом уже были загружены в его `ObjectStateManager`. Если сущность с помощью тех же самых ключей уже существует EF будет включить в результаты запроса. Несмотря на то, что EF по-прежнему будет выдавать запрос к базе данных, это поведение можно работать в обход большей стоимости материализация сущности несколько раз.

3.1.1 Получение сущности из объекта кэша с помощью метода `DbContext Find`

В отличие от обычных запросов метод `Find` в `DbSet` (интерфейсов API, включенных в первый раз, в EF 4.1) выполнит поиск в памяти перед даже выдачей запроса к базе данных. Важно отметить, что два разных экземпляра `ObjectContext` будет иметь два разных экземпляров `ObjectStateManager`, это означает, что они имеют отдельный объект кэша.

Поиск использует значение первичного ключа попытку найти сущность, отслеживаемые по контексту. Если сущность не находится в контексте затем запрос будет выполнен и вычисляется в базе данных и возвращается значение `null`, если сущность не найдена в контексте или в базе данных. Обратите внимание, что найти также возвращает сущности, которые были добавлены в контекст, но еще не были сохранены в базе данных.

Есть вопросы производительности, которые должны выполняться при с помощью метода `Find`. Вызовы этого метода по умолчанию активирует проверку кэша объектов для обнаружения изменения, ожидающие выполнения фиксации к базе данных. Этот процесс может быть очень дорогим в том случае, если существует большое количество объектов в кэше объектов или в виде графа больших объектов, добавляемый в кэш объектов, но также может быть отключена. В некоторых случаях может заметить на порядок разницы в вызов `Find`, метод, при отключении автоматического обнаружения изменений. Еще второго порядка воспринимается, когда объект фактически соотношения когда объект должен быть извлечен из базы данных. Ниже приведен пример графа, с помощью изменений, выполненных с помощью некоторых из наших микротестами, выраженный в миллисекундах, с нагрузкой 5000 сущностей.



Ниже приведен пример поиска с изменениями автоопределения отключена.

```
context.Configuration.AutoDetectChangesEnabled = false;
var product = context.Products.Find(productId);
context.Configuration.AutoDetectChangesEnabled = true;
...
```

Что необходимо учитывать при использовании метода Find является:

- Если объект не находится в кэше преимущества поиска отрицательное, но синтаксис по-прежнему проще, чем запрос по ключу.
- Включено ли автоматическое обнаружение изменений может увеличиться стоимость метод Find один порядок или более в зависимости от сложности модели и количество сущностей в объект кэша.

Кроме того Имейте в виду, что найти только возвращает сущность, которую вы ищете, и он не автоматически загружает его связанные сущности, если они отсутствуют в кэше объектов. Если вам необходимо получить связанные сущности, можно использовать запрос по ключу с безотлагательной загрузкой. Дополнительные сведения см. в разделе **8.1 vs отложенной загрузки. Безотложная загрузка**.

3.1.2 производительностью, когда кэш объектов большим количеством сущностей

Кэш объектов позволяет увеличить общую скорость реагирования Entity Framework. Тем не менее при кэша объектов имеет очень большой объем сущностей, загрузки, это может повлиять на некоторые операции, такие как добавление, удаление, найти запись, SaveChanges и многое другое. В частности операции, вызывающие вызов DetectChanges отрицательно влияет кэши очень большой объект. Метод DetectChanges синхронизирует граф объекта с диспетчер состояния объекта и его обеспечивает производительность, напрямую зависит от размера графа объектов. Дополнительные сведения о DetectChanges, см. в разделе [отслеживание изменений в сущностях POCO](#).

При использовании Entity Framework 6, разработчики имеют возможность вызвать AddRange и RemoveRange непосредственно в DbSet, вместо итерации по коллекции и вызова Add один раз на каждый экземпляр. С помощью методов диапазон удобен тем, что стоимость DetectChanges выплачивается только один раз для всего набора сущностей, а не один раз в каждой сущности, добавляемого.

3.2 кэширование плана запросов

Запрос выполняется, впервые проходит внутренний план компилятору преобразовать концептуальные запрос в команду хранилища (например, T-SQL выполняется при выполнении SQL-сервере). Если включено кэширование планов запросов, в следующий раз, запрос будет выполнен хранилище команда получаются непосредственно из кэша планов запросов для выполнения, минуя компилятора плана.

Кэша планов запросов совместно используется экземплярами ObjectContext в том же домене приложения. Не нужно удерживать экземпляр ObjectContext можно было использовать кэширование планов запросов.

3.2.1 некоторые замечания о планировании кэширование запросов

- Кэш планов запроса является общим для все типы запросов: Entity SQL, LINQ to Entities и объекты CompiledQuery.
- По умолчанию кэширование планов запросов включена для запросов Entity SQL, выполнения с помощью

EntityCommand или с помощью ObjectQuery. Он также включается по умолчанию для LINQ для запросов сущностей в Entity Framework в .NET 4.5, а в Entity Framework 6

- Кэширование плана запроса можно отключить, задав свойство EnablePlanCaching (на EntityCommand или ObjectQuery) значение false. Пример:

```
var query = from customer in context.Customer
            where customer.CustomerId == id
            select new
            {
                customer.CustomerId,
                customer.Name
            };
ObjectQuery oQuery = query as ObjectQuery;
oQuery.EnablePlanCaching = false;
```

- Для параметризованных запросов изменив значение параметра по-прежнему достигнет кэшированных запросов. Но изменение аспекты параметров (например, размер, точность или масштаб) достигнет другую запись в кэше.
- При использовании Entity SQL, строка запроса является частью ключа. Все изменения запроса приведет к записи кэша, даже если запросы функционально эквивалентны. Сюда входят изменения регистра или пробел.
- При использовании LINQ, запрос обрабатывается для создания частью ключа. Изменение выражения LINQ таким образом создаст другой ключ.
- Другие технические ограничения могут применяться; Дополнительные сведения см. Autocompiled запросов.

3.2.2 алгоритм вытеснения кэша

Понимание того, как работает внутренняя алгоритм поможет выяснить, чтобы включить или отключить кэширование плана запроса. Алгоритм очистки выглядит следующим образом:

1. Когда кэш содержит определенный набор записей (800), мы начнем таймер, периодически (один раз в минуту) выполняет очистку кэша.
2. Во время переходов кэша записи удаляются из кэша на LFRU (наименее часто — недавно использованные) основы. Этот алгоритм учитывает число попаданий и возраст решая, какие операции, исключаются из кэша.
3. В конце каждого Очистка кэша кэш снова содержит 800 записей.

Все записи кэша обрабатываются одинаково, определяя, какие операции удаления. Это означает, что команда хранилища для CompiledQuery имеет вероятностью вытеснения, как команда хранилища для запроса Entity SQL.

Обратите внимание, что в запускается таймер вытеснения кэша, при наличии 800 сущностей в кэше, но кэш только лежит 60 секунд после запуска этого таймера. Это означает, что до 60 секунд кэша может увеличиваться достаточно большие.

3.2.3 проверить метрики, демонстрация производительности кэширования плана запроса

Чтобы продемонстрировать влияние на производительность приложения кэширование плана запроса, мы выполнили теста где мы выполним несколько запросов Entity SQL к модели Navision. См. в приложении описание модели Navision и типы запросов, которые были выполнены. В этом тесте мы сначала итерацию по списку запросов и выполнения каждого из них один раз для добавления их в кэш (если включено кэширование). Этот шаг является untimed. Далее мы спящий режим главного потока для более чем 60 секунд разрешить кэшировать широкими вступили в силу; Наконец выполняется перебор списка второй время выполнения кэшированные запросы. Кроме того он кэша планов SQL Server очищается перед выполнением каждого набора запросов, чтобы случаи, когда мы получаем точно отражают преимущества, предоставляемых кэша планов запросов.

3.2.3.1 результаты теста

ПРОВЕРКА	EF5 БЕЗ КЭШИРОВАНИЯ	EF5 КЭШИРОВАНИЯ	EF6 БЕЗ КЭШИРОВАНИЯ	EF6 КЭШИРОВАНИЯ
Перечисление всех 18723 запросов	124	125.4	124.3	125.3
Как избежать поворота (просто сначала 800 запросов, вне зависимости от сложности)	41.7	5.5	40,5	5,4
Только запросы AggregatingSubtotals (178 всего - что избавляет от поворота)	39.5	4.5	38,1	4.6

Все значения времени в секундах.

Моральная - при выполнении много возможностей различных запросов (например, динамически созданные запросы), кэширование не увеличит производительность и итоговый Очистка кэша можно сохранить запросы, использующие преимущества наиболее кэширование плана из его фактическое использование.

Запросы AggregatingSubtotals являются наиболее сложных запросов, протестированных с. Как и ожидалось, тем более сложный запрос выполнен, тем больше преимуществ, вы увидите благодаря кэшированию плана запроса.

Сравнение CompiledQuery и эквивалентный запрос Entity SQL CompiledQuery именно с его кэшированный план запроса LINQ, должны быть такой же результат. На самом деле Если приложение содержит массу динамических запросов Entity SQL, заполнение кэша запросов также приведет CompiledQueries «декомпилировать», когда они удаляются из кэша. В этом случае производительность может быть повышенена путем отключения кэширования на динамические запросы для определения приоритетов CompiledQueries. Еще лучше само собой, можно переписать приложение, чтобы использовать параметризованные запросы вместо динамических запросов.

3.3. Использование CompiledQuery для повышения производительности с помощью запросов LINQ

Наши тесты подтверждают, что с помощью CompiledQuery можно перенести преимущество 7% autocompiled запросы LINQ; Это означает, что позволяет тратить 7% меньше времени на выполнение кода из стека Entity Framework; Это означает, что ваше приложение будет 7% быстрее. Вообще говоря затраты на написание и обслуживании объектов CompiledQuery в EF 5.0 могут стоить возникли проблемы, по сравнению с преимущества. Детализация проработки может различаться, поэтому Упражнение этот параметр, если требуются дополнительные Push-уведомления в вашем проекте. Обратите внимание, что CompiledQueries только совместим с производным ObjectContext моделях и не совместим с производным DbContext моделей.

Дополнительные сведения о создании и вызове CompiledQuery см. в разделе [компилированные запросы \(LINQ to Entities\)](#).

Есть две рекомендации, которые необходимо выполнить при использовании CompiledQuery, а именно: требование использовать статические экземпляры и проблем, что они имеют с Компонуемость. Ниже следует подробное разъяснение эти две рекомендации.

3.3.1 используйте статические экземпляры CompiledQuery

Поскольку компиляции запроса LINQ занимает много времени, мы не хотим делать каждый раз, нам нужно

получить данные из базы данных. Экземпляры CompiledQuery позволяют один раз скомпилировать и запустить несколько раз, но следует соблюдать осторожность и приобретение повторно использовать один и тот же экземпляр CompiledQuery каждый раз, а не его снова и снова компиляции. Использование статических членов для хранения экземпляров CompiledQuery становится необходимым; в противном случае вы не увидите каких-либо преимуществ.

Например предположим, что на странице имеется следующий текст метод для обработки, отображение продуктов выбранной категории:

```
// Warning: this is the wrong way of using CompiledQuery
using (NorthwindEntities context = new NorthwindEntities())
{
    string selectedCategory = this.categoriesList.SelectedValue;

    var productsForCategory = CompiledQuery.Compile<NorthwindEntities, string, IQueryable<Product>>(
        (NorthwindEntities nwnd, string category) =>
            nwnd.Products.Where(p => p.Category.CategoryName == category)
    );

    this.productsGrid.DataSource = productsForCategory.Invoke(context, selectedCategory).ToList();
    this.productsGrid.DataBind();
}

this.productsGrid.Visible = true;
```

В этом случае вы создадите новый экземпляр CompiledQuery в режиме реального времени каждый раз при вызове метода. Вместо выигрыш в производительности, получая команда хранилища из кэша планов запросов, CompiledQuery будет проходить через компилятор плана каждый раз при создании нового экземпляра. На самом деле будут избавляют кэша плана запроса с помощью новой записи CompiledQuery каждый раз при вызове метода.

Вместо этого необходимо создать статический экземпляр скомпилированного запроса, чтобы каждый раз при вызове метода вызываются один и тот же скомпилированный запрос. Один из способов это, добавив экземпляр CompiledQuery в качестве члена объекта контекста. Затем можно сделать вещей немного более чистым и читабельным доступ к CompiledQuery через вспомогательный метод:

```
public partial class NorthwindEntities : ObjectContext
{
    private static readonly Func<NorthwindEntities, string, IEnumerable<Product>> productsForCategoryCQ =
        CompiledQuery.Compile(
            (NorthwindEntities context, string categoryName) =>
                context.Products.Where(p => p.Category.CategoryName == categoryName)
        );

    public IEnumerable<Product> GetProductsForCategory(string categoryName)
    {
        return productsForCategoryCQ.Invoke(this, categoryName).ToList();
    }
}
```

Этот вспомогательный метод будет вызываться следующим образом:

```
this.productsGrid.DataSource = context.GetProductsForCategory(selectedCategory);
```

3.3.2 составление через CompiledQuery

Возможность составляемый по любой запрос LINQ является очень полезным. Чтобы сделать это, вы просто вызываете метод после IQueryble например `Skip()` или `Count()`. Тем не менее это по сути, возвращает объект IQueryble. Хотя нет ничего должно остановить вас от составления через CompiledQuery с технической точки зрения, это приведет к созданию нового объекта IQueryble, снова требуется через компилятор плана.

Некоторые компоненты будут работать с IQueryables составных объектов для включения дополнительных функций. Например, ASP. NET GridView может быть связан с данными на объект IQueryables через свойство SelectMethod. GridView будет сочетать этот объект IQueryables, чтобы разрешить сортировку и разбиение по страницам по модели данных. Как вы видите, с помощью CompiledQuery для GridView не мог попасть скомпилированный запрос, но будет создать новый запрос autocompiled.

Группа консультирования клиентов рассказывает об этом в их «Потенциальных производительности проблемы с компиляции LINQ запросов повторно компилирует» записи блога:

<http://blogs.msdn.com/b/appfabriccat/archive/2010/08/06/potential-performance-issues-with-compiled-linq-query-re-compiles.aspx>.

Одно место, где вы можете столкнуться это при добавлении прогрессивного фильтры в запрос. Например предположим, что имеется страница клиентов с несколько раскрывающихся списков для необязательные фильтры (например, "Страна" и "OrdersCount"). Эти фильтры можно составляемый по результатам IQueryables из CompiledQuery. Однако это приведет к в новый запрос, который проходит через компилятор плана каждый раз при его выполнении.

```
using (NorthwindEntities context = new NorthwindEntities())
{
    IQueryable<Customer> myCustomers = context.InvokeCustomersForEmployee();

    if (this.orderCountFilterList.SelectedItem.Value != defaultFilterText)
    {
        int orderCount = int.Parse(orderCountFilterList.SelectedValue);
        myCustomers = myCustomers.Where(c => c.Orders.Count > orderCount);
    }

    if (this.countryFilterList.SelectedItem.Value != defaultFilterText)
    {
        myCustomers = myCustomers.Where(c => c.Address.Country == countryFilterList.SelectedValue);
    }

    this.customersGrid.DataSource = myCustomers;
    this.customersGrid.DataBind();
}
```

Чтобы избежать повторной компиляции, можно переписать CompiledQuery учитывать возможные фильтры:

```
private static readonly Func<NorthwindEntities, int, int?, string, IQueryable<Customer>>
customersForEmployeeWithFiltersCQ = CompiledQuery.Compile(
    (NorthwindEntities context, int empId, int? countFilter, string countryFilter) =>
    context.Customers.Where(c => c.Orders.Any(o => o.EmployeeID == empId))
        .Where(c => countFilter.HasValue == false || c.Orders.Count > countFilter)
        .Where(c => countryFilter == null || c.Address.Country == countryFilter)
);
```

Который будет вызываться в пользовательском Интерфейсе, например:

```

using (NorthwindEntities context = new NorthwindEntities())
{
    int? countFilter = (this.orderCountFilterList.SelectedIndex == 0) ?
        (int?)null :
        int.Parse(this.orderCountFilterList.SelectedValue);

    string countryFilter = (this.countryFilterList.SelectedIndex == 0) ?
        null :
        this.countryFilterList.SelectedValue;

    IQueryable<Customer> myCustomers = context.InvokeCustomersForEmployeeWithFilters(
        countFilter, countryFilter);

    this.customersGrid.DataSource = myCustomers;
    this.customersGrid.DataBind();
}

```

Здесь компромисс — команда созданного хранилища всегда будет иметь фильтры с проверками "null", но они должны быть относительно простыми для сервера базы данных для оптимизации:

```

...
WHERE ((0 = (CASE WHEN (@p_linq_1 IS NOT NULL) THEN cast(1 as bit) WHEN (@p_linq_1 IS NULL) THEN cast(0 as bit) END)) OR ([Project3].[C2] > @p_linq_2)) AND (@p_linq_3 IS NULL OR [Project3].[Country] = @p_linq_4)

```

3.4 кэширование метаданных

Платформа Entity Framework также поддерживает кэширование метаданных. Это по сути кэширует сведения о типе и сведения о сопоставлении типа для базы данных в разных способа подключения к той же модели. Кэш метаданных уникален в пределах одного домена приложения.

3.4.1 кэширование метаданных алгоритма

- Сведения о метаданных для модели хранится в коллекцию ItemCollection для каждого EntityConnection.
 - Заметим существуют разные объекты ItemCollection для разных частей модели. Например StoreItemCollections содержит сведения о модели базы данных; ObjectItemCollection содержит сведения о модели данных; EdmItemCollection содержит сведения о концептуальной модели.
- Если два соединения используют ту же строку подключения, они будут совместно использовать один и тот же экземпляр ItemCollection.
- Строки подключения функционально эквивалентны, но текстовой форме различные может привести разные метаданные кэшей. Мы помечать строки подключения, просто изменив порядок токенов следует привести общие метаданные. Однако две строки подключения, которые кажутся функционально могут не оцениваться идентичными после разметки.
- Коллекция ItemCollection периодически проверяется для использования. Если выяснилось, что рабочая область не обращались недавно, он будет помечена для очистки на Далее Очистка кэша.
- Простое создание объект EntityConnection приведет к кэшу метаданных создаваемой (хотя коллекции элементов в нем не будут инициализированы до открытия подключения). Эта рабочая область будет оставаться в памяти до кэширования алгоритм определяет, что он не «используется».

Группа консультирования клиентов написал в блоге, описывающий содержит ссылку на коллекцию ItemCollection избежание «устаревание» при использовании больших моделей:

<http://blogs.msdn.com/b/appfabriccat/archive/2010/10/22/metadataworkspace-reference-in-wcf-services.aspx>.

3.4.2 связь между кэширование метаданных и планирование кэширование запросов

Экземпляр кэша плана запроса находится в области MetadataWorkspace ItemCollection типов хранилища. Это

означает, что команды кэшированном хранилище будет использоваться для запросов к каким-либо контекстом, создается с помощью заданной области MetadataWorkspace. Это также означает, что если у вас есть две строки подключения, которые немного отличаются и не соответствуют после разделения на лексемы, у вас будет другой запрос планирование экземпляров кэша.

3.5 кэширование результатов

Вместе с результатами кэширование (также называется «второго уровня кэширование») и сохранять результаты запросов в локальном кэше. При выполнении запроса, вы увидите результаты доступны локально, прежде чем запрос к хранилищу. Хотя кэширование результатов не поддерживаются Entity Framework, это можно добавить кэша второго уровня с помощью поставщика упаковки. Поставщик упаковки примере с помощью кэша второго уровня — в Alachisoft [зависимости кэша второго уровня Entity Framework от NCache](#).

Эта реализация кэширования второго уровня является внедренного функциональные возможности, которая принимает место после оценки выражения LINQ (и funcletized) и вычисляется или получить из кэша первого уровня плана выполнения запроса. Кэша второго уровня будут храниться только необработанных результатов, поэтому конвейер материализации по-прежнему выполняется позже.

3.5.1 Дополнительные ссылки для результатов, кэширование с поставщиком упаковки

- Джули Лерман написал статьи MSDN «Второго уровня кэширования в Entity Framework и Windows Azure», которая включает в себя обновление упаковки в образец поставщика, для использования кэширования Windows Server AppFabric: <https://msdn.microsoft.com/magazine/hh394143.aspx>
- Если вы работаете с Entity Framework 5, в блоге группы разработчиков имеет post, в котором описано, как восстановить нормальную с поставщиком кэширования для Entity Framework 5: <<http://blogs.msdn.com/b/adonet/archive/2010/09/13/ef-caching-with-jarek-kowalski-s-provider.aspx>>. Она также включает шаблон T4 для автоматизации, добавление второго уровня кэширования в проект.

4 Autocompiled запросов

Во время запроса к базе данных с помощью Entity Framework, то он должен проходить через ряд шагов до фактически материализации результатов; один шаг — компиляции запроса. Известно, что запросы Entity SQL иметь хорошую производительность, как они автоматически сохраняются в кэше, что второй или третий раз выполнять тот же запрос, его можно пропустить компилятора плана и вместо этого использовать кэшированный план.

Entity Framework 5 появился автоматическое кэширование для LINQ для запросов сущностей. В прошлых выпусках платформы Entity Framework, создание CompiledQuery для ускорения производительность была обычной практикой, так как это делает LINQ to Entities в запрос может быть кэширован. Так как кэширование теперь выполняется автоматически без использования CompiledQuery, мы называем эту функцию «autocompiled запросы». Дополнительные сведения о кэше планов запросов и его механизмы см. в разделе Кэширование планов запросов.

Платформа Entity Framework обнаруживает, когда запрос требует повторной компиляции, и делает это, при вызове даже если компилировали до запроса. Ниже приведены распространенные условия, вызывающие повторную компиляцию запроса.

- Изменение MergeOption, связанный в запрос. Кэшированный запрос не будет использоваться, вместо этого компилятор план будет выполняться снова и кэшируемый только что созданный план.
- Изменение значения ContextOptions.UseCSharpNullComparisonBehavior. Вы получаете тот же эффект, что изменение MergeOption.

Другие условия можно запретить запроса из кэша. Ниже приведены распространенные примеры.

- С помощью IEnumerable<T>. Содержит<>(значение T).
- Использование функций, которые создают запросы с константами.

- С помощью свойств объекта не сопоставлен.
- Связывание запроса в другой запрос, который требует повторной компиляции.

4.1 с помощью `IEnumerable<T>`. Содержит<T>(значение T)

Платформа Entity Framework не кэширует запросы, которые вызывают `IEnumerable<T>.Содержит<T>`(значение T) по коллекции в памяти, так как значения коллекции считаются временными. В следующем примере запроса не будут кэшироваться, поэтому он всегда будет обрабатываться компилятором плана:

```
int[] ids = new int[10000];
...
using (var context = new MyContext())
{
    var query = context.MyEntities
        .Where(entity => ids.Contains(entity.Id));

    var results = query.ToList();
    ...
}
```

Обратите внимание, что выполняемого размер `IEnumerable` от содержащий определяет, насколько быстро или медленно как запрос компилируется. Может наблюдаться снижение производительности значительно при использовании больших коллекций, как показано в приведенном выше примере.

Оптимизировано Entity Framework 6 как `IEnumerable<T>.Содержит<T>(значение T)` работает при выполнении запросов. Код SQL, который создается во много раз быстрее создавать и более удобном для чтения, и в большинстве случаев также выполняется быстрее на сервере.

4.2 с помощью функций, которые создают запросы с константами

Операторы `Skip()`, методы `Take()`, `Contains()` и `DefaultIfEmpty()` LINQ не вызывают SQL-запросы с параметрами, но вместо этого поместите значения, передаваемые им как константы. По этой причине запросы, которые в противном случае могут быть идентичными окажутся избавляют запрос Планируйте кэша, как в стеке EF, так и на сервере базы данных и не получить `reutilized`, если же константы не используются при выполнении последующих запросов. Пример:

```
var id = 10;
...
using (var context = new MyContext())
{
    var query = context.MyEntities.Select(entity => entity.Id).Contains(id);

    var results = query.ToList();
    ...
}
```

В этом примере каждый раз при выполнении этого запроса с другим значением для идентификатора запроса будут скомпилированы в новый план.

В определенной следует обратить внимание на использование предложения `Skip` и `Take` при выполнении разбиения по страницам. В EF6 этих методов есть перегрузки, лямбда-выражения, который фактически делает кэшированного плана запроса для повторного использования, так как EF можно фиксировать переменные, передаваемый для этих методов и преобразовывать их в параметров `SQLparameters`. Это также помогает сохранять кэш чище, поскольку в противном случае каждый запрос с различные константы для предложения `Skip` и `Take` бы получает собственную запись кэша плана запроса.

Рассмотрим следующий код, который не является оптимальным, но предназначен только для того чтобы проиллюстрировать этот класс запросов:

```
var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i < count; ++i)
{
    var currentCustomer = customers.Skip(i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}
```

Более быстрой версии этого кода будет включать в себя вызов Skip с лямбда-выражения:

```
var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i < count; ++i)
{
    var currentCustomer = customers.Skip(() => i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}
```

Второй фрагмент могут выполняться до 11% быстрее, потому что каждый раз выполняется запрос, сохраняющий время ЦП и позволяет избежать избавляют кэша запросов, используется один и тот же план запроса. Кроме того так как параметр Skip, в замыкании код может также выглядеть теперь:

```
var i = 0;
var skippyCustomers = context.Customers.OrderBy(c => c.LastName).Skip(() => i);
for (; i < count; ++i)
{
    var currentCustomer = skippyCustomers.FirstOrDefault();
    ProcessCustomer(currentCustomer);
}
```

4.3 с помощью свойств объекта несопоставленные

Когда запрос использует свойства типа несопоставленные объекта как параметр, а затем запрос не будет помещаться. Пример:

```
using (var context = new MyContext())
{
    var myObject = new NonMappedType();

    var query = from entity in context.MyEntities
               where entity.Name.StartsWith(myObject.MyProperty)
               select entity;

    var results = query.ToList();
    ...
}
```

В этом примере предполагается, что класс NonMappedType не является частью модели сущности. Этот запрос можно легко изменить, не используйте тип несопоставленные и вместо этого использовать локальную переменную в качестве параметра к запросу:

```

using (var context = new MyContext())
{
    var myObject = new NonMappedType();
    var myValue = myObject.MyProperty;
    var query = from entity in context.MyEntities
                where entity.Name.StartsWith(myValue)
                select entity;

    var results = query.ToList();
    ...
}

```

В этом случае запрос будет помещаться и выиграет от кэша планов запросов.

4.4 связывание запросов, которые требует повторной компиляции

Следующие тот же пример, как описано выше при наличии второй запрос, который основывается на запросе, который нужно перекомпилировать, весь второй запрос будет также перекомпилирован. Ниже приведен пример, чтобы продемонстрировать этот сценарий:

```

int[] ids = new int[10000];
...
using (var context = new MyContext())
{
    var firstQuery = from entity in context.MyEntities
                     where ids.Contains(entity.Id)
                     select entity;

    var secondQuery = from entity in context.MyEntities
                      where firstQuery.Any(otherEntity => otherEntity.Id == entity.Id)
                      select entity;

    var results = secondQuery.ToList();
    ...
}

```

Пример является универсальным, но он иллюстрирует, как связывание с firstQuery вызывает secondQuery смогут окажутся кэшированными. Если firstQuery не был запрос, который требует повторной компиляции, затем secondQuery бы были кэшированы.

5 NoTracking запросов

5.1 отключить отслеживание изменений, чтобы сократить расходы на управление состоянием

Если вы являетесь в сценарии только для чтения и хотите избежать дополнительной нагрузки при загрузке объектов в диспетчере ObjectStateManager, можно выполнять запросы «No отслеживания». Отслеживание изменений можно отключить на уровне запроса.

Обратите внимание, что путем отключения отслеживания изменений равносильно отключению этого кэша объектов. При запросе сущности, мы не можем пропустить материализации, используя результаты запроса материализуются ранее из ObjectStateManager. При запросе несколько раз для той же сущности, на том же контексте, может увидеть извлечь выгоду от включения отслеживания изменений производительности.

При запросе с помощью ObjectContext, экземпляры ObjectQuery и ObjectSet запомнят MergeOption после того как оно задано, и запросы, которые составляются на них будут наследовать действующие MergeOption родительского запроса. При использовании DbContext, отслеживания можно отключить, вызвав модификатор AsNoTracking() на DbSet.

5.1.1 Отключение отслеживания изменений для запроса при использовании DbContext

Можно переключить режим запроса для NoTracking с помощью цепочки из вызова метода AsNoTracking() в

запросе. В отличие от ObjectQuery DbSet и DbQuery классы в DbContext API не имеют изменяемые свойства для MergeOption.

```
var productsForCategory = from p in context.Products.AsNoTracking()
                           where p.Category.CategoryName == selectedCategory
                           select p;
```

5.1.2 Отключение отслеживания на уровне запроса с помощью ObjectContext изменений

```
var productsForCategory = from p in context.Products
                           where p.Category.CategoryName == selectedCategory
                           select p;

((ObjectQuery)productsForCategory).MergeOption = MergeOption.NoTracking;
```

5.1.3 Отключение отслеживания изменений для всего набора с помощью ObjectContext сущностей

```
context.Products.MergeOption = MergeOption.NoTracking;

var productsForCategory = from p in context.Products
                           where p.Category.CategoryName == selectedCategory
                           select p;
```

5.2 показатели, демонстрирующие преимущества высокой производительности запросов NoTracking теста

В этом тесте мы будем за счет заполнения, сравнивая отслеживания к запросам NoTracking для модели Navision ObjectStateManager. См. в приложении описание модели Navision и типы запросов, которые были выполнены. В этом тесте мы итерацию по списку запросов и выполнения каждого из них один раз. Мы запустили две вариации теста, один раз с NoTracking запросов и один раз с параметром слияния по умолчанию «AppendOnly». Мы выполнили налагается три раза и взять среднее значение выполнения. Между двумя тестами Очистка кэша запросов на SQL Server и сжатие базы данных tempdb, выполнив следующие команды:

1. DBCC DROPCLEANBUFFERS
2. DBCC FREEPROCCACHE
3. DBCC SHRINKDATABASE (tempdb, 0)

Результаты тестов, медианы более чем трех запуски:

	ОТСЛЕЖИВАНИЕ — РАБОЧИЙ НАБОР	НЕТ — ВРЕМЯ	ИНКРЕМЕНТИРУЕМЫЕ — РАБОЧИЙ НАБОР	ДОБАВЛЕНИЕ ТОЛЬКО — ВРЕМЯ
Entity Framework 5	460361728	1163536 ms	596545536	1273042 ms
Entity Framework 6	647127040	190228 ms	832798720	195521 ms

Entity Framework 5 будет иметь небольшого объема памяти в конце выполнения, чем Entity Framework 6. Дополнительная память, занятая Entity Framework 6 является результатом дополнительную память структуры и кода, включение новых функций и повышения производительности.

Имеется также различия в объем памяти при использовании ObjectStateManager. Entity Framework 5 увеличить ее объем, 30%, когда отслеживает все сущности, которые мы материализованных из базы данных. Entity Framework 6 увеличить ее объем, 28%, при этом.

С точки зрения времени Entity Framework 6 превосходит по производительности Entity Framework 5 в этом

тесте, большие поля. Entity Framework 6 выполнить тест в примерно 16% времени, затраченное Entity Framework 5. Кроме того Entity Framework 5 занимает 9% больше времени при использовании ObjectStateManager. В отличие от этого Entity Framework 6 используется 3% больше времени, при использовании ObjectStateManager.

6 параметры выполнения запроса

Платформа Entity Framework предлагает несколько разных способов для запроса. Мы рассмотрим следующие параметры, сравнить преимущества и недостатки каждого из них и проверить соответствующие характеристики производительности:

- LINQ to Entities.
- Нет отслеживания LINQ to Entities.
- Язык Entity SQL через ObjectQuery.
- Язык Entity SQL через EntityCommand.
- ExecuteStoreQuery.
- SQL-запрос.
- CompiledQuery.

6.1 запросы LINQ to Entities

```
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
```

Преимущества

- Подходит для операций CUD.
- Полностью материализованные объекты.
- Самый простой для записи с синтаксисом, встроенные в язык программирования.
- Хорошую производительность.

Недостатки

- Некоторые технические ограничения, такие как:
 - Шаблоны, использование DefaultIfEmpty для запросов, OUTER JOIN привести к более сложные запросы, чем простые инструкции OUTER JOIN в Entity SQL.
 - Вы по-прежнему не удастся использовать ПОДОБНЫЕ с совпадающими общим шаблон.

6.2 не отслеживания LINQ для запросов сущностей

Когда контекст наследуется ObjectContext:

```
context.Products.MergeOption = MergeOption.NoTracking;  
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
```

Когда контекст наследуется DbContext:

```
var q = context.Products.AsNoTracking()  
    .Where(p => p.Category.CategoryName == "Beverages");
```

Преимущества

- Повышенная производительность по сравнению обычных запросов LINQ.
- Полностью материализованные объекты.
- Самый простой для записи с синтаксисом, встроенные в язык программирования.

Недостатки

- Не подходит для операций CUD.
- Некоторые технические ограничения, такие как:
 - Шаблоны, использование DefaultIfEmpty для запросов, OUTER JOIN привести к более сложные запросы, чем простые инструкции OUTER JOIN в Entity SQL.
 - Вы по-прежнему не удается использовать ПОДОБНЫЕ с совпадающими общий шаблон.

Обратите внимание на то, что запросы, скалярные свойства проекта не отслеживаются даже если NoTracking не указан. Пример:

```
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages").Select(p => new { p.ProductName });
});
```

Этот запрос не указывает явно, NoTracking, но так как он не материализации тип, известно, что диспетчер состояния объекта затем материализованный результат не отслеживается.

6.3 entity SQL через ObjectQuery

```
ObjectQuery<Product> products = context.Products.Where("it.Category.CategoryName = 'Beverages'");
```

Преимущества

- Подходит для операций CUD.
- Полностью материализованные объекты.
- Поддерживает кэширование плана запросов.

Недостатки

- Включает в себя строки текстовых запросов, которые являются более могут совершать ошибки чем конструкции запросов, встроенные в язык.

6.4 entity SQL через команду сущности

```
EntityCommand cmd = eConn.CreateCommand();
cmd.CommandText = "Select p From NorthwindEntities.Products As p Where p.Category.CategoryName = 'Beverages'";

using (EntityDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
{
    while (reader.Read())
    {
        // manually 'materialize' the product
    }
}
```

Преимущества

- Кэширование плана в .NET 4.0 (кэширование плана поддерживается всеми типами запросов в .NET 4.5) запросов поддерживает.

Недостатки

- Включает в себя строки текстовых запросов, которые являются более могут совершать ошибки чем конструкции запросов, встроенные в язык.
- Не подходит для операций CUD.
- Результаты материализуются автоматически и должны быть считаны из считывателя данных.

6.5 SqlQuery и ExecuteStoreQuery

SQL-запрос в базе данных:

```
// use this to obtain entities and not track them
var q1 = context.Database.SqlQuery<Product>("select * from products");
```

SQL-запрос на DbSet:

```
// use this to obtain entities and have them tracked
var q2 = context.Products.SqlQuery("select * from products");
```

ExecuteStoreQuery:

```
var beverages = context.ExecuteStoreQuery<Product>(
    @"      SELECT          P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued, P.DiscontinuedDate
        FROM          Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
        WHERE          (C.CategoryName = 'Beverages')"
);
```

Преимущества

- Обычно высокая производительность, так как план компилятора пропускается.
- Полностью материализованные объекты.
- Подходит для операций CUD при использовании из DbSet.

Недостатки

- Запрос является текстовым и подвержено ошибкам.
- Запрос привязывается к конкретной серверной части с использованием семантики хранилища вместо концептуальная семантика.
- При наличии наследования собственноручно запрос должен учитывать условия сопоставления для запрошенного типа.

6.6 CompiledQuery

```
private static readonly Func<NorthwindEntities, string, IQueryable<Product>> productsForCategoryCQ =
CompiledQuery.Compile(
    (NorthwindEntities context, string categoryName) =>
    context.Products.Where(p => p.Category.CategoryName == categoryName)
);

...
var q = context.InvokeProductsForCategoryCQ("Beverages");
```

Преимущества

- Доступно до 7% производительность через обычные запросы LINQ.
- Полностью материализованные объекты.
- Подходит для операций CUD.

Недостатки

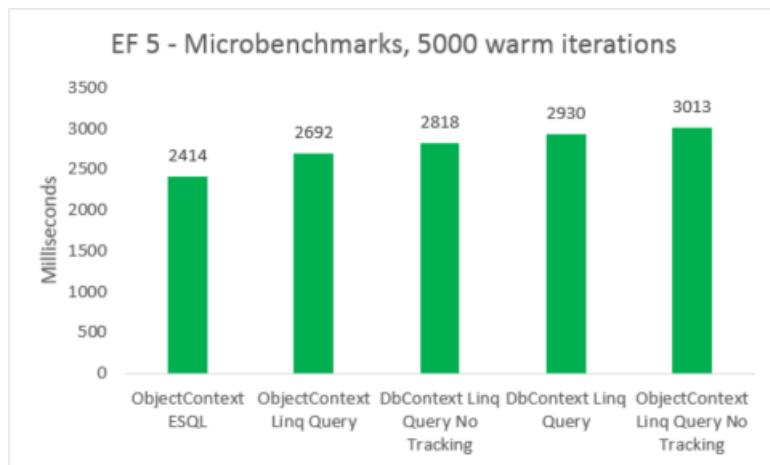
- Увеличить сложность и издержки программирования.
- Улучшение производительности теряется при составлении поверх скомпилированный запрос.
- Нельзя записать некоторые запросы LINQ в качестве CompiledQuery - например, проекции, анонимных

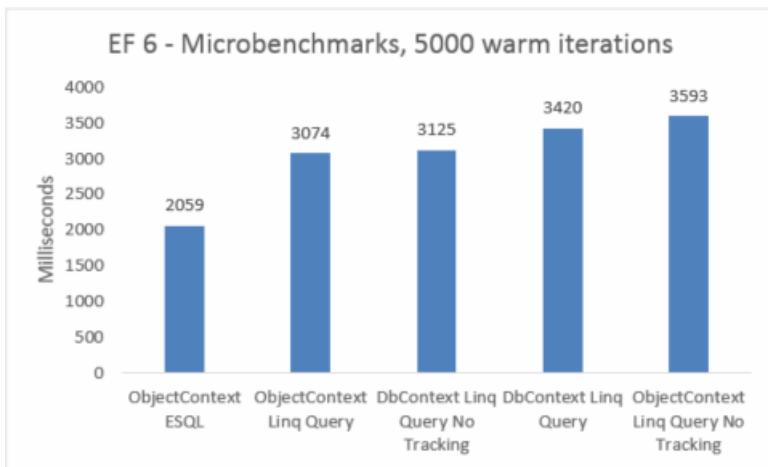
типов.

6.7 Сравнение производительности параметров различных запросов

Простой микротестами, где создания контекста не истек, были помещены в тест. Мы разработали, запрашивая 5000 раз набор сущностей без кэширования в контролируемой среде. Эти числа должны браться с предупреждением: они не отражают фактические значения, созданные приложением, но они являются очень точные измерения уровня использования разница в производительности при сравнении разных составления запросов яблоки требуют, за исключением затрат на создание нового контекста.

EF	ПРОВЕРКА	ВРЕМЯ (МС)	ПАМЯТЬ
EF5	ObjectContext ESQL	2414	38801408
EF5	Запрос Linq ObjectContext	2692	38277120
EF5	DbContext Linq-запроса без отслеживания	2818	41840640
EF5	Запрос Linq DbContext	2930	41771008
EF5	ObjectContext Linq-запроса без отслеживания	3013	38412288
EF6	ObjectContext ESQL	2059	46039040
EF6	Запрос Linq ObjectContext	3074	45248512
EF6	DbContext Linq-запроса без отслеживания	3125	47575040
EF6	Запрос Linq DbContext	3420	47652864
EF6	ObjectContext Linq-запроса без отслеживания	3593	45260800



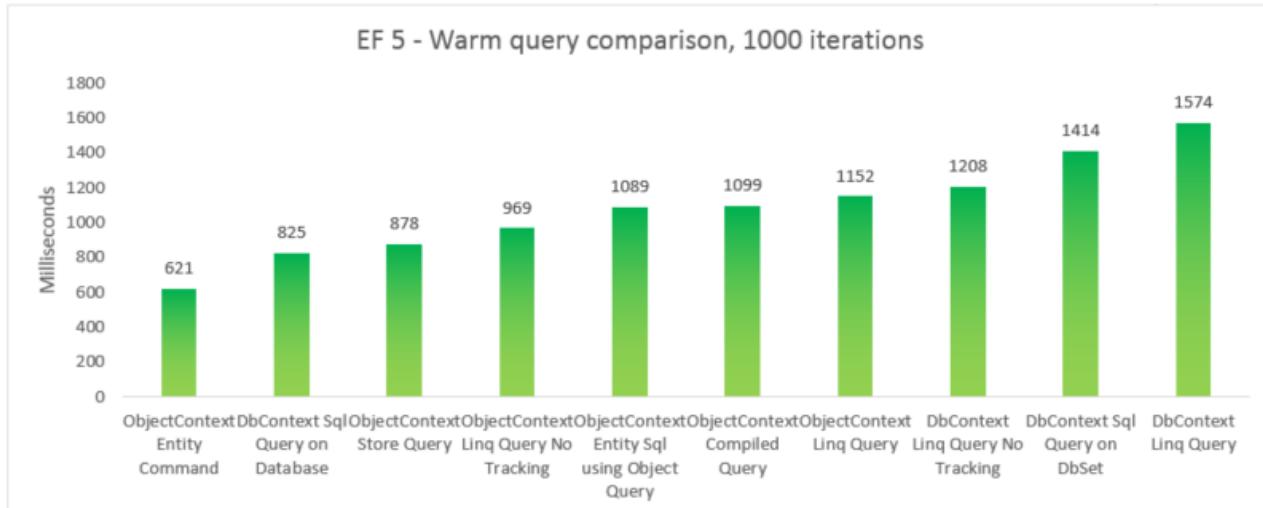


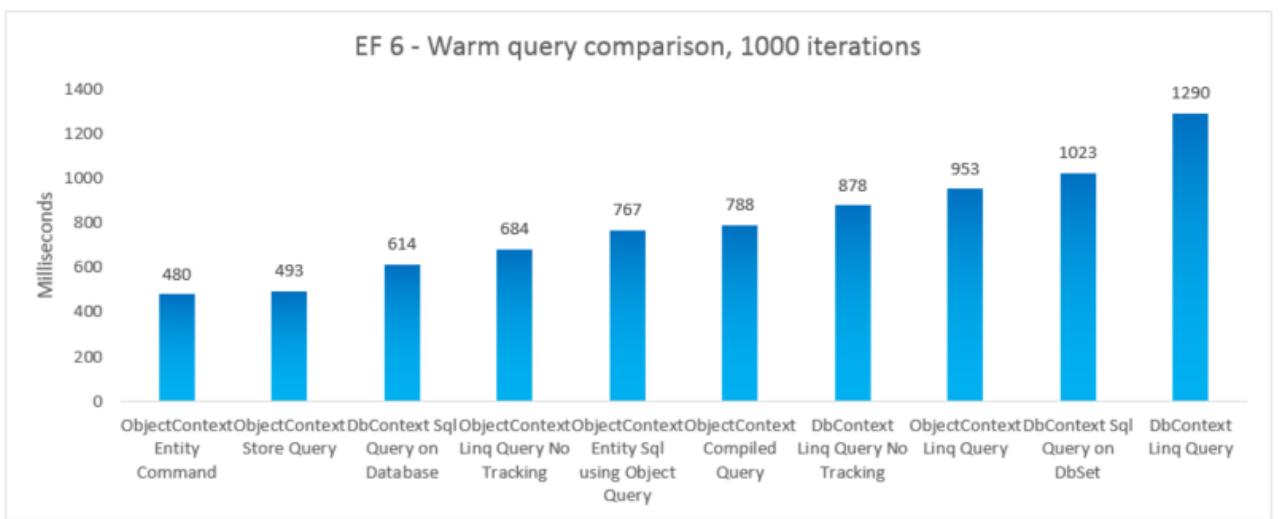
Микротестами очень чувствительны к небольшие изменения в коде. В этом случае разница между стоимостью Entity Framework 5 и Entity Framework 6 возникает из-за появлениями [перехвата](#) и [транзакций улучшения](#). Однако эти числа микротестами являются смещаться представления к платформе Entity Framework достигается очень небольшой фрагмент. Сценарии из реальной жизни "горячего" резервирования запросов должны отствовать снижение производительности при обновлении с Entity Framework 5 для Entity Framework 6.

Чтобы сравнить производительность реальных параметры другой запрос, мы создали 5 отдельный тестовый вариации, где используется параметр другой запрос для выбора всех продуктов, которого зовут категории «Напитки». Каждая итерация включает затрат на создание контекста и стоимость материализации все возвращаемые сущности. Перед созданием суммы истекло время ожидания 1000 итераций, выполняются untimed 10 итераций. Показаны не все результаты выполнения медианы, взятое из 5 запусков каждого теста. Дополнительные сведения см. приложение Б, которая включает код для теста.

EF	ПРОВЕРКА	ВРЕМЯ (МС)	ПАМЯТЬ
EF5	Команда ObjectContext сущности	621	39350272
EF5	DbContext Sql-запроса для базы данных	825	37519360
EF5	Query Store ObjectContext	878	39460864
EF5	ObjectContext Linq-запроса без отслеживания	969	38293504
EF5	ObjectContext Entity Sql с помощью запроса объектов	защищены	38981632
EF5	Скомпилированный запрос ObjectContext	1099	38682624
EF5	Запрос Linq ObjectContext	1152	38178816
EF5	DbContext Linq-запроса без отслеживания	1208	41803776
EF5	DbContext Sql-запроса для DbSet	1414	37982208

EF	ПРОВЕРКА	ВРЕМЯ (МС)	ПАМЯТЬ
EF5	Запрос Linq DbContext	1574	41738240
EF6	Команда ObjectContext сущности	480	47247360
EF6	Query Store ObjectContext	493	46739456
EF6	DbContext Sql-запроса для базы данных	614	41607168
EF6	ObjectContext Linq-запроса без отслеживания	684	46333952
EF6	ObjectContext Entity Sql с помощью запроса объектов	767	48865280
EF6	Скомпилированный запрос ObjectContext	788	48467968
EF6	DbContext Linq-запроса без отслеживания	878	47554560
EF6	Запрос Linq ObjectContext	953	47632384
EF6	DbContext Sql-запроса для DbSet	1023	41992192
EF6	Запрос Linq DbContext	1290	47529984





NOTE

Для полноты информации мы включили как вариант, где мы выполнить запрос Entity SQL в EntityCommand. Тем не менее, так как результаты материализуются для таких запросов, сравнение не обязательно яблоки требуют. Тест включает приближенное к материализации также попытаться создать при более равноправном сравнения.

В данном случае end-to-end, Entity Framework 6 превосходит по производительности Entity Framework 5 из-за повышение производительности на несколько частей стека, в том числе гораздо скромнее инициализации DbContext и быстрее MetadataCollection<T> уточняющих запросов.

7 вопросы производительности разработки

7.1 стратегии наследования

Еще один нюанс производительности при использовании Entity Framework является использования стратегии наследования. Платформа Entity Framework поддерживает три основных типа наследования и их комбинации:

- Таблица на иерархию (TPH) — устанавливаемый карты в таблицу со столбцом дискриминатора, чтобы указать, какой конкретный тип в иерархии наследования каждого представляется в строке.
- Таблица на тип (TPT) — где каждый тип имеет собственную таблицу в базе данных; дочерние таблицы только определить столбцы, которые не содержат родительской таблицы.
- Таблицы на класс (TPC) — где каждый тип имеет собственную полную таблицу в базе данных; дочерние таблицы определите всех их полях, включая определенные в родительских типов.

Если модель использует наследование TPT, запросы, которые создаются будет сложнее, чем те, которые были созданы с помощью наследования стратегиях, которые могут на более длительное время выполнения в хранилище. Обычно займет больше времени для формирования запросов на основе модели TPT и для материализации полученные объекты.

См. в разделе «Сведения о производительности при использовании наследования TPT (одна таблица на тип) на платформе Entity Framework» в блоге MSDN:

<<http://blogs.msdn.com/b/adonet/archive/2010/08/17/performance-considerations-when-using-tpt-table-per-type-inheritance-in-the-entity-framework.aspx>>.

7.1.1 Предотвращение TPT в приложениях сначала модель или «сначала код»

При создании модели поверх существующей базы данных со схемой, TPT, у вас нет множество параметров. Но при создании приложения с помощью Model First или Code First, следует избегать наследования TPT для проблем с производительностью.

При использовании Model First в мастере конструктора сущностей, вы получите TPT для любой

наследования в модели. Переключиться на стратегию наследование TPH с Model First следует можно использовать «сущности конструктор базы данных поколения Power Pack» доступны из коллекции Visual Studio (<<http://visualstudiogallery.msdn.microsoft.com/df3541c3-d833-4b65-b942-989e7ec74c87/>>).

При использовании Code First для настройки сопоставления модели с наследованием, по умолчанию EF будет использовать TPH, поэтому все сущности в иерархии наследования сопоставляется с той же таблицы. См. в разделе «Сопоставление с Fluent API» в статье «Код первой в сущности Framework4.1» в журнале MSDN Magazine (<http://msdn.microsoft.com/magazine/hh126815.aspx>) для получения дополнительных сведений.

7.2 обновляете EF4 для улучшения формирования модели времени

Это улучшение связанные с SQL Server для алгоритма, который создает уровень хранилища (SSDL) модели доступен в Entity Framework 5 и 6, а также как обновление для Entity Framework 4 при установке Visual Studio 2010 с пакетом обновления 1. Ниже приведены результаты теста демонстрируют улучшения при генерации очень большие модели, в этом случае Navision модели. Дополнительные сведения о ней см.

Модель содержит наборы сущностей 1005 и 4227 наборы ассоциаций.

КОНФИГУРАЦИЯ	ДЕКОМПОЗИЦИЯ ЗАТРАЧЕННОЕ ВРЕМЯ
Visual Studio 2010, платформа Entity Framework 4	Создание языка SSDL: 2 часа 27 мин. Создание сопоставления: 1 секунда Создание языка CSDL: 1 секунда Создание ObjectLayer: 1 секунда Создание представлений: 2 ч 14 мин.
Visual Studio 2010 с пакетом обновления 1, Entity Framework 4	Создание языка SSDL: 1 секунда Создание сопоставления: 1 секунда Создание языка CSDL: 1 секунда Создание ObjectLayer: 1 секунда Создание представлений: 1 час 53 мин.
Visual Studio 2013, Entity Framework 5	Создание языка SSDL: 1 секунда Создание сопоставления: 1 секунда Создание языка CSDL: 1 секунда Создание ObjectLayer: 1 секунда Создание представлений: 65 минут
Visual Studio 2013, Entity Framework 6	Создание языка SSDL: 1 секунда Создание сопоставления: 1 секунда Создание языка CSDL: 1 секунда Создание ObjectLayer: 1 секунда Создание представлений: 28 секунд.

Стоит отметить, что при создании на языке SSDL, нагрузки почти полностью расходуется на SQL Server, пока клиентский компьютер для разработки находится в состоянии ожидания простоя, результаты могут возвращаться с сервера. Администраторы баз данных особенно следует оценить это улучшение. Стоит также отметить, что по сути всего затраты на создание модели происходит в средств создания представления теперь.

7.3 разделение большие модели с базой данных, сначала и Model First

По мере увеличения размера модели рабочей области конструктора становится загроможденным и сложно использовать. Обычно мы рассмотрим модель с более чем 300 сущностями слишком большой для эффективного использования конструктора. В следующей записи блога описывается несколько вариантов для разделения больших моделей: <<http://blogs.msdn.com/b/adonet/archive/2008/11/25/working-with-large-models-in-entity-framework-part-2.aspx>>.

Post был написан для первой версии Entity Framework, но по-прежнему применяются действия.

7.4 вопросы производительности с элементом управления источником данных сущности

Мы видели вариантов в многопоточных производительности и нагрузочные тесты, где значительно снижается производительность веб-приложение, использующее элемент управления EntityDataSource. Причиной является то, что EntityDataSource многократно вызывает MetadataWorkspace.LoadFromAssembly для сборок, указанных для веб-приложения для обнаружения типов для использования в качестве сущностей.

Решение — присвоить ContextTypeName элемента управления EntityDataSource имя типа производного класса ObjectContext. Это отключает механизм, который сканирует все связанные сборки для типов сущностей.

Установив в поле ContextTypeName также предотвращает функциональной проблемы, где EntityDataSource в .NET 4.0 выдает исключения ReflectionTypeLoadException при его не удастся загрузить тип из сборки с помощью отражения. Эта проблема была устранена в .NET 4.5.

7.5 сущностей РОСО и прокси, отслеживающих изменения

Платформа Entity Framework позволяет использовать пользовательские классы данных с моделью данных без внесения изменений в классах данных. Это означает, что с моделью данных могут быть использованы традиционные объекты среды CLR (POCO), например существующие объекты домена. Классы данных РОСО (также известный как игнорирующих сохраняемость объектов), которые сопоставлены с сущностями, которые определены в модели данных, поддерживают большинство того же запроса, вставки, обновления и удаления, что и типы сущностей, создаваемых средствами модели EDM.

Платформа Entity Framework можно также создать прокси-классы, производные от типов РОСО, которые используются для включения функций, таких как отложенная загрузка и автоматическое отслеживание изменений для сущностей РОСО. Классы РОСО должны соответствовать определенным требованиям, чтобы разрешить Entity Framework для использования прокси-серверы, как описано здесь:

<http://msdn.microsoft.com/library/dd468057.aspx>.

Возможность отслеживания учетных записей-посредников уведомит диспетчер состояния объекта каждый раз, в каких-либо свойств сущностей имеет свое значение, поэтому платформа Entity Framework знает фактическое состояние сущностей постоянно. Это делается путем добавления события уведомления в тело методов задания свойств и необходимости обработки таких событий диспетчера состояния объектов. Обратите внимание, что создание учетной записи-посредника сущности обычно будет быть дороже, чем создание сущности РОСО без прокси-сервера из-за добавлен набор события, созданные с Entity Framework.

Когда нет прокси отслеживания изменений для сущности РОСО, изменения можно найти, сравнением содержимого строк в одной сущности для копии предыдущее сохраненное состояние. Это сравнение глубокого станет длительное время при наличии нескольких сущностей в контексте, или когда сущностей имеют очень большой объем свойства, даже если ни одна из них изменен с момента последнего сравнения.

В сводке: вам нужно оплатить производительность при создании прокси отслеживания изменений, но отслеживания изменений поможет вам ускорить процесс обнаружения изменений, когда сущностей имеют много свойств, или при наличии нескольких сущностей в модели. Для сущностей с небольшим числом свойств, где объем сущностей не увеличиваться слишком много наличие прокси отслеживания изменений может оказаться много преимуществ.

8 загрузка связанных сущностей

8.1 vs отложенной загрузки. Активная загрузка

Платформа Entity Framework предлагает несколько способов загрузки сущностей, связанных с вашей целевой сущности. Например, при запросе для продуктов, можно разными способами загружаются связанные заказы в диспетчер состояния объекта. С точки зрения производительности будет крупнейших

вопрос загрузки связанных сущностей, необходимость использования отложенной загрузки или Безотложная загрузка.

Если вы используете, упреждающая загрузка, связанные сущности загружаются вместе с вашей целевой набор сущностей. При использовании инструкции `Include` в запросе, чтобы указать, какие связанные сущности, которые вы хотите перенести.

При использовании отложенной загрузки, начальный запрос вернет только в целевой набор сущностей. Но при каждом обращении к свойству навигации, другой запрос, получающий хранилище, чтобы загрузить связанные сущности.

После загрузки сущности, любые дальнейшие запросы для сущности будет загружать непосредственно из диспетчера состояний объекта, при использовании отложенной загрузки или Безотложная загрузка.

8.2 о выборе между отложенной загрузки и упреждающей загрузка

Важно понимать разницу между отложенной загрузки и упреждающей загрузка, таким образом, чтобы сделать правильный выбор для вашего приложения. Это поможет вам оценить компромисс между нескольких запросов в базе данных и один запрос, который может содержать большой объем данных. Возможно, следует использовать безотложную загрузку в некоторые части приложения и отложенной загрузки в других частях.

Пример того, что происходит за кулисами Предположим, что вам нужно создать запрос для клиентов, проживающих в Великобритании и их число заказов.

Используя Безотложную загрузку

```
using (NorthwindEntities context = new NorthwindEntities())
{
    var ukCustomers = context.Customers.Include(c => c.Orders).Where(c => c.Address.Country == "UK");
    var chosenCustomer = AskUserToPickCustomer(ukCustomers);
    Console.WriteLine("Customer Id: {0} has {1} orders", customer.CustomerID, customer.Orders.Count);
}
```

Использование отложенной загрузки

```
using (NorthwindEntities context = new NorthwindEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;

    //Notice that the Include method call is missing in the query
    var ukCustomers = context.Customers.Where(c => c.Address.Country == "UK");

    var chosenCustomer = AskUserToPickCustomer(ukCustomers);
    Console.WriteLine("Customer Id: {0} has {1} orders", customer.CustomerID, customer.Orders.Count);
}
```

При использовании Безотложная загрузка, будет выдавать запрос, возвращает всех заказчиков и все заказы. Команда хранилища выглядит как:

```

SELECT
[Project1].[C1] AS [C1],
[Project1].[CustomerID] AS [CustomerID],
[Project1].[CompanyName] AS [CompanyName],
[Project1].[ContactName] AS [ContactName],
[Project1].[ContactTitle] AS [ContactTitle],
[Project1].[Address] AS [Address],
[Project1].[City] AS [City],
[Project1].[Region] AS [Region],
[Project1].[PostalCode] AS [PostalCode],
[Project1].[Country] AS [Country],
[Project1].[Phone] AS [Phone],
[Project1].[Fax] AS [Fax],
[Project1].[C2] AS [C2],
[Project1].[OrderID] AS [OrderID],
[Project1].[CustomerID1] AS [CustomerID1],
[Project1].[EmployeeID] AS [EmployeeID],
[Project1].[OrderDate] AS [OrderDate],
[Project1].[RequiredDate] AS [RequiredDate],
[Project1].[ShippedDate] AS [ShippedDate],
[Project1].[ShipVia] AS [ShipVia],
[Project1].[Freight] AS [Freight],
[Project1].[ShipName] AS [ShipName],
[Project1].[ShipAddress] AS [ShipAddress],
[Project1].[ShipCity] AS [ShipCity],
[Project1].[ShipRegion] AS [ShipRegion],
[Project1].[ShipPostalCode] AS [ShipPostalCode],
[Project1].[ShipCountry] AS [ShipCountry]
FROM (
  SELECT
    [Extent1].[CustomerID] AS [CustomerID],
    [Extent1].[CompanyName] AS [CompanyName],
    [Extent1].[ContactName] AS [ContactName],
    [Extent1].[ContactTitle] AS [ContactTitle],
    [Extent1].[Address] AS [Address],
    [Extent1].[City] AS [City],
    [Extent1].[Region] AS [Region],
    [Extent1].[PostalCode] AS [PostalCode],
    [Extent1].[Country] AS [Country],
    [Extent1].[Phone] AS [Phone],
    [Extent1].[Fax] AS [Fax],
    1 AS [C1],
    [Extent2].[OrderID] AS [OrderID],
    [Extent2].[CustomerID] AS [CustomerID1],
    [Extent2].[EmployeeID] AS [EmployeeID],
    [Extent2].[OrderDate] AS [OrderDate],
    [Extent2].[RequiredDate] AS [RequiredDate],
    [Extent2].[ShippedDate] AS [ShippedDate],
    [Extent2].[ShipVia] AS [ShipVia],
    [Extent2].[Freight] AS [Freight],
    [Extent2].[ShipName] AS [ShipName],
    [Extent2].[ShipAddress] AS [ShipAddress],
    [Extent2].[ShipCity] AS [ShipCity],
    [Extent2].[ShipRegion] AS [ShipRegion],
    [Extent2].[ShipPostalCode] AS [ShipPostalCode],
    [Extent2].[ShipCountry] AS [ShipCountry],
    CASE WHEN ([Extent2].[OrderID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C2]
  FROM [dbo].[Customers] AS [Extent1]
  LEFT OUTER JOIN [dbo].[Orders] AS [Extent2] ON [Extent1].[CustomerID] = [Extent2].[CustomerID]
  WHERE N'UK' = [Extent1].[Country]
) AS [Project1]
ORDER BY [Project1].[CustomerID] ASC, [Project1].[C2] ASC

```

При использовании отложенной загрузки, изначально будет выполнен следующий запрос:

```

SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
FROM [dbo].[Customers] AS [Extent1]
WHERE N'UK' = [Extent1].[Country]

```

И каждый раз при обращении к свойству навигации заказы клиента в хранилище выдается другой запрос следующим образом:

```

exec sp_executesql N'SELECT
[Extent1].[OrderID] AS [OrderID],
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[EmployeeID] AS [EmployeeID],
[Extent1].[OrderDate] AS [OrderDate],
[Extent1].[RequiredDate] AS [RequiredDate],
[Extent1].[ShippedDate] AS [ShippedDate],
[Extent1].[ShipVia] AS [ShipVia],
[Extent1].[Freight] AS [Freight],
[Extent1].[ShipName] AS [ShipName],
[Extent1].[ShipAddress] AS [ShipAddress],
[Extent1].[ShipCity] AS [ShipCity],
[Extent1].[ShipRegion] AS [ShipRegion],
[Extent1].[ShipPostalCode] AS [ShipPostalCode],
[Extent1].[ShipCountry] AS [ShipCountry]
FROM [dbo].[Orders] AS [Extent1]
WHERE [Extent1].[CustomerID] = @EntityKeyValue1',N'@EntityKeyValue1 nchar(5)',@EntityKeyValue1=N'AROUT'

```

Дополнительные сведения см. в разделе [загрузка связанных объектов](#).

8.2.1 отложенную загрузку и упреждающая загрузка Памятка

Нет такого понятия, как универсальная выборе безотложную загрузку и отложенной загрузки. Сначала попытайтесь понять различия между обеими стратегиями, поэтому можно также обоснованное решение; Кроме того стоит, подходит ли ваш код в любой из следующих сценариев:

СЦЕНАРИЙ	НАШИ ПРЕДЛОЖЕНИЯ
Требуется для доступа к многие свойства навигации из выбранных объектов?	<p>Не — возможно, сделает оба варианта. Тем не менее если полезные данные, которые приносит запроса не является слишком большим, возможны выигрыш в производительности, используя Безотложную загрузку, так как он не требует сетевого обходов для материализации объектов.</p> <p>Да — Если вам требуется доступ к многие свойства навигации с сущностями, это делается, что с помощью нескольких включить операторы в запросе с безотлагательной загрузкой. Дополнительные сущности включаются, чем больше полезных данных, которые возвращает запрос. Когда вы включаете три или нескольких сущностей в запрос, подумайте о переходе в режим Lazy загрузки.</p>

СЦЕНАРИЙ	НАШИ ПРЕДЛОЖЕНИЯ
Знаете ли вы, точно какие данные потребуются во время выполнения?	<p>Не — отложенная загрузка будет лучше для вас. В противном случае может появиться запрос данных, что вам не потребуется.</p> <p>Да — упреждающая загрузка, возможно, является лучшим решением; будет проще ускорения загрузки всей наборов. Если запрос требует выборки очень большой объем данных, и это становится слишком медленно, попробуйте вместо загрузки Lazy.</p>
Ваш код выполняется далеко от базы данных? (задержка сети)	<p>Не — когда сетевая задержка не является проблемой, с помощью отложенная загрузка может упростить код. Помните, что топология приложения могут измениться, поэтому не принимают с учетом расположения базы данных как должное.</p> <p>Да — Если в сети только для проблем, можно решить, что лучше подходит для вашего сценария. Как правило, так как он требуется меньше циклов Безотложная загрузка будет лучше.</p>

8.2.2 включает в себя несколько, касающиеся производительности

Когда мы слышим вопросы производительности, включающие проблемы времени отклика сервера, источником проблемы является часто запросы с несколькими инструкциями `Include`. Во время, включая связанные сущности в запросе обладает широкими возможностями, важно понимать, что происходит на самом деле.

Занимает достаточно долго для запроса с несколькими инструкциями `Include` в его, чтобы перейти по нашей внутренней плана компилятора для создания команды хранилища. Большую часть этого времени расходуется оптимизации результирующий запрос. Команда созданного хранилища будет содержать `Outer Join` или `Union` для каждого `Include`, в зависимости от вашей сопоставления. Подобные запросы добавит больших графах подключенных из базы данных в одной полезной нагрузке, которая будет `acerbate` проблемы пропускной способности, особенно в том случае, если есть много избыточности в полезных данных (например, при использовании нескольких уровней `Include` для обхода ассоциации в направлении один ко многим).

Можно проверить для случаев, где запросы возвращается слишком большого объема полезных данных путем доступа к базовой TSQL для запроса с помощью `ToTraceString`, выполнив команду хранилища в SQL Server Management Studio, чтобы просмотреть размер полезных данных. В таких случаях вы можете повторить, чтобы уменьшить число операторов `Include` в запросе для только что перенесите нужные данные. Или можно разбить запрос на небольших последовательность вложенных запросов, например:

До следующей запроса:

```
using (NorthwindEntities context = new NorthwindEntities())
{
    var customers = from c in context.Customers.Include(c => c.Orders)
                    where c.LastName.StartsWith(lastNameParameter)
                    select c;

    foreach (Customer customer in customers)
    {
        ...
    }
}
```

После разбив его:

```

using (NorthwindEntities context = new NorthwindEntities())
{
    var orders = from o in context.Orders
                 where o.Customer.LastName.StartsWith(lastNameParameter)
                 select o;

    orders.Load();

    var customers = from c in context.Customers
                    where c.LastName.StartsWith(lastNameParameter)
                    select c;

    foreach (Customer customer in customers)
    {
        ...
    }
}

```

Это будет работать только в отслеживаемых запросах, как мы выполняем использование возможности контекст должен автоматически выполнить адресная привязка ассоциация и разрешение идентификаторов.

Как и в случае отложенной загрузки компромисс будут дополнительные запросы для меньшие порции данных. Проекции отдельные свойства также можно использовать явно установите только необходимые данные из каждой сущности, но не загружается сущностей в этом случае обновление не поддерживается.

8.2.3 рекомендации для получения отложенная загрузка свойств

В настоящее время платформа Entity Framework не поддерживает отложенную загрузку скалярного или сложного свойства. Однако в случаях, где у вас есть таблицу, содержащую больших объектов, таких как большой двоичный объект, можно использовать разделение таблицы для разделения больших свойства в отдельную сущность. Например предположим, что имеется таблица продукции со столбцом photo varbinary. Если вам не нужно часто доступ к этому свойству в запросах, можно использовать для перевода в частях сущность, которая обычно необходимо разбиение таблиц. Сущность, представляющая фотография продукта будет загружаться только в тех случаях, когда она явно нужна.

Хороший ресурс, в котором показано, как включить разделение таблицы является Джил Финк «Таблица разделения в Entity Framework» в блоге: <<http://blogs.microsoft.co.il/blogs/gilf/archive/2009/10/13/table-splitting-in-entity-framework.aspx>>.

9. другие ВОПРОСЫ

9.1 сборка мусора сервера

Некоторые пользователи могут испытывать конфликта ресурсов, который ограничивает параллелизм, они ожидают, когда сборщик мусора не настроен должным образом. Каждый раз, когда EF используется в сценарии или в любом приложении, аналогичный системы на сервере, не забудьте включить сборка мусора сервера. Это делается с помощью простой параметр в файле конфигурации приложения:

```

<?xmlversion="1.0" encoding="utf-8" ?>
<configuration>
    <runtime>
        <gcServer enabled="true" />
    </runtime>
</configuration>

```

Это следует уменьшить конфликты вашей потоков и повысить пропускную способность, до 30% в сценариях пропускная способность ЦП. В целом вы должны всегда тестировать поведение приложения с помощью классической сборки мусора (настроенный лучше для сценариев на стороне пользовательского интерфейса и клиента) а также сборка мусора сервера.

9.2 AutoDetectChanges

Как упоминалось ранее, Entity Framework может показывать проблем с производительностью, когда кэш объектов большим количеством сущностей. Некоторые операции, например добавления, удаления, поиска, запись и SaveChanges, активировать вызовы к DetectChanges, который может использовать большой объем ЦП, в зависимости от размера кэша объектов стал. Причиной этого является то, что кэш объектов и повторите диспетчер состояния объекта оставаться как синхронизированным, при каждой операции, выполняемые к контексту, таким образом, производимых данных гарантированно будет правильно под широкий набор сценариев.

Обычно рекомендуется оставить Entity Framework автоматическое обнаружение изменений для всего жизненного цикла приложения. Если вашего сценария отрицательно влияет высокой загрузки ЦП и профили указывают, что выяснилось, это вызов DetectChanges, рассмотрите возможность временного отключения AutoDetectChanges в важные части кода:

```
try
{
    context.Configuration.AutoDetectChangesEnabled = false;
    var product = context.Products.Find(productId);
    ...
}
finally
{
    context.Configuration.AutoDetectChangesEnabled = true;
}
```

Прежде чем отключать AutoDetectChanges, важно понимать, что это может привести к платформе Entity Framework теряет способность отслеживать определенные сведения об изменениях, которые выполняются в сущности. Если обрабатывается неправильно, это может привести к несогласованности данных на приложение. Дополнительные сведения об отключении AutoDetectChanges см. в статье <<http://blog.oneunicorn.com/2012/03/12/secrets-of-detectchanges-part-3-switching-off-automatic-detectchanges/>>.

9.3 контекст каждого запроса

Контексты Entity Framework предназначены для использования в качестве небольшим временем существования экземпляров, чтобы обеспечить максимальную производительность работы. Контексты должны быть коротким и выполнявшейся удаляются и таким образом были реализованы очень простая и reutilize метаданных, когда это возможно. В сценариях веб-важно иметь в виду и обладает контекстом дольше, чем длительность одного запроса. Аналогичным образом в сценариях веб-контекста следует удалить зависимости от понимания различные уровни кэширования в Entity Framework. Вообще говоря один следует избегать наличия экземпляра контекста на протяжении всего жизненного цикла приложения, а также контексты каждого потока и статические контекстов.

9.4 null-семантика базы данных

Entity Framework по умолчанию будет генерировать код SQL, который имеет C# значение null, семантика сравнения. Рассмотрим следующий запрос:

```

int? categoryId = 7;
int? supplierId = 8;
decimal? unitPrice = 0;
short? unitsInStock = 100;
short? unitsOnOrder = 20;
short? reorderLevel = null;

var q = from p in context.Products
        where p.Category.CategoryName == "Beverages"
            || (p.CategoryID == categoryId
                || p.SupplierID == supplierId
                || p.UnitPrice == unitPrice
                || p.UnitsInStock == unitsInStock
                || p.UnitsOnOrder == unitsOnOrder
                || p.ReorderLevel == reorderLevel)
        select p;

var r = q.ToList();

```

В этом примере мы сравнивает ряд переменных, допускающие значение NULL по обнуляемые свойства сущности, такие как SupplierID и «цена». Созданный код SQL для этого запроса запросит в том случае, если значение параметра является таким же, как значение столбца, или если оба параметра, так и значения столбцов имеют значение null. Это будет скрыт способ сервера базы данных обрабатывает значения NULL и будет предоставлять согласованные C# null взаимодействие через различные поставщики баз данных. С другой стороны, созданный код немного запутанным и не может выполнять when хорошо количество сравнений в where инструкции запроса увеличивается на большом числе.

Один из способов решения данной проблемы — с помощью семантику null базы данных. Обратите внимание, что это потенциально может работать по-разному для C# null семантику, так как теперь Entity Framework будет генерировать более простой код SQL, который предоставляет способ СУБД обрабатывает значения null. Семантика null базы данных может быть активированы на контекста с помощью одной строки одной конфигурации для контекста конфигурации:

```
context.Configuration.UseDatabaseNullSemantics = true;
```

Предприятиям среднего размера запросы не отобразит заметного производительность при использовании семантику null базы данных, но разница станут заметными на запросы с большим числом потенциальных сравнений со значением null.

В приведенном выше примере запроса разница в производительности был меньше, чем 2% microbenchmark, запущенных в управляемой среде.

9.5 Async

Поддержка Entity Framework 6 появились асинхронных операций, при выполнении в .NET 4.5 или более поздней версии. По большей части, приложений, имеющих операций ввода-ВЫВОДА, связанных с конфликтов будут наиболее выгодны с помощью асинхронного запроса и операции сохранения. Если приложение не страдал состязание ввода-ВЫВОДА, использование async в лучшем случае синхронного выполнения и возвращают результат, в то же количество времени в виде синхронного вызова или в худшем случае, просто отложить выполнение асинхронной задачи и добавьте дополнительный Тим е для выполнения вашего сценария.

Сведения о асинхронных задач программирования, которые помогут вам решить, если асинхронный улучшит производительность приложения посетите <http://msdn.microsoft.com/library/hh191443.aspx>. Дополнительные сведения об использовании асинхронных операций в Entity Framework, см. в разделе [асинхронный запрос и сохранить](#).

9.6 NGEN

Entity Framework 6 не поставляется в установке по умолчанию платформы .NET framework. Таким образом сборки .NET Framework сущности не являются NGEN будет по умолчанию, это означает, что весь код Entity Framework регулируется же затраты JIT'ing как любые другие сборки MSIL. Это может привести к снижению качества F5 во время разработки, а также холодного запуска приложения в рабочей среде. Чтобы снизить затраты на ЦП и памяти JIT'ing рекомендуется NGEN, Entity Framework образы соответствующим образом. Дополнительные сведения о том, как повысить производительность при запуске Entity Framework 6 с помощью NGEN см. в разделе [повышение производительности запуска с помощью NGen](#).

9.7 code First и EDMX

Entity Framework достигнуты проблему потери соответствия между объектно-ориентированного программирования и реляционных баз данных благодаря наличию хранимое в памяти представление концептуальной модели (объекты), схемы хранения (база данных) и сопоставление между двумя. Эти метаданные называется Entity Data Model – EDM для краткости. Из этой модели EDM Entity Framework являются производными представления для данных обмена данными объектов в памяти в базу данных и обратно.

При использовании Entity Framework с EDMX-файла, формально указывает концептуальной модели, схемы хранения и сопоставление, а затем стадию загрузки модели имеет только для проверки правильности модели EDM (например, убедитесь, что нет сопоставлений отсутствуют), затем Создание представлений, проверки представлений и иметь эти метаданные, готовой к использованию. Только может затем запрос можно выполнять, или сохранить новые данные в хранилище данных.

Подход Code First является, по существу, сложных генератор модели EDM. Имеет Entity Framework для создания EDM из предоставленного кода; Это достигается путем анализа классов, используемых в модели, применение соглашений и Настройка модели с помощью Fluent API. После построения модели EDM Entity Framework по сути работает так же образом, как он бы присутствовали в проект EDMX-файла. Таким образом при построении модели из Code First добавляет дополнительные осложнения, который преобразует в медленнее, время запуска для Entity Framework по сравнению с наличие EDMX. Стоимость зависит от размера и сложности модели, который строится.

Если вы решили использовать EDMX и Code First, важно знать, что гибкость, представленное в Code First увеличивается стоимость создания модели в первый раз. Если приложение может выдержать затраты на такую загрузку первого то обычно Code First будет предпочтительным способом для перехода.

10 Исследование производительности

10.1 с помощью Visual Studio Profiler

Если возникают проблемы с производительностью с помощью Entity Framework, можно использовать профилировщик, аналогичный приведенному, встроенных в Visual Studio для см. в разделе, где приложение тратит свое время. Это средство, мы использовали для создания круговых диаграмм в записи блога «Обзор производительности платформы ADO.NET Entity Framework — часть 1» (<http://blogs.msdn.com/b/adonet/archive/2008/02/04/exploring-the-performance-of-the-ado-net-entity-framework-part-1.aspx>) , показывающие, где Entity Framework тратит свое время во время выполнения запросов "холодных" и "горячего" резервирования.

В записи блога «Профилирования Entity Framework с помощью Visual Studio Profiler 2010», написанной данных и моделирования группы консультирования клиентов представляет реальных пример того, как их использовать профилировщик для изучения проблемы производительности. (<http://blogs.msdn.com/b/dmcat/archive/2010/04/30/profiling-entity-framework-using-the-visual-studio-2010-profiler.aspx>). Эта запись была написана для приложения windows. Если необходимо профилировать веб-приложение средства записи производительности Windows (WPR) и анализатор производительности Windows (WPA) может работать лучше, чем работе из Visual Studio. WPR и WPA являются частью Windows Performance Toolkit, который входит в состав комплекта средств для развертывания и оценки Windows (<http://www.microsoft.com/en-US/download/details.aspx?id=39982>).

10.2 приложения или базы данных профилирования

Такие средства, как профилировщик, встроенных в Visual Studio сообщит о том, где приложение тратит время. Другой тип профилировщик доступен, выполняет динамический анализ для запущенного приложения, в рабочей среде или подготовительной среде, в зависимости от потребностей и ищет общие проблемы и антишаблоны доступа к базе данных.

Два имеющихся на рынке профилировщика являются Entity Framework Profiler (<http://efprof.com>) и ORMProfiler (<http://ormprofiler.com>).

Если приложение является приложение MVC с использованием Code First, можно использовать в StackExchange MiniProfiler. Скотт Хансельман описывает этот инструмент в его блог по адресу: <http://www.hanselman.com/blog/NuGetPackageOfTheWeek9ASPNETMiniProfilerFromStackExchangeRocksYourWorld.aspx>.

Дополнительные сведения о профилирование операций базы данных приложения, см. в статье журнала MSDN Magazine Джули Лерман под названием [профилирование операций с базой данных в Entity Framework](#).

10.3 средство ведения журнала базы данных

Если вы используете Entity Framework 6 также рассмотреть возможность использования функции встроенного ведения журнала. Можно также указать свойства контекста базы данных для входа своих действий через простую конфигурацию одной строки:

```
using (var context = newQueryComparison.DbC.NorthwindEntities())
{
    context.Database.Log = Console.WriteLine;
    var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
    q.ToList();
}
```

В этом примере активность базы данных будет записываться в консоль, но свойство журнала можно настроить для вызова любого действия<строка> делегировать.

Если вы хотите включить ведение журнала базы данных без повторной компиляции и вы используете Entity Framework 6.1 или более поздней версии, это можно сделать, добавив при этом перехватчик в файле web.config или app.config вашего приложения.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Path\To\My\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

Дополнительные сведения о том, как добавление ведения журнала без повторной компиляции перейдите в раздел <http://blog.oneunicorn.com/2014/02/09/ef-6-1-turning-on-logging-without-recompiling/>.

Приложение 11

11.1 А. тестовой среды

Эта среда использует настройки машины 2 с базой данных на отдельном компьютере из клиентского приложения. Машины находятся в одной стойке, поэтому задержки в сети является сравнительно мало, но более реалистичный чем среде одного компьютера.

11.1.1 сервер приложений

11.1.1.1 программной среде

- Программная среда Entity Framework 4
 - Имя ОС: Windows Server 2008 R2 Enterprise с пакетом обновления 1.
 - Visual Studio 2010 Ultimate.
 - Visual Studio 2010 с пакетом обновления 1 (только для некоторых сравнения).
- Программная среда Entity Framework 5 и 6
 - Имя ОС: Windows 8.1 Enterprise
 - Visual Studio 2013 — Ultimate.

11.1.1.2 аппаратную среду

- Двухъядерный процессор: W3530 L5520 ЦП Intel(R) Xeon(R) с частотой 2,27 ГГц, 2261 Mhz8 ГГц, 4 ядер, 84 логических процессоров.
- RamRAM 2412 ГБ.
- 136 ГБ SCSI250GB SATA 7200 об/мин 3 ГБИТ/с диска разделить на 4 разделов.

11.1.2 сервера базы данных

11.1.2.1 программной среде

- Имя ОС: Windows Server 2008 R2.8.1 Enterprise с пакетом обновления 1.
- R22012 SQL Server 2008.

11.1.2.2 аппаратную среду

- Один процессор: L5520 ЦП Intel(R) Xeon(R) с частотой 2,27 ГГц, 2261 MhzES-1620 0 @ 3,60 ГГц, 4 ядер, 8 логических процессоров.
- RamRAM 824 ГБ.
- 465 ГБ ATA500GB SATA 7200 об/мин 6 ГБИТ/с диска разделить на 4 разделов.

11.2 проверяет запрос б. Сравнение производительности

Модель Northwind был использован для выполнения этих тестов. Он был создан из базы данных с помощью конструктора Entity Framework. Затем следующий код был использован для сравнения производительности параметры выполнения запроса:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Common;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;
using System.Data.Objects;
using System.Linq;

namespace QueryComparison
{
    public partial class NorthwindEntities : ObjectContext
    {
        private static readonly Func<NorthwindEntities, string, IQueryable<Product>> productsForCategoryCQ =
            CompiledQuery.Compile(
                (NorthwindEntities context, string categoryName) =>
                    context.Products.Where(p => p.Category.CategoryName == categoryName)
            );

        public IQueryable<Product> InvokeProductsForCategoryCQ(string categoryName)
        {
            return productsForCategoryCQ(this, categoryName);
        }
    }

    public class QueryTypePerfComparison
    {
        private static string entityConnectionString =
            @"metadata=res://*/Northwind.csdl|res://*/Northwind.ssdl|res://*/Northwind.msl;provider=System.Data.SqlClient;
provider connection string='data source=.;initial catalog=Northwind;integrated
security=True;MultipleActiveResultSets=True;App=EntityDataSource1";
    }
}
```

```
security=false;multipleActiveResults=true;App=EntityFramework ;
```

```
public void LINQIncludingContextCreation()
{
    using (NorthwindEntities context = new NorthwindEntities())
    {
        var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
        q.ToList();
    }
}

public void LINQNoTracking()
{
    using (NorthwindEntities context = new NorthwindEntities())
    {
        context.Products.MergeOption = MergeOption.NoTracking;

        var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
        q.ToList();
    }
}

public void CompiledQuery()
{
    using (NorthwindEntities context = new NorthwindEntities())
    {
        var q = context.InvokeProductsForCategoryCQ("Beverages");
        q.ToList();
    }
}

public void ObjectQuery()
{
    using (NorthwindEntities context = new NorthwindEntities())
    {
        ObjectQuery<Product> products = context.Products.Where("it.Category.CategoryName =
'Beverages'");
        products.ToList();
    }
}

public void EntityCommand()
{
    using (EntityConnection eConn = new EntityConnection(entityConnectionString))
    {
        eConn.Open();
        EntityCommand cmd = eConn.CreateCommand();
        cmd.CommandText = "Select p From NorthwindEntities.Products As p Where p.Category.CategoryName
= 'Beverages'";

        using (EntityDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
        {
            List<Product> productsList = new List<Product>();
            while (reader.Read())
            {
                DbDataRecord record = (DbDataRecord)reader.GetValue(0);

                // 'materialize' the product by accessing each field and value. Because we are
materializing products, we won't have any nested data readers or records.
                int fieldCount = record.FieldCount;

                // Treat all products as Product, even if they are the subtype DiscontinuedProduct.
                Product product = new Product();

                product.ProductID = record.GetInt32(0);
                product.ProductName = record.GetString(1);
                product.SupplierID = record.GetInt32(2);
                product.CategoryID = record.GetInt32(3);
                product.QuantityPerUnit = record.GetString(4);
            }
        }
    }
}
```

```

                product.UnitPrice = record.GetDecimal(5);
                product.UnitsInStock = record.GetInt16(6);
                product.UnitsOnOrder = record.GetInt16(7);
                product.ReorderLevel = record.GetInt16(8);
                product.Discontinued = record.GetBoolean(9);

                productsList.Add(product);
            }
        }
    }

    public void ExecuteStoreQuery()
    {
        using (NorthwindEntities context = new NorthwindEntities())
        {
            ObjectResult<Product> beverages = context.ExecuteStoreQuery<Product>(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
    FROM        Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE       (C.CategoryName = 'Beverages')"
);
            beverages.ToList();
        }
    }

    public void ExecuteStoreQueryDbContext()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var beverages = context.Database.SqlQuery<QueryComparison.DbC.Product>(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
    FROM        Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE       (C.CategoryName = 'Beverages')"
);
            beverages.ToList();
        }
    }

    public void ExecuteStoreQueryDbSet()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var beverages = context.Products.SqlQuery(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
    FROM        Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE       (C.CategoryName = 'Beverages')"
);
            beverages.ToList();
        }
    }

    public void LINQIncludingContextCreationDbContext()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
            q.ToList();
        }
    }

    public void LINQNoTrackingDbContext()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var q = context.Products.AsNoTracking().Where(p => p.Category.CategoryName == "Beverages");
            q.ToList();
        }
    }
}

```

```
        }
    }
}
```

Модель Navision 11.3 C.

База данных Navision находится большой базы данных, используемый для демонстрации Microsoft Dynamics – NAV. Создаваемых концептуальных моделях содержит наборы сущностей 1005 и 4227 наборы ассоциаций. Модель, используемая в teste «плоский» — не поддерживают наследование был добавлен к нему.

11.3.1 запросы, используемые для тестов Navision

Список запросов, используемый с моделью Navision содержит 3 категорий запросов Entity SQL:

11.3.1.1 поиск

Простые поисковые запросы с нет агрегатов

- Число: 16232
- Пример

```
<Query complexity="Lookup">
  <CommandText>Select value distinct top(4) e.Idle_Time From NavisionFKContext.Session as e</CommandText>
</Query>
```

11.3.1.2 SingleAggregating

Обычный запрос бизнес-Аналитики с помощью нескольких статистических выражений, но не подытоги (одного запроса)

- Число: 2313
- Пример

```
<Query complexity="SingleAggregating">
  <CommandText>NavisionFK.MDF_SessionLogin_Time_Max()</CommandText>
</Query>
```

Где MDF_SessionLogin_время_Max() определен в модели в виде:

```
<Function Name="MDF_SessionLogin_Time_Max" ReturnType="Collection(DateTime)">
  <DefiningExpression>SELECT VALUE Edm.Min(E.Login_Time) FROM NavisionFKContext.Session as E</DefiningExpression>
</Function>
```

11.3.1.3 AggregatingSubtotals

Запрос бизнес-Аналитики с помощью агрегатов и промежуточные итоги (посредством объединения всех)

- Число: 178
- Пример

```

<Query complexity="AggregatingSubtotals">
    <CommandText>
using NavisionFK;
function AmountConsumed(entities Collection([CRONUS_International_Ltd__Zone])) as
(
    Edm.Sum(select value N.Block_Movement FROM entities as E, E.CRONUS_International_Ltd__Bin as N)
)
function AmountConsumed(P1 Edm.Int32) as
(
    AmountConsumed(select value e from NavisionFKContext.CRONUS_International_Ltd__Zone as e where
e.Zone_Ranking = P1)
)
-----
-----
(
    select top(10) Zone_Ranking, Cross_Dock_Bin_Zone, AmountConsumed(GroupPartition(E))
    from NavisionFKContext.CRONUS_International_Ltd__Zone as E
    where AmountConsumed(E.Zone_Ranking) > @MinAmountConsumed
    group by E.Zone_Ranking, E.Cross_Dock_Bin_Zone
)
union all
(
    select top(10) Zone_Ranking, Cast(null as Edm.Byte) as P2, AmountConsumed(GroupPartition(E))
    from NavisionFKContext.CRONUS_International_Ltd__Zone as E
    where AmountConsumed(E.Zone_Ranking) > @MinAmountConsumed
    group by E.Zone_Ranking
)
union all
{
    Row(Cast(null as Edm.Int32) as P1, Cast(null as Edm.Byte) as P2, AmountConsumed(select value E
                                                from
NavisionFKContext.CRONUS_International_Ltd__Zone as E
                                                where AmountConsumed(E.Zone_Ranking)
> @MinAmountConsumed))
}</CommandText>
<Parameters>
    <Parameter Name="MinAmountConsumed" DbType="Int32" Value="10000" />
</Parameters>
</Query>

```

Быстрый запуск с помощью NGen

13.09.2018 • 9 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Платформа .NET Framework поддерживает создание образов в машинном коде для управляемых приложений и библиотек, как способ защиты приложения запускаются быстрее, а также в некоторых случаях использовать меньше памяти. Образы в машинном коде создаются путем перевода сборки с управляемым кодом в файлы, содержащие инструкции машинного перед выполнением приложения, освобождая разработчика компилятора .NET JIT (Just-In-Time) от необходимости создания собственного инструкциям, приведенным в Среда выполнения приложения.

До версии 6 библиотеки среды выполнения EF core были частью .NET Framework и образы в машинном коде были автоматически созданы для них. Начиная с версии 6 всей среды выполнения EF объединены в пакет EntityFramework NuGet. Образы в машинном коде для теперь быть созданы с помощью средства командной строки NGen.exe для получения результатов похожих.

Эмпирическим наблюдениям показывают, что образы в машинном коде для сборки среды выполнения EF можно вырезать от 1 до 3 секунд времени запуска приложения.

Как использовать NGen.exe

Основная функция средством NGen.exe — «установка» (то есть для создания и сохранения на диск) образов в машинном коде для сборки и всех его прямых зависимостей. Вот, как это можно сделать:

1. Откройте окно командной строки с правами администратора
2. Измените расположение сборок, которые требуется создать образы в машинном коде для текущего рабочего каталога:

```
cd <*Assemblies location*>
```

3. В зависимости от операционной системы и конфигурации для приложения может потребоваться для создания образов в машинном коде для 32-разрядной архитектуры, 64-разрядной архитектуры или одновременно обеих платформ.

Для запуска 32-разрядная версия:

```
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\ngen install <Assembly name>
```

Для 64-разрядных запуска:For 64 bit run:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\ngen install <Assembly name>
```

TIP

Создание образов в машинном коде для неправильной архитектурой является распространенной ошибкой. Если сомневаетесь, можно просто создать образы в машинном коде для всех архитектур, которые применяются к операционной системы, установленной на компьютере.

NGen.exe также поддерживает другие функции, такие как удаление и Отображение установленных образов в машинном коде, очереди создания нескольких образов и т. д. Дополнительные сведения об использовании в статье [документации NGen.exe](#).

Когда следует использовать NGen.exe

Когда дело доходит до решить, какие сборки для создания образов в машинном коде для в приложении, в зависимости от EF 6 или более поздней версии, учитывайте следующие параметры.

- **Основной сборки среды выполнения EF, EntityFramework.dll:** типичного приложения на основе EF выполняет значительный объем кода из этой сборки, при запуске или при его первом доступе к базе данных. Следовательно создание образов в машинном коде для этой сборки приведет крупнейших выигрыш в производительности при запуске.
- **Любой сборки поставщика EF, используемой приложением:** время запуска также могут немного выиграть от генерации машинных образов из них. Например если приложение использует поставщика EF для SQL Server требуется создать образ в машинном коде для EntityFramework.SqlServer.dll.
- **Сборок приложения и другие зависимости:** [документации NGen.exe](#) рассматриваются общие критерии для выбора сборки для создания образов в машинном коде и влияние образов в машинном коде на безопасность, Дополнительные параметры, такие как «жесткая привязка», сценарии, как использование образов в машинном коде в отладки и профилирования, сценарии и т. д.

TIP

Убедитесь в том, что вы тщательно оценить влияние использования образов в машинном коде на производительность при запуске и общую производительность приложения и сравните их с фактические требования. Образы в машинном коде обычно помогает повысить запуска показатель производительности и в некоторых случаях снизить использование памяти, столь же отразится не для всех сценариев. К примеру в устойчивом состоянии выполнения (то есть после хотя бы один раз после вызова всех методов, которые используются приложением) код, сгенерированный JIT-компилятор может фактически дать немного лучшую производительность, чем образы в машинном коде.

С помощью NGen.exe на машине разработки

Во время разработки .NET JIT компилятора предложит установить общую наилучшего для кода, который часто изменяется. Создание образов в машинном коде для скомпилированного зависимости, такие как сборки среды выполнения EF может помочь ускорить процесс разработки и тестирования, вырезание несколько секунд в начале каждого выполнения.

Отлично подходит для поиска сборки среды выполнения EF является расположение пакета NuGet для решения. Например, для приложения с помощью EF 6.0.2 с SQL Server и предназначенных для .NET 4.5 или более поздней можно ввести следующее в окне командной строки (не забудьте открыть его в качестве администратора):

```
cd <Solution directory>\packages\EntityFramework.6.0.2\lib\net45  
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\ngen install EntityFramework.SqlServer.dll  
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\ngen install EntityFramework.SqlServer.dll
```

NOTE

Это позволяет пользоваться преимуществами тот факт, что установка образы в машинном коде для поставщика EF для SQL Server также по умолчанию установит образы в машинном коде для основной сборки среды выполнения EF. Это работает, поскольку NGen.exe можно определить, что EntityFramework.dll прямых зависимостей сборки EntityFramework.SqlServer.dll, расположенной в том же каталоге.

Создание образов в машинном коде во время установки

Набор средств WiX поддерживает очереди создания образов в машинном коде для управляемых сборок во время установки, описанные в этой [практическом руководстве](#). Другой вариант — Создать задачу пользовательской установки, воспользуйтесь командой NGen.exe.

Проверка того, что образы в машинном коде используются для Entity FRAMEWORK

Убедитесь, что определенное приложение использует машинные сборки для загружаемых сборок, которые имеют расширение «.ni.dll» «или» «.ni.exe». Например образ в машинном коде для сборки основной среды выполнения EF будет вызываться EntityFramework.ni.dll. Простой способ проверки загруженных сборок .NET процесса является использование [Process Explorer](#).

Следует учитывать следующие моменты

Создание образа в машинном коде сборки не следует путать с регистрацией сборки в GAC (глобальный кэш сборок). NGen.exe позволяет создавать образы сборок, которые не находятся в глобальном кэше СБОРОК, а на самом деле, несколько приложений, использующих определенную версию EF могут совместно использовать же образа в машинном коде. Хотя Windows 8 может автоматически создавать образы в машинном коде для сборок, помещенных в глобальный кэш СБОРОК, исполняющая среда EF оптимизирован для развертывания вместе с приложения, и мы не рекомендуем его регистрация в глобальном кэше СБОРОК, как это негативно влияет на разрешения сборки и обслуживание приложений среди других аспектах.

Сопоставление заранее созданные представления

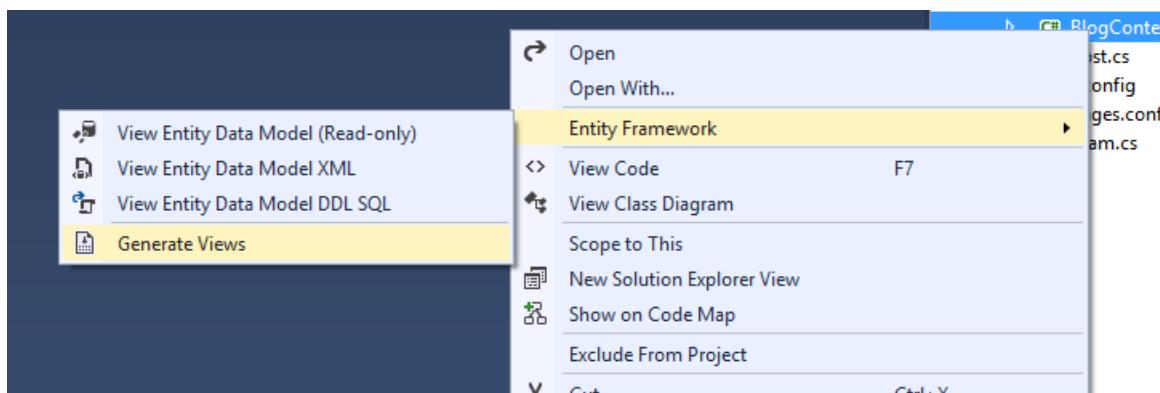
01.10.2018 • 7 minutes to read • [Edit Online](#)

Прежде чем платформа Entity Framework можно выполнить запрос или сохранить изменения в источнике данных, ей необходимо создать набор сопоставления представлений для доступа к базе данных. Эти представления сопоставления представляют собой набор оператор Entity SQL, отображают состояние базы данных отвлеченно и являются частью метаданных, кэшируемых для каждого домена приложения. При создании нескольких экземпляров того же контекста, в том же домене приложения, они будут повторно использовать сопоставление представления из кэшированных метаданных, а не формируются заново. Сопоставление создание представлений составляет значительную долю общей стоимости выполнения первого запроса, Entity Framework позволяет заранее создавать представления сопоставления и включать их в скомпилированный проект. Дополнительные сведения см. в разделе [вопросы производительности \(Entity Framework\)](#).

Создание сопоставления представлений с помощью EF Power Tools Community Edition

Чтобы заранее создать представления проще всего использовать [EF Power Tools Community Edition](#). Если у вас установлены инструменты Power имеется пункт меню, чтобы создать представления, как показано ниже.

- Для **Code First** моделей, щелкните правой кнопкой мыши на файл кода, который содержит ваш класс DbContext.
- Для **конструктор EF** моделей, щелкните правой кнопкой мыши на файле EDMX.



После завершения процесса вы получите следующую созданный класс

A screenshot of the Solution Explorer in Visual Studio. It shows a project named 'BlogApp' with files like 'Blog.cs', 'BlogContext.cs', and 'BlogContext.Views.cs'. The 'BlogContext.Views.cs' file is selected. In the main code editor window, the following C# code is displayed:

```
1 //-----  
2 // <auto-generated>  
3 //   This code was generated by a tool.  
4 //  
5 //   Changes to this file may cause incorrect behavior and will be lost if  
6 //   the code is regenerated.  
7 // </auto-generated>  
8 //-----  
9  
10 using System.Data.Entity.Infrastructure.MappingViews;  
11  
12 [assembly: DbMappingViewCacheTypeAttribute(  
13     typeof(BlogApp.Models.BlogContext),  
14     typeof(Edm_EntityMappingGeneratedViews.ViewsForBaseEntitySetsa0b843f03dd29abee99789e190a6fb70ce8e93dc97945d437d9a58fb8e2af2e))]  
15  
16 namespace Edm_EntityMappingGeneratedViews  
17 {  
18     using System;  
19     using System.CodeDom.Compiler;  
20     using System.Data.Entity.Core.Metadata.Edm;  
21  
22     /// <summary>  
23     /// Implements a mapping view cache.  
24     /// </summary>  
25     [GeneratedCode("Entity Framework Power Tools", "0.9.0.0")]  
26     internal sealed class ViewsForBaseEntitySetsa0b843f03dd29abee99789e190a6fb70ce8e93dc97945d437d9a58fb8e2af2e : DbMappingViewCache  
27     {  
28         /// <summary>  
29         /// Gets a hash value computed over the mapping closure.  
30         /// </summary>  
31     }  
32 }
```

Теперь при запуске приложения EF будет использовать этот класс для загрузки представления при необходимости. Если изменения модели и вы не повторно создаете этот класс EF вызовет исключение.

Создание представления сопоставления из кода - EF6 и выше

Еще одним способом создавать представления является использование API-интерфейсов с EF. При использовании этого метода, вы можете свободно для сериализации в представлениях, однако вам удобно, но необходимо также загрузить представления самостоятельно.

NOTE

EF6 и выше только -API, как показано в этом разделе были представлены в Entity Framework 6. При использовании более ранней версии эти сведения не применяется.

Создание представлений

API-интерфейсы для создания представлений, находятся в классе System.Data.Entity.Core.Mapping.StorageMappingItemCollection. StorageMappingCollection для контекста, можно получить с помощью MetadataWorkspace ObjectContext. Если вы используете более новый API DbContext, а затем можно получить доступ к этому с помощью IObjectContextAdapter, например ниже, в этом коде у нас есть экземпляр вашем производном DbContext называется dbContext:

```
var objectContext = ((IObjectContextAdapter) dbContext).ObjectContext;
var mappingCollection = (StorageMappingItemCollection) objectContext.MetadataWorkspace
    .GetItemCollection(DataSpace.CSSpace);
```

При наличии StorageMappingItemCollection можно получить доступ к методам GenerateViews и ComputeMappingHashValue.

```
public Dictionary<EntitySetBase, DbMappingView> GenerateViews(IList<EdmSchemaError> errors)
public string ComputeMappingHashValue()
```

Первый метод создает словарь с записью для каждого представления в сопоставлении контейнеров. Второй метод вычисляет значение хэша для сопоставления одного контейнера и используется во время выполнения для проверки что модель не был изменен, так как представления были предварительно созданных. Для сложных сценариев, включающих несколько сопоставлений контейнера предоставляются переопределения двух методов.

При создании представления будет вызывать метод GenerateViews и затем записать итоговый EntitySetBase и DbMappingView. Также необходимо будет хранить хэша, созданного с помощью метода ComputeMappingHashValue.

Загрузка представления

Чтобы загрузить представления, сформированные с помощью метода GenerateViews, вы можете предоставить EF класс, наследующий от абстрактного класса DbMappingViewCache. DbMappingViewCache определяет два метода, которые необходимо реализовать:

```
public abstract string MappingHashValue { get; }
public abstract DbMappingView GetView(EntitySetBase extent);
```

Свойство MappingHashValue должно возвращать хэша, созданного с помощью метода ComputeMappingHashValue. Когда EF, которая будет служить для представления сначала создаст и сравнивать хэш-значение модели с хешем, возвращаемый этим свойством. Если они не совпадают EF

вызовет исключение EntityCommandCompilationException.

Метод GetView примет EntitySetBase и требуется возвращать что DbMappingView, содержащий язык EntitySql, который был создан для этого был связан с заданной EntitySetBase в словаре, созданной методом GenerateViews. Если EF запрашивает представление, что у вас нет затем GetView должен возвращать значение null.

Ниже приведен фрагмент DbMappingViewCache, который создается с помощью Power Tools, как описано выше, в нем мы видим, один из способов для хранения и извлечения языка EntitySql, которые требуется.

```
public override string MappingHashValue
{
    get { return "a0b843f03dd29abee99789e190a6fb70ce8e93dc97945d437d9a58fb8e2af2e"; }
}

public override DbMappingView GetView(EntitySetBase extent)
{
    if (extent == null)
    {
        throw new ArgumentNullException("extent");
    }

    var extentName = extent.EntityContainer.Name + "." + extent.Name;

    if (extentName == "BlogContext.Blogs")
    {
        return GetView2();
    }

    if (extentName == "BlogContext.Posts")
    {
        return GetView3();
    }

    return null;
}

private static DbMappingView GetView2()
{
    return new DbMappingView(@"
        SELECT VALUE -- Constructing Blogs
        [BlogApp.Models.Blog](T1.Blog_BlogId, T1.Blog_Test, T1.Blog_title, T1.Blog_Active,
        T1.Blog_SomeDecimal)
        FROM (
        SELECT
            T.BlogId AS Blog_BlogId,
            T.Test AS Blog_Test,
            T.title AS Blog_title,
            T.Active AS Blog_Active,
            T.SomeDecimal AS Blog_SomeDecimal,
            True AS _from0
        FROM CodeFirstDatabase.Blog AS T
        ) AS T1");
}
```

Чтобы использовать EF вашей DbMappingViewCache, добавлении используйте DbMappingViewCacheTypeAttribute, указав контекст, в котором он был создан для. В приведенном ниже коде мы связываем с классом MyMappingViewCache BlogContext.

```
[assembly: DbMappingViewCacheType(typeof(BlogContext), typeof(MyMappingViewCache))]
```

Для более сложных сценариев можно указать экземпляров кэша представление сопоставления, задайте

фабрику кэша представление сопоставления. Это можно сделать путем реализации абстрактного класса System.Data.Entity.Infrastructure.MappingViews.DbMappingViewCacheFactory. Экземпляр фабрики кэша представление сопоставления, используемый можно извлечь или задать с помощью StorageMappingItemCollection.MappingViewCacheFactoryproperty.

Поставщики Entity Framework 6

13.09.2018 • 7 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Сейчас Entity Framework разрабатывается в рамках лицензии на ПО с открытым исходным кодом, поэтому EF6 и более поздние версии не будут поставляться в составе платформы .NET Framework. Такая ситуация дает множество преимуществ, но также требует, чтобы поставщики EF были перестроены в соответствии со сборками EF6. Это означает, что поставщики EF для EF5 и предыдущих версий будут работать с EF6 только после их перестройки.

Какие поставщики доступны для EF6?

В число известных перестроенных для EF6 поставщиков входят следующие.

- **Поставщик Microsoft SQL Server**
 - Создан на [базе открытого исходного кода Entity Framework](#).
 - Поставляется в составе [пакета NuGet для EntityFramework](#).
- **Поставщик Microsoft SQL Server Compact Edition**
 - Создан на [базе открытого исходного кода Entity Framework](#).
 - Поставляется в [пакете NuGet EntityFramework.SqlServerCompact](#).
- **Поставщики данных Devart dotConnect**
 - Существуют сторонние поставщики от компании [Devart](#) для различных баз данных, включая Oracle, MySQL, PostgreSQL, SQLite, Salesforce, DB2 и SQL Server.
- **Поставщики CData Software**
 - Существуют сторонние поставщики от [CData Software](#) для различных хранилищ данных, включая Salesforce, хранилище таблиц Azure, MySql и многие другие.
- **Поставщик Firebird**
 - Доступен как [пакет NuGet](#).
- **Поставщик Visual Fox Pro**
 - Доступен как [пакет NuGet](#).
- **MySQL**
 - [MySQL Connector для Net](#)
- **PostgreSQL**
 - Npgsql доступен как [пакет NuGet](#).
- **Oracle**
 - ODP.NET доступен как [пакет NuGet](#).

Обратите внимание, что при включении в этот список не указывается уровень функциональности или поддержки для данного поставщика. Это значит только то, что стала доступной сборка для EF6.

Регистрация поставщиков EF

Начиная с Entity Framework 6, поставщики EF можно регистрировать, используя либо конфигурацию на основе кода, либо файл конфигурации приложения.

Регистрация в файле конфигурации

Регистрация поставщика EF в файлах app.config или web.config имеет следующий формат.

```
<entityFramework>
  <providers>
    <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
  </providers>
</entityFramework>
```

Обратите внимание: часто, если поставщик EF устанавливается из пакета NuGet, пакет NuGet автоматически добавит эту регистрацию в файл конфигурации. При установке пакета NuGet в проект, который не является запускаемым проектом для вашего приложения, может потребоваться скопировать регистрацию в файл конфигурации для запускаемого проекта.

invariantName в этой регистрации является тем же инвариантным именем, которое используется для идентификации поставщика ADO.NET. Его можно найти в виде атрибута invariant в регистрации DbProviderFactory и в виде атрибута providerName в регистрации строки подключения. Используемое инвариантное имя также должно быть включено в документацию для поставщика. Примеры инвариантных имен: System.Data.SqlClient для SQL Server и System.Data.SqlServerCe.4.0 для SQL Server Compact.

type в данной регистрации — это имя с указанием сборки для типа поставщика, который является производным от System.Data.Entity.Core.Common.DbProviderServices. Например, для SQL Compact будет использоваться строка System.Data.Entity.SqlServerCompact.SqlCeProviderServices, EntityFramework.SqlServerCompact. Используемый здесь тип должен быть включен в документацию по поставщику.

Регистрация на основе кода

Начиная с Entity Framework 6, конфигурацию в масштабе приложения для EF можно указывать в коде. Все сведения см. в разделе [Entity Framework Code-Based Configuration](#) (Конфигурация Entity Framework на основе кода). Стандартным способом регистрации поставщика EF с помощью конфигурации на основе кода является создание класса, производного от System.Data.Entity.DbConfiguration, и его размещение в одной сборке с классом DbContext. Затем класс DbConfiguration должен зарегистрировать поставщик в своем конструкторе. Например, чтобы зарегистрировать поставщик SQL Compact, класс DbConfiguration выглядит следующим образом.

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices(
            SqlCeProviderServices.ProviderInvariantName,
            SqlCeProviderServices.Instance);
    }
}
```

В этом коде SqlCeProviderServices.ProviderInvariantName является удобным способом выражения строки инвариантного имени поставщика SQL Server Compact (System.Data.SqlServerCe.4.0), а SqlCeProviderServices.Instance возвращает единственный экземпляр поставщика SQL Compact EF.

Что делать, если нужный поставщик недоступен?

Если поставщик доступен в предыдущих версиях EF, мы рекомендуем обратиться к владельцу поставщика и попросить его создать версию EF6. Необходимо включить ссылку на [документацию к модели поставщика](#)

EF6.

Можно ли создать поставщик самостоятельно?

Конечно, вы можете создать поставщик EF самостоятельно несмотря на то, что эта процедура не должна считаться тривиальной задачей. Лучше всего начать с приведенной выше ссылки о модели поставщика EF6. Кроме того, возможно, вы сочтете полезным использовать качестве отправной точки или в справочных целях код для поставщика SQL Server и SQL CE, включенный в [базу открытого исходного кода EF](#).

Обратите внимание, что, начиная с EF6, поставщик EF не так тесно связан с поставщиком ADO.NET. Это упрощает создание поставщика EF без написания или упаковки классов ADO.NET.

Модели поставщика Entity Framework 6

27.09.2018 • 27 minutes to read • [Edit Online](#)

Модель поставщика Entity Framework позволяет Entity Framework для использования с различными типами сервера базы данных. Например один поставщик можно подключить разрешающее EF для использования Microsoft SQL Server, хотя можно подключить другого поставщика к разрешающее EF для использования Microsoft SQL Server Compact Edition. Поставщики для EF6, нам известно можно найти на [поставщики Entity Framework](#) страницы.

Некоторые изменения требовались так, как EF взаимодействует с поставщиками, чтобы разрешить EF освобождения лицензии с открытым исходным кодом. Эти изменения к перестройке поставщиков EF относительно к сборкам EF6, а также новые механизмы для регистрации поставщика.

Перестроение

С помощью EF6 базовый код, который ранее был частью .NET Framework теперь поставляется в качестве сборки (OOB)-каналу. Сведения о том, как создавать приложения для EF6 можно найти на [обновление приложений для EF6](#) страницы. Поставщики также необходимо перестроить с помощью следующей процедуры.

Общие сведения о типах поставщика

Поставщик EF действительно является коллекцией из поставщика служб, определяемых типов CLR, эти службы расширяет из (для базового класса) или не реализует (для интерфейса). Два из этих служб, являются основными и необходимые для Entity FRAMEWORK работать вообще. Другие являются необязательными и только должны быть реализованы, если определенные функции необходимо и/или реализации по умолчанию из этих служб не работает для конкретной базы данных сервера.

Поставщик фундаментальные типы

DbProviderFactory

EF зависит от наличия типа, производного от [System.Data.Common.DbProviderFactory](#) для выполнения всех базы данных низкого уровня доступа. DbProviderFactory не фактически является частью EF, а вместо этого является, класс в .NET Framework, которая служит точкой входа для поставщиков ADO.NET может использоваться платформой EF, другие O/RMs или непосредственно приложением для получения экземпляров подключения, команды, параметры и другие абстракции ADO.NET в поставщике узлам. Дополнительные сведения о DbProviderFactory см. в [документацию MSDN для ADO.NET](#).

DbProviderServices

EF зависит от наличия типа, производного от DbProviderServices для добавления новых функциональных возможностей, необходимых EF на основе функциональности, уже предоставляемых поставщиком ADO.NET. В более старых версиях EF класса DbProviderServices входил в состав .NET Framework и был найден в пространстве имен System.Data.Common. Начиная с EF6 этот класс теперь является частью EntityFramework.dll и находится в пространстве имен System.Data.Entity.Core.Common.

Дополнительные сведения об основных возможностях DbProviderServices реализации можно найти на [MSDN](#). Тем не менее Обратите внимание на то, что по состоянию на момент написания статьи эта информация не обновляется для EF6 несмотря на то, что большинства понятий, по-прежнему допустимы. Для реализации SQL Server и SQL Server Compact DbProviderServices также возвращены в [codebase открытым исходным кодом](#) и могут служить полезных ссылок для других реализаций.

В более старых версиях EF реализация DbProviderServices для использования были получены напрямую из поставщика ADO.NET. Это было сделано, приведение DbProviderFactory IServiceProvider и вызвать метод GetService. Это тесно связан поставщик EF с DbProviderFactory. Такое объединение заблокирован EF из .NET Framework и поэтому для EF6 тесной связи была удалена и реализация DbProviderServices зарегистрирован непосредственно в файле конфигурации приложения или на основе кода конфигурации, как описано более подробно *регистрации DbProviderServices* разделе ниже.

Дополнительные службы

Помимо основных служб, описанных выше существуют также многие другие службы, используемые EF, всегда или иногда от поставщика. Реализация поставщика по умолчанию этих служб может быть предоставленный реализация DbProviderServices. Приложения можно также переопределить реализации этих служб или предоставить реализации, если тип DbProviderServices не предоставляет значение по умолчанию. Это описано более подробно в *разрешение дополнительных служб* разделе ниже.

Ниже перечислены типы дополнительная служба, поставщик может представлять интерес для поставщика. Дополнительные сведения о каждом из типов служб можно найти в документации по API.

IDbExecutionStrategy

Это дополнительная служба, которая позволяет поставщику для реализации повторных попыток или другого поведения, при выполнении запросов и команд в базе данных. Если реализации не указаны, затем EF будет просто выполните команды и распространить все исключения. Для SQL Server эта служба используется для предоставления политику повтора, что особенно полезно при запуске для серверов базы данных на основе облака, таких как SQL Azure.

IDbConnectionFactory

Это дополнительная служба, которая позволяет поставщику для создания объектов DbConnection по соглашению при наличии только имени базы данных. Обратите внимание, что хотя эта служба может разрешить реализация DbProviderServices он был присутствовавшая EF 4.1 и можно также задать явным образом в файле конфигурации или в коде. Поставщик получит только возможность самому разрешить эту службу, если она зарегистрирована как поставщик по умолчанию (см. в разделе *поставщика по умолчанию* ниже) и если фабрика соединений по умолчанию не было задано в другом месте.

DbSpatialServices

Это дополнительные службы, которые позволяет поставщику добавить поддержку Пространственные типы geography и geometry. Реализация этой службы должны предоставляться в приложение для использования EF с Пространственные типы. DbSptialServices запрашивается двумя способами. Во-первых, поставщика пространственной службы запрашиваются с помощью объекта DbProviderInfo (который содержит инвариант имя и маркер манифеста) как ключ. Во-вторых DbSpatialServices могут запрашиваться без ключа. Это используется для разрешения «глобальные пространственных поставщик», используемый при создании автономного DbGeography или DbGeometry типов.

MigrationSqlGenerator

Это дополнительная служба, позволяющий миграции EF, используемые для создания SQL, используемый в создании и изменении схем баз данных Code First. Реализация необходим для поддержки миграции. Если реализация затем он будет использоваться при создании базы данных с помощью инициализаторов базы данных или метода Database.Create.

Func < DbConnection, строка, HistoryContextFactory >

Это дополнительная служба, которая позволяет поставщику настроить сопоставление HistoryContext для `_MigrationHistory` таблицу, используемую миграции EF. HistoryContext является первый DbContext кода и могут быть настроены в обычном fluent API для изменения действия, такие как имя таблицы и спецификации сопоставления столбцов. Реализация по умолчанию эта служба, возвращенных EF для

всех поставщиков может работать для заданной базы данных сервера, если этот поставщик поддерживает все сопоставления по умолчанию таблиц и столбцов. В этом случае поставщик не нужно предоставлять реализацию этой службы.

IDbProviderFactoryResolver

Это дополнительная служба для получения правильного DbProviderFactory из указанного объекта DbConnection. Реализация по умолчанию эта служба, возвращенных EF для всех поставщиков предназначен для работы для всех поставщиков. Тем не менее, при выполнении в .NET 4, объект DbProviderFactory не является общедоступной с одним из его DbConnections. Таким образом EF использует часть эвристических правил для поиска зарегистрированных поставщиков для поиска совпадений. Вполне возможно, что некоторые поставщики таких методов завершится ошибкой, и в таких ситуациях поставщик должен предоставить новую реализацию.

Регистрация DbProviderServices

Реализации DbProviderServices для использования могут быть зарегистрированы в файл конфигурации (app.config или web.config) или с помощью конфигурации на основе кода приложения. В любом случае регистрации использует «неизменяемое имя поставщика» в качестве ключа. Благодаря этому несколько поставщиков зарегистрировать и использовать их в одном приложении. Инвариантное имя, используемое для регистрации EF совпадает неизменяемое имя, используемое для регистрации и соединения строки поставщика ADO.NET. Например для SQL Server неизменяемое имя «System.Data.SqlClient» используется.

Регистрация в файле конфигурации

Тип DbProviderServices для использования регистрируется как поставщик элемент в списке поставщиков entityFramework раздела файла конфигурации приложения. Пример:

```
<entityFramework>
  <providers>
    <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
  </providers>
</entityFramework>
```

Тип строка должна быть реализация DbProviderServices нужно использовать имя типа с указанием сборки.

Регистрация на основе кода

Начиная с EF6 поставщиков также можно зарегистрировать с помощью кода. Это позволяет поставщику EF использовать без изменения в файл конфигурации приложения. Для использования конфигурации на основе кода приложения необходимо создать класс DbConfiguration, как описано в разделе [документации конфигурация на основе кода](#). Конструктор класса DbConfiguration необходимо вызвать SetProviderServices для регистрации поставщика EF. Пример:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices("My.New.Provider", new MyProviderServices());
    }
}
```

Разрешение дополнительных служб

Как упоминалось выше в *Обзор типов поставщика* разделе DbProviderServices класс может также

использоваться для разрешения дополнительных служб. Это возможно, поскольку DbProviderServices реализует IDbDependencyResolver и каждого зарегистрированного типа DbProviderServices добавляется в качестве «Сопоставитель по умолчанию». Более подробно описан механизм IDbDependencyResolver **разрешение зависимостей**. Тем не менее необязательно знать все понятия, описанные в этой спецификации для разрешения дополнительных служб в поставщике.

Наиболее распространенным способом для поставщика для разрешения дополнительных служб является вызов DbProviderServices.AddDependencyResolver для каждой службы в конструктор класса DbProviderServices. Например SqlProviderServices (EF поставщика для SQL Server) имеет примерно такой для инициализации код:

```
private SqlProviderServices()
{
    AddDependencyResolver(new SingletonDependencyResolver<IDbConnectionFactory>(
        new SqlConnectionFactory()));

    AddDependencyResolver(new ExecutionStrategyResolver<DefaultSqlExecutionStrategy>(
        "System.data.SqlClient", null, () => new DefaultSqlExecutionStrategy()));

    AddDependencyResolver(new SingletonDependencyResolver<Func<MigrationSqlGenerator>>(
        () => newSqlServerMigrationSqlGenerator(), "System.data.SqlClient"));

    AddDependencyResolver(new SingletonDependencyResolver<DbSpatialServices>(
        SqlSpatialServices.Instance,
        k =>
    {
        var asSpatialKey = k as DbProviderInfo;
        return asSpatialKey == null
            || asSpatialKey.ProviderInvariantName == ProviderInvariantName;
    }));
}
```

Этот конструктор использует следующие вспомогательные классы:

- SingletonDependencyResolver: предоставляет простой способ устранить одноэлементные службы — то есть службы, для которых тот же экземпляр возвращается каждый раз, что вызывается GetService. Временные службы часто регистрируются как одноэлементная фабрика, которая будет использоваться для создания временных экземпляров по запросу.
- ExecutionStrategyResolver: арбитра для возвращения IExecutionStrategy реализаций.

Вместо использования DbProviderServices.AddDependencyResolver можно также переопределить DbProviderServices.GetService и непосредственно устранить дополнительных служб. Этот метод будет вызываться, когда EF определены определенного типа и, в некоторых случаях для заданного ключа службы. Метод должен вернуть службы, способно или возвращать значение null, чтобы отказаться от возврата службы и позволит другому классу для ее устранения. Например чтобы устранить фабрика соединений по умолчанию код в GetService может выглядеть примерно следующим образом:

```
public override object GetService(Type type, object key)
{
    if (type == typeof(IDbConnectionFactory))
    {
        return new SqlConnectionFactory();
    }
    return null;
}
```

Порядок регистрации

Когда несколько реализаций DbProviderServices регистрируются в файле конфигурации приложения

они добавляются в качестве вторичного сопоставителей в порядке, в котором они перечислены. Так как сопоставители всегда добавляются к верхней части цепочки вторичной Сопоставитель, это означает, что поставщик в конце списка получит возможность разрешать зависимости раньше других. (Это может показаться немного неочевидным на первый, но имеет смысл если представить выполнением каждого поставщика из списка и размещения его поверх существующих поставщиков).

Этот порядок обычно не имеет значения, так как большинство поставщика службы поставщика и с помощью команд, неизменяемое имя поставщика. Тем не менее для служб, не идентифицируются по неизменяемое имя поставщика и некоторые другие поставщика ключ, служба будет разрешаться на основе этого порядка. Например если не задано явным образом по-разному где-то иначе, фабрика соединений по умолчанию получаются из верхних поставщик в цепи.

Файл регистрации дополнительные конфигурации

Имеется возможность явно зарегистрировать некоторые дополнительного поставщика службы, описанной выше непосредственно в файле конфигурации приложения. После этого регистрации в файле конфигурации будет использоваться вместо ничего возвращаемый метод GetService реализация DbProviderServices.

Регистрация фабрики подключения по умолчанию

Начиная с EF5 пакета EntityFramework NuGet автоматически зарегистрировать фабрику соединений SQL Express или LocalDb фабрики подключения в файле конфигурации.

Пример:

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">
  </defaultConnectionFactory>
</entityFramework>
```

Type — имя типа с указанием сборки для фабрика соединений по умолчанию, которая должна реализовывать интерфейс IDbConnectionFactory.

Мы рекомендуем что пакета NuGet поставщика фабрика соединений по умолчанию таким образом, при установке. См. в разделе *пакеты NuGet для поставщиков* ниже.

Дополнительные изменения поставщика EF6

Изменения пространственных поставщика

Поставщики, которые поддерживают Пространственные типы теперь необходимо реализовать некоторые дополнительные методы в классах, производных от DbSpatialDataReader:

- `public abstract bool IsGeographyColumn(int ordinal)`
- `public abstract bool IsGeometryColumn(int ordinal)`

Существует также новые асинхронные версии существующих методов, которые рекомендуются для переопределения, как делегировать синхронные методы реализации по умолчанию и поэтому не следует вызывать асинхронно.

- `public virtual Task<DbGeography> GetGeographyAsync(int ordinal, CancellationToken cancellationToken)`
- `public virtual Task<DbGeometry> GetGeometryAsync(int ordinal, CancellationToken cancellationToken)`

Встроенная поддержка Enumerable.Contains

EF6 появился новый тип выражения, DblInExpression, которая была добавлена для устранения проблем производительности, использовании Enumerable.Contains в запросах LINQ. Класс DbProviderManifest

имеет новый виртуальный метод `SupportsInExpression`, который вызывается объектом EF, чтобы определить, если поставщик обрабатывает новый тип выражения. Для обеспечения совместимости с существующими реализациями поставщика, метод возвращает значение `false`. Чтобы извлечь пользу из этого улучшения, поставщик EF6 можно добавить код для обработки `DbInExpression` и переопределить `SupportsInExpression` возвращающее значение `true`. Экземпляр `DbInExpression` могут создаваться путем вызова метода `DbExpressionBuilder.In`. Экземпляр `DbInExpression` представляет собой `DbExpression`, обычно представляющих столбец таблицы и списка `DbConstantExpression` для проверки на соответствие.

Пакеты NuGet для поставщиков

Один из способов сделать доступным один из поставщиков EF6 является выпустить ее в виде пакета NuGet. С помощью пакета NuGet имеет следующие преимущества:

- Это простой в использовании NuGet для добавления регистрации поставщика в файле конфигурации приложения
- Можно вносить дополнительные изменения в файл конфигурации для задания фабрика соединений по умолчанию, соединений, производимых по соглашению будут использовать зарегистрированный поставщик
- NuGet обрабатывает добавление переадресации привязок, таким образом, чтобы поставщик EF должны продолжать работать даже в том случае, после выпуска нового пакета EF

Примером этого является, входящей в состав пакета `EntityFramework.SqlServerCompact` [открытым исходным кодом codebase](#). Этот пакет предоставляет хороший шаблон для создания поставщика EF пакеты NuGet.

Команды PowerShell

При установке пакета EntityFramework NuGet он регистрирует модуль PowerShell, который содержит две команды, которые очень полезны для поставщика пакетов:

- Добавление `EFProvider` добавляет новую сущность для поставщика в файле конфигурации целевого проекта и гарантирует, что он находится в конце списка зарегистрированных поставщиков.
- Добавление `EFDConnectionFactory` добавляет или обновляет регистрацию `defaultConnectionFactory` в файле конфигурации целевого проекта.

Обе эти команды выполнит Добавление раздел `entityFramework` в файл конфигурации и Добавление коллекции поставщиков, при необходимости.

Предполагается, что эти команды вызываться из `install.ps1` сценария NuGet. Например `install.ps1` SQL Compact поставщика выглядит примерно так:

```
param($installPath, $toolsPath, $package, $project)
Add-EFDefaultConnectionFactory $project 'System.Data.Entity.Infrastructure.SqlCeConnectionFactory,
EntityFramework' -ConstructorArguments 'System.Data.SqlServerCe.4.0'
Add-EFProvider $project 'System.Data.SqlServerCe.4.0'
'System.Data.Entity.SqlServerCompact.SqlCeProviderServices, EntityFramework.SqlServerCompact'</pre>
```

Дополнительные сведения об этих командах можно получить с помощью `get-help` в окне консоли диспетчера пакетов.

Перенос поставщиков

Перенос поставщик — поставщик EF и/или ADO.NET, который создает оболочку для существующего поставщика расширить ее с другими функциями, например профилирования или возможностей отслеживания. Упаковки поставщики могут регистрироваться обычным способом, но часто бывает более удобно для настройки поставщика упаковки во время выполнения путем перехвата разрешения,

связанные с поставщиком служб. Для этого используется статическое событие OnLockingConfiguration DbConfiguration класса.

OnLockingConfiguration вызывается после EF определяет, где все конфигурации EF для домена приложения будут получены из, но до его блокировки для использования. При запуске приложения (до использования EF) приложение должно зарегистрироваться обработчик событий для данного события. (Мы рассматриваем, добавляя поддержку регистрации этого обработчика в файле конфигурации, но это еще не поддерживается). Обработчик событий, затем следует сделать вызов к ReplaceService для каждой службы, который должен быть заключен.

Например программы-оболочки для IDbConnectionFactory и DbProviderService, должен быть зарегистрирован обработчик примерно так:

```
DbConfiguration.OnLockingConfiguration +=
    (_, a) =>
{
    a.ReplaceService<DbProviderServices>(
        (s, k) => new MyWrappedProviderServices(s));

    a.ReplaceService<IDbConnectionFactory>(
        (s, k) => new MyWrappedConnectionFactory(s));
};
```

Служба, которая будет разрешена и следует упаковывать вместе с ключом, который был использован для разрешения службы, передаются в обработчик. Обработчик можно переносить этой службы и заменить полученную упакованного версией.

Разрешение DbProviderFactory с EF

DbProviderFactory является одним из фундаментальных поставщика типов Затребован EF, как описано в разделе *Обзор типов поставщика* предыдущем разделе. Как уже упоминалось, это не является типом EF и регистрации обычно не является частью конфигурации EF, а вместо этого является обычной регистрации поставщика ADO.NET в файле machine.config и в файле конфигурации приложения.

Несмотря на это EF по-прежнему использует его обычной зависимостью механизм разрешения конфликтов при поиске класса DbProviderFactory для использования. Сопоставитель по умолчанию использует обычный регистрации ADO.NET в файлах конфигурации, так что это обычно понятен. Однако из-за разрешения обычной зависимостью используется механизм это означает, что IDbDependencyResolver может использоваться для разрешения класса DbProviderFactory даже в том случае, если не было сделано обычной регистрации ADO.NET.

Разрешение DbProviderFactory таким образом ряд последствий:

- Приложения с помощью конфигурации на основе кода можно добавить вызовы в классе DbConfiguration регистрируемый соответствующий объект DbProviderFactory. Это особенно полезно для приложений, которые не требуется (или не может) сделать вообще использовать какой-либо настройки на основе файлов.
- Служба может переноситься или заменить с помощью ReplaceService, как описано в разделе *упаковки поставщики* выше
- Теоретически реализация DbProviderServices удалось разрешить класса DbProviderFactory.

Важно следить помнить о выполнении этих операций, то, что они будут влиять только на поиск DbProviderFactory платформой EF. Другой код не EF может по-прежнему требуется, чтобы поставщик ADO.NET для регистрации обычным способом и может завершиться ошибкой, если регистрация не найден. По этой причине лучше обычно для класса DbProviderFactory для регистрации обычным способом ADO.NET.

Связанные службы

Если EF используется для разрешения класса DbProviderFactory, то его необходимо разрешить IProviderInvariantName и IDbProviderFactoryResolver служб.

IProviderInvariantName — это служба, который используется для определения неизменяемое имя поставщика для данного типа DbProviderFactory. Реализация по умолчанию эта служба использует Регистрация поставщика ADO.NET. Это означает, что если поставщик ADO.NET не зарегистрирован обычным способом, так как объект DbProviderFactory разрешается путем EF, затем он также будет необходимо для устранения этой службы. Обратите внимание на то, что арбитр для этой службы автоматически добавляется при использовании метода DbConfiguration.SetProviderFactory.

Как описано в разделе *Обзор типов поставщика* разделе выше, IDbProviderFactoryResolver используется для получения правильного DbProviderFactory из указанного объекта DbConnection. Реализация по умолчанию эта служба при выполнении в .NET 4 использует Регистрация поставщика ADO.NET. Это означает, что если поставщик ADO.NET не зарегистрирован обычным способом, так как объект DbProviderFactory разрешается путем EF, затем он также будет необходимо для устранения этой службы.

Поддержка пространственных типов

13.09.2018 • 4 minutes to read • [Edit Online](#)

Платформа Entity Framework поддерживает работу с пространственными данными через классы `DbGeography` или `DbGeometry`. Эти классы используют функциональные возможности конкретной базы данных, предлагаемых поставщиком Entity Framework. Не все поставщики поддерживают Пространственные данные и которые, возможно, дополнительные необходимые компоненты, например, установка сборок пространственный тип. Дополнительные сведения о поддержке поставщика для пространственных типов приведены ниже.

Дополнительные сведения об использовании пространственных типов в приложении можно найти в двух пошаговых руководствах, один для `Code First`, другой — для `Database First` или `Model First`:

- [Типы пространственных данных в коде сначала](#)
- [Типы пространственных данных в конструкторе EF](#)

Выпуски EF, которые поддерживают Пространственные типы

Поддержка пространственных типов появилась в EF5. Тем не менее в EF5 Пространственные типы поддерживаются только если предназначен для приложения и запускается на .NET 4.5.

Начиная с EF6, Пространственные типы поддерживаются для приложений, предназначенных для .NET 4 и .NET 4.5.

Поставщики EF, которые поддерживают Пространственные типы

EF5

Поставщики Entity Framework для EF5, нам известно, что поддержка пространственных типов:

- Поставщик Microsoft SQL Server
 - Этот поставщик поставляется как часть EF5.
 - Этот поставщик зависит от некоторые дополнительные библиотеки низкого уровня, которые может потребоваться установить — дополнительные сведения приведены ниже.
- [Devart dotConnect для Oracle](#)
 - Это стороннего поставщика из Devart.

Если вы знаете об EF5 поставщика, поддерживающего Пространственные типы затем запустить контакта, и мы будем рады добавить его в этот список.

EF6

Поставщики Entity Framework для EF6, нам известно, что поддержка пространственных типов:

- Поставщик Microsoft SQL Server
 - Этот поставщик поставляется как часть EF6.
 - Этот поставщик зависит от некоторые дополнительные библиотеки низкого уровня, которые может потребоваться установить — дополнительные сведения приведены ниже.
- [Devart dotConnect для Oracle](#)
 - Это стороннего поставщика из Devart.

Если вы знаете из EF6 поставщика, поддерживающего пространственных типов можно затем запустить обратитесь в службу, и мы будем рады добавить его в этот список.

Необходимые условия для пространственных типов с помощью Microsoft SQL Server

Пространственная поддержка SQL Server зависит от низкого уровня, связанные с SQL Server типов SqlGeography и SqlGeometry. Эти типы расположены в сборку Microsoft.SqlServer.Types.dll, и эта сборка не поставляется как часть EF или как часть платформы .NET Framework.

При установке Visual Studio также часто установит версию SQL Server, и это будет включать копию файла Microsoft.SqlServer.Types.dll.

Если SQL Server не установлен на компьютере, где вы хотите использовать Пространственные типы или Пространственные типы были исключены из установки SQL Server, вам потребуется установить их вручную. Типы могут быть установлены с помощью `SQLSysClrTypes.msi`, который является частью пакета дополнительных компонентов Microsoft SQL Server. Пространственные типы — SQL Server от версии, поэтому рекомендуется [поиск «SQL Server Feature Pack»](#) в центре загрузки Майкрософт, затем выберите и загрузите соответствующий до версии SQL Server, будет использоваться.

Работа с прокси

13.09.2018 • 4 minutes to read • [Edit Online](#)

При создании экземпляров типов сущностей POCO, Entity Framework часто создает экземпляры динамически создаваемого производный тип, который выступает в качестве прокси для сущности. Этот прокси-сервер определяет некоторые виртуальные свойства сущности, которую нужно вставить обработчик для выполнения действий автоматически, при обращении к свойству. Например этот механизм используется для поддержки отложенная загрузка связей. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Отключение создания прокси-сервера

Иногда полезно запретить Entity Framework создавать экземпляры прокси-сервера. Например сериализация экземпляров без прокси-сервера — значительно легче, чем сериализации экземпляров прокси-сервера. Создание прокси-сервер можно отключить, сняв флаг ProxyCreationEnabled. Централизованно, это можно сделать это в конструкторе контекста. Пример:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.ProxyCreationEnabled = false;
    }

    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

Обратите внимание, что EF не будет создавать учетные записи-посредники для типов там, где нет ничего для прокси-сервера для выполнения. Это означает, что прокси-серверы можно избежать благодаря использованию типов, которые являются запечатанными или нет виртуальных свойств.

Явного создания экземпляра прокси-сервера

Экземпляр прокси-сервера не создаются при создании экземпляра сущности с помощью оператора new. Это может не быть проблемой, но если вам нужно создать экземпляр прокси-сервера (например, благодаря чему отложенной загрузки или прокси-сервера отслеживания изменений будет работать) затем это можно сделать с помощью метода Create объекта DbSet. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Create();
```

Если вы хотите создать экземпляр производного типа сущности используется универсальная версия инструкции Create. Пример:

```
using (var context = new BloggingContext())
{
    var admin = context.Users.Create<Administrator>();
```

Обратите внимание, что метод Create не добавить или присоединить к контексту созданной сущности.

Обратите внимание, что метод Create просто создаст экземпляр самого типа сущности при создании прокси-типа сущности будет иметь значение не, поскольку он бы не выполняет никаких действий. Например если тип сущности является запечатанным и/или не имеет виртуальных свойств, а затем создать будет просто создайте экземпляр типа сущности.

Получение типа фактические сущности из типа прокси-сервера

Прокси-типы имеют имена, которые выглядят примерно следующим образом:

System.Data.Entity.DynamicProxies.Blog_5E43C6C196972BF0754973E48C9C941092D86818CD94005E9A759B70BF6E48E6

Тип сущности можно найти для этого типа прокси-сервера, с помощью метода GetObjectType от ObjectContext. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    var entityType = ObjectContext.GetObjectType(blog.GetType());
}
```

Обратите внимание, что если тип, передаваемый GetObjectType является экземпляром типа сущности, не является типом прокси-сервера, затем тип сущности, все равно будет возвращена. Это означает, что этот метод всегда можно использовать для получения фактического объекта типа без каких-либо других проверок для просмотра, если тип является типом прокси-сервера, или нет.

Тестирование с помощью макетирования

06.10.2018 • 12 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

При написании тестов для приложения часто желательно избежать обращения к базе данных. Платформа Entity Framework позволяет добиться этого путем создания контекста — с поведение, определяемое тестов —, в котором используется данных в памяти.

Параметры для создания тестовых дублеров

Существует два различных подхода, которые могут использоваться для создания в памяти версии контекста.

- **Создать свои собственные тестовые дублеры** — этот подход подразумевает написание собственной реализации в памяти DbSets и данные контекста. Это обеспечивает большую контроль над тем, как ведут себя классы, но может включать в себя написание и владельцев разумный объем кода.
- **Использовать платформа имитации для создания тестовых дублеров** — использование платформы имитированной (например, Moq) может иметь реализаций в памяти из вас контекст и наборов, созданных динамически во время выполнения для вас.

В этой статье будет работать при использовании платформа имитации. Для создания собственных тестовых дублеров см. в разделе [тестирование с помощью вашей собственной тестирования двойной точности](#).

Чтобы продемонстрировать использование EF с макетирования мы собираемся использовать Moq. Самый простой способ получить Moq является установка [Moq пакет из NuGet](#).

Тестирование с помощью EF6 предварительной версии

Сценарии, приведенном в этой статье, зависит от некоторых изменений, внесенных в DbSet в EF6.

Тестирование с помощью EF5 и более ранней версии см. в разделе [тестирования с контекстом имитировать](#).

Ограничения EF in-Memory тестовых дублеров

В памяти тестовых дублеров может быть хорошим средством предоставления уровня покрытия битов приложения, использующие EF модульного теста. Тем не менее при этом при использовании LINQ to Objects для выполнения запросов к данным в памяти. Это может привести другое поведение, чем при использовании поставщика платформы EF LINQ (LINQ to Entities) для преобразования запросов в SQL, которая выполняется в базе данных.

Примером такой разницы загрузка связанных данных. Если создать серию блогов, в которых тематических записях блога, то при использовании данных в памяти связанных сообщений всегда будут загружаться для каждого блога. Тем не менее при выполнении в базе данных будет загружаться только при использовании метода `Include`.

По этой причине рекомендуется всегда включать некоторый уровень end-to-end тестирования (в дополнение к модульные тесты), чтобы ваше приложение работает правильно для базы данных.

Следуя указаниям в этой статье

В этой статье приводятся полные Листинги кода, которые можно скопировать в Visual Studio для выполнения этой процедуры при необходимости. Проще всего создать **проект модульного теста** и требуется целевой объект **.NET Framework 4.5** для выполнения в разделах, которые используют асинхронный.

Модель EF

Службы, мы собираемся тестирования использует EF модели BloggingContext и классы blog и Post. Этот код был создан в конструкторе EF или быть модели Code First.

```
using System.Collections.Generic;
using System.Data.Entity;

namespace TestingDemo
{
    public class BloggingContext : DbContext
    {
        public virtual DbSet<Blog> Blogs { get; set; }
        public virtual DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}
```

Виртуальные свойства DbSet, с помощью конструктора EF

Обратите внимание на то, что свойства DbSet в контексте помечаются как виртуальные. Таким образом, платформа имитации для наследования от наш контекст и переопределение этих свойств с реализацией макеты.

Если вы используете Code First, то ваши классы можно изменить напрямую. Если вы используете конструктор EF необходимо изменить шаблон T4, который создает контекст. Откройте <model_name>.Context.tt файл, который вложен в вы EDMX-файла, найдите следующий фрагмент кода и добавьте в ключевое слово `virtual`, как показано.

```

public string DbSet(EntitySet entitySet)
{
    return string.Format(
        CultureInfo.InvariantCulture,
        "{0} virtual DbSet<{1}> {2} {{ get; set; }}",
        Accessibility.ForReadOnlyProperty(entitySet),
        _typeMapper.GetTypeName(entitySet.ElementType),
        _code.Escape(entitySet));
}

```

Служба проверяемый

Для демонстрации тестирование с помощью выполняющейся в памяти тестовых дублеров мы хотим написание нескольких тестов для BlogService. Служба, которая позволяет создавать новые веб-дневники (AddBlog) и возвращает все блоги упорядоченные по имени (GetAllBlogs). В дополнение к GetAllBlogs мы также предоставляем метод, который получит асинхронно все блоги, упорядоченные по имени (GetAllBlogsAsync).

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public Blog AddBlog(string name, string url)
        {
            var blog = _context.Blogs.Add(new Blog { Name = name, Url = url });
            _context.SaveChanges();

            return blog;
        }

        public List<Blog> GetAllBlogs()
        {
            var query = from b in _context.Blogs
                       orderby b.Name
                       select b;

            return query.ToList();
        }

        public async Task<List<Blog>> GetAllBlogsAsync()
        {
            var query = from b in _context.Blogs
                       orderby b.Name
                       select b;

            return await query.ToListAsync();
        }
    }
}

```

Сценарии тестирования не связанным с запросом

Это все, что необходимо сделать, чтобы начать тестирование методы без запросов. Следующий тест использует Moq для создания контекста. Затем он создает коллекцию DbSet<блог> и настраивающий возвращается из свойства контекста блоги. Затем контекст используется для создания новых BlogService, который затем используется для создания нового блога — с помощью метода AddBlog. Наконец тест проверяет, что служба добавлен новый блог и вызове метода SaveChanges в контексте.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Data.Entity;

namespace TestingDemo
{
    [TestClass]
    public class NonQueryTests
    {
        [TestMethod]
        public void CreateBlog_saves_a_blog_via_context()
        {
            var mockSet = new Mock<DbSet<Blog>>();

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(m => m.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            service.AddBlog("ADO.NET Blog", "http://blogs.msdn.com/adonet");

            mockSet.Verify(m => m.Add(It.IsAny<Blog>()), Times.Once());
            mockContext.Verify(m => m.SaveChanges(), Times.Once());
        }
    }
}
```

Сценарии тестирования запроса

Чтобы иметь возможность выполнять запросы к нашей DbSet тестовый дублер нам нужно настроить реализации IQueryble. Первым шагом является создание некоторых данных в памяти — мы используем список<блог>. Теперь создадим контекст и DBSet<блог> затем подключите реализации IQueryble DbSet — они просто делегировании в поставщике LINQ to Objects, работающий со списком<T>.

Затем можно создать BlogService, в зависимости от наших тестовых дублеров и убедитесь, что данные, которые мы получаем от GetAllBlogs упорядочен по имени.

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace TestingDemo
{
    [TestClass]
    public class QueryTests
    {
        [TestMethod]
        public void GetAllBlogs_orders_by_name()
        {
            var data = new List<Blog>
            {
                new Blog { Name = "BBB" },
                new Blog { Name = "ZZZ" },
                new Blog { Name = "AAA" },
            }.AsQueryable();

            var mockSet = new Mock<DbSet<Blog>>();
            mockSet.As<IQueryable<Blog>>().Setup(m => m.Provider).Returns(data.Provider);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.Expression).Returns(data.Expression);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.ElementType).Returns(data.ElementType);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.GetEnumerator()).Returns(data.GetEnumerator());

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(c => c.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            var blogs = service.GetAllBlogs();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

Тестирование с помощью асинхронных запросов

Entity Framework 6 был представлен набор методов расширения, которые могут использоваться для асинхронного выполнения запроса. Эти методы примеры `ToListAsync`, `FirstAsync`, `ForEachAsync` и т. д.

Так как запросы Entity Framework используют LINQ, методы расширения определяются в `IQueryable` и `IEnumerable`. Тем не менее так как они предназначены только для использования с Entity Framework может появиться следующая ошибка при попытке использовать их в запрос LINQ, который не запрос Entity Framework:

Источник `IQueryable` не реализует `IDbAsyncEnumerable{0}`. Можно использовать только те источники, которые реализуют `IDbAsyncEnumerable` для асинхронных операций Entity Framework. Дополнительные сведения см. <http://go.microsoft.com/fwlink/?LinkId=287068>.

Пока асинхронные методы поддерживаются только в том случае, при запуске для запроса на EF, можно использовать их выполнения с в памяти тестирования двойной из `DbSet` в модульном teste.

Чтобы использовать асинхронные методы, нам нужно создать `DbAsyncQueryProvider` в памяти для обработки асинхронного запроса. Хотя можно было бы настроить поставщик запросов, с помощью Moq, гораздо проще создать реализацию `double` теста в коде. Ниже приведен код для этой реализации:

```

using System.Collections.Generic;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Linq.Expressions;
using System.Threading;
using System.Threading.Tasks;

namespace TestingDemo
{
    internal class TestDbAsyncQueryProvider<TEntity> : IDbAsyncQueryProvider
    {
        private readonly IQueryProvider _inner;

        internal TestDbAsyncQueryProvider(IQueryProvider inner)
        {
            _inner = inner;
        }

        public IQueryable CreateQuery(Expression expression)
        {
            return new TestDbAsyncEnumerable<TEntity>(expression);
        }

        public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
        {
            return new TestDbAsyncEnumerable<TElement>(expression);
        }

        public object Execute(Expression expression)
        {
            return _inner.Execute(expression);
        }

        public TResult Execute<TResult>(Expression expression)
        {
            return _inner.Execute<TResult>(expression);
        }

        public Task<object> ExecuteAsync(Expression expression, CancellationToken cancellationToken)
        {
            return Task.FromResult(Execute(expression));
        }

        public Task<TResult> ExecuteAsync<TResult>(Expression expression, CancellationToken cancellationToken)
        {
            return Task.FromResult(Execute<TResult>(expression));
        }
    }

    internal class TestDbAsyncEnumerable<T> : EnumerableQuery<T>, IDbAsyncEnumerable<T>, IQueryable<T>
    {
        public TestDbAsyncEnumerable(IEnumerable<T> enumerable)
            : base(enumerable)
        { }

        public TestDbAsyncEnumerable(Expression expression)
            : base(expression)
        { }

        public IDbAsyncEnumerator<T> GetAsyncEnumerator()
        {
            return new TestDbAsyncEnumerator<T>(this.AsEnumerable().GetEnumerator());
        }

        IDbAsyncEnumerator IDbAsyncEnumerable.GetAsyncEnumerator()
        {
            return GetAsyncEnumerator();
        }
    }
}

```

```
    IQueryProvider IQueryable.Provider
    {
        get { return new TestDbAsyncQueryProvider<T>(this); }
    }
}

internal class TestDbAsyncEnumerator<T> : IDbAsyncEnumerator<T>
{
    private readonly IEnumerator<T> _inner;

    public TestDbAsyncEnumerator(IEnumerator<T> inner)
    {
        _inner = inner;
    }

    public void Dispose()
    {
        _inner.Dispose();
    }

    public Task<bool> MoveNextAsync(CancellationToken cancellationToken)
    {
        return Task.FromResult(_inner.MoveNext());
    }

    public T Current
    {
        get { return _inner.Current; }
    }

    object IDbAsyncEnumerator.Current
    {
        get { return Current; }
    }
}
```

Теперь, когда у нас есть поставщик асинхронный запрос можно написать модульный тест для нашего нового метода GetAllBlogsAsync.

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    [TestClass]
    public class AsyncQueryTests
    {
        [TestMethod]
        public async Task GetAllBlogsAsync_orders_by_name()
        {

            var data = new List<Blog>
            {
                new Blog { Name = "BBB" },
                new Blog { Name = "ZZZ" },
                new Blog { Name = "AAA" },
            }.AsQueryable();

            var mockSet = new Mock<DbSet<Blog>>();
            mockSet.As< IDbAsyncEnumerable<Blog>>()
                .Setup(m => m.GetAsyncEnumerator())
                .Returns(new TestDbAsyncEnumerator<Blog>(data.GetEnumerator()));

            mockSet.As< IQueryable<Blog>>()
                .Setup(m => m.Provider)
                .Returns(new TestDbAsyncQueryProvider<Blog>(data.Provider));

            mockSet.As< IQueryable<Blog>>().Setup(m => m.Expression).Returns(data.Expression);
            mockSet.As< IQueryable<Blog>>().Setup(m => m.ElementType).Returns(data.ElementType);
            mockSet.As< IQueryable<Blog>>().Setup(m => m.GetEnumerator()).Returns(data.GetEnumerator());

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(c => c.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            var blogs = await service.GetAllBlogsAsync();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

Тестирование с помощью собственных тестовых дублеров

13.09.2018 • 13 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

При написании тестов для приложения часто желательно избежать обращения к базе данных. Платформа Entity Framework позволяет добиться этого путем создания контекста — с поведение, определяемое тестов —, в котором используется данных в памяти.

Параметры для создания тестовых дублеров

Существует два различных подхода, которые могут использоваться для создания в памяти версии контекста.

- **Создать свои собственные тестовые дублеры** — этот подход подразумевает написание собственной реализации в памяти DbSets и данные контекста. Это обеспечивает большую контроль над тем, как ведут себя классы, но может включать в себя написание и владельцев разумный объем кода.
- **Использовать платформа имитации для создания тестовых дублеров** — использование платформы имитированной (например, Moq) может иметь реализаций в памяти из вас контекст и наборов, созданных динамически во время выполнения для вас.

В этой статье будет иметь дело с созданием собственных тестов double. Сведения об использовании платформа имитации см. в разделе [тестирование на платформе, макетирование](#).

Тестирование с помощью EF6 предварительной версии

Код, показанный в этой статье совместим с EF6. Тестирование с помощью EF5 и более ранней версии см. в разделе [тестирования с контекстом имитировать](#).

Ограничения EF in-Memory тестовых дублеров

В памяти тестовых дублеров может быть хорошим средством предоставления уровня покрытия битов приложения, использующие EF модульного теста. Тем не менее при этом при использовании LINQ to Objects для выполнения запросов к данным в памяти. Это может привести другое поведение, чем при использовании поставщика платформы EF LINQ (LINQ to Entities) для преобразования запросов в SQL, которая выполняется в базе данных.

Примером такой разницы загрузка связанных данных. Если создать серию блогов, в которых тематических записях блога, то при использовании данных в памяти связанных сообщений всегда будут загружаться для каждого блога. Тем не менее при выполнении в базе данных будет загружаться только при использовании метода `Include`.

По этой причине рекомендуется всегда включать некоторый уровень end-to-end тестирования (в дополнение к модульные тесты), чтобы ваше приложение работает правильно для базы данных.

Следуя указаниям в этой статье

В этой статье приводятся полные Листинги кода, которые можно скопировать в Visual Studio для выполнения этой процедуры при необходимости. Проще всего создать **проект модульного теста** и требуется целевой объект **.NET Framework 4.5** для выполнения в разделах, которые используют асинхронный.

Создание интерфейса контекста

Мы собираемся тестирования службы, который использует EF модели. Чтобы иметь возможность заменять наш контекст EF с версией в памяти для тестирования, мы определим интерфейс что наш контекст EF (и double в памяти) будет implement.

Службы, с которой мы хотим проверить запрос и изменение данных с помощью свойства DbSet наш контекст и также вызывать метод SaveChanges для внесения изменений в базу данных. Поэтому мы добавим эти члены в интерфейсе.

```
using System.Data.Entity;

namespace TestingDemo
{
    public interface IBloggingContext
    {
        DbSet<Blog> Blogs { get; }
        DbSet<Post> Posts { get; }
        int SaveChanges();
    }
}
```

Модель EF

Службы, мы собираемся тестирования использует EF модели BloggingContext и классы блог и Post. Этот код был создан в конструкторе EF или быть модели Code First.

```
using System.Collections.Generic;
using System.Data.Entity;

namespace TestingDemo
{
    public class BloggingContext : DbContext, IBloggingContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}
```

Реализация интерфейса контекста с помощью конструктора EF

Обратите внимание на то, что наш контекст реализует интерфейс IBloggingContext.

Если вы используете Code First можно изменить текущий контекст непосредственно реализуют интерфейс. Если вы используете конструктор EF необходимо изменить шаблон T4, который создает контекст. Откройте <model_name>.Context.tt файл, который вложен в вы EDMX-файла, найдите следующий фрагмент кода и добавьте в интерфейсе, как показано.

```
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> : DbContext, IBloggingContext
```

Служба проверяемый

Для демонстрации тестирование с помощью выполняющейся в памяти тестовых дублеров мы хотим написание нескольких тестов для BlogService. Служба, которая позволяет создавать новые веб-дневники (AddBlog) и возвращает все блоги упорядоченные по имени (GetAllBlogs). В дополнение к GetAllBlogs мы также предоставляем метод, который получит асинхронно все блоги, упорядоченные по имени (GetAllBlogsAsync).

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class BlogService
    {
        private IBloggingContext _context;

        public BlogService(IBloggingContext context)
        {
            _context = context;
        }

        public Blog AddBlog(string name, string url)
        {
            var blog = new Blog { Name = name, Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();

            return blog;
        }

        public List<Blog> GetAllBlogs()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return query.ToList();
        }

        public async Task<List<Blog>> GetAllBlogsAsync()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return await query.ToListAsync();
        }
    }
}

```

Создание теста в памяти удваивает

Теперь, когда у нас есть реальной модели EF и службой, можно использовать его, пора создать тест в памяти, double, можно использовать для тестирования. Мы создали теста TestContext double для наш контекст. В тестовых дублеров, которые мы приступим к выберите поведение, мы хотим, чтобы обеспечить поддержку тестов мы собираемся выполнить. В этом примере мы просто захватываются количество раз, когда в вызове метода SaveChanges, но можно включить любую логику, необходимым для проверки сценария, который вы тестируете.

Мы также создали TestDbSet, который предоставляет реализацию в памяти DbSet. Мы предоставляем полный реализации для всех методов на DbSet (за исключением находит), но необходимо реализовать члены, которые будут использовать сценария тестирования.

TestDbSet использует некоторые другие классы инфраструктуры, которые мы включили, чтобы убедиться, что обработку асинхронных запросов.

```

using System;
using System.Collections.Generic;

```

```

using System.Collections.ObjectModel;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Linq.Expressions;
using System.Threading;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class TestContext : IBloggingContext
    {
        public TestContext()
        {
            this.Blogs = new TestDbSet<Blog>();
            this.Posts = new TestDbSet<Post>();
        }

        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
        public int SaveChangesCount { get; private set; }
        public int SaveChanges()
        {
            this.SaveChangesCount++;
            return 1;
        }
    }

    public class TestDbSet<TEntity> : DbSet<TEntity>, IQueryble, IEnumerable<TEntity>,
        IDbAsyncEnumerable<TEntity>
        where TEntity : class
    {
        ObservableCollection<TEntity> _data;
        IQueryable _query;

        public TestDbSet()
        {
            _data = new ObservableCollection<TEntity>();
            _query = _data.AsQueryable();
        }

        public override TEntity Add(TEntity item)
        {
            _data.Add(item);
            return item;
        }

        public override TEntity Remove(TEntity item)
        {
            _data.Remove(item);
            return item;
        }

        public override TEntity Attach(TEntity item)
        {
            _data.Add(item);
            return item;
        }

        public override TEntity Create()
        {
            return Activator.CreateInstance<TEntity>();
        }

        public override TDerivedEntity Create<TDerivedEntity>()
        {
            return Activator.CreateInstance<TDerivedEntity>();
        }
    }
}

```

```
public override ObservableCollection< TEntity> Local
{
    get { return _data; }
}

Type IQueryable.ElementType
{
    get { return _query.ElementType; }
}

Expression IQueryable.Expression
{
    get { return _query.Expression; }
}

IQueryProvider IQueryable.Provider
{
    get { return new TestDbAsyncQueryProvider< TEntity>(_query.Provider); }
}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return _data.GetEnumerator();
}

IEnumerator< TEntity> IEnumerable< TEntity>.GetEnumerator()
{
    return _data.GetEnumerator();
}

IDbAsyncEnumerator< TEntity> IDbAsyncEnumerable< TEntity>.GetAsyncEnumerator()
{
    return new TestDbAsyncEnumerator< TEntity>(_data.GetEnumerator());
}
}

internal class TestDbAsyncQueryProvider< TEntity> : IDbAsyncQueryProvider
{
    private readonly IQueryProvider _inner;

    internal TestDbAsyncQueryProvider(IQueryProvider inner)
    {
        _inner = inner;
    }

    public IQueryable CreateQuery(Expression expression)
    {
        return new TestDbAsyncEnumerable< TEntity>(expression);
    }

    public IQueryable< TElement> CreateQuery< TElement>(Expression expression)
    {
        return new TestDbAsyncEnumerable< TElement>(expression);
    }

    public object Execute(Expression expression)
    {
        return _inner.Execute(expression);
    }

    public TResult Execute< TResult>(Expression expression)
    {
        return _inner.Execute< TResult>(expression);
    }

    public Task< object> ExecuteAsync(Expression expression, CancellationToken cancellationToken)
    {
        return Task.FromResult(Execute(expression));
    }
}
```

```

        public Task<TResult> ExecuteAsync<TResult>(Expression expression, CancellationToken cancellationToken)
    {
        return Task.FromResult(Execute<TResult>(expression));
    }
}

internal class TestDbAsyncEnumerable<T> : EnumerableQuery<T>, IDbAsyncEnumerable<T>, IQueryable<T>
{
    public TestDbAsyncEnumerable(IEnumerable<T> enumerable)
        : base(enumerable)
    { }

    public TestDbAsyncEnumerable(Expression expression)
        : base(expression)
    { }

    public IDbAsyncEnumerator<T> GetAsyncEnumerator()
    {
        return new TestDbAsyncEnumerator<T>(this.AsEnumerable().GetEnumerator());
    }

    IDbAsyncEnumerator IDbAsyncEnumerable.GetAsyncEnumerator()
    {
        return GetAsyncEnumerator();
    }

    IQueryProvider IQueryable.Provider
    {
        get { return new TestDbAsyncQueryProvider<T>(this); }
    }
}

internal class TestDbAsyncEnumerator<T> : IDbAsyncEnumerator<T>
{
    private readonly IEnumerator<T> _inner;

    public TestDbAsyncEnumerator(IEnumerator<T> inner)
    {
        _inner = inner;
    }

    public void Dispose()
    {
        _inner.Dispose();
    }

    public Task<bool> MoveNextAsync(CancellationToken cancellationToken)
    {
        return Task.FromResult(_inner.MoveNext());
    }

    public T Current
    {
        get { return _inner.Current; }
    }

    object IDbAsyncEnumerator.Current
    {
        get { return Current; }
    }
}
}

```

Реализация поиска

Метод Find трудно реализовать в первоначальном виде. Если необходимо протестировать код, который

делает использование метода Find, проще всего создать тест DbSet для каждого из типов сущностей, которые должны поддерживать поиск. После этого можно написать логику для поиска конкретного типа сущности, как показано ниже.

```
using System.Linq;

namespace TestingDemo
{
    class TestBlogDbSet : TestDbSet<Blog>
    {
        public override Blog Find(params object[] keyValues)
        {
            var id = (int)keyValues.Single();
            return this.SingleOrDefault(b => b.BlogId == id);
        }
    }
}
```

Написание некоторых тестов

Это все, что необходимо сделать, чтобы начать тестирование. Следующий тест создает TestContext, а затем — это служба, исходя из этого контекста. Служба затем используется для создания нового блога — с помощью метода AddBlog. Наконец тест проверяет, что служба добавлен новый блог свойства блоги контекста и вызове метода SaveChanges в контексте.

Это всего лишь примером типа из вещей, которые можно проверить с помощью двойных теста в памяти, и можно настроить логику тестовых дублеров и проверку в соответствии с требованиями.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestingDemo
{
    [TestClass]
    public class NonQueryTests
    {
        [TestMethod]
        public void CreateBlog_saves_a_blog_via_context()
        {
            var context = new TestContext();

            var service = new BlogService(context);
            service.AddBlog("ADO.NET Blog", "http://blogs.msdn.com/adonet");

            Assert.AreEqual(1, context.Blogs.Count());
            Assert.AreEqual("ADO.NET Blog", context.Blogs.Single().Name);
            Assert.AreEqual("http://blogs.msdn.com/adonet", context.Blogs.Single().Url);
            Assert.AreEqual(1, context.SaveChangesCount());
        }
    }
}
```

Вот еще один пример теста — это время, который выполняет запрос. Тест начинается с создания контекста теста с данными в его блоге свойство — Обратите внимание, что данные не в алфавитном порядке. Затем можно создать BlogService, исходя из наших контекста теста и убедитесь, что данные, которые мы получаем от GetAllBlogs упорядочен по имени.

```

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestingDemo
{
    [TestClass]
    public class QueryTests
    {
        [TestMethod]
        public void GetAllBlogs_orders_by_name()
        {
            var context = new TestContext();
            context.Blogs.Add(new Blog { Name = "BBB" });
            context.Blogs.Add(new Blog { Name = "ZZZ" });
            context.Blogs.Add(new Blog { Name = "AAA" });

            var service = new BlogService(context);
            var blogs = service.GetAllBlogs();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

Наконец, мы напишем еще один тест, который использует наших асинхронный метод, чтобы убедиться, что инфраструктура async, мы включили в [TestDbSet](#) работает.

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    [TestClass]
    public class AsyncQueryTests
    {
        [TestMethod]
        public async Task GetAllBlogsAsync_orders_by_name()
        {
            var context = new TestContext();
            context.Blogs.Add(new Blog { Name = "BBB" });
            context.Blogs.Add(new Blog { Name = "ZZZ" });
            context.Blogs.Add(new Blog { Name = "AAA" });

            var service = new BlogService(context);
            var blogs = await service.GetAllBlogsAsync();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

Пригодность для тестирования и Entity Framework 4.0

27.09.2018 • 75 minutes to read • [Edit Online](#)

Скотт Аллен

Опубликовано: Май 2010 г.

Вступление

В этом техническом документе описывается и демонстрируется способ записи тестируемого кода с помощью ADO.NET Entity Framework 4.0 и Visual Studio 2010. В этом документе не пытается сосредоточиться на конкретных тестирования методологии разработки через тестирование (TDD) или разработки на основе поведения (BDD). Вместо этого в этом документе основное внимание уделяется как написать код, который использует ADO.NET Entity Framework, но остается легко изолировать и проверить в автоматическом режиме. Мы рассмотрим распространенные шаблоны разработки, упростить тестирование в данных сценариях доступа и применение этих шаблонов, при использовании платформы. Мы также рассмотрим особенности платформы framework, чтобы увидеть работу этих возможностей в тестируемого кода.

Что такое тестируемого кода?

Возможность проверки компонента с помощью автоматических модульных тестов по предоставляет множество преимуществ нежелательно. Все знают, что хорошие тесты сократит число дефектов программного обеспечения в приложения и увеличение, качество приложения -, однако наличие модульных тестов в месте далеко выходит за рамки, просто обнаружения ошибок.

Набор хороших модульных тестов позволяет группе разработчиков для экономии времени и сохраняете контроль над создаваемое ими программное обеспечение. Команды можно внести изменения существующего кода, рефакторинг и переработки и реструктуризации программного обеспечения к новым требованиям и добавить новые компоненты в приложение при этом знать набор тестов можно проверить поведение приложения. Модульные тесты являются частью цикла быструю обратную связь для упрощения изменений и сохранить удобства обслуживания программного обеспечения, при увеличении сложности.

Модульное тестирование в состав цену, тем не менее. Проектной группе приходится вкладывать время для создания и обслуживания модульных тестов. Трудозатраты, необходимые для создания этих тестов непосредственно связано с **пригодности для тестирования** базового программного обеспечения. Насколько просто программное обеспечение для тестирования? Группы разработки программного обеспечения учитывались возможности тестирования эффективных тестов создается быстрее, чем группа, работающая с нетестируемого программного обеспечения.

Корпорация Microsoft разработала ADO.NET Entity Framework 4.0 (EF4) учитывались возможности тестирования. Это не означает, что разработчики будет записывать модульные тесты для сам код framework. Вместо этого цели тестирования EF4 упрощают создание тестируемого кода, который создает на основе framework. Прежде чем мы рассмотрим конкретные примеры, стоит понять качества кода для тестирования.

Качество тестируемого кода

Код, который легко тестировать всегда будет обладать по крайней мере два признаков. Во-первых, пригодного для тестирования кода легко **наблюдать за**. Учитывая некоторый набор входных данных, должно быть легко наблюдать за результат выполнения кода. Например тестирование следующий метод несложно, так как метод напрямую возвращает результат вычисления.

```
public int Add(int x, int y) {
    return x + y;
}
```

Тестирование метода является сложной задачей, если метод записывает вычисляемое значение в сетевому сокету, таблицы базы данных или файла аналогично следующему коду. Тест должен выполнить дополнительную работу, чтобы получить значение.

```
public void AddAndSaveToFile(int x, int y) {
    var results = string.Format("The answer is {0}", x + y);
    File.WriteAllText("results.txt", results);
}
```

Во-вторых, тестируемого кода легко **изолировать**. Давайте используем следующий псевдокод неверный пример кода для тестирования.

```
public int ComputePolicyValue(InsurancePolicy policy) {
    using (var connection = new SqlConnection("dbConnection"))
    using (var command = new SqlCommand(query, connection)) {

        // business calculations omitted ...

        if (totalValue > notificationThreshold) {
            var message = new MailMessage();
            message.Subject = "Warning!";
            var client = new SmtpClient();
            client.Send(message);
        }
    }
    return totalValue;
}
```

Метод легко наблюдать за — мы можем передать в страховой полис и убедиться, что возвращаемое значение совпадает с ожидаемым результатом. Тем не менее для тестирования метода необходимо установить базу данных с помощью правильную схему и настроить SMTP-сервера в случае, если метод пытается отправить сообщение электронной почты.

Только хочет проверить логику вычислений внутри метода модульного теста, но тест может завершиться ошибкой, поскольку серверу электронной почты находится в автономном режиме, или из-за перемещения сервера базы данных. Оба эти ошибки связаны с поведение, которое желает проверить теста. Поведение трудно локализовать.

Разработчики программного обеспечения, стремиться к созданию кода для тестирования часто стремятся Ведение разделения проблем в коде, что они записывают. Приведенный выше метод необходимо сосредоточиться на бизнес-расчетов и делегировать детали реализации базы данных и адрес электронной почты для других компонентов. Роберт с. Мартин вызывает этот принципа персональной ответственности. Объект следует инкапсулировать один узкий ответственности, скажем, расчет значение политики. Вся остальная работа базы данных и уведомления должно быть ответственность за некоторому другому объекту. Код, написанный таким образом, проще ее изолировать, так как она сосредоточена на одну задачу.

В .NET есть абстракции, нам нужно следовать принципа персональной ответственности и добиться изоляции. Можно использовать интерфейс определения и принудительного кода, чтобы использовать абстракции интерфейс вместо конкретного типа. Далее в этом документе мы рассмотрим, как метод как неправильный приведенного выше примера можно работать с интерфейсы, выглядеть , как они будут обмениваться данными к базе данных. Во время тестирования мы может заменить фиктивные реализация, которая не обращаться к базе данных, но вместо этого хранит данные в памяти. Это фиктивная реализация будет

изолировать код из несвязанных проблем в модуле кода доступа к данным или базы данных конфигурации.

Существуют дополнительные преимущества изоляции. Бизнес-расчета в последний метод обычно занимает лишь несколько миллисекунд для выполнения, но сам тест могут работать на несколько секунд в качестве кода прыжков в сети и взаимодействует с различными серверами. Модульные тесты должны выполняться *fast*, чтобы упростить небольшие изменения. Модульные тесты должны также быть повторяемым и не сбоем, так как возникла проблема с компонентом, не имеющих отношения к тесту. Создание кода, легко наблюдать и изолировать означает, что разработчики получают это облегчит написание тестов для кода, тратить меньше времени на ожидание исполнение тестов и более важно, тратят меньше времени, ошибок, которые не существуют.

Будем надеяться, что можно оценить преимущества тестирования и понять качеств, которые видно, тестируемого кода. Мы должны решить, как написать код, который работает с помощью EF4 для сохранения данных в базу данных, сохраняя при этом наблюдаемых и легко изолировать, но сначала мы будем сузить основное внимание обсудить тестируемые архитектуры для доступа к данным.

Шаблоны разработки для постоянного хранения

В обоих примерах неправильный, представленного выше было слишком большого количества обязанностей. Первый пример неправильный пришлось выполнить расчет и записи в файл. Во втором примере неправильный требовалось считывать данные из базы данных и выполнения расчетов и Отправка сообщения электронной почты. Создав небольших методов, которые разделения функций и делегировать ответственность за другими компонентами вносятся большой шаг вперед для написания тестируемого кода. Целью является построение функциональные возможности путем составления действий с небольшим и направленным абстракции.

Когда дело доходит до постоянного хранения небольшую и фокусом абстракции, которые мы ищем настолько распространено, они были задокументированы как шаблоны проектирования. Книги (Martin Fowler) Patterns of Enterprise Application Architecture была первой работой для описания этих шаблонов в печати. Прежде чем мы покажем, как эти платформы ADO.NET Entity Framework реализует и работает с этими шаблонами, мы предоставим краткое описание этих шаблонов в следующих разделах.

Шаблон репозитория

Фаулер, — говорит репозиторием «является посредником между доменом и данных уровней сопоставления, с помощью интерфейса обобщенное для доступа к объектам домена». Шаблон репозитория призвано изолировать код из рутинной работе доступа к данным, и как мы видели ранее изоляции является обязательным характерной для тестирования.

Ключом к изоляции является, как хранилище предоставляет объекты, с помощью интерфейса коллекции по принципу. Записи можно использовать хранилище имеет ни малейшего представления как логика репозитории материализуется объекты, которые можно запросить. Хранилище может взаимодействовать с базой данных, или он может просто возвращает объекты из коллекции в памяти. Ваш код должен знать всего лишь репозитории отображается для обслуживания коллекции, что можно получить, добавление и удаление объектов из коллекции.

Конкретный репозиторий в существующих приложений .NET часто наследуется от универсального интерфейса следующим образом:

```
public interface IRepository<T> {
    IEnumerable<T> FindAll();
    IEnumerable<T> FindBy(Expression<Func<T, bool>> predicate);
    T FindById(int id);
    void Add(T newEntity);
    void Remove(T entity);
}
```

Сделаем некоторые изменения в определение интерфейса мы предоставляем реализацию для EF4, когда основной принцип остается неизменным. Код с помощью репозиторием конкретные реализации этого интерфейса можно получить сущность по значению первичного ключа, для получения коллекции сущностей, по результатам вычисления предиката, или просто получить все доступные сущности. Код можно также добавлять и удалять сущности через интерфейс репозитория.

Заданных объектов IRepository Employee, код может выполнить следующие операции.

```
var employeesNamedScott =
    repository
        .FindBy(e => e.Name == "Scott")
        .OrderBy(e => e.HireDate);
var firstEmployee = repository.FindById(1);
var newEmployee = new Employee() { /*... */};
repository.Add(newEmployee);
```

Поскольку код использует интерфейс (IRepository сотрудника), мы можем предоставить код с помощью различных реализаций интерфейса. Одна реализация может быть реализацией, предлагая EF4 и сохранение объектов в базу данных Microsoft SQL Server. Другую реализацию (один, который используется во время тестирования) может быть подкреплен сотрудника из списка объектов в памяти. Интерфейс поможет добиться изоляции в коде.

Обратите внимание, что IRepository<T> интерфейс не предоставляет операции сохранения. Как обновить существующие объекты? Часто встречаются IRepository определений, которые включают операции сохранения и реализации этих репозиториев необходимо немедленно сохранить объект в базу данных. Тем не менее во многих приложениях мы не хотим сохранять объекты по отдельности. Вместо этого мы хотим вдохните жизнь в объекты, возможно из разных репозиториев, изменить эти объекты как часть бизнес-деятельности и затем сохранить все объекты в рамках одной атомарной операции. К счастью есть шаблон, чтобы разрешить этот тип поведения.

Шаблон единицы работы

Фаулер, — говорит единица работы «сохранит список объектов зависит от бизнес-транзакции и координирует записи изменений и разрешение проблем параллелизма». Это ответственность за единицу работы для отслеживания изменений объектов мы воплотить в жизнь из репозитория и сохранять любые изменения, внесенных в объекты, когда сообщается о единице работы для фиксации изменений. Это также отвечают за единицу работы для новых объектов, мы добавили ко всем репозиториям и вставить объекты в базе данных, а также управление ими удаления.

Если вы выполняли никакой работы с наборами данных ADO.NET затем вы будете уметь шаблон единицы работы. Наборы данных ADO.NET имел возможность отслеживать наши обновления, удаления и вставки объектов DataRow и удалось (с помощью адаптера таблицы) выверить все изменения в базе данных. Тем не менее объекты DataSet модель отключенных подмножество основной базе данных. Шаблон единицы работы выполняет то же поведение, но работает с бизнес-объекты и объекты домена, которые изолированы от кода доступа к данным и без базы данных.

Абстракция для моделирования единицу работы, в коде .NET может выглядеть следующим образом:

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<Order> Orders { get; }
    IRepository<Customer> Customers { get; }
    void Commit();
}
```

Предоставляя ссылки репозитория из единицу работы, можно обеспечить единый рабочий объект имеет возможность отслеживать все сущности, материализуется во время бизнес-транзакции. Метод Commit для

реальных единицы работы реализуется все Магия для согласования изменений в памяти с базой данных.

Получает ссылку на IUnitOfWork, код можно внести изменения в бизнес-объекты, полученные из одного или нескольких репозиториев и сохранить все изменения, с помощью атомарные операции фиксации.

```
var firstEmployee = unitofWork.Employees.FindById(1);
var firstCustomer = unitofWork.Customers.FindById(1);
firstEmployee.Name = "Alex";
firstCustomer.Name = "Christopher";
unitofWork.Commit();
```

Шаблон отложенной загрузки

Фаулер использует имя отложенной загрузки для описания «объект, который не содержит все данные, вам требуется, но знает, как его получить». Прозрачный отложенная загрузка является важной характеристикой для после написания кода, пригодного для тестирования бизнеса и работа в реляционной базе данных.

Например рассмотрим следующий код.

```
var employee = repository.FindById(id);
// ... and later ...
foreach(var timeCard in employee.TimeCards) {
    // .. manipulate the timeCard
}
```

Как табелях коллекция заполняется? Существует два возможных варианта ответа. Один ответ – что хранилище сотрудника, в ответ на запрос для получения сотрудника, выдает запрос для получения сотрудника вместе с сотрудниками связанные сведения. В реляционных базах данных, это обычно требуется запрос с предложением JOIN и может повлечь за извлечение больше информации, чем приложения должен. Что делать, если приложение никогда не должно touch свойство табелях?

Ответ на второй вопрос — загрузка свойство табелях «по требованию». Эта отложенная загрузка прозрачно для бизнес-логики и неявные потому, что код не вызывает специальные API, чтобы получить сведения. В коде предполагается, что сведения о карте время присутствует, при необходимости. Чудеса с отложенной загрузки, который обычно включает перехват вызовов методов среды выполнения. Перехват кода несет ответственность за обращения к базе данных и получение сведений о времени карты оставив бизнес-логика может быть бизнес-логики. Эти невероятные вещи отложенную загрузку позволяет бизнес-кода для изоляции сам с операциями получения данных и чему больший объем кода, пригодного для тестирования.

Недостатком вызывает отложенную загрузку является, когда приложение *does* необходимые сведения о карте времени, код будет выполняться дополнительный запрос. Это не является проблемой для многих приложений, но для производительности уязвимых приложений или приложений цикла по несколько сотрудников объектов и выполнения запроса для получения карточек времени во время каждой итерации цикла (проблемы часто называют N + 1 задачи с запросами), отложенная загрузка — это перетаскивания. В этих сценариях приложение может потребоваться заблаговременной загрузки сведения в наиболее эффективным способом невозможно.

К счастью мы увидим, как EF4 поддерживает оба неявное отложенной загрузки и эффективный интенсивно загружается информация по мере перемещения в следующем разделе и реализации этих шаблонов.

Реализация шаблонов с платформой Entity Framework

Хорошо то, что все шаблоны разработки, описанные в предыдущем разделе, просто реализуется с помощью EF4. Для демонстрации мы собираемся использовать простое приложение ASP.NET MVC для редактирования и отображения сотрудников и их связанные сведения. Мы начнем с помощью следующих «старые объекты CLR» (POCO).

```

public class Employee {
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime HireDate { get; set; }
    public ICollection<TimeCard> TimeCards { get; set; }
}

public class TimeCard {
    public int Id { get; set; }
    public int Hours { get; set; }
    public DateTime EffectiveDate { get; set; }
}

```

Эти определения классов немного меняется, как мы рассмотрим различные подходы и особенностей EF4, но цель состоит в том, чтобы сохранить эти классы как игнорирующих сохраняемость (PI) максимально. Объект PI не знает *как*, или даже *Если*, состояние, в ней находится внутри базы данных. PI и РОСО идут рука об руку с тестируемого программного обеспечения. Объекты, с использованием подхода РОСО являются менее ограниченного более гибким и проще тестировать, поскольку они могут работать без базы данных существует.

С помощью РОСО на месте мы можем создать Entity Data Model (EDM) в Visual Studio (см. рис. 1). Мы не будем использовать для создания кода для наших сущностей модели EDM. Вместо этого мы хотим использовать сущностей, которые мы любовно создавать вручную. Модель EDM используется только для создания схемы базы данных и введите метаданные EF4 должен сопоставить объекты в базе данных.

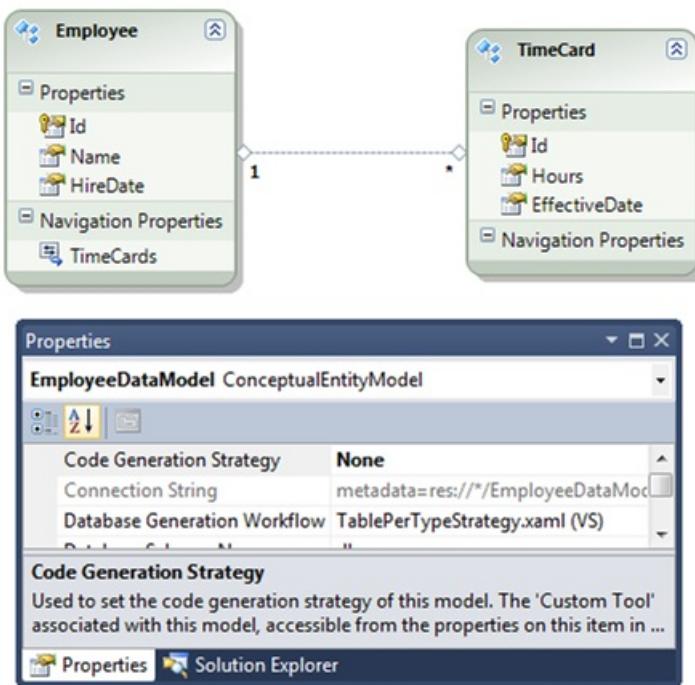


Рис. 1

Примечание: Если вы хотите разрабатывать модель EDM, во-первых, это невозможно создать очистки код РОСО из модели EDM. Это можно сделать с помощью расширения Visual Studio 2010, предоставляемые группе программирования данных. Чтобы скачать расширение, запустите диспетчер расширений из меню "Сервис" в Visual Studio и выполните поиск веб-коллекции шаблонов «РОСО» (см. рис. 2). Существует несколько шаблонов РОСО для Entity FRAMEWORK. Дополнительные сведения об использовании шаблона см. в разделе "[Пошаговое руководство: шаблон РОСО для Entity Framework](#)".

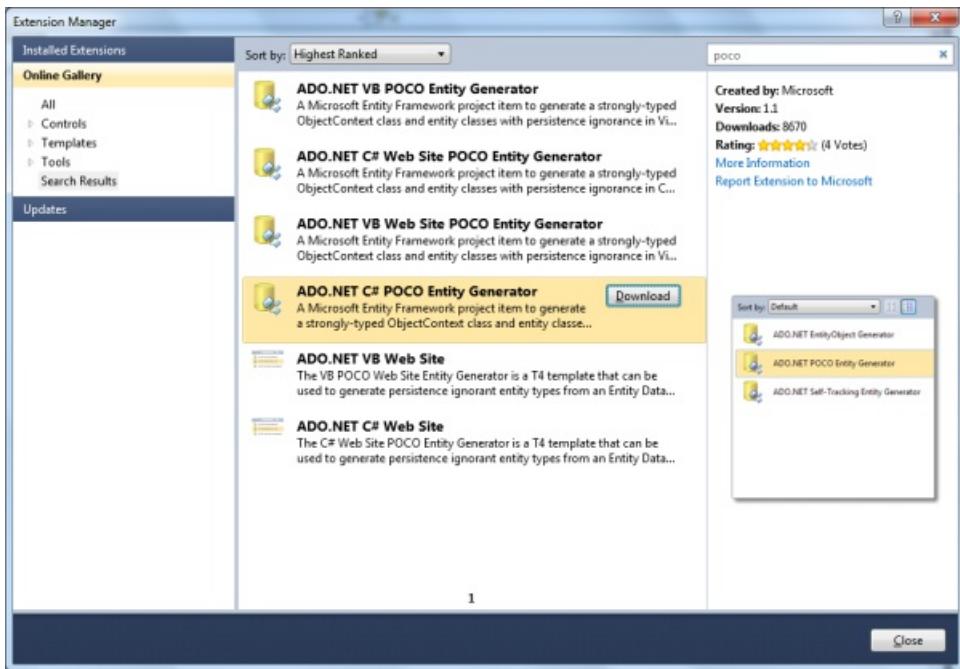


Рис. 2

Из этого POCO, начальная точка мы рассмотрим два различных подхода для тестируемого кода. Первый подход, я вызываю метод EF, так как он использует абстракции от Entity Framework API для реализации единицы работы и репозиториев. Второй способ мы создадим собственных абстракций пользовательский репозиторий и затем см. в разделе преимущества и недостатки каждого подхода. Мы начнем с изучения подход EF.

Ориентированные на реализацию EF

Рассмотрим следующее действие контроллера из проекта ASP.NET MVC. Действие извлекает объект Employee и возвращает результат, чтобы отобразить подробное представление сотрудника.

```
public ViewResult Details(int id) {
    var employee = _unitOfWork.Employees
        .Single(e => e.Id == id);
    return View(employee);
}
```

Можно протестировать код? Существует по крайней мере два теста, необходимо проверить его поведение. Во-первых мы бы хотели убедиться, что это действие возвращает правильный режим — простым тестом. Мы бы также требовалось написать тест, чтобы проверить действие извлекает правильный сотрудника, и мы бы хотели сделать это без выполнения кода в базе данных. Помните, что нам нужно изолировать тестируемый код. Изоляция будет убедиться, что тест не произошел сбой из-за ошибки в модуле кода доступа к данным или базы данных конфигурации. Если тест не пройден, мы знаем, что у нас есть ошибки в логике контроллера, а не в некоторых нижнего уровня системного компонента.

Для обеспечения изоляции, потребуются некоторые абстракции, как и интерфейсы, которые мы представляем выше для репозиториев и единиц работы. Помните, что шаблон репозитория предназначен в качестве посредника между объектами домена и уровень сопоставления данных. В этом сценарии EF4 — слоя сопоставления данных и уже предоставляет абстракцию хранилища с именем помошью `IObjectSet<T>` (из пространства имен `System.Data.Objects`). Определение интерфейса выглядит следующим образом.

```

public interface IObjectSet<TEntity> :
    IQueryable<TEntity>,
    IEnumerable<TEntity>,
    IQueryable,
    IEnumerable
    where TEntity : class
{
    void AddObject(TEntity entity);
    void Attach(TEntity entity);
    void DeleteObject(TEntity entity);
    void Detach(TEntity entity);
}

```

Помощью `IObjectSet<T>` соответствует требованиям для репозитория, так как он напоминает коллекции объектов (через интерфейс `IEnumerable<T>`) и предоставляет методы для добавления и удаления объектов из коллекции имитации. Методы присоединения и отсоединения предоставляют дополнительные возможности EF4 API. Для использования с помощью `IObjectSet<T>` как интерфейс для репозиториев, мы должны единицу работы абстракции для связать репозитории вместе.

```

public interface IUnitOfWork {
    IObjectSet<Employee> Employees { get; }
    IObjectSet<TimeCard> TimeCards { get; }
    void Commit();
}

```

Одну конкретную реализацию этого интерфейса будет обращаться к SQL Server и можно легко создать с помощью класса `ObjectContext` с помощью EF4. Класс `ObjectContext` является реальной единицей работы в EF4 API.

```

public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        var connectionString =
            ConfigurationManager
                .ConnectionStrings[ConnectionStringName]
                .ConnectionString;
        _context = new ObjectContext(connectionString);
    }

    public IObjectSet<Employee> Employees {
        get { return _context.CreateObjectSet<Employee>(); }
    }

    public IObjectSet<TimeCard> TimeCards {
        get { return _context.CreateObjectSet<TimeCard>(); }
    }

    public void Commit() {
        _context.SaveChanges();
    }

    readonly ObjectContext _context;
    const string ConnectionStringName = "EmployeeDataModelContainer";
}

```

Перенос с помощью `IObjectSet<T>` в жизнь так же прост, что и вызов метод `CreateObjectSet` объекта `ObjectContext`. За кулисами платформа будет использовать метаданные мы предоставили в модели EDM для создания конкретных `ObjectSet<T>`. Поместим при возврате помощью `IObjectSet<T>` интерфейс, так как он поможет сохранить пригодности для тестирования в клиентском коде.

Это конкретная реализация полезно в рабочей среде, но нам нужно сосредоточиться на том, как мы будем

использовать наши IUnitOfWork абстракции для упрощения тестирования.

Тестовых дублеров

Чтобы изолировать действия контроллера нам потребуется возможность переключаться между реальными единицами работы (поддерживаемый ObjectContext) и тест double или «фальшивая» единица работы (выполнение операций в памяти). Распространенный подход для выполнения такого рода переключение является не могут создать экземпляр единицы работы, но вместо этого передайте единицу работы в качестве параметра конструктора контроллеров MVC.

```
class EmployeeController : Controller {
    public EmployeeController(IUnitOfWork unitOfWork) {
        _unitOfWork = unitOfWork;
    }
    ...
}
```

Приведенный выше код является примером внедрения зависимостей. Мы не позволяем на контроллере, чтобы создать его зависимостей (единица работы), но внедрения зависимости в контроллер. В проекте MVC довольно часто использовать фабрику настраиваемый контроллер в сочетании с инверсией управления (IoC) контейнера для автоматизации внедрения зависимостей. Эти темы выходят за рамки данной статьи, но можно прочитать по более следующие ссылки в конце этой статьи.

Единое фиктивные реализации работы, который можно использовать для тестирования может выглядеть следующим образом.

```
public class InMemoryUnitOfWork : IUnitOfWork {
    public InMemoryUnitOfWork() {
        Committed = false;
    }
    public IObjectSet<Employee> Employees {
        get;
        set;
    }

    public IObjectSet<TimeCard> TimeCards {
        get;
        set;
    }

    public bool Committed { get; set; }
    public void Commit() {
        Committed = true;
    }
}
```

Обратите внимание, что фиктивные единицу работы предоставляет свойство были применены. Иногда полезно добавить компоненты в класс фиктивными, упростить тестирование. В этом случае проще наблюдать, если код фиксирует единицу работы, путем проверки свойства были применены.

Нам также потребуется фальшивые помощью IObjectSet<T> для хранения сотрудников и TimeCard объектов в памяти. Мы можем предоставить единую реализацию использований универсальных шаблонов.

```

public class InMemoryObjectSet<T> : IObjectSet<T> where T : class
{
    public InMemoryObjectSet()
        : this(Enumerable.Empty<T>()) {
    }

    public InMemoryObjectSet(IEnumerable<T> entities) {
        _set = new HashSet<T>();
        foreach (var entity in entities) {
            _set.Add(entity);
        }
        _queryableSet = _set.AsQueryable();
    }

    public void AddObject(T entity) {
        _set.Add(entity);
    }

    public void Attach(T entity) {
        _set.Add(entity);
    }

    public void DeleteObject(T entity) {
        _set.Remove(entity);
    }

    public void Detach(T entity) {
        _set.Remove(entity);
    }

    public Type ElementType {
        get { return _queryableSet.ElementType; }
    }

    public Expression Expression {
        get { return _queryableSet.Expression; }
    }

    public IQueryProvider Provider {
        get { return _queryableSet.Provider; }
    }

    public IEnumerator<T> GetEnumerator() {
        return _set.GetEnumerator();
    }

    IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }

    readonly HashSet<T> _set;
    readonly IQueryable<T> _queryableSet;
}

```

Этот тестовый дублер делегирует большинство операций на базовый класс `HashSet<T>` объекта. Обратите внимание, помошью `IObjectSet<T>` требует универсального ограничения применения `T` в качестве класса (ссылочный тип), а также заставляет нас реализацию `IQueryable<T>`. Это легко сделать в коллекцию в памяти, которые отображаются как `IQueryable<T>` с помощью стандартного оператора LINQ `AsQueryable`.

Тесты

Традиционные модульные тесты будут использовать одного класса тестов для хранения всех тестов для всех действий в один контроллер MVC. Мы можем написать эти тесты, или любого типа модульного теста, используя в памяти имитаций мы создали. Однако в этой статье, в которой мы будем избегать монолитного приложения тестирования подход на основе класса и сгруппировать связанные наших тестов, чтобы сосредоточиться на определенной части функциональных возможностей. Например, «создание нового сотрудника» может быть функциональные возможности, которые мы хотим проверить, поэтому мы будем использовать одного класса тестов, чтобы проверить, как одного контроллера работает отвечает за создание нового сотрудника.

Есть некоторые распространенные код настройки, необходимые для все эти детально тестовых классов.

Например необходимо всегда создать наших репозиториях в памяти и фальшивые единицу работы.

Необходимо также экземпляра контроллера сотрудника с фиктивными единицу работы вставлен. Мы будем совместного использования общего кода установки тестовых классов с помощью базового класса.

```

public class EmployeeControllerTestBase {
    public EmployeeControllerTestBase() {
        _employeeData = EmployeeObjectMother.CreateEmployees()
            .ToList();
        _repository = new InMemoryObjectSet<Employee>(_employeeData);
        _unitOfWork = new InMemoryUnitOfWork();
        _unitOfWork.Employees = _repository;
        _controller = new EmployeeController(_unitOfWork);
    }

    protected IList<Employee> _employeeData;
    protected EmployeeController _controller;
    protected InMemoryObjectSet<Employee> _repository;
    protected InMemoryUnitOfWork _unitOfWork;
}

```

Мы используем в базовом классе «объект мама» — один из распространенных шаблонов для создания тестовых данных. Мама объект содержит фабричные методы для создания экземпляра сущности теста для использования в нескольких средствах тестирования.

```

public static class EmployeeObjectMother {
    public static IEnumerable<Employee> CreateEmployees() {
        yield return new Employee() {
            Id = 1, Name = "Scott", HireDate=new DateTime(2002, 1, 1)
        };
        yield return new Employee() {
            Id = 2, Name = "Poonam", HireDate=new DateTime(2001, 1, 1)
        };
        yield return new Employee() {
            Id = 3, Name = "Simon", HireDate=new DateTime(2008, 1, 1)
        };
    }
    // ... more fake data for different scenarios
}

```

Мы используем EmployeeControllerTestBase как базовый класс для числа средств тестирования (см. рис. 3). Каждый средство тестирования определяет свое действие контроллера. Например, одно средство тестирования основное внимание уделяется тестирование создания действия, используемое при выполнении запроса HTTP GET (чтобы отобразить представление для создания сотрудника), и другой тестовый стенд основное внимание уделяется созданию действия, используемое в запросе HTTP POST (будет извлекать сведения, отправленные пользователю для создания сотрудника). Каждый производный класс отвечает только за настройки, необходимые в его определенном контексте, а также для предоставления утверждения, необходимые для проверки результатов для контекста конкретного теста.

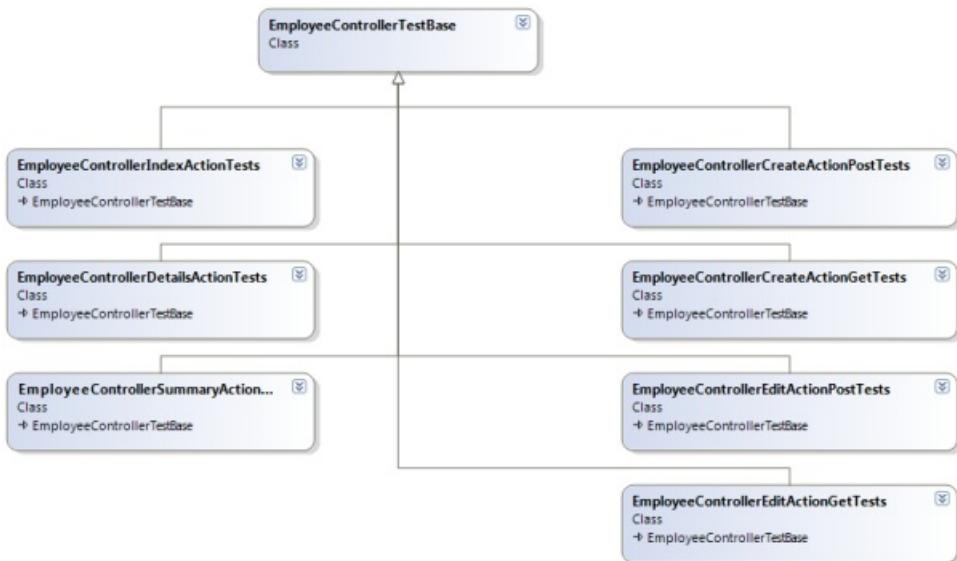


Рис. 3

Соглашение о вызовах и тестирования стиля именования представленные здесь, не является обязательным для тестируемого кода — это лишь одним из подходов. Рис. 4 показано, что тесты, выполняемые в Resharper полуширье Jet тестов средство запуска подключаемого модуля для Visual Studio 2010.

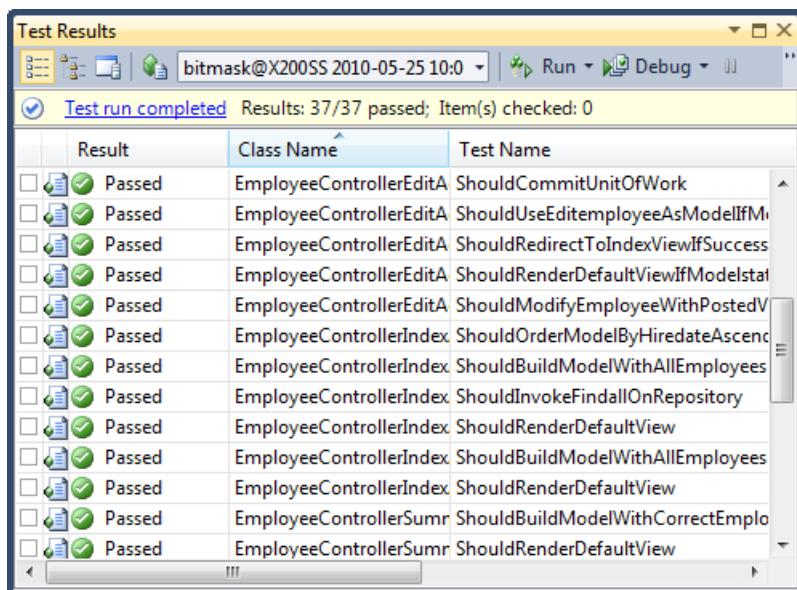


Рис. 4

С базовым классом для обработки ее кода сетевую установку модульные тесты для каждого действия контроллера небольшим и простым для записи. Тесты будут выполняться быстро (поскольку мы занимаемся операций в памяти) и не должны завершаться сбоем из-за несвязанных инфраструктуры или охраной окружающей среды (поскольку мы определили, что модульного теста).

```

[TestClass]
public class EmployeeControllerCreateActionPostTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldAddNewEmployeeToRepository() {
        _controller.Create(_newEmployee);
        Assert.IsTrue(_repository.Contains(_newEmployee));
    }
    [TestMethod]
    public void ShouldCommitUnitOfWork() {
        _controller.Create(_newEmployee);
        Assert.IsTrue(_unitOfWork.Committed);
    }
    // ... more tests

    Employee _newEmployee = new Employee() {
        Name = "NEW EMPLOYEE",
        HireDate = new System.DateTime(2010, 1, 1)
    };
}

```

В этих тестах базовый класс не выполняет большую часть работы программы установки. Помните, что конструктор базового класса создает хранилище в памяти, фиктивный единица работы и экземпляр класса EmployeeController. Тестовый класс является производным от этого базового класса и фокусируется на особенности тестирования метода Create. В этом случае особенности подразделяются на «упорядочить, действовать и утвердить» шаги, которые можно будет увидеть в любой модульное тестирование процедуры:

- Создайте объект newEmployee для имитации входящих данных.
- Вызвать действие Create EmployeeController и передайте newEmployee.
- Убедитесь, что действие создания получить желаемые результаты, (сотрудника отображается в репозитории).

Мы создали позволяет нам протестировать EmployeeController действия. Например при написании тестов для действия Index контроллера сотрудника мы может наследовать от базового класса теста, чтобы установить эту же базовая установка для наших тестов. Еще раз базовый класс создаст хранилище в памяти, фиктивный единица работы и экземпляр EmployeeController. Тесты для действия индекса только необходимо сосредоточиться на вызове действия индекса и тестирование качества модели действие возвращает.

```

[TestClass]
public class EmployeeControllerIndexActionTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldBuildModelWithAllEmployees() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.Count() == _employeeData.Count);
    }
    [TestMethod]
    public void ShouldOrderModelByHiredateAscending() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.SequenceEqual(
            _employeeData.OrderBy(e => e.HireDate)));
    }
    // ...
}

```

Тесты создаются с помощью fakes в памяти на достижение тестирования *состояние программного*

обеспечения. Например при тестировании действия создания, нам нужно проверять состояние репозитория, после выполнения действия создания — репозиторий хранит новый сотрудник?

```
[TestMethod]
public void ShouldAddNewEmployeeToRepository() {
    _controller.Create(_newEmployee);
    Assert.IsTrue(_repository.Contains(_newEmployee));
}
```

Позже мы рассмотрим взаимодействие на основе тестирования. Тестирование взаимодействия на основе запрашивает Если тестируемый код вызывается соответствующие методы объектов и получает правильные параметры. Сейчас мы перейдем на обложке другой design pattern — отложенной загрузки.

Безотложная загрузка и отложенная загрузка

В определенный момент в ASP.NET MVC веб-приложение, которое мы можем потребоваться просмотреть информацию сотрудника, а также включать сотрудника связанные время карты. Например может использоваться карта времени отображения сводки, который отображает имя сотрудника и общее количество времени карт в системе. Существует несколько подходов, которые мы можем предпринять для реализации этой возможности.

Проекция

— Это один из простых способов создания сводки для построения модели, выделенные с данными, которые нужно отобразить в представлении. В этом случае модель может выглядеть следующим образом.

```
public class EmployeeSummaryViewModel {
    public string Name { get; set; }
    public int TotalTimeCards { get; set; }
}
```

Обратите внимание, что EmployeeSummaryViewModel не является сущностью, — другими словами это не то, что мы должны сохраняться в базе данных. Только мы собираемся использовать этот класс распределяет данные в представление в режиме строгой типизации. Передача данных объекта (DTO), так как в нем нет поведения (методы не) — только свойства аналогична модели представления. Свойства будет содержать данные, необходимые для перемещения. Его можно легко создать экземпляр этой модели представления, с помощью LINQ проекции стандартный оператор — оператор Select.

```
public ViewResult Summary(int id) {
    var model = _unitOfWork.Employees
        .Where(e => e.Id == id)
        .Select(e => new EmployeeSummaryViewModel
        {
            Name = e.Name,
            TotalTimeCards = e.TimeCards.Count()
        })
        .Single();
    return View(model);
}
```

Существуют две важные функции в приведенный выше код. Сначала — код можно легко проверить, так как он по-прежнему позволяет легко наблюдать и изолировать. Оператор Select точно так же работает с нашей fakes в памяти, как в случае с реальной единицей работы.

```
[TestClass]
public class EmployeeControllerSummaryActionTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldBuildModelWithCorrectEmployeeSummary() {
        var id = 1;
        var result = _controller.Summary(id);
        var model = result.ViewData.Model as EmployeeSummaryViewModel;
        Assert.IsTrue(model.TotalTimeCards == 3);
    }
    // ...
}
```

Вторая функция, важные заключается в том, как код позволяет EF4 для формирования запроса единый, эффективных сборка сотрудника и времени сведения о карте друг с другом. Мы загрузили данные о сотрудниках и сведения в тот же объект без использования любые специальные интерфейсы API. Код просто выражен сведения, что он требует использования стандартных операторов LINQ, которые работают с источниками данных в памяти, а также удаленным источникам данных. EF4 была возможность перевода дерева выражений, создаваемый запрос LINQ и C# компилятора в единый и эффективный запрос T-SQL.

```
SELECT
[Limit1].[Id] AS [Id],
[Limit1].[Name] AS [Name],
[Limit1].[C1] AS [C1]
FROM (SELECT TOP (2)
[Project1].[Id] AS [Id],
[Project1].[Name] AS [Name],
[Project1].[C1] AS [C1]
FROM (SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Name] AS [Name],
(SELECT COUNT(1) AS [A1]
    FROM [dbo].[TimeCards] AS [Extent2]
    WHERE [Extent1].[Id] =
        [Extent2].[EmployeeTimeCard_TimeCard_Id]) AS [C1]
    FROM [dbo].[Employees] AS [Extent1]
    WHERE [Extent1].[Id] = @p__linq__0
) AS [Project1]
) AS [Limit1]
```

Существуют в других случаях, когда мы не хотим работать с модели представления или объект DTO, но реальные сущности. Когда мы знаем, нам нужно сотрудник и карт время сотрудника, мы можем заранее загрузить связанные данные ненавязчивого и эффективным образом.

Явная Безотложная загрузка

Когда мы хотим заранее загрузить сведения о связанной сущности, необходимо, чтобы какой-либо механизм для бизнес-логики (или в этом случае логика действия контроллера) для выражения необходимость в репозиторий. Для класса ObjectQuery EF4<T> класс определяет метод `Include`, чтобы задать связанные объекты должны быть получены при запросе. Помните, EF4 ObjectContext предоставляет сущности через конкретный `ObjectSet<T>` класс, который наследуется от `ObjectQuery<T>`. Если мы использовали `ObjectSet<T>` ссылок в наши действия контроллера, можно было бы написать следующий код, чтобы указать диапазон безотложной загрузки сведения для каждого сотрудника.

```
_employees.Include("TimeCards")
    .Where(e => e.HireDate.Year > 2009);
```

Тем не менее, так как мы пытаемся хранить наш код пригодного для тестирования мы не предоставляет `ObjectSet<T>` из за пределами реальных класс рабочих единиц. Вместо этого мы полагаемся на помощь

`IObjectSet<T>` интерфейс, который гораздо проще подделки, но помошью `IObjectSet<T>` не определяет метод `Include`. Красота LINQ — это, можно создать собственный оператор `Include`.

```
public static class QueryableExtensions {
    public static IQueryble<T> Include<T>
        (this IQueryble<T> sequence, string path) {
        var objectQuery = sequence as ObjectQuery<T>;
        if(objectQuery != null)
        {
            return objectQuery.Include(path);
        }
        return sequence;
    }
}
```

Обратите внимание на то, как метод расширения определен оператор `Include` для `IQueryble<T>` вместо помошью `IObjectSet<T>`. Это дает возможность использовать метод с более широкий диапазон возможных типов, включая `IQueryble<T>`, помошью `IObjectSet<T>`, `ObjectQuery<T>` и `ObjectSet<T>`. В случае базовой последовательности, не является подлинной `ObjectQuery EF4<T>`, то нет никакого вреда Готово, и оператор `Include` является холостой. Если основной последовательности — `ObjectQuery<T>` (или производный от `ObjectQuery<T>`), то EF4 будет см. в разделе требования дополнительные данные и формулировка правильного SQL запрос.

С помощью этого нового оператора, в месте мы явно запросить диапазон безотложной загрузки сведения из репозитория.

```
public ViewResult Index() {
    var model = _unitOfWork.Employees
        .Include("TimeCards")
        .OrderBy(e => e.HireDate);
    return View(model);
}
```

При выполнении для реальных `ObjectContext`, код создает следующий один запрос. Запрос собирает достаточно информации из базы данных в один обращения к материализации объектов `employee` и полностью заполнить свойству их труда.

```

SELECT
[Project1].[Id] AS [Id],
[Project1].[Name] AS [Name],
[Project1].[HireDate] AS [HireDate],
[Project1].[C1] AS [C1],
[Project1].[Id1] AS [Id1],
[Project1].[Hours] AS [Hours],
[Project1].[EffectiveDate] AS [EffectiveDate],
[Project1].[EmployeeTimeCard_TimeCard_Id] AS [EmployeeTimeCard_TimeCard_Id]
FROM ( SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Name] AS [Name],
[Extent1].[HireDate] AS [HireDate],
[Extent2].[Id] AS [Id1],
[Extent2].[Hours] AS [Hours],
[Extent2].[EffectiveDate] AS [EffectiveDate],
[Extent2].[EmployeeTimeCard_TimeCard_Id] AS
[EmployeeTimeCard_TimeCard_Id],
CASE WHEN ([Extent2].[Id] IS NULL) THEN CAST(NULL AS int)
ELSE 1 END AS [C1]
FROM [dbo].[Employees] AS [Extent1]
LEFT OUTER JOIN [dbo].[TimeCards] AS [Extent2] ON [Extent1].[Id] = [Extent2].
[EmployeeTimeCard_TimeCard_Id]
) AS [Project1]
ORDER BY [Project1].[HireDate] ASC,
[Project1].[Id] ASC, [Project1].[C1] ASC

```

Хорошая новость в том, остается полностью тестируемый код внутри метода действия. Не требуется предоставить дополнительные компоненты для наших fakes для поддержки оператора Include. Плохая новость заключается в том, что нам было необходимо использовать оператор Include внутри кода, мы хотели бы сохранить игнорирующих сохраняемость. Это — отличный пример типа необходимых действий, которые необходимо оценить при построении тестируемого кода. Бывают случаи, когда необходимо разрешить сохраняемости проблемы утечки за пределами абстракции репозитория в соответствии с целевыми показателями производительности.

Альтернативой Безотложной загрузке является отложенной загрузки. Отложенная загрузка означает, что мы делаем не требуется код нашей бизнес-логики явно объявить о требование для связанных данных. Вместо этого мы используем наш сущностей в приложении и ли дополнительные данные, необходимые Entity Framework будет загружать данные по запросу.

Отложенная загрузка

Очень просто Представьте себе ситуацию, где мы не знаем, что потребуется данных часть бизнес-логики. Мы знаем, логика должна объект employee, но мы может выполнять переход в разные пути выполнения которых некоторые из этих путей требуют сведения от сотрудника, а некоторые нет. Такие сценарии, как это идеально подходят для неявного отложенной загрузки, так как данные волшебным образом отображаются по мере необходимости.

Отложенная загрузка, также известный как отложенная загрузка, поместите некоторые требования объектов сущности. Любые требования из уровня сохраняемости не сопряжено с POCO с обеспечением сохраняемости значение true, но true сохраняемости практически невозможно достичь. Вместо этого мы измеряем сохраняемости в градусах относительный. Было бы нежелательным, если нужно наследовать от базового класса ориентированного сохраняемости или использовать специализированные коллекции для достижения отложенной загрузки в POCO. К счастью EF4 содержит щадящим решения.

Практически невозможно обнаружить

При работе с объектами POCO, EF4 может динамически создавать прокси-серверы среди выполнения для сущностей. Эти учетные записи-посредники незаметно wrap материализованные POCO и предоставить дополнительные службы, перехватывая каждого свойства, которые получают операций установки и

выполнить дополнительные действия. Из таких служб — это возможность отложенной загрузки, которую мы ищем. Другая служба — это эффективный механизм, который можно записать, когда она изменяет значения свойств сущности отслеживания изменений. Список изменений используется контекст `ObjectContext` во время выполнения метода `SaveChanges` для сохранения любого измененных сущностей, с помощью команды обновления.

Для этих учетных записей-посредников для работы Однако они нужен способ обработки свойства `get` и набора операций на сущности и прокси-серверы достижения этой цели путем переопределения виртуальных членов. Таким образом Если мы хотим использовать неявную отложенную загрузку и эффективное отслеживание изменений необходимо вернуться к нашей определения классов РОСО и пометки свойства как виртуальный.

```
public class Employee {
    public virtual int Id { get; set; }
    public virtual string Name { get; set; }
    public virtual DateTime HireDate { get; set; }
    public virtual ICollection<TimeCard> TimeCards { get; set; }
}
```

Мы по-прежнему может сообщать, что сущность сотрудника является главным образом игнорирующих сохраняемость. Единственное требование заключается в том, чтобы использовать виртуальные члены, и это не влияет на пригодность для тестирования кода. Мы не должны наследовать от любого специального базового класса, или даже использовать специальную подборку, выделенных для отложенной загрузки. Как показано в коде, любой класс, реализующий интерфейс `ICollection<T>` доступен для хранения связанных сущностей.

Имеется также одним небольшим изменением, что необходимо сделать внутри нашей единицей работы. Отложенная загрузка — это `off` по умолчанию при работе непосредственно с объекта `ObjectContext`. Свойства, которые можно установить для свойства `ContextOptions` для включения отложенной загрузки, и мы можем установить это свойство внутри наших реальных единица работы, если необходимо, чтобы везде отложенной загрузки.

```
public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        // ...
        _context = new ObjectContext(connectionString);
        _context.ContextOptions.LazyLoadingEnabled = true;
    }
    // ...
}
```

Неявные отложенная загрузка включена, код приложения можно использовать сотрудника и сотрудника связанных карт время сохраняя при этом не нужно знать о работы, необходимой для Entity FRAMEWORK загрузить дополнительные данные.

```
var employee = _unitOfWork.Employees
    .Single(e => e.Id == id);
foreach (var card in employee.TimeCards) {
    // ...
}
```

Отложенная загрузка упрощает код приложения для записи и с помощью волшебной команды прокси-сервера остается полностью пригодного для тестирования кода. Fakes в памяти, единицы работы просто можно предварительно загрузить фиктивные сущности с связанные данные, при необходимости во время теста.

На этом этапе мы обратим наше внимание от создания репозиториев с помощью помощью `IObjectSet<T>` и посмотрите на абстракции, чтобы скрыть все знаки платформы сохраняемости.

Пользовательские репозитории

Когда мы впервые представлена шаблон единицы работы конструктора в этой статье мы предоставили пример кода для как может выглядеть единицу работы. Давайте снова представить эту идею исходного, с помощью сотрудников и карта времени ситуацию с сотрудниками, мы работали с.

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<TimeCard> TimeCards { get; }
    void Commit();
}
```

— Это основное различие между этой единицей работы и единицей работы, мы создали в предыдущем разделе, как эта единица работы не использовать такие абстракции, с помощью EF4 framework (имеется не помощью `IObjectSet<T>`). Помощью `IObjectSet<T>` работает, а также интерфейс репозитория, но он предоставляет API могут быть не согласованы полностью с потребностями приложения. При таком подходе предстоящих будем представлять репозитории, с помощью пользовательских `IRepository<T>` абстракции.

Многие разработчики, которым следуют разработки через тестирование, проектирование на основе поведения и методологии разработки предметно-ориентированное предпочитают `IRepository<T>` подход по следующим причинам. Во-первых, `IRepository<T>` интерфейс представляет слой «защиты от повреждения». Как описано в своей книге проблемно-ориентированное проектирование (Eric Evans) уровень защиты от повреждения хранит код предметной области от инфраструктуры API-интерфейсы, такие как API сохраняемости. Во-вторых разработчики могут создавать методы в репозиторий, отвечают потребностям приложения, (обнаруженные во время записи тестов). Например мы часто может потребоваться найти одну сущность, с помощью значений Идентификаторов, поэтому мы можем добавить метод `FindById` интерфейс репозитория. Наши `IRepository<T>` определение будет выглядеть следующим образом.

```
public interface IRepository<T>
    where T : class, IEntity {
    IQueryable<T> FindAll();
    IQueryable<T> FindWhere(Expression<Func<T, bool>> predicate);
    T FindById(int id);
    void Add(T newEntity);
    void Remove(T entity);
}
```

Обратите внимание на то, будет отправлено обратно с помощью `IQueryable<T>` интерфейс для предоставления коллекции сущностей. `IQueryable<T>` позволяет деревьях выражений LINQ, поступающие в EF4 поставщика и предоставление поставщика целостное представление запроса. Второй вариант будет возвращать объект `IEnumerable<T>`, это означает, поставщик EF4 LINQ будет видеть только выражения, построенные в репозитории. Любой группирования, сортировки и проекции, вне репозитория будет состоять не в команду SQL, отправленную в базу данных, которая может вызвать снижение производительности. С другой стороны, хранилище, возвращая только `IEnumerable<T>` результаты будут никогда не удивлять с помощью новой команды SQL. Оба подхода будут работать, и оба подхода остаются пригодного для тестирования.

Это просто для предоставления единую реализацию `IRepository<T>` интерфейс, с помощью универсальных типов и EF4 `ObjectContext` API.

```

public class SqlRepository<T> : IRepository<T>
{
    where T : class, IEntity {
        public SqlRepository(ObjectContext context) {
            _objectSet = context.CreateObjectSet<T>();
        }
        public IQueryable<T> FindAll() {
            return _objectSet;
        }
        public IQueryable<T> FindWhere(
            Expression<Func<T, bool>> predicate) {
            return _objectSet.Where(predicate);
        }
        public T FindById(int id) {
            return _objectSet.Single(o => o.Id == id);
        }
        public void Add(T newEntity) {
            _objectSet.AddObject(newEntity);
        }
        public void Remove(T entity) {
            _objectSet.DeleteObject(entity);
        }
        protected ObjectSet<T> _objectSet;
    }
}

```

IRepository<T> подход дает нам некоторые дополнительный контроль над нашими запросами так, как клиент должен вызвать метод, чтобы перейти к сущности. Внутри метода мы предоставляем дополнительные проверки и операторы LINQ для принудительного применения ограничения для приложения. Обратите внимание, что интерфейс имеет два ограничения для параметра универсального типа. Первое ограничение – недостатки класс нарушить проверку необходимых ObjectSet<T>, и второе ограничение принудительно сущностями для реализации IEntity — это абстракция, созданного для приложения. Интерфейс IEntity заставляет сущностей в удобном для чтения свойств Id, и можно затем использовать это свойство в метод FindById. IEntity определяется следующим кодом.

```

public interface IEntity {
    int Id { get; }
}

```

IEntity можно считать небольшой нарушение сохраняемости, поскольку сущностями необходимы для реализации этого интерфейса. Помните, принцип независимости сохраняемости очень о необходимости поиска компромиссов, и для многих функций FindById перевешивают ограничение, наложенное интерфейс. Интерфейс не оказывает влияния на пригодность для тестирования.

Создание экземпляра динамической IRepository<T> требуется контекст ObjectContext EF4, поэтому конкретные единицы реализации рабочих должен управлять создание экземпляра.

```

public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        var connectionString =
            ConfigurationManager
                .ConnectionStrings[ConnectionStringName]
                .ConnectionString;

        _context = new ObjectContext(connectionString);
        _context.ContextOptions.LazyLoadingEnabled = true;
    }

    public IRepository<Employee> Employees {
        get {
            if (_employees == null) {
                _employees = new SqlRepository<Employee>(_context);
            }
            return _employees;
        }
    }

    public IRepository<TimeCard> TimeCards {
        get {
            if (_timeCards == null) {
                _timeCards = new SqlRepository<TimeCard>(_context);
            }
            return _timeCards;
        }
    }

    public void Commit() {
        _context.SaveChanges();
    }
}

SqlRepository<Employee> _employees = null;
SqlRepository<TimeCard> _timeCards = null;
readonly ObjectContext _context;
const string ConnectionStringName = "EmployeeDataModelContainer";
}

```

С помощью пользовательского репозитория

С помощью нашего пользовательского репозитория незначительно отличается от использования репозитория, в соответствии с помощью `IObjectSet<T>`. Вместо применения операторов LINQ непосредственно к свойству, мы сначала необходимо вызвать один методы репозитория, чтобы скопировать `IQueryable<T>` ссылки.

```

public ViewResult Index() {
    var model = _repository.FindAll()
        .Include("TimeCards")
        .OrderBy(e => e.HireDate);
    return View(model);
}

```

Обратите внимание на то, что пользовательский оператор `Include`, которую мы применили ранее будет работать без изменений. Метод `FindByid` репозитория удаляет повторяющиеся логики из действия, при попытке извлечь одну сущность.

```

public ViewResult Details(int id) {
    var model = _repository.FindByid(id);
    return View(model);
}

```

Нет нет существенной разницы в двух подходов, которые мы рассмотрели возможности тестирования. Мы можем решить фиктивных реализаций из `IRepository<T>`, создав конкретные классы, поддерживаемый `HashSet<сотрудника>` — так же, как мы сделали в последнем разделе. Тем не менее некоторые разработчики предпочитают использовать макеты объектов и макетирования платформ объекта вместо сборки fakes. Мы рассмотрим использование макетов для тестирования нашей реализации и обсудим различия между макетами и имитацией в следующем разделе.

Тестирование с помощью макетов

Существуют различные способы создания какие вызовы (Martin Fowler) «тестовый дублер». Тестовый дублер (например, фильма *stunt double*) — это объект, создаваемых «вместо» для реальных рабочих объектов во время выполнения тестов. Репозитории в памяти, созданной нами — тестовых дублеров для репозиториев, которые обращаются к SQL Server. Мы узнали, как использовать эти тестовые дублеры во время модульных тестов для изоляции кода и сохранить быстрому запуску тестов.

Тестовых дублеров, которые мы создали имеют реализации реальных, работа. За кулисами каждый из них хранит коллекцию объектов, и они будут добавлять и удалять объекты из данной коллекции, как мы управляем хранилище во время теста. Некоторые разработчики предпочитаю создавать их тестовых дублеров таким образом — с реальный код и работу реализации. Эти тестовые дублеры так называемые *fakes*. Они имеют рабочей реализации, но они не реальных, для использования в рабочей среде. Имитацией хранилища фактически выполняет запись в базе данных. Фиктивные SMTP-сервера не фактически отправляет сообщение электронной почты по сети.

Макеты и имитаций

Есть другой тип теста, `double`, известный как *макетирование*. Хотя fakes рабочий реализаций, макеты поставляются с нет. С помощью платформы макетов объектов создается во время выполнения эти макеты объектов и использовать их в качестве тестовых дублеров. В этом разделе мы будем использовать макетирование framework Moq открытым кодом. Ниже приведен простой пример с помощью Moq для динамического создания теста `double` для репозиторием сотрудника.

```
Mock< IRepository<Employee>> mock =
    new Mock< IRepository<Employee>>();
 IRepository<Employee> repository = mock.Object;
 repository.Add(new Employee());
 var employee = repository.FindById(1);
```

Мы запрашиваем `IRepository Moq<сотрудника>` реализации и он создает его динамически. Мы можем получить для объекта, реализующего `IRepository<сотрудника>` путем обращения к свойству объекта фиктивного объекта `<T>` объекта. Это этот внутренний объект, которые можно передать в наши контроллеры, и он может не знать, если это тестовый дублер или реальные репозитории. Мы можно вызывать методы в объекте, так же, как мы бы вызывать методы на объект с фактической реализацией.

Вы должны спросить, что будете выполнять фиктивного репозитория, когда мы вызываем метод `Add`. Так как не реализацию макета объекта, добавить не выполняет никаких действий. Нет конкретных коллекции за кулисами, как у нас было с помощью fakes, который был записан, чтобы сотрудник удаляется. Как насчет возвращаемое значение `FindById`? В этом случае макета объекта выполняет единственное, что его можно сделать, это возвращаемое значение по умолчанию. Так как мы возвращается ссылочным типом (`сотрудник`), возвращается значение `null`.

Макеты может показаться бесполезным; Тем не менее существуют две дополнительные возможности макетов, которые мы не затрагивали. Во-первых Moq framework записывает все вызовы, выполняемые для макета объекта. Далее в коде мы просим Moq Если любой пользователь вызывается метод `Add`, или в том случае, если любой пользователь вызывает метод `FindById`. Мы рассмотрим, как можно использовать эту функцию «черного ящика» запись в тестах.

Второй отличная возможность заключается в том, как можно использовать Moq программировать макет

объекта с ожидания. Ожидание указывает макета объекта как отвечать на любые данного действия. Например мы программируем ожидание в наш макет и сообщить его, чтобы вернуть объект employee, когда кто-то вызывает FindById. Моq инфраструктура использует API установки и лямбда-выражения для программировать эти ожидания.

```
[TestMethod]
public void MockSample() {
    Mock< IRepository< Employee >> mock =
        new Mock< IRepository< Employee >>();
    mock.Setup(m => m.FindById(5))
        .Returns(new Employee { Id = 5 });
    IRepository< Employee > repository = mock.Object;
    var employee = repository.FindById(5);
    Assert.IsTrue(employee.Id == 5);
}
```

В этом примере мы просим Моq динамически создать репозиторий, а затем мы используем при программировании репозиторий с Ожидание. Исходя из предположения о том, макета объекта для возврата объекта сотрудника с идентификатором 5, когда кто-то вызывает метод FindById, передавая значение 5. Этот тест проходит успешно, и нам не нужно создавать полную реализацию, чтобы поддельное IRepository< T >.

Давайте вернемся к тестов, который был записан ранее и переработать их на использование макетов вместо fakes. Так же, как раньше, мы будем использовать базовый класс для настройки общие части инфраструктуры, которые нужны для всех тестов контроллера.

```
public class EmployeeControllerTestBase {
    public EmployeeControllerTestBase() {
        _employeeData = EmployeeObjectMother.CreateEmployees()
            .AsQueryable();
        _repository = new Mock< IRepository< Employee >>();
        _unitOfWork = new Mock< IUnitOfWork >();
        _unitOfWork.Setup(u => u.Employees)
            .Returns(_repository.Object);
        _controller = new EmployeeController(_unitOfWork.Object);
    }

    protected IQueryable< Employee > _employeeData;
    protected Mock< IUnitOfWork > _unitOfWork;
    protected EmployeeController _controller;
    protected Mock< IRepository< Employee >> _repository;
}
```

Программный код установки большей части остается неизменным. Вместо использования fakes, мы будем использовать Моq для создания макетов объектов. Базовый класс упорядочивает макетов единицу работы для возврата фиктивного репозитория, когда код вызывает свойство сотрудников. Остальная часть макета настройки будет иметь место внутри средства тестирования, выделенных для каждого конкретного сценария. Например средство тестирования для действия индекса настроим фиктивного репозитория, возвращающий список сотрудников, когда действие вызывает метод FindAll фиктивного репозитория.

```

[TestClass]
public class EmployeeControllerIndexActionTests
    : EmployeeControllerTestBase {
    public EmployeeControllerIndexActionTests() {
        _repository.Setup(r => r.FindAll())
            .Returns(_employeeData);
    }
    // ... tests
    [TestMethod]
    public void ShouldBuildModelWithAllEmployees() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.Count() == _employeeData.Count());
    }
    // ... and more tests
}

```

За исключением ожиданий наши тесты выглядеть тесты, которые мы использовали ранее. Тем не менее с возможностью записи макетов платформы мы может приблизить тестирования с другого ракурса. Мы рассмотрим это новая Перспектива в следующем разделе.

Состояния и тестирование взаимодействия

Существуют различные методы, которые можно использовать для тестирования программного обеспечения макеты объектов. Одним из подходов является использование состояний на основе тестирования, это, что мы проделали в настоящей документации в данный момент. Состояние на основе тестирования делает утверждения о состоянии программного обеспечения. В последней проверки мы вызова метода действия контроллера и утверждение о модели, он должен быть построен. Ниже приведены некоторые другие примеры тестирования состояния.

- Убедитесь, что хранилище содержит новый объект employee, после выполнения Create.
- Убедитесь, что модель содержит список всех сотрудников, после выполнения индекса.
- Убедитесь, что хранилище не содержит данного сотрудника после выполнения удаления.

Другой подход, вы увидите с макеты объектов — для проверки взаимодействия. Хотя состояния на основе тестирования делает предположения о состоянии объектов, взаимодействие на основе тестирования делает утверждения о взаимодействии объектов. Пример:

- Убедитесь, что контроллер вызывает метод Add репозитория при выполнении Create.
- Убедитесь, что контроллер вызывает метод FindAll репозитория, при выполнении индекса.
- Убедитесь, что контроллер вызывает единицы работы метода Commit для сохранения изменений, при выполнении редактирования.

Тестирование взаимодействия часто требуется меньше проверочных данных, так как мы не поковыряться внутри коллекции, а также проверка счетчиков. Например если мы знаем, что на действие Details вызывает метод FindById репозитория с правильным значением - нажмите действие, вероятно, работает правильно. Мы можем проверить это без настройки любые тестовые данные для возврата из FindById.

```

[TestClass]
public class EmployeeControllerDetailsActionTests
    : EmployeeControllerTestBase {
    // ...
    [TestMethod]
    public void ShouldInvokeRepositoryToFindEmployee() {
        var result = _controller.Details(_detailsId);
        _repository.Verify(r => r.FindById(_detailsId));
    }
    int _detailsId = 1;
}

```

Только установки, необходимый в выше средство тестирования состоит в настройке, предоставляемых базовым классом. Когда вызывается действие контроллера, Moq приводит к записи взаимодействия с фиктивного репозитория. С помощью API проверки из Moq, мы можем попросить Moq Если контроллер вызывается `FindById` с правильным значением идентификатора. Если контроллер не происходит вызов метода, или вызывается метод со значением непредвиденный параметр, проверьте метод вызовет исключение, и тест завершится ошибкой.

Вот еще один пример, чтобы убедиться, что действие создания вызывает фиксации в текущей единице работы.

```

[TestMethod]
public void ShouldCommitUnitOfWork() {
    _controller.Create(_newEmployee);
    _unitOfWork.Verify(u => u.Commit());
}

```

Один недостаток тестирование взаимодействия является тенденции к через определяют взаимодействие. Способность макета объекта для записи и убедитесь, что любое взаимодействие с макета объекта не значит, тест следует попытаться проверить каждого диалога. Некоторые взаимодействия являются подробностями реализации и необходимо только проверить взаимодействия требуется для удовлетворения текущего теста.

Выбор между макеты и имитаций во многом зависит от системы, тестировании и личную (или объединении адаптеров) предпочтения. Макеты объектов может значительно сократить объем кода, необходимо реализовать тестовых дублеров, но не всем удобна, программирование ожиданиям и проверке взаимодействия.

Выходы

В этом документе мы рассмотрели несколько подходов к созданию кода для тестирования при использовании ADO.NET Entity Framework для постоянного хранения. Можно использовать встроенные абстракций наподобие помощью `IObjectSet<T>`, или создания собственных абстракций наподобие `IRepository<T>`. В обоих случаях поддержка РОКО в Entity Framework ADO.NET 4.0 позволяет пользователям эти абстракции сохраняются неведении и широкими возможностями тестирования. Дополнительные возможности EF4 неявную отложенную загрузку позволяет приложению и бизнес-службы, как код работать, не заботясь о деталях реляционного хранилища данных. Наконец эти абстракции, которые будут созданы просты в макетируйте и не Имитируйте внутри модульных тестов, и мы можем использовать эти тестовые дублеры для достижения быстрое выполнение, высокой изолированным и надежные тесты.

Дополнительные ресурсы

- Роберт с. Мартин, " [принцип единственной обязанности](#)"
- (Martin Fowler) [каталог шаблонов из шаблоны Enterprise Application Architecture](#)
- Caprio Гриффин " [внедрения зависимостей](#)"

- Блог программистов данных, « [Пошаговое руководство: разработка с помощью Entity Framework 4.0 на основе тестирования](#)».
- Блог программистов данных, « [шаблонов с помощью репозитория и единица работы с Entity Framework 4.0](#)»
- Дейв Astels, " [введение BDD](#)"
- Йенсен Аарон " [Общие сведения о спецификации машин](#)"
- Эрик ли " [BDD с использованием MSTest](#)"
- (Eric Evans), " [предметно-ориентированное проектирование](#)"
- (Martin Fowler) « [макетов не только заглушки](#)»
- (Martin Fowler) « [тестирования двойной](#)»
- Джереми Миллер " [состояние и тестирование взаимодействия](#)"
- [MOQ](#)

Биография

Скотт Аллен является членом технического персонала в Pluralsight и учредитель OdeToCode.com. В 15-летним коммерческой разработки программных продуктов Скотт участвовал в решения для всех операций из 8-разрядных устройств embedded позволяет высокомасштабируемых веб-приложений ASP.NET. Скоттом можно связаться на его блог по адресу OdeToCode или в Твиттере <http://twitter.com/OdeToCode> .

Создание модели

18.09.2018 • 5 minutes to read • [Edit Online](#)

Модель EF хранит сведения о сопоставлении классов и свойств приложений с таблицами и столбцами баз данных. Модель EF можно создать двумя способами.

- **С помощью Code First:** разработчик пишет код, чтобы указать модель. Во время выполнения EF создает модели и сопоставления на основе классов сущностей и дополнительной конфигурации модели, предоставленных разработчиком.
- **С помощью Entity Framework Designer:** разработчик рисует поля и строки для указания модели с помощью Entity Framework Designer. Результирующая модель хранится в виде XML-файла с расширением EDMX. Объекты предметной области приложения обычно создаются автоматически на основе концептуальной модели.

Рабочие процессы EF

Оба этих подхода можно применять для использования в целевой базе данных или создания базы данных, что в итоге составляет 4 различных рабочих процесса. Узнайте, какие из них подходят вам лучше всего.

	МНЕ НУЖНО ПРОСТО НАПИСАТЬ КОД...	Я ХОЧУ ИСПОЛЬЗОВАТЬ КОНСТРУКТОР...
Я создаю базу данных	Используйте Code First для определения модели в коде, а затем создайте базу данных.	Используйте Model First для определения модели с помощью полей и строк, а затем создайте базу данных.
Мне требуется доступ к существующей базе данных	Используйте Code First для создания модели на основе кода, которая сопоставляется с существующей базой данных.	Используйте Database First для создания модели полей и строк, которая сопоставляется с существующей базой данных.

Просмотрите видео о выборе подходящего рабочего процесса EF.

В этом коротком видеоролике объясняются различия рабочих процессов и приводятся советы по выбору оптимального процесса.

Представляет: Роэн Миллер (Rowan Miller)



[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Если после просмотра видеоролика вы все еще не можете решить, что будете использовать — Entity Framework Designer или Code First, — изучите оба подхода.

Взгляд изнутри

Независимо от того, используете ли вы Code First или Entity Framework Designer, модель EF всегда состоит из нескольких компонентов.

- Объекты предметной области приложения или сами типы сущностей. Этот уровень часто называют уровнем объектов.

- Концептуальная модель, состоящая из соответствующих предметной области типов сущностей и связей, описываемых с помощью [модели EDM](#). Этот уровень часто указывается с буквой *C*, означающей *conceptual* (*концептуальный*).
- Модель хранения, представляющая таблицы, столбцы и связи, как определено в базе данных. Этот уровень часто указывается с буквой *S*, означающей *S* (*хранение*).
- Сопоставление концептуальной модели со схемой базы данных. Это сопоставление, часто называют сопоставлением *C-S*.

Подсистема сопоставления EF использует сопоставление *C-S* для преобразования операций с сущностями (создание, чтение, обновление и удаление) в эквивалентные операции для таблиц в базе данных.

Сопоставление концептуальной модели и объектов приложения часто называют сопоставлением *O-C*. По сравнению с сопоставлением *C-S* сопоставление *O-C* является неявным и поддерживающим тип "один к одному": сущности, свойства и связи, определенные в концептуальной модели, необходимы для соответствия фигурам и типам объектов .NET. В EF4 и более поздних версиях уровень объектов может состоять из простых объектов со свойствами без зависимостей от EF. Они обычно называются традиционными объектами среды CLR (POCO), и сопоставление типов и свойств выполняется на основе соглашений о соответствии имен. Ранее в EF 3.5 существовали определенные ограничения для уровня объектов, например, сущности должны быть производными от класса `EntityObject` и должны использовать атрибуты EF для реализации сопоставления *O-C*.

Code First в новой базе данных

27.09.2018 • 18 minutes to read • [Edit Online](#)

В этом пошаговом руководстве видео и пошаговые познакомят вас с разработки Code First, предназначенные для новой базы данных. Этот сценарий включает предназначенные для базы данных, которая не существует и Code First создаст пустую базу данных, Code First добавит новые таблицы для. Во-первых, код позволяет определить модель с помощью C## или классам VB.Net. Дополнительная настройка при необходимости выполняются с помощью атрибутов для классов и свойств или с помощью текущего API.

Просмотреть видео

Этот видеоролик представляет собой введение разработки Code First, предназначенные для новой базы данных. Этот сценарий включает предназначенные для базы данных, которая не существует и Code First создаст пустую базу данных, Code First добавит новые таблицы для. Во-первых, код позволяет определить модель с помощью классов C# или VB.Net. Дополнительная настройка при необходимости выполняются с помощью атрибутов для классов и свойств или с помощью текущего API.

Представляет: Роэн Миллер (Rowan Miller)

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Для выполнения этого пошагового руководства необходимо иметь по крайней мере Visual Studio 2010 или Visual Studio 2012.

Если вы используете Visual Studio 2010, также необходимо будет иметь [NuGet](#) установлен.

1. Создание приложения

Для простоты мы собираемся создать простое консольное приложение, которое использует Code First для осуществления доступа к данным.

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в меню слева и **консольного приложения**
- Введите **CodeFirstNewDatabaseSample** как имя
- Нажмите кнопку **OK**

2. Создание модели

Давайте определим очень простую модель с помощью классов. Мы просто определяем их в файл Program.cs, но в случае реального приложения должно разбивать классы out в отдельных файлах и, возможно, отдельный проект.

Ниже определении класса программы в файле Program.cs добавьте следующие два класса.

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}

```

Вы заметите, что мы создаем два свойства навигации (Blog.Posts и Post.Blog) виртуальный. Это позволяет функции отложенной загрузки в Entity Framework. Отложенная загрузка означает, что содержимое этих свойств будет автоматически загружаться из базы данных при попытке доступа к ним.

3. Создайте контекст

Пришло время для определения производного контекста, который представляет сеанс подключения с базой данных, позволяет запрашивать и сохранять данные. Мы определяем контекст, который является производным от System.Data.Entity.DbContext и предоставляет типизированный DbSet< TEntity > для каждого класса в нашей модели.

Теперь мы запускаем следует использовать типы из Entity Framework, поэтому нам нужно добавить пакет EntityFramework NuGet.

- **Проект —> управление пакетами NuGet...** Примечание: Если у вас нет **управление пакетами NuGet...** параметр, необходимо установить [последнюю версию NuGet](#)
- Выберите **Online** вкладку
- Выберите **EntityFramework** пакета
- Нажмите кнопку **установки**

Добавить с помощью инструкции для System.Data.Entity в верхней части Program.cs.

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

Под Post класса в файле Program.cs добавьте следующие производного контекста.

```

public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}

```

Ниже приведен полный листинг что еще теперь должно содержать файл Program.cs.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;

namespace CodeFirstNewDatabaseSample
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }

    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }
}
```

Это весь код, который необходимо запустить, хранения и извлечения данных. Очевидно, что есть довольно много происходит за кулисами, и мы рассмотрим краткий обзор, что в момент, однако сначала посмотрим на это в действии.

4. Чтение и запись данных

Реализуйте метод Main в файле Program.cs, как показано ниже. Этот код создает новый экземпляр класса наш контекст, а затем использует его для вставки нового блога. Затем он использует запрос LINQ для извлечения из базы данных, представлены в алфавитном порядке по названию все блоги.

```

class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

Теперь можно запустить приложение и протестировать его.

```

Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...

```

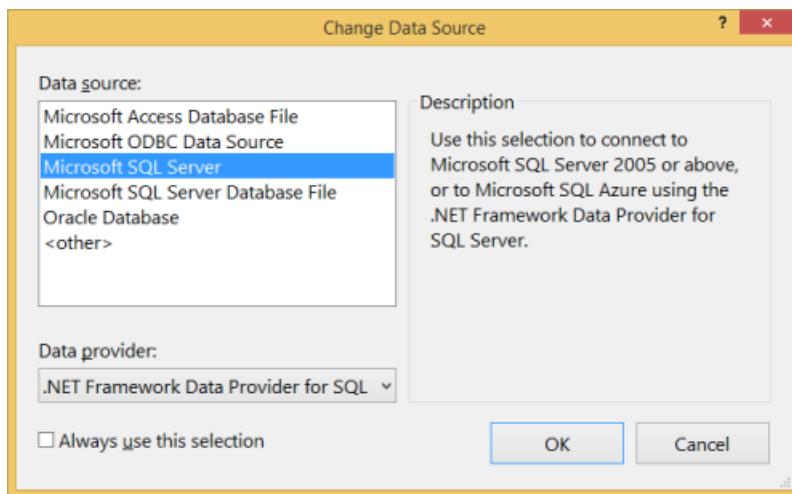
Где мои данные?

По соглашению DbContext автоматически создала базу данных.

- Если локальный экземпляр SQL Express (устанавливается по умолчанию вместе с Visual Studio 2010) затем Code First базы данных на этом экземпляре создается
- Если SQL Express недоступен, то Code First будет попробуйте и использовать [LocalDB](#) (устанавливается по умолчанию с помощью Visual Studio 2012)
- База данных называется после полное имя производного контекста, в нашем случае это **CodeFirstNewDatabaseSample.BloggingContext**

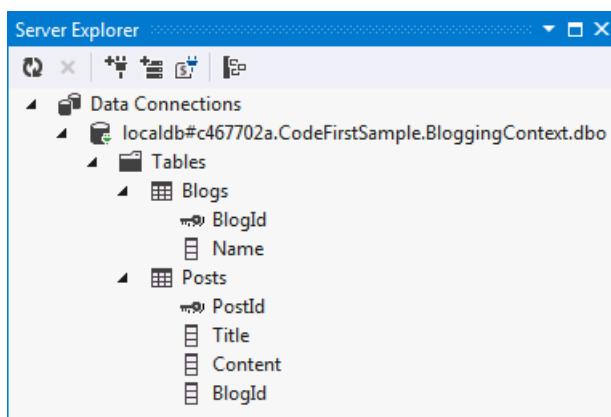
Это просто соглашения по умолчанию существуют различные способы изменения базы данных, который использует Code First, Дополнительные сведения можно найти в **как DbContext обнаруживает модель и подключение к базе данных** раздела. Можно подключиться к этой базе данных, с помощью обозревателя серверов в Visual Studio

- **Представление —> обозревателя серверов**
- Щелкните правой кнопкой мыши **подключения к данным** и выберите **добавить соединение...**
- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам потребуется выбрать в качестве источника данных Microsoft SQL Server



- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них установки

Теперь можно проверить, Code First создает схему.



Классы для включения в модель, просмотрев свойства DbSet, которые мы определили работы DbContext. Затем набор соглашения Code First, по умолчанию используется для определения имен таблиц и столбцов, определения типов данных, найти первичные ключи и т. д. Далее в этом пошаговом руководстве мы рассмотрим, как можно переопределить эти соглашения.

5. Изменения модели

Теперь пора внести некоторые изменения в нашей модели, когда мы внести эти изменения, необходимо также обновить схему базы данных. Для этого мы собираемся использовать средство Code First Migrations и миграции для краткости.

Миграция позволяет нам упорядоченный набор шагов, которые описывают обновление (или понижение) нашей схемы базы данных. Каждая из этих действий, известных как миграцию, содержит код, который описывает изменения, применяемые.

Первым шагом является включение Code First Migrations для наших BloggingContext.

- Средства —> диспетчер пакетов библиотеки -> консоль диспетчера пакетов**
- Запустите команду **Enable-Migrations** в консоли диспетчера пакетов.
- Новая папка миграции был добавлен в наш проект, содержащий два элемента:
 - Configuration.cs** — этот файл содержит параметры, которые будут использовать миграции для переноса BloggingContext. Нет необходимости изменять что-либо в этом пошаговом руководстве, но Вот где можно указать начальное значение данных, поставщики register для других баз данных, изменяет пространство имен что миграции создаются в и т.д.

- <Метка времени>_InitialCreate.cs – это первую миграцию, он представляет изменения, которые уже были применены к базы данных, пользоваться не пустую базу данных, который включает в себя блогов и записей таблицы . Несмотря на то, что мы позволим Code First автоматически создавать эти таблицы для нас, раз уж мы выбрали для миграций, они были преобразованы в миграции. Код сначала также записывается в нашей локальной базы данных, что уже был применен такой миграции. Метка времени в имени файла используется для сортировки целей.

Теперь давайте внести изменения в нашей модели, добавьте в класс блог свойства URL-адреса:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public virtual List<Post> Posts { get; set; }
}
```

- Запустите **AddUrl Add-Migration** команду в консоли диспетчера пакетов. Команда Add-Migration проверяет наличие изменений с момента последнего перехода и формирует шаблоны новой миграции с любыми изменениями, которые находятся. Мы можем дать миграций имени; в этом случае мы называем миграции «AddUrl». Шаблонный код говорит, что нам нужно добавить столбец URL-адрес, который может содержать строковые данные, dbo. Таблица блоги. При необходимости, мы может изменить сформированный код, но в данном случае это не обязательно.

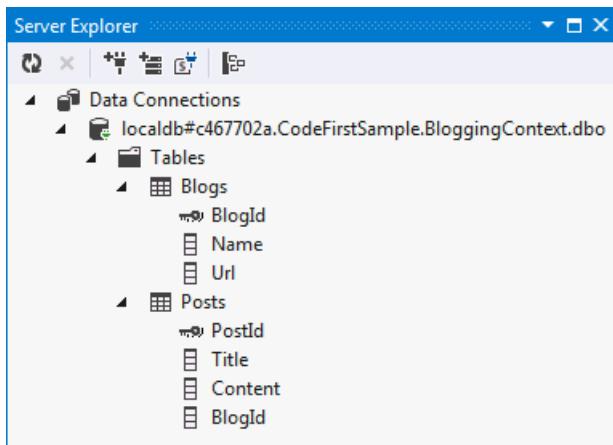
```
namespace CodeFirstNewDatabaseSample.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddUrl : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Blogs", "Url", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Blogs", "Url");
        }
    }
}
```

- Запустите **Update-Database** команду в консоли диспетчера пакетов. Эта команда применит все незавершенные миграции к базе данных. Миграция это InitialCreate уже был применен, поэтому миграция будет просто применить нашей новой миграции AddUrl. Совет: Можно использовать – **Verbose** при вызове Update-Database, чтобы увидеть SQL, который выполняется в базе данных.

Теперь новый столбец URL-адрес добавляется в блоги таблицу в базе данных:



6. Заметки к данным

Пока мы просто позволили EF обнаруживать модели с помощью его соглашения по умолчанию, но будут ли раз, когда наших классов не соответствуют соглашениям об, и нам нужно иметь возможность дальнейшей настройки. Существует два варианта. Мы рассмотрим заметок к данным в этом разделе, а затем текущего API в следующем разделе.

- Добавим к нашей модели пользовательского класса

```
public class User
{
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- Необходимо также добавить набор в нашей производного контекста

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }
}
```

- Если мы попытались добавить миграцию, то получили бы ошибку "EntityType «User» имеет ключ не определен. Определите ключ для этого типа EntityType." так как EF не имеет возможности узнать, что имя пользователя должно быть первичный ключ для пользователя.
- Мы собираемся теперь использовать заметки к данным, поэтому нам нужно добавить с помощью оператора в верхней части Program.cs

```
using System.ComponentModel.DataAnnotations;
```

- Теперь Добавление заметок свойство Username, чтобы определить, что он является первичным ключом

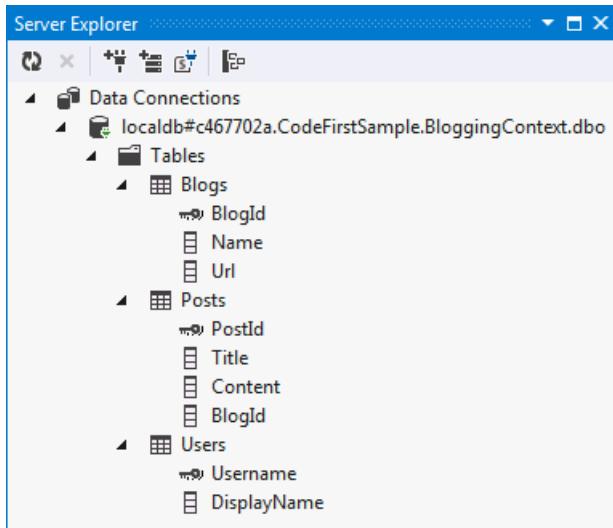
```
public class User
{
    [Key]
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- Используйте **AddUser Add-Migration** команду, чтобы сформировать шаблон миграции для

применения этих изменений в базу данных

- Запустите **Update-Database** команду, чтобы применить созданную миграцию к базе данных

Теперь новая таблица добавляется в базу данных:



Приведен полный список заметок, поддерживаемых EF.

- [KeyAttribute](#)
- [StringLengthAttribute](#)
- [MaxLengthAttribute](#)
- [ConcurrencyCheckAttribute](#)
- [RequiredAttribute](#)
- [TimestampAttribute](#)
- [ComplexTypeAttribute](#)
- [ColumnAttribute](#)
- [TableAttribute](#)
- [InversePropertyAttribute](#)
- [ForeignKeyAttribute](#)
- [DatabaseGeneratedAttribute](#)
- [NotMappedAttribute](#)

7. Текущий API

В предыдущем разделе мы рассмотрели использование заметок к данным для дополнения или переопределить, что была обнаружена по соглашению. Другой способ настройки модели — с помощью Code First fluent API.

Большинство конфигураций модели можно сделать с помощью простых данных заметок. Текущий API — это более сложных способ указания модели конфигурации, который охватывает все, что заметки к данным можно делать в дополнение к некоторые более сложные конфигурации, недоступные в заметки к данным. Заметки к данным и текущий API могут использоваться совместно.

Для доступа к fluent API переопределить метод `OnModelCreating` в `DbContext`. Предположим, мы хотели бы переименовать столбец, который `User.DisplayName` хранится в для отображения_имя.

- Переопределите метод `OnModelCreating` на `BloggingContext` следующим кодом

```

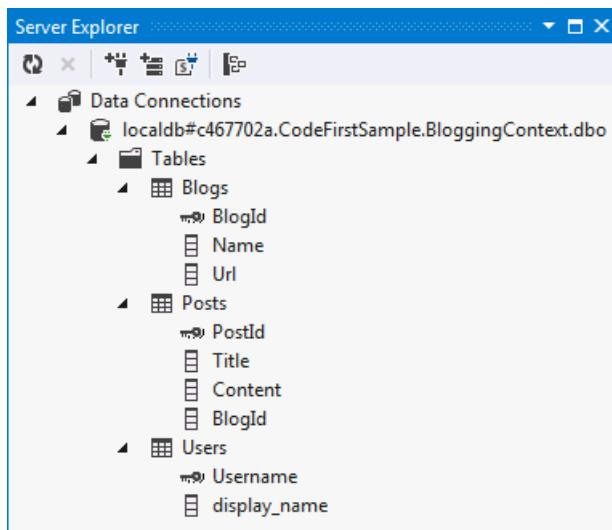
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
            .Property(u => u.DisplayName)
            .HasColumnName("display_name");
    }
}

```

- Используйте **ChangeDisplayName Add-Migration** команду, чтобы сформировать шаблон миграции для применения этих изменений в базу данных.
- Запустите **Update-Database** команду, чтобы применить созданную миграцию к базе данных.

Теперь в столбце DisplayName переименовывается для отображения_имя:



Сводка

В этом пошаговом руководстве мы рассмотрели шаблона разработки Code First с помощью новой базы данных. Мы определена модель, с помощью классов, затем использовать эту модель для создания базы данных и хранения и извлечения данных. После создания базы данных мы использовали Code First Migrations для изменения схемы, как наша модель развитие. Мы также увидели, как настроить модель с помощью заметок к данным и Fluent API.

Code First для существующей базы данных

13.09.2018 • 9 minutes to read • [Edit Online](#)

В этом пошаговом руководстве видео и пошаговые познакомят вас с разработки Code First, предназначенных для существующей базы данных. Во-первых, код позволяет определить модель с помощью C# или классам VB.Net. При необходимости дополнительной настройки могут выполняться с помощью атрибутов для классов и свойств или с помощью текущего API.

Просмотреть видео

Этот видеоролик — [теперь доступна на канале Channel 9](#).

Предварительные требования

Необходимо иметь **Visual Studio 2012** или **Visual Studio 2013** для выполнения этого пошагового руководства.

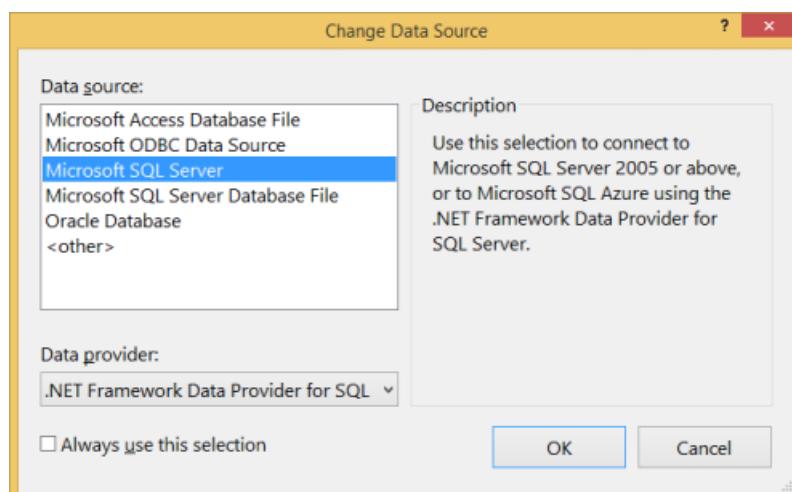
Вам также потребуется версии **6.1** (или более поздней версии) из **средства платформы Entity Framework для Visual Studio** установлен. См. в разделе [получение Entity Framework](#) сведения об установке последней версии средства платформы Entity Framework.

1. Создание базы данных

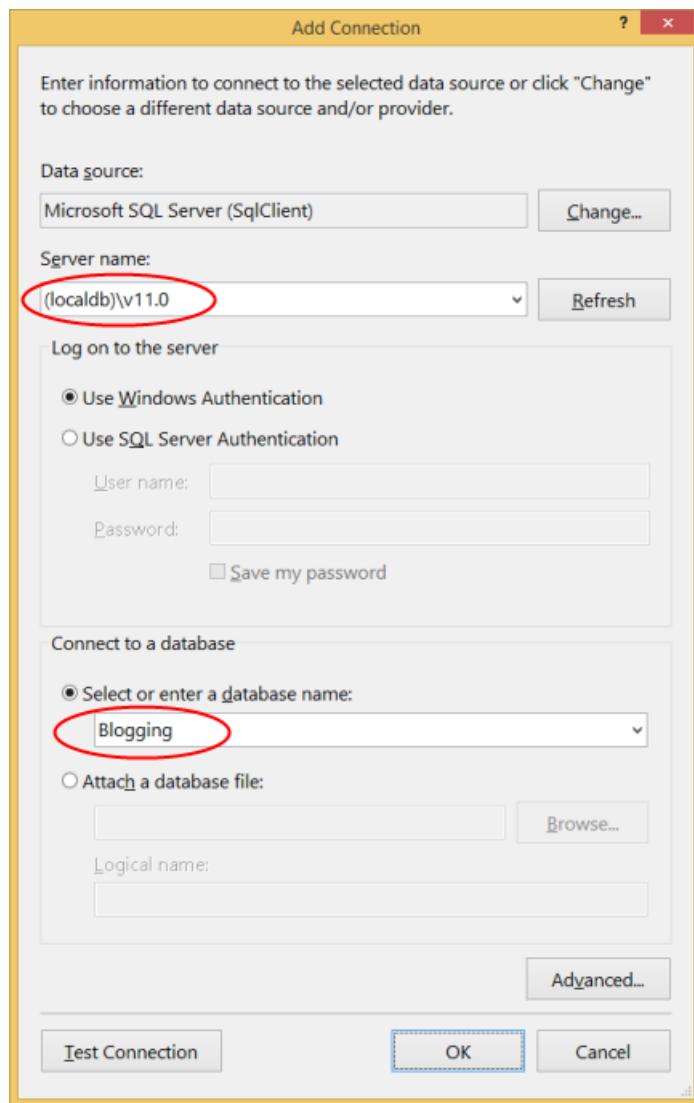
Обычно при ориентировании существующей базы данных, он будет уже создан, но для этого пошагового руководства необходимо создать базу данных для доступа к.

Перейдем дальше и создать базу данных.

- Открытие Visual Studio
- **Представление —> обозревателя серверов**
- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**
- Если вы не подключились к базе данных из **обозревателя серверов** прежде, чем вам нужно будет выбрать **Microsoft SQL Server** как источник данных



- Подключитесь к экземпляру LocalDB и введите **ведения блогов** имя базы данных



- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**



- Новой базы данных будут отображаться в обозревателе сервера щелкните его правой кнопкой мыши и выберите **новый запрос**
- Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**

```

CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId]) ON
DELETE CASCADE
);

INSERT INTO [dbo].[Blogs] ([Name],[Url])
VALUES ('The Visual Studio Blog', 'http://blogs.msdn.com/visualstudio/')

INSERT INTO [dbo].[Blogs] ([Name],[Url])
VALUES ('.NET Framework Blog', 'http://blogs.msdn.com/dotnet/')

```

2. Создание приложения

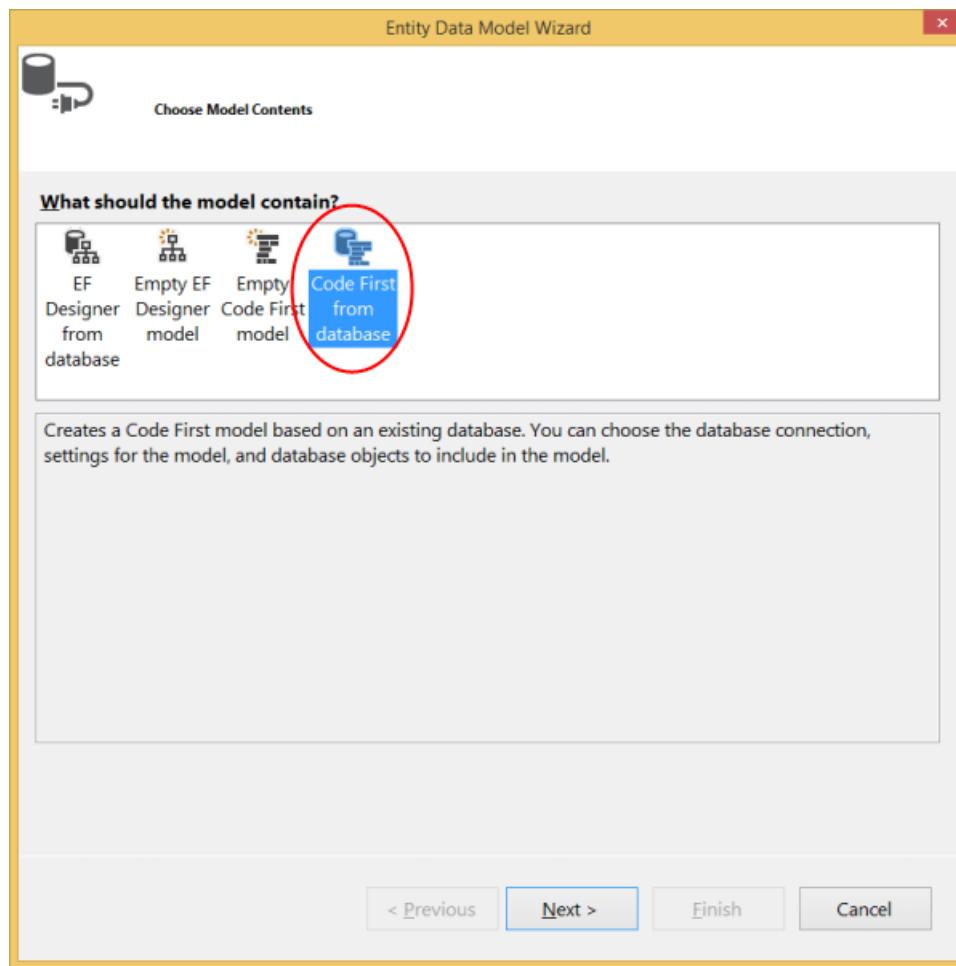
Для простоты мы собираемся создать простое консольное приложение, которое использует Code First для осуществления доступа к данным:

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в меню слева и **консольного приложения**
- Введите **CodeFirstExistingDatabaseSample** как имя
- Нажмите кнопку **OK**

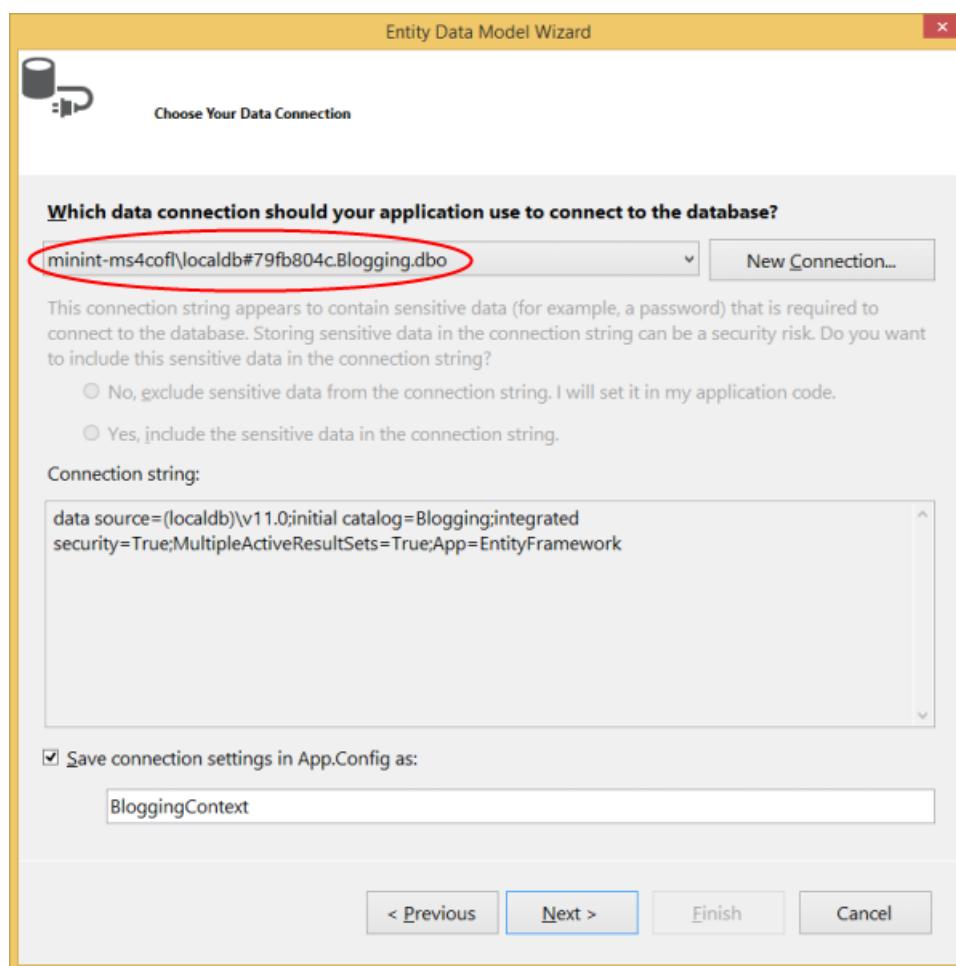
3. Реконструирование модели

Мы будем использовать инструменты Entity Framework для Visual Studio позволяет нам создать некоторый исходный код для сопоставления с базой данных. Эти средства просто генерирует код, который также можно ввести вручную при необходимости.

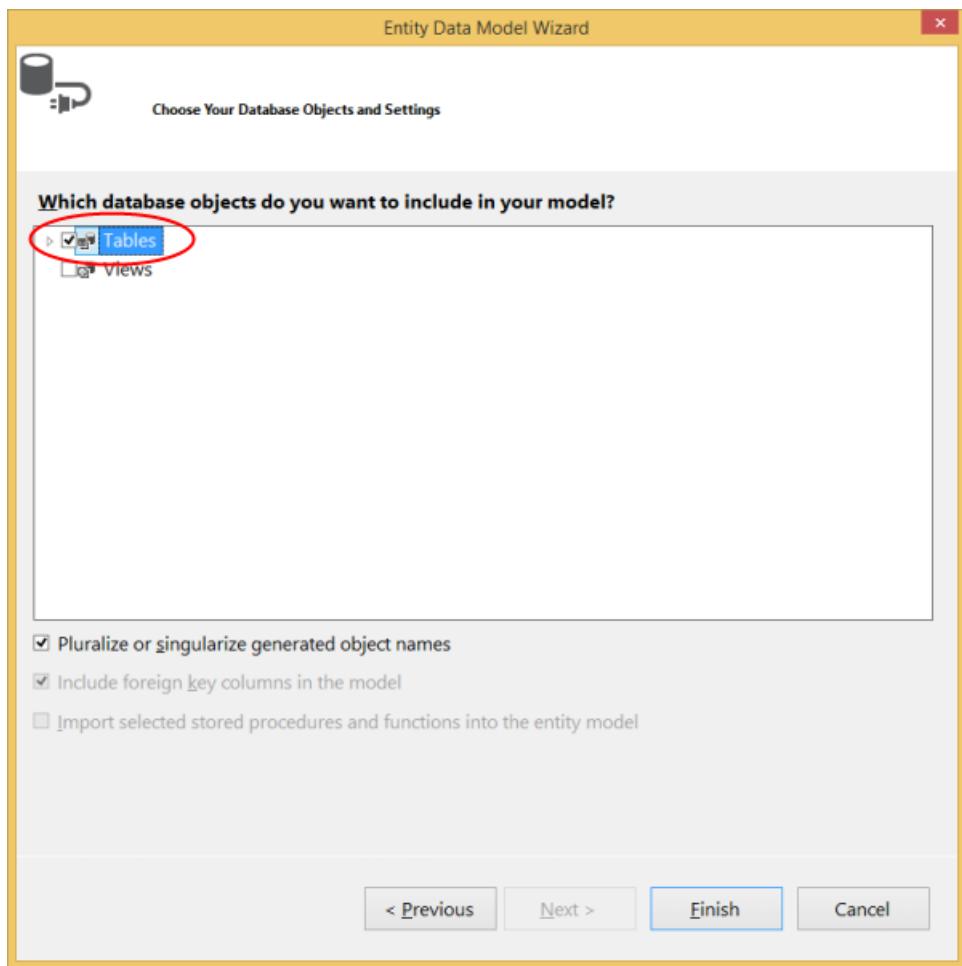
- **Проект -> добавить новый элемент...**
- Выберите **данных** в меню слева и затем **модель EDM ADO.NET**
- Введите **BloggingContext** имя и нажмите кнопку **OK**
- Это откроет **мастер моделей EDM**
- Выберите **Code First, из базы данных** и нажмите кнопку **Далее**



- Выберите соединение с базой данных, созданной в первом разделе и нажмите кнопку **Далее**



- Установите флажок рядом с полем **таблиц** для импорта всех таблиц и нажмите кнопку **Готово**



После завершения процесса реконструирования с числом элементов будут добавлены в проект, давайте взглянем на то, что добавляется.

файл конфигурации

Файл App.config был добавлен в проект, в этот файл содержит строку подключения к существующей базе данных.

```
<connectionStrings>
  <add
    name="BloggingContext"
    connectionString="data source=(localdb)\mssqllocaldb;initial catalog=Blogging;integrated
    security=True;MultipleActiveResultSets=True;App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

Вы заметите некоторые параметры в файле конфигурации слишком, это параметры по умолчанию EF, которые сообщают, Code First для создания баз данных. Так как мы сопоставляемся существующей базы данных, эти параметры будут игнорироваться в нашем приложении.

Производного контекста

Объект **BloggingContext** класс будет добавлен в проект. Контекст представляет собой сеанс с базой данных, позволяет запрашивать и сохранять данные. Предоставляет контекст **DbSet< TEntity >** для каждого типа в нашей модели. Вы также заметите, что конструктор по умолчанию вызывает конструктор базового класса с помощью **имя = синтаксис**. Это сообщает Code First, что строка подключения, необходимо использовать в этом контексте должна быть загружена из файла конфигурации.

```

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

    public virtual DbSet<Blog> Blogs { get; set; }
    public virtual DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
    }
}

```

Следует всегда использовать **имя** = синтаксиса при использовании строки подключения в файле конфигурации. Это гарантирует, что если строке подключения отсутствует затем Entity Framework будет генерировать вместо создания новой базы данных по соглашению.

Классы модели

Наконец **блог** и **Post** класс также были добавлены в проект. Это доменных классов, входящих в состав нашей модели. Вы увидите заметки данных применяются к классам для указания конфигурации где соглашения Code First бы не согласуются со структурой существующей базы данных. Например, вы увидите **StringLength** заметки на **Blog.Name** и **Blog.Url** так, как они имеют длину не более **200** в базы данных (Code First по умолчанию разрешено использовать максимальное длину, поддерживаемую поставщиком баз данных - **nvarchar(max)** в SQL Server).

```

public partial class Blog
{
    public Blog()
    {
        Posts = new HashSet<Post>();
    }

    public int BlogId { get; set; }

    [StringLength(200)]
    public string Name { get; set; }

    [StringLength(200)]
    public string Url { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}

```

4. Чтение и запись данных

Теперь, когда у нас есть модель, настала пора использовать ее для доступа к некоторые данные. Реализуйте **Main** метод в **Program.cs** как показано ниже. Этот код создает новый экземпляр класса наш контекст и затем используется, чтобы вставить новый **блог**. Затем она использует запрос LINQ для извлечения всех **блоги** из базы данных, в алфавитном порядке по **Title**.

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Теперь можно запустить приложение и протестировать его.

```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
.NET Framework Blog
ADO.NET Blog
The Visual Studio Blog
Press any key to exit...
```

Что делать, если Моя база данных изменилась?

Code First для базы данных мастер предназначен для создания начальной точки набора классов, которые можно настроить и изменить. При изменении схемы базы данных можно вручную изменить классы или выполнить другую реконструкции перезаписать классы.

С помощью Code First Migrations для существующей базы данных

Если вы хотите использовать Code First Migrations с существующей базой данных, см. в разделе [Code First Migrations для существующей базы данных](#).

Сводка

В этом пошаговом руководстве мы рассмотрели шаблона разработки Code First с помощью существующей базы данных. Мы использовали средства платформы Entity Framework для Visual Studio, чтобы реконструировать набор классов, которые сопоставляются с базой данных и может использоваться для хранения и извлечения данных.

Заметки данных Code First

29.09.2018 • 27 minutes to read • [Edit Online](#)

NOTE

EF4.1 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в версии 4.1 платформы Entity Framework. При использовании более ранней версии, некоторые или все эти сведения не применяются.

Содержимое на этой странице взято из статьи, первоначально написан Майклом Джули Лерман (<<http://thedatafarm.com>>).

Entity Framework Code First позволяет использовать собственные классы домена для представления модели EF полагается на выполнять запросы, изменение отслеживания и обновления функций. Код сначала использует шаблон программирования, называется «соглашение относительно настройки». Код сначала будет предполагаться, что ваши классы соответствуют соглашениям об платформы Entity Framework и в этом случае будут автоматически работать способа выполнения задания его. Тем не менее если ваши классы не выполняйте эти соглашения, вы можете для добавления конфигурации к классам, чтобы предоставить EF требуемую информацию.

Во-первых, код предоставляет два способа добавления этих конфигураций в пользовательские классы. Один с помощью простых атрибутов, вызывается DataAnnotations, а второй с помощью Code First Fluent API, который предоставляет способ описания конфигураций, принудительно, в коде.

В этой статье основное внимание уделяется с помощью DataAnnotations (в пространстве имен System.ComponentModel.DataAnnotations) для настройки классов — выделение наиболее общие конфигурации. DataAnnotations также понимает ряд приложений .NET, таких как ASP.NET MVC, который позволяет этим приложениям использовать одинаковые примечания для проверки на стороне клиента.

Модель

Я продемонстрирую DataAnnotations первого кода с парой простых классов: блог и Post.

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

Значения, блог и Post классы удобно соответствуют соглашению об первый код и требуют нет изменений для включения совместимости EF. Тем не менее можно также использовать заметки для предоставления

EF Дополнительные сведения о классах и базы данных, к которому они сопоставлены.

Ключ

Платформа Entity Framework использует каждой сущности, который используется для отслеживания сущности значение ключа. Одно соглашение о Code First — неявное ключевых свойств; Код сначала ищет свойство с именем «`Id`», или сочетание имени класса и «`Id`», например «`BlogId`». Это свойство сопоставляется столбцом первичного ключа в базе данных.

Блог и Post классов следуют соглашению. Что делать, если они не? Что делать, если имя используется блог `PrimaryTrackingKey` вместо, или даже `foo`? Если код сначала не удается найти свойство, которое соответствует этому соглашению вызовет исключение из-за требования платформы Entity Framework, что необходимо иметь ключевое свойство. Ключа заметки можно использовать для указания того, какое свойство будет использоваться в качестве `EntityKey`.

```
public class Blog
{
    [Key]
    public int PrimaryTrackingKey { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}
```

Если вы являетесь сначала с помощью кода — использовать функцию создания базы данных, в блоге таблице будет иметь столбец первичного ключа с именем `PrimaryTrackingKey`, который также определен как удостоверение по умолчанию.

A screenshot of the SQL Server Object Explorer. It shows a tree structure under 'dbo'. Under 'Tables', there is a folder named 'Blogs'. Inside 'Blogs', there is a folder named 'Columns'. Inside 'Columns', there are three entries: 'PrimaryTrackingKey' (PK, int, not null), 'Title' (nvarchar(128), null), and 'BloggerName' (nvarchar(128), null).

Составные ключи

Платформа Entity Framework поддерживает составные ключи - первичные ключи, которые состоят из нескольких свойств. Например можно создать первичный ключ которой представляет собой сочетание `PassportNumber` и `IssuingCountry` класс `Passport`.

```
public class Passport
{
    [Key]
    public int PassportNumber { get; set; }
    [Key]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

Попытка использовать приведенного выше класса в модели EF приведет к `InvalidOperationException`:

Не удалось определить составного первичного ключа упорядочения для типа «Passport». Чтобы указать порядок для составные первичные ключи, используя `ColumnAttribute` или метод `HasKey`.

Чтобы использовать составные ключи, Entity Framework необходимо определять порядок ключевых свойств. Это можно сделать с помощью заметок к столбцам, чтобы задать порядок.

NOTE

Порядок сортировки является относительным (а не на базе индекса), чтобы использовать все значения. Например 100 и 200 подойдет вместо 1 и 2.

```
public class Passport
{
    [Key]
    [Column(Order=1)]
    public int PassportNumber { get; set; }

    [Key]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

При наличии сущностей с помощью составного внешние ключи, необходимо указать тот же столбец, упорядочение, которое использовалось для соответствующего свойства первичного ключа.

Только относительный порядок в пределах свойств внешнего ключа должен быть тем же, точные значения, присвоенные **порядок** не обязательно должны совпадать. Например в следующем классе, 3 и 4 может использоваться вместо 1 и 2.

```
public class PassportStamp
{
    [Key]
    public int StampId { get; set; }
    public DateTime Stamped { get; set; }
    public string StampingCountry { get; set; }

    [ForeignKey("Passport")]
    [Column(Order = 1)]
    public int PassportNumber { get; set; }

    [ForeignKey("Passport")]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }

    public Passport Passport { get; set; }
}
```

Обязательно

Требуется аннотация указывает EF, что определенное свойство является обязательным.

Добавление необходимых для свойства Title приведет к EF (и MVC), чтобы убедиться, что свойство имеет данные в ней.

```
[Required]
public string Title { get; set; }
```

Нет дополнительных нет изменений кода или разметки в приложении, MVC-приложении будет выполнять проверки на стороне клиента, даже динамическое создание сообщения с использованием имени свойства и заметки.

Create

Blog

Title	<input type="text"/> The Title field is required.
BloggerName	<input type="text"/> Julie

Create

Обязательный атрибут также влияет на создаваемой базы данных, сделав сопоставленного свойства не допускающие значения NULL. Обратите внимание на то, что поле заголовка изменилось на «не null».

NOTE

В некоторых случаях может оказаться невозможным для столбца в базе данных и не допускало, несмотря на то, что свойство является обязательным. Например, когда с помощью данных стратегии наследование TPH для нескольких типов хранится в одной таблице. Если производный тип содержит свойство столбца невозможно не допускающие значения NULL, так как не все типы в иерархии, это свойство будет задано.

dbo.Blogs
Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), not null)
BloggerName (nvarchar(128), null)

MaxLength и MinLength

MaxLength и MinLength атрибуты позволяют пользователю указать дополнительных свойств в комплексе, как это делалось с обязательным.

Вот BloggerName с требованиями к длине. Пример также демонстрирует использование атрибутов.

```
[MaxLength(10),MinLength(5)]  
public string BloggerName { get; set; }
```

Заметка MaxLength влияет на базе данных путем задания свойства длины 10.

Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), not null)
BloggerName (nvarchar(10), null)

Заметки клиентские MVC и EF 4.1 заметки на сервере оба учитывает проверки снова динамическое создание сообщение об ошибке: «поле BloggerName должно быть типом строки или массива с максимальной длиной "10".» Это сообщение имеет несколько большую длину. Многие заметки позволяют указать сообщение об ошибке с атрибутом ErrorMessage.

```
[MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]  
public string BloggerName { get; set; }
```

Сообщение об ошибке можно также указать необходимые заметки.

Create

Blog

Title

BloggerName
 BloggerName must be 10 characters or less

NotMapped

Соглашение об первый код определяет, что каждое свойство, которое имеет тип данных, поддерживаемых представляется в базе данных. Но это не всегда так, в ваших приложениях. Например можно использовать свойство в классе блог, который создает код, основанный на поля Title и BloggerName. Это свойство можно создавать динамически и не должны храниться. Можно пометить все свойства, которые не сопоставляются с аннотацией NotMapped, например, это свойство BlogCode базы данных.

```
[NotMapped]
public string BlogCode
{
    get
    {
        return Title.Substring(0, 1) + ":" + BloggerName.Substring(0, 1);
    }
}
```

ComplexType

Нередко для описания сущности предметной области через набор классов и затем уровня этих классов для описания в завершенную сущность. Например может добавьте класс с именем BlogDetails в модель.

```
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

Обратите внимание на то, что BlogDetails не имеет ключевого свойства любого типа. В предметно-ориентированное проектирование BlogDetails упоминается как объект значения. Платформа Entity Framework ссылается на объекты-значения как сложные типы. Сложные типы не отслеживаются сами по себе.

Тем не менее как свойство в классе блог BlogDetails, он будет отслеживаться как часть объекта блога. В порядке для code first для распознавания это BlogDetails класс необходимо пометить как ComplexType.

```
[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

Теперь можно добавить свойство в блоге классу для представления BlogDetails для этого блога.

```
public BlogDetails BlogDetail { get; set; }
```

В базе данных блог таблица будет содержать все свойства блога, включая свойства, содержащиеся в его свойстве BlogDetail. По умолчанию каждый из них предшествует имени сложного типа, BlogDetail.

```
dbo.Blogs
Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), not null)
BloggerName (nvarchar(10), null)
BlogDetail_DateCreated (datetime, null)
BlogDetail_Description (nvarchar(250), null)
```

Другой Интересно отметить, что несмотря на то, что свойство DateCreated было определено как не допускающие значения NULL даты и времени в классе, поле соответствующей базы данных допускает значения NULL. Если вы хотите повлиять на схему базы данных, необходимо использовать необходимые заметки.

ConcurrencyCheck

Заметка ConcurrencyCheck позволяет пометить одно или несколько свойств, используемых для проверки в базе данных, когда пользователь изменяет или удаляет сущность параллелизма. Если вы работали с конструктором, EF, это соответствует соглашению присвоить свойство ConcurrencyMode значение Fixed.

Давайте посмотрим, как работает ConcurrencyCheck, добавьте его в свойство BloggerName.

```
[ConcurrencyCheck, MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"), MinLength(5)]
public string BloggerName { get; set; }
```

При вызове метода SaveChanges, из-за Аннотация ConcurrencyCheck в поле BloggerName исходное значение этого свойства будет использоваться в обновлении. Команда попытается найти правильные строки путем фильтрации не только на значение ключа, но также и на ее оригинальном значении BloggerName. Ниже приведены важные обновления команда, Посланная в базу данных, где вы увидите, команда будет обновить строку, которая имеет PrimaryTrackingKey — 1 и BloggerName «Julie», который был исходное значение, при блоге был извлечен из базы данных.

```
where (([PrimaryTrackingKey] = @4) and ([BloggerName] = @5))
@4=1,@5=N'Julie'
```

Если кто-то изменил название блоггер для блога, в то же время, это обновление завершится сбоем, и вы получите DbUpdateConcurrencyException, которые потребуются для обработки.

Метка времени

Это чаще всего используется поля rowversion или метку времени для проверки параллелизма. Но вместо использования ConcurrencyCheck заметки, можно использовать более конкретные заметки времени, до тех пор, пока тип свойства — массив байтов. Код сначала будет обрабатывать Timestamp свойства так же, как свойства ConcurrencyCheck, но также гарантирует, что поле базы данных, код сначала создает не допускающие значения NULL. В одном классе может иметь только одно свойство метки времени.

Добавление в класс блог следующее свойство:

```
[Timestamp]
public Byte[] TimeStamp { get; set; }
```

в результате получается код сначала создается столбец типа timestamp, не допускающие значения NULL в таблице базы данных.

The screenshot shows the 'dbo.Blogs' table structure. It has six columns: PrimaryTrackingKey (PK, int, not null), Title (nvarchar(128), not null), BloggerName (nvarchar(10), null), TimeStamp (timestamp, not null), BlogDetail_DateCreated (datetime, null), and BlogDetail_Description (nvarchar(250), null).

Таблицы и столбца

Если вы предоставляете Code First создания базы данных, можно изменить имя таблицы и столбцы, которые он создает. Можно также использовать Code First с существующей базой данных. Но это не всегда так, что имена классов и свойств в вашем домене соответствуют именам таблиц и столбцов базы данных.

Мой класс называется блог и по соглашению, код сначала рассматривает это приведет к сопоставлению в таблицу с именем блоги. Если это не так с атрибутом таблицы можно указать имя таблицы. Здесь к примеру, заметка указывается, что имя таблицы является InternalBlogs.

```
[Table("InternalBlogs")]
public class Blog
```

Пометка столбца является более детальное представление при указании атрибутов сопоставленного столбца. Можно указать имя, тип данных или даже порядок, в котором отображается столбец в таблице. Вот пример атрибута.

```
[Column("BlogDescription", TypeName="ntext")]
public String Description {get;set;}
```

Не путайте атрибут TypeName столбца с типом данных DataAnnotation. Тип данных является заметки, используемый для пользовательского интерфейса и поэтому пропускается при Code First.

Здесь приведена таблица после его генерируется. Имя таблицы изменилось на InternalBlogs и описание столбца из сложного типа теперь BlogDescription. Так как имя указано в аннотации, код сначала не будет использовать соглашение для имени столбца, начиная с имени сложного типа.

└─	└─	dbo.InternalBlogs
└─	└─	Columns
└─	└─	PrimaryTrackingKey (PK, int, not null)
└─	└─	Title (nvarchar(128), not null)
└─	└─	BloggerName (nvarchar(10), null)
└─	└─	TimeStamp (timestamp, not null)
└─	└─	BlogDetail_DateCreated (datetime, null)
└─	└─	BlogDescription (ntext, null)

DatabaseGenerated

Функции важные базы данных является возможность вычисляемые свойства. При сопоставлении Code First классы для таблиц, содержащих вычисляемые столбцы, Entity Framework, чтобы попытаться обновить эти столбцы не нужно. Но вы хотите EF для получения этих значений из базы данных после были вставлены или обновлены данные. DatabaseGenerated заметки можно использовать для пометки этих свойств в классе, а также перечисления вычисленные. Другие перечисления являются None и удостоверений.

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public DateTime DateCreated { get; set; }
```

Можно использовать базу данных, создается байтов или метки времени столбцов при код сначала создает базу данных, в противном случае вам следует использовать только при указывает на существующие базы данных, так как код сначала не сможет определить формулу для данного вычисляемого столбца.

Ключевое свойство, которое должно быть целым числом, уже говорилось, по умолчанию станет ключа удостоверения в базе данных. Что бы так же, как присвоить DatabaseGenerated DatabaseGeneratedOption.Identity. Если вы не хотите его ключ удостоверения, можно задать значение для DatabaseGeneratedOption.None.

Индекс

NOTE

EF6.1 и более поздних версий только -индекс, представляющий атрибут появился в Entity Framework 6.1. При использовании более ранней версии сведения в этом разделе не применяется.

Можно создать индекс для одного или нескольких столбцов с помощью **IndexAttribute**. Добавление атрибута в один или несколько свойств будет приводит к EF для создания соответствующего индекса в базе данных, при создании базы данных, или сформировать шаблон соответствующего **CreateIndex** вызывает при использовании Code First Migrations.

Например, следующий код приведет к индекса при создании на основе **Оценка** столбец **сообщения** таблицы в базе данных.

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    [Index]
    public int Rating { get; set; }
    public int BlogId { get; set; }
}
```

По умолчанию будут присваиваться индекс **IX_<имя свойства>** (IX_оценка в приведенном выше примере). Можно также указать имя индекса хотя. В следующем примере указывается, что индекс должен

быть назван **PostRatingIndex**.

```
[Index("PostRatingIndex")]
public int Rating { get; set; }
```

По умолчанию, неуникальных индексов, но можно использовать **IsUnique** именованный параметр, чтобы указать, что индекс должен быть уникальным. В следующем примере представлены уникальный индекс на **пользователя**на имя входа.

```
public class User
{
    public int UserId { get; set; }

    [Index(IsUnique = true)]
    [StringLength(200)]
    public string Username { get; set; }

    public string DisplayName { get; set; }
}
```

Индексы с несколькими столбцами

Индексы, которые охватывают несколько столбцов, указываются с помощью тем же именем в несколько заметок индекса для данной таблицы. Создавая индексы с несколькими столбцами, необходимо указать порядок для столбцов в индексе. Например, следующий код создает многостолбцовый индекс для **Оценка** и **BlogId** вызывается **IX_BlogAndRating**. **BlogId** является первым столбцом в индексе и **Оценка** является вторым.

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    [Index("IX_BlogIdAndRating", 2)]
    public int Rating { get; set; }
    [Index("IX_BlogIdAndRating", 1)]
    public int BlogId { get; set; }
}
```

Связь атрибутов: `InverseProperty` и `ForeignKey`

NOTE

Эта страница содержит сведения о настройке связи в модели Code First с помощью заметок к данным. Общие сведения о связях в EF и способах доступа к данных и управления ими с помощью связей см. в разделе [связей и свойств навигации](#). *

Соглашение об первый кода позаботится о наиболее распространенных связи в модели, но существуют случаи, где он нуждается в помощи.

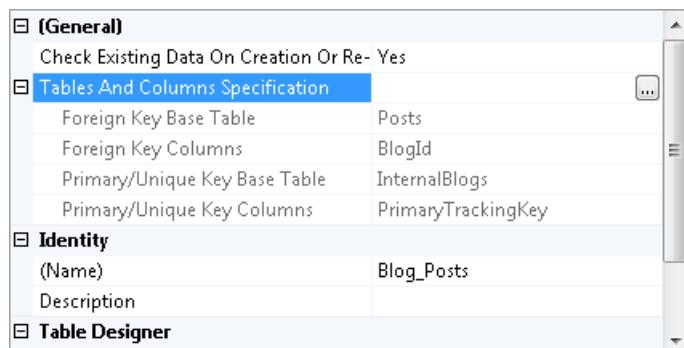
Изменение имени ключевого свойства в класс блог, созданный на проблему с его связь с Post.

При создании базы данных, код сначала увидит `BlogId` свойства в классе `Post` и распознает ее, по соглашению, что он соответствует имени класса, а также «`Id`», как внешний ключ к классу блога. Но нет `BlogId` свойства в классе блога. Решением будет создать свойство навигации в блога и использовать внешний `DataAnnotation` для кода сначала понять, как создать связь между двумя классами, используя

свойство Post.BlogId — а также способы указания ограничений в База данных.

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    [ForeignKey("BlogId")]
    public Blog Blog { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

Ограничение в базе данных отображает связь между InternalBlogs.PrimaryTrackingKey и Posts.BlogId.



InverseProperty используется в том случае, если у вас есть несколько связей между классами.

В классе Post, может потребоваться хранить список кто написал в блоге и кто его изменил. Ниже приведены два новых свойства навигации для класса Post.

```
public Person CreatedBy { get; set; }
public Person UpdatedBy { get; set; }
```

Также необходимо добавить в класс Person, ссылается на эти свойства. Класс Person имеет свойства навигации обратно к записи, одну для всех записей, написанной person и одну для всех записей, обновлены этим пользователем.

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Post> PostsWritten { get; set; }
    public List<Post> PostsUpdated { get; set; }
}
```

Сначала код не сможет сопоставлять свойства в класс сам по себе. В таблице базы данных для записей должны иметь один внешний ключ для пользователя, Кем создано и один для UpdatedBy человека, но код сначала создать свойства внешнего ключа четыре будет: Person_идентификатор, Person_Id1, Кем создано_идентификатор и UpdatedBy_идентификатор.

		dbo.Posts
		Columns
		Id (PK, int, not null)
		Title (nvarchar(128), null)
		DateCreated (datetime, not null)
		Content (nvarchar(128), null)
		BlogId (FK, int, not null)
		Person_Id (FK, int, null)
		Person_Id1 (FK, int, null)
		CreatedBy_Id (FK, int, null)
		UpdatedBy_Id (FK, int, null)

Чтобы устраниТЬ эти проблемы, InverseProperty заметки можно использовать для указания выравнивания свойств.

```
[InverseProperty("CreatedBy")]
public List<Post> PostsWritten { get; set; }

[InverseProperty("UpdatedBy")]
public List<Post> PostsUpdated { get; set; }
```

Так как свойство PostsWritten лично знает, что это относится к типу Post, сборка будет связи Post.CreatedBy. Аналогичным образом PostsUpdated будут подключены к Post.UpdatedBy. И код сначала не создает дополнительных внешних ключей.

		dbo.Posts
		Columns
		Id (PK, int, not null)
		Title (nvarchar(128), null)
		DateCreated (datetime, not null)
		Content (nvarchar(128), null)
		BlogId (FK, int, not null)
		CreatedBy_Id (FK, int, null)
		UpdatedBy_Id (FK, int, null)

Сводка

DataAnnotations не только позволяют описывать проверки на стороне клиента и сервера в классах первого кода, но они также позволяют улучшать и даже устраниТЬ предположения, которые код сначала облегчает о классов на основе его соглашений. С помощью DataAnnotations вы можете не только обеспечивать создание схемы базы данных, но также можно сопоставить классы первый код и уже существующей базы данных.

Хотя они являются достаточно гибкими, помните, что DataAnnotations предоставляют только наиболее часто используемые конфигурации изменения, внесенные в классах первого кода. Чтобы настроить свои классы для некоторых крайних случаев, следует проверить механизм альтернативная конфигурация, Code First Fluent API.

Определение DbSets

13.09.2018 • 2 minutes to read • [Edit Online](#)

При разработке с помощью Code First рабочего процесса вы определяете производном DbContext, который представляет сеанс с базой данных и предоставляет DbSet для каждого типа в модели. В этом разделе рассматриваются различные способы, можно определить свойства DbSet.

DbContext с помощью свойства DbSet

Распространенный случай, показано в примерах Code First является DbContext с помощью общедоступного автоматические свойства DbSet для типов сущностей модели. Пример:

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

При использовании в режиме Code First, это настроит блогов и записей как типы сущностей, а также настройка других доступен на основе этих типов. Кроме DbContext автоматически вызывает метод задания для каждого из этих свойств, чтобы задать экземпляр соответствующего DbSet.

DbContext со свойствами IDbSet

Существуют ситуации, например при создании макетов или fakes, где это удобнее для объявления свойств набора с помощью интерфейса. В таких случаях IDbSet интерфейс можно использовать вместо DbSet.

Пример:

```
public class BloggingContext : DbContext
{
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
}
```

Этот контекст работает точно так же, как контекст, который использует класс DbSet для его задания свойств.

DbContext со свойствами только для чтения

Если вы не хотите предоставлять открытые методы задания свойств DbSet или IDbSet вместо этого можно создать свойства только для чтения и создания экземпляров набора самостоятельно. Пример:

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs
    {
        get { return Set<Blog>(); }
    }

    public DbSet<Post> Posts
    {
        get { return Set<Post>(); }
    }
}
```

Обратите внимание на то, что DbContext кэширует экземпляр DbSet, возвращается из метода Set, чтобы каждое из этих свойств возвращает тот же экземпляр при каждом вызове.

Обнаружение типов сущностей для Code First работает так же, как и для свойства с помощью общедоступного методы get и set.

Поддержка перечислений — Code First

27.09.2018 • 7 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Это видео и пошаговое руководство показано, как использовать типы перечисления с Entity Framework Code First. Также демонстрируется использование перечислений в запросе LINQ.

В этом пошаговом руководстве будет использовать Code First для создания новой базы данных, но можно также использовать [Code First для сопоставления существующей базы данных](#).

В Entity Framework 5 добавлена поддержка перечисления. Чтобы использовать новые функции, например перечисления, Пространственные типы данных и функции, возвращающие табличные значения, необходимо ориентироваться .NET Framework 4.5. Visual Studio 2012 предназначенного для .NET 4.5 по умолчанию.

В Entity Framework, что перечисление может иметь следующие базовые типы: **байтов**, **Int16**, **Int32**, **Int64**, или **SByte**.

Просмотреть видео

В этом видео показано, как использовать типы перечисления с Entity Framework Code First. Также демонстрируется использование перечислений в запросе LINQ.

Представленный: Юлия Корнич

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Необходимо будет установлен выпуск Visual Studio 2012 Ultimate, Premium, Professional и Web Express для выполнения этого пошагового руководства.

Настройка проекта

1. Откройте Visual Studio 2012
2. На **файл** последовательно выберите пункты **New**, а затем нажмите кнопку **проекта**
3. В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона
4. Введите **EnumCodeFirst** как имя проекта и нажмите кнопку **OK**

Определения новой модели с помощью Code First

При использовании шаблона разработки Code First обычно начинается с написания классов .NET Framework, которые определяют модель концептуальный (домен). Приведенный ниже код определяет класс отдела.

Код также определяет перечисление DepartmentNames. По умолчанию имеет перечисления **int** типа. Свойство Name класса Department имеет тип DepartmentNames.

Откройте файл Program.cs и вставьте следующие определения класса.

```
public enum DepartmentNames
{
    English,
    Math,
    Economics
}

public partial class Department
{
    public int DepartmentID { get; set; }
    public DepartmentNames Name { get; set; }
    public decimal Budget { get; set; }
}
```

Определить тип, производный от DbContext

Кроме определения сущностей, необходимо определить класс, который является производным от DbContext и DbSet предоставляет < TEntity > свойства. DbSet< TEntity > свойства let контекст знать, какие типы, которые вы хотите включить в модель.

Экземпляр типа DbContext производным управляет объектов сущности во время выполнения, который включает заполнение объектов с данными из базы данных, изменение отслеживания и сохранения данных в базу данных.

Типы DbContext и DbSet определенные в сборке EntityFramework. Мы добавим ссылку на эту библиотеку DLL с помощью пакета EntityFramework NuGet.

1. В обозревателе решений щелкните правой кнопкой мыши имя проекта.
2. Выберите **управление пакетами NuGet...**
3. В диалоговом окне «Управление пакетами NuGet» выберите **Online** вкладку и выберите **EntityFramework** пакета.
4. Нажмите кнопку **установки**

Обратите внимание, что помимо сборки EntityFramework, ссылки на сборки System.ComponentModel.DataAnnotations и System.Data.Entity добавляются также.

В верхней части файла Program.cs, добавьте следующий оператор using:

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

В Program.cs добавьте определение контекста.

```
public partial class EnumTestContext : DbContext
{
    public DbSet<Department> Departments { get; set; }
}
```

Сохранения и извлечения данных

Откройте файл Program.cs, в котором определен метод Main. Добавьте следующий код в функцию Main. Код добавляет новый объект отдела в контекст. Затем она сохраняет данные. Код также выполняется запрос LINQ, возвращающий подразделение, где имя задается DepartmentNames.English.

```
using (var context = new EnumTestContext())
{
    context.Departments.Add(new Department { Name = DepartmentNames.English });

    context.SaveChanges();

    var department = (from d in context.Departments
                      where d.Name == DepartmentNames.English
                      select d).FirstOrDefault();

    Console.WriteLine(
        "DepartmentID: {0} Name: {1}",
        department.DepartmentID,
        department.Name);
}
```

Скомпилируйте и запустите приложение. Программа выдает следующие результаты.

```
DepartmentID: 1 Name: English
```

Представление создаваемой базы данных

При запуске приложение в первый раз, Entity Framework создает базу данных. Поскольку у нас есть установленной Visual Studio 2012, базе данных создается на экземпляре LocalDB. По умолчанию Entity Framework содержит имя базы данных после полное имя производного контекста (например, **EnumCodeFirst.EnumTestContext**). Последующие случаи, когда будет использоваться существующая база данных.

Обратите внимание, что если внести изменения в модель после создания базы данных, необходимо использовать Code First Migrations для обновления схемы базы данных. См. в разделе [Code First для новой базы данных](#) пример с помощью миграций.

Чтобы просмотреть базы данных и данных, сделайте следующее:

1. В главном меню Visual Studio 2012 выберите **представление - > обозреватель объектов SQL Server**.
2. Если LocalDB отсутствует в списке серверов, щелкните правой кнопкой мыши **SQL Server** и выберите **добавить SQL Server** используйте значение по умолчанию **проверки подлинности Windows** для подключения к Экземпляр LocalDB
3. Разверните узел LocalDB
4. Unfold **баз данных** папку для новой базы данных см. в разделе и перейдите к **отдел** таблицы Обратите внимание, что Code First не создает таблицу, которая сопоставляется с типом перечисления
5. Чтобы просмотреть данные, щелкните правой кнопкой мыши по таблице и выберите **Просмотр данных**

Сводка

В этом пошаговом руководстве мы рассмотрели способы использования типов перечисления с Entity Framework Code First.

Пространственные - Code First

27.09.2018 • 8 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Видео и пошаговые руководства показано, как сопоставить пространственных типов с помощью Entity Framework Code First. Также показано, как использовать LINQ-запрос для поиска расстояние между двумя расположениями.

В этом пошаговом руководстве будет использовать Code First для создания новой базы данных, но можно также использовать [Code First для существующей базы данных](#).

В Entity Framework 5 появилась поддержка пространственных типов. Обратите внимание на то, что чтобы использовать новые функции, например пространственного типа, перечислимые типы и функции, возвращающие табличные значения, необходимо ориентироваться .NET Framework 4.5. Visual Studio 2012 предназначенного для .NET 4.5 по умолчанию.

Для использования пространственных типов данных необходимо также использовать с поставщиком Entity Framework с поддержкой пространственных. См. в разделе [поддержка пространственных типов](#) Дополнительные сведения.

Существует два типа основных пространственных данных: geography и geometry. Тип данных сохраняет эллиптические данные (например, GPS координаты широты и долготы). Тип данных geometry представляет Евклидовой (плоской) системе координат.

Просмотреть видео

В этом видео показано, как сопоставить пространственных типов с помощью Entity Framework Code First. Также показано, как использовать LINQ-запрос для поиска расстояние между двумя расположениями.

Представленный: Юлия Корнич

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Необходимо будет установлен выпуск Visual Studio 2012 Ultimate, Premium, Professional и Web Express для выполнения этого пошагового руководства.

Настройка проекта

1. Откройте Visual Studio 2012
2. На **файл** последовательно выберите пункты **New**, а затем нажмите кнопку **проекта**
3. В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона
4. Введите **SpatialCodeFirst** как имя проекта и нажмите кнопку **OK**

Определения новой модели с помощью Code First

При использовании шаблона разработки Code First обычно начинается с написания классов .NET Framework, которые определяют модель концептуальный (домен). Приведенный ниже код определяет класс университета.

В университете ведется свойство Location типа DbGeography. Чтобы использовать тип DbGeography, необходимо добавить ссылку на сборку System.Data.Entity и также добавить System.Data.Spatial, с помощью инструкции.

Откройте файл Program.cs и вставьте следующие операторы using в верхней части файла:

```
using System.Data.Spatial;
```

Добавьте следующее определение класса университета в файл Program.cs.

```
public class University
{
    public int UniversityID { get; set; }
    public string Name { get; set; }
    public DbGeography Location { get; set; }
}
```

Определить тип, производный от DbContext

Кроме определения сущностей, необходимо определить класс, который является производным от DbContext и DbSet предоставляет < TEntity > свойства. DbSet< TEntity > свойства let контекст знать, какие типы, которые вы хотите включить в модель.

Экземпляр типа DbContext производным управляет объектов сущности во время выполнения, который включает заполнение объектов с данными из базы данных, изменение отслеживания и сохранения данных в базу данных.

Типы DbContext и DbSet определенные в сборке EntityFramework. Мы добавим ссылку на эту библиотеку DLL с помощью пакета EntityFramework NuGet.

1. В обозревателе решений щелкните правой кнопкой мыши имя проекта.
2. Выберите **управление пакетами NuGet...**
3. В диалоговом окне «Управление пакетами NuGet» выберите **Online** вкладку и выберите **EntityFramework** пакета.
4. Нажмите кнопку **установки**

Обратите внимание, что помимо сборки EntityFramework, ссылку на сборку System.ComponentModel.DataAnnotations также добавляется.

В верхней части файла Program.cs, добавьте следующий оператор using:

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

В Program.cs добавьте определение контекста.

```
public partial class UniversityContext : DbContext
{
    public DbSet<University> Universities { get; set; }
}
```

Сохранения и извлечения данных

Откройте файл Program.cs, в котором определен метод Main. Добавьте следующий код в функцию Main.

Код добавляет два новых университета объектов к контексту. Пространственные свойства инициализируются с помощью метода DbGeography.FromText. Географическая точка представить в виде WellKnownText передается в метод. Затем код сохраняет данные. Затем запрос LINQ, который возвращает объект университета, где она будет находиться в указанное расположение, ближайшее составлением и выполнением.

```
using (var context = new UniversityContext ())
{
    context.Universities.Add(new University()
    {
        Name = "Graphic Design Institute",
        Location = DbGeography.FromText("POINT(-122.336106 47.605049)"),
    });

    context.Universities.Add(new University()
    {
        Name = "School of Fine Art",
        Location = DbGeography.FromText("POINT(-122.335197 47.646711)"),
    });

    context.SaveChanges();

    var myLocation = DbGeography.FromText("POINT(-122.296623 47.640405)");

    var university = (from u in context.Universities
                      orderby u.Location.Distance(myLocation)
                      select u).FirstOrDefault();

    Console.WriteLine(
        "The closest University to you is: {0}.",
        university.Name);
}
```

Скомпилируйте и запустите приложение. Программа выдает следующие результаты.

```
The closest University to you is: School of Fine Art.
```

Представление создаваемой базы данных

При запуске приложение в первый раз, Entity Framework создает базу данных. Поскольку у нас есть установленной Visual Studio 2012, базе данных создается на экземпляре LocalDB. По умолчанию Entity Framework содержит имя базы данных после полное имя производного контекста (в этом примере, **SpatialCodeFirst.UniversityContext**). Последующие случаи, когда будет использоваться существующая база данных.

Обратите внимание, что если внести изменения в модель после создания базы данных, необходимо использовать Code First Migrations для обновления схемы базы данных. См. в разделе [Code First для новой базы данных](#) пример с помощью миграций.

Чтобы просмотреть базы данных и данных, сделайте следующее:

1. В главном меню Visual Studio 2012 выберите **представление - > обозреватель объектов SQL Server**.
2. Если LocalDB отсутствует в списке серверов, щелкните правой кнопкой мыши **SQL Server** и выберите **добавить SQL Server** используйте значение по умолчанию **проверки подлинности Windows** для подключения к экземпляру LocalDB

3. Разверните узел LocalDB
4. Unfold **баз данных** папку для новой базы данных см. в разделе и перейдите к **университеты** таблицы
5. Чтобы просмотреть данные, щелкните правой кнопкой мыши по таблице и выберите **Просмотр данных**

Сводка

В этом пошаговом руководстве мы рассмотрели способы использования пространственных типов с Entity Framework Code First.

Первый соглашения о коде

13.09.2018 • 9 minutes to read • [Edit Online](#)

Во-первых, код позволяет описания модели с помощью классов C# или Visual Basic .NET. Основные фигуры модели определяется с использованием соглашений. Соглашения являются наборы правил, которые используются для автоматической настройки концептуальной модели на основе определений класса при работе в режиме Code First. Правила определяются в пространстве имен System.Data.Entity.ModelConfiguration.Conventions.

Можно продолжить настройку модели с помощью заметок к данным или текущего API. Преимущество отдается конфигурацию с помощью текущего API, следуют заметок к данным и соглашения.

Дополнительные сведения см. в разделе [заметок к данным](#), [Fluent API - связи](#), [Fluent API - типов и свойств](#), и [Fluent API с использованием VB.NET](#).

Подробный список соглашения Code First доступен в [документации по API](#). Здесь представлен обзор соглашения, используемые механизмом Code First.

Обнаружение типа

При использовании шаблона разработки Code First обычно начинается с написания классов .NET Framework, которые определяют модель концептуальный (домен). Помимо определения классов, необходимо также разрешить **DbContext** знать, какие типы, которые вы хотите включить в модель. Чтобы сделать это, вы определяется класс контекста, производный от **DbContext** и предоставляет **DbSet** свойств для типов, которые должны быть частью модели. Код сначала будет включать эти типы и также будет получать в любой ссылочных типов, даже если ссылочных типов определяются в другой сборке.

Если типы участвуют в иерархии наследования, его достаточно, чтобы определить **DbSet** свойство для базового класса и производных типов, будут автоматически включены, если они находятся в той же сборке, в качестве базового класса.

В следующем примере имеется только один **DbSet** свойство, определенное для **SchoolEntities** класс (**отделов**). Код сначала использует это свойство для обнаружения и по запросу в любой ссылочных типов.

```

public class SchoolEntities : DbContext
{
    public DbSet<Department> Departments { get; set; }
}

public class Department
{
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public string Location { get; set; }
    public string Days { get; set; }
    public System.DateTime Time { get; set; }
}

```

Если вы хотите исключить тип из модели, используйте **NotMapped** атрибут или **DbModelBuilder.Ignore** текущего API.

```
modelBuilder.Ignore<Department>();
```

Соглашение первичного ключа

Код сначала определяет, что свойство является первичным ключом, если свойство класса, называется «ID» (без учета регистра), или имя класса, добавив «ID». Если тип со свойством первичного ключа является числом или GUID будет настроен в качестве столбца идентификаторов.

```

public class Department
{
    // Primary key
    public int DepartmentID { get; set; }

    . .

}

```

Соглашение о связи

В Entity Framework свойства навигации позволяют для перехода по связи между двумя типами сущностей.

Каждый объект может обладать свойством навигации для каждого отношения, в котором участвует.

Свойства навигации позволяют перейти связей и управление ими в обоих направлениях, возвращая объект

ссылки (если атрибут кратности имеет один или ноль или один) или коллекции (Если кратность равна много).

Код сначала определяет связи, основанные на свойства навигации, определенные для типов.

В дополнение к свойствам навигации рекомендуется включить свойства внешнего ключа на типы, представляющие зависимые объекты. Любое свойство с тем же типом данных, как основной свойство первичного ключа и с именем, которое следует одной из следующих форматов представляет внешний ключ для связи: "<имя свойства навигации><участника Имя свойство первичного ключа>','<имя основного класса><имя свойство первичного ключа>", или"<имя участника-свойство первичного ключа>". Если найдено несколько совпадений, преимущество отдается в порядке, указанном выше. Обнаружение внешнего ключа не учитывается регистр символов. При обнаружении свойство внешнего ключа, Code First определяет кратность связи, в зависимости от того, поддерживает ли внешний ключ. Если свойство имеет значение NULL затем связь регистрируется как необязательный; в противном случае — регистрируется связи при необходимости.

Если внешний ключ для зависимой сущности не допускает значения NULL, то Code First устанавливает каскадное удаление в отношении. Если внешний ключ для зависимой сущности допускает значения NULL, Code First не устанавливает каскадное удаление для связи, и при удалении участника внешний ключ устанавливается в значение null. Поведение, обнаруживаемые при удалении кратность и cascade соглашение можно переопределить с помощью текущего API.

В следующем примере свойства навигации и внешний ключ используются для определения связи между классами отдела и курса.

```
public class Department
{
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
}
```

NOTE

Если у вас есть несколько связей между теми же типами (предположим, например, можно определить **Person** и **книги** классы, где **Person** класс содержит **ReviewedBooks** и **AuthoredBooks** свойства навигации и **книги** класс содержит **автор** и **Редактор** свойства навигации) необходимо вручную настроить отношения с помощью заметок к данным или текущего API. Дополнительные сведения см. в разделе [заметок к данным - связи](#) и [Fluent API - связи](#).

Соглашение о сложных типов

Когда Code First обнаруживает определение класса, где не может быть определен первичный ключ, а первичный ключ не зарегистрирован при помощи заметки к данным или fluent API, тип автоматически регистрируется как сложный тип. Определение сложных типов также требует, что тип не имеет свойства, которые ссылаются на типы сущностей и нет ссылок из свойства коллекции другого типа. Учитывая следующие определения класса Code First бы получен, **сведения** — это сложный тип, так как он не имеет первичного ключа.

```
public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}
```

Соглашение о строке подключения

Дополнительные сведения о соглашениях об что DbContext использует для обнаружения соединение для использования см. в разделе [подключений и модели](#).

Удаление соглашения

Вы можете удалить правила, определенным в пространстве имен System.Data.Entity.ModelConfiguration.Conventions. В следующем примере удаляется **PluralizingTableNameConvention**.

```
public class SchoolEntities : DbContext
{
    . .

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention, the generated tables
        // will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Соглашения об именовании

Соглашения об именовании, поддерживаются в EF6 и выше. Дополнительные сведения см. в разделе [первый соглашения о написании пользовательского кода](#).

Первый соглашения о написании пользовательского кода

13.09.2018 • 19 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

При использовании Code First модели вычисляется на основе классов с помощью набора соглашений. Значение по умолчанию [первый соглашения о коде](#) определяет, как и в которых свойство становится первичным ключом сущности, имя таблицы, сопоставляет сущности и какие точность и масштаб столбца decimal имеет по умолчанию.

Иногда эти соглашения по умолчанию не идеально подходят для вашей модели, и вам нужно решить их, настроив множество отдельных сущностей, с помощью заметок к данным или Fluent API. Пользовательские соглашения о коде позволяют определять собственные соглашения, которые предоставляют значения конфигурации по умолчанию для модели. В этом пошаговом руководстве мы рассмотрим различные типы соглашения об именовании и создание каждого из них.

Правила на основе моделей

В этой статье описываются API-интерфейса DbModelBuilder для соглашения об именовании. Этот API должно быть достаточно для создания большинства соглашения об именовании. Тем не менее есть также возможность создавать модели с использованием соглашений - соглашения, которые управляют окончательная версия модели, после ее создания — для обработки более сложных сценариев. Дополнительные сведения см. в разделе [модели с использованием соглашений](#).

Наша модель

Начнем с определения простой модели, который можно использовать с помощью наших соглашений. Добавьте следующие классы в проект.

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ProductContext>());
    }

    public DbSet<Product> Products { get; set; }
}

public class Product
{
    public int Key { get; set; }
    public string Name { get; set; }
    public decimal? Price { get; set; }
    public DateTime? ReleaseDate { get; set; }
    public ProductCategory Category { get; set; }
}

public class ProductCategory
{
    public int Key { get; set; }
    public string Name { get; set; }
    public List<Product> Products { get; set; }
}

```

Знакомство с соглашениями об именовании

Давайте напишем соглашение, которое настраивает любое свойство с именем ключа в качестве первичного ключа для своего типа сущности.

Соглашения о включены построитель модели, к которому можно получить путем переопределения OnModelCreating в контексте. Обновите класс ProductContext следующим образом:

```

public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ProductContext>());
    }

    public DbSet<Product> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Properties()
            .Where(p => p.Name == "Key")
            .Configure(p => p.IsKey());
    }
}

```

Теперь, любое свойство в нашей модели, с именем ключа будет настроен в качестве первичного ключа сущности, независимо от его частью.

Можно также сделать соглашения более точным, фильтруя данные по типу свойства, который будет использоваться для настройки:

```
modelBuilder.Properties<int>()
    .Where(p => p.Name == "Key")
    .Configure(p => p.IsKey());
```

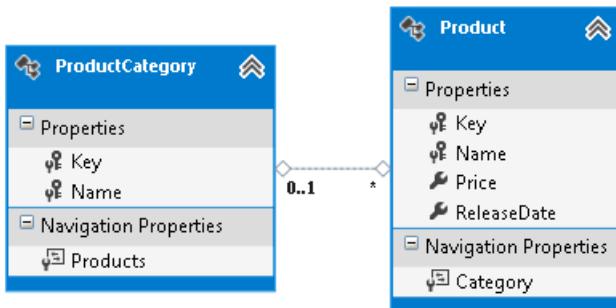
Будут настроены все свойства, называемые ключ основного ключа их сущности, но только в том случае, если это целое число.

Интересной особенностью метод IsKey является его аддитивный характер. То есть, если вы вызываете IsKey на несколько свойств, и все они станут частью составного ключа. Единственным условием для этого является то, что при указании нескольких свойств для ключа также необходимо указать, заказ для этих свойств. Это можно сделать, вызвав HasColumnOrder, аналогичный метод:

```
modelBuilder.Properties<int>()
    .Where(x => x.Name == "Key")
    .Configure(x => x.IsKey().HasColumnOrder(1));

modelBuilder.Properties()
    .Where(x => x.Name == "Name")
    .Configure(x => x.IsKey().HasColumnOrder(2));
```

Этот код будет настроить типы в нашей модели иметь составной ключ, состоящий из столбца Key int и строкового имени столбца. Если просмотреть модель в конструкторе оно выглядело следующим образом:



Другой пример соглашений свойство — настроить все свойства даты и времени в моей модели для сопоставления с типом datetime2 в SQL Server вместо даты и времени. Это можно сделать с помощью следующих:

```
modelBuilder.Properties<DateTime>()
    .Configure(c => c.HasColumnType("datetime2"));
```

Соглашение о классах

Определение соглашения еще один способ — использовать соглашение о класс для инкапсуляции соглашению об. При использовании класса соглашение о создании типа, наследуемого от класса соглашение в пространстве имен System.Data.Entity.ModelConfiguration.Conventions.

Мы можем создать класс соглашение об с datetime2 соглашение, которое мы продемонстрировали ранее, сделав следующее:

```
public class DateTime2Convention : Convention
{
    public DateTime2Convention()
    {
        this.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
    }
}
```

Чтобы сообщить EF, чтобы использовать это соглашение, добавить его в коллекцию соглашений в `OnModelCreating`, что если вы выполняли с пошаговым руководством, будет выглядеть следующим образом:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Properties<int>()
        .Where(p => p.Name.EndsWith("Key"))
        .Configure(p => p.IsKey());

    modelBuilder.Conventions.Add(new DateTime2Convention());
}
```

Как вы видите, мы добавим экземпляр нашей соглашения в коллекцию соглашений. Наследование от соглашение предоставляет удобный способ группировки и совместное использование соглашения несколькими командами или проектов. Например, можно создать общий набор соглашений, что все организации проекты, использующие библиотеки классов.

Настраиваемые атрибуты

Чтобы включить новые атрибуты для использования при настройке модели — еще одно отличное применение соглашений. Чтобы проиллюстрировать это, давайте создадим атрибут, который можно использовать для пометки свойств строки как Юникод.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class NonUnicode : Attribute
{}
```

Теперь давайте создадим соглашение о этот атрибут применяется к нашей модели:

```
modelBuilder.Properties()
    .Where(x => x.GetCustomAttributes(false).OfType<NonUnicode>().Any())
    .Configure(c => cIsUnicode(false));
```

Данное соглашение мы можем добавить атрибут `NonUnicode` к любому из наших свойства строки, что означает столбец в базе данных будет храниться как `varchar`, а не `nvarchar`.

Следует отметить о это соглашение о том, что если поместить атрибут `NonUnicode` на ничего, кроме строковое свойство, а затем он вызовет исключение. Это связано с `IsUnicode` невозможно настроить для любого типа, кроме строки. В этом случае после этого можно выполнять соглашению об более точным, позволяя отфильтровывает все, что не является строкой.

Хотя выше соглашение работает для определения настраиваемых атрибутов нет другого API, который может быть гораздо проще в использовании, особенно если вы хотите использовать свойства класса атрибутов.

В этом примере мы собираемся обновить наш атрибут и измените его атрибут IsUnicode, чтобы он выглядел следующим образом:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
internal class IsUnicode : Attribute
{
    public bool Unicode { get; set; }

    public IsUnicode(bool isUnicode)
    {
        Unicode = isUnicode;
    }
}
```

Получив это, можно установить логическое значение для наших атрибутов, чтобы понять в соответствии с соглашением, ли свойство должно быть Юникодом. Это можно делать в соглашение, которое уже имеется, обратившись к ClrProperty класса конфигурации следующим образом:

```
modelBuilder.Properties()
    .Where(x => x.GetCustomAttributes(false).OfType<IsUnicode>().Any())
    .Configure(c => cIsUnicode(c.Clr PropertyInfo.GetCustomAttribute<IsUnicode>().Unicode));
```

Это достаточно просто, но есть более краткий способ сделать это с помощью Having метод соглашений API. Having, метод имеет параметр типа Func<PropertyInfo, T>, принимающий класс PropertyInfo так же, как Where метод, но должен вернуть объект. Если возвращаемый объект имеет значение null, то свойство не будет настроен, что может сделать возможной фильтрацию свойств с ним так же, как Where, но он отличается тем, что он также помещается возвращаемый объект и передать его в метод Configure. Это работает следующим образом:

```
modelBuilder.Properties()
    .Having(x =>x.GetCustomAttributes(false).OfType<IsUnicode>().FirstOrDefault())
    .Configure((config, att) => configIsUnicode(att.Unicode));
```

Пользовательские атрибуты не единственная причина для использования Having метод, полезно в любом месте, необходимо делать выводы о том, что фильтрации по при настройке типами или свойствами.

Настройка типов

До сих всех соглашений были для свойств, но существует еще одна область API соглашений для настройки типов в модели. Процесс выполняется аналогично соглашения об именах, который мы видели в данный момент, но параметры настройки внутри будет объекты вместо свойства уровня.

Среди прочего, соглашения об уровне типов может быть очень полезно при изменении таблицы соглашение об именовании, сопоставляемый существующую схему, отличную от EF по умолчанию или создать новую базу данных с различных соглашения об именовании. Для этого необходимо сначала метод, который может принимать TypeInfo типа в нашей модели и возвращать имя таблицы для этого типа, которое должно быть:

```
private string GetTableName(Type type)
{
    var result = Regex.Replace(type.Name, ".[A-Z]", m => m.Value[0] + "_" + m.Value[1]);

    return result.ToLower();
}
```

Этот метод принимает значение типа и возвращает строку, которая используется нижний регистр с символами подчеркивания, а не CamelCase. В нашей модели это означает, что таблица с именем продукта будет сопоставлено класс ProductCategory_категории вместо ProductCategories.

После получения этого метода можно обращаться к нему в соглашении следующим образом:

```
modelBuilder.Types()
    .Configure(c => c.ToTable(GetTableName(cClrType)));
```

Это соглашение настраивает каждый тип в нашей модели, которое будет сопоставлено имя таблицы, которое возвращается из нашего метода GetTableName. Это соглашение эквивалентен вызову totable-метод для каждой сущности в модели с помощью Fluent API.

Следует заметить, об этом, что при вызове totable-EF будет иметь строку, указанных в качестве точное имя таблицы, без каких-либо преобразования во множественную форму, что обычно делается при определении имен таблиц. Вот почему имя таблицы из наших соглашения — продукт_категории вместо продукта_категории. Чтобы устраниТЬ, в нашем соглашении, делая вызов к службе преобразования во множественную форму, сами.

В следующем коде мы будем использовать [разрешение зависимостей](#) компонент, добавленный в EF6, обращение к службе преобразования во множественную форму, приходилось использовать EF и pluralize наших имён таблицы.

```
private string GetTableName(Type type)
{
    var pluralizationService = DbConfiguration.DependencyResolver.GetService<IPluralizationService>();

    var result = pluralizationService.Pluralize(type.Name);

    result = Regex.Replace(result, ".[A-Z]", m => m.Value[0] + "_" + m.Value[1]);

    return result.ToLower();
}
```

NOTE

Универсальная версия GetService является методом расширения в пространстве имен System.Data.Entity.Infrastructure.DependencyResolution, необходимо добавить с помощью инструкции к контексту, чтобы их использовать.

Totable- и наследование

Еще один важный аспект ToTable том, что если вы явным образом сопоставить тип для данной таблицы, а затем его можно изменить стратегию сопоставления, который будет использовать EF. Если вы вызываете метод ToTable для каждого типа в иерархии наследования, передавая имя типа как имя таблицы, как это делалось выше, затем мы изменим стратегию сопоставления по умолчанию таблица на иерархию (ТРН) таблица на тип (ТРТ). Лучший способ описания — with конкретный пример:

```

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Manager : Employee
{
    public string SectionManaged { get; set; }
}

```

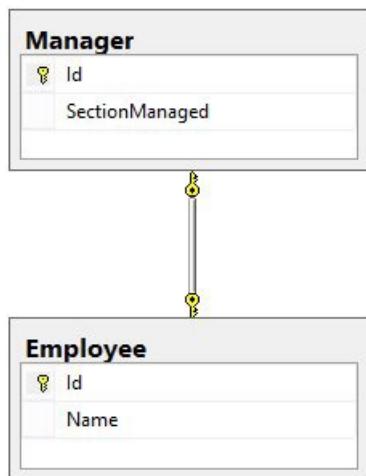
По умолчанию сотрудника и руководителя сопоставляются с той же таблицей (сотрудники) в базе данных. Таблица будет содержать сотрудники и менеджеры со столбцом дискриминатора, который сообщит, какой тип экземпляра хранится в каждой строке. Такое сопоставление ТРН, так как для одной таблицы для иерархии. Тем не менее если вызвать метод ToTable в обоих классах затем каждого типа будет вместо этого можно сопоставить с собственную таблицу, а также называется ТРТ, так как каждый тип имеет собственную таблицу.

```

modelBuilder.Types()
    .Configure(c=>c.ToTable(cClrType.Name));

```

Приведенный выше код будет сопоставлен структуру таблицы, которая выглядит следующим образом:



Можно избежать этого и поддерживать ТРН сопоставление по умолчанию, несколькими способами:

1. Вызовите метод ToTable с тем же именем таблицы, для каждого типа в иерархии.
2. Вызовите метод ToTable только на базовый класс для иерархии, в нашем примере, которая была бы сотрудник.

Порядок выполнения

Соглашения о работают в виде последнего wins, так же, как Fluent API. Это означает, что при написании два соглашения, настройте один и тот же параметр этого свойства, а затем последнее из них для выполнения wins. Например в приведенном ниже коде Максимальная длина для всех строк имеет значение 500, но мы затем настройте все свойства «Name» в модели, чтобы максимальная длина 250.

```
modelBuilder.Properties<string>()
    .Configure(c => c.HasMaxLength(500));

modelBuilder.Properties<string>()
    .Where(x => x.Name == "Name")
    .Configure(c => c.HasMaxLength(250));
```

Так как соглашение об установке максимальной длины до 250 после того, который задает все строки до 500, все свойства «Name» в нашей модели будет иметь MaxLength 250 при других строк, например, описания, составит 500. Подобное использование соглашения означает, что может предоставить общие соглашения для типов или свойств в модели и затем переопределить их подмножества, которые отличаются.

Fluent API и заметки к данным может также использоваться для переопределения соглашения в конкретных случаях. В примере выше если бы мы использовали Fluent API для задания Максимальная длина свойства затем было бы поместить ее до или после соглашения, так как более конкретный Fluent API бьет соглашение по дополнительной конфигурации.

Встроенные соглашения

Так как соглашения об именовании может повлиять на соглашения Code First по умолчанию, может быть полезно добавить соглашения для выполнения до или после другим соглашением. Для этого можно использовать методы AddBefore и AddAfter коллекции соглашения в вашем производном DbContext. Добавить приведенный ниже класс соглашение, созданную ранее будет выполняться перед встроенные в соглашении обнаруживать ключ.

```
modelBuilder.Conventions.AddBefore<IdKeyDiscoveryConvention>(new DateTime2Convention());
```

Это будет наиболее полезны при добавлении соглашения, которые должны выполняться до или после встроенных правил, список встроенных соглашений можно найти здесь: [пространства имен System.Data.Entity.ModelConfiguration.Conventions](#).

Можно также удалить соглашения, которые необходимо применить к модели. Чтобы удалить соглашение, используйте метод Remove. Ниже приведен пример удаления PluralizingTableNameConvention.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
}
```

Правила на основе моделей

27.09.2018 • 9 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Соглашения для модели на основе являются расширенный способ настройки модели на основе соглашения. В большинстве случаев [Custom соглашение об API для Code First на DbModelBuilder](#) следует использовать. Понимание соглашения об API-интерфейса DbModelBuilder рекомендуется перед использованием соглашений для модели на основе.

Соглашения для модели на основе разрешение на создание соглашений, которые влияют на свойства и таблиц, которые не могут быть изменены через стандартные соглашения. Примеры этих являются столбцами дискриминатора в одна таблица на иерархию модели и столбцами независимом сопоставлении.

Создание соглашения

Первым шагом создания модели на основе соглашения, выбирая при в конвейере соглашению необходимо применить к модели. Существует два типа соглашения для модели, концептуальная (C-Space) и Store (S-Space). C-Space соглашение применяется к модели, сборка приложения, тогда как S-Space соглашение применяется к версии модели, которая представляет базу данных и элементы управления, такими как автоматически созданные столбцы имеют имена.

Соглашение для модели — это класс, расширяющий `IConceptualModelConvention` или `IStoreModelConvention`. Эти интерфейсы, принимающие универсальный тип, который может иметь тип `MetadataItem`, который используется для фильтрации типа данных, к которому применяется соглашение.

Добавление соглашения

Соглашения для модели, добавляются в так же, как классы регулярных соглашения. В `OnModelCreating` метод, добавить соглашение в список соглашений для модели.

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

public class BlogContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Add<MyModelBasedConvention>();
    }
}
```

Также можно добавить соглашение по отношению к другим соглашением, с помощью

Conventions.AddBefore<> или Conventions.AddAfter<> методы. Дополнительные сведения о правилах, к которым применяется Entity Framework см. в разделе "Примечания".

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.AddAfter<IdKeyDiscoveryConvention>(new MyModelBasedConvention());
}
```

Пример: Соглашение для модели дискриминатора

Переименование столбцов, созданных EF является примером того, что невозможно выполнить с помощью другие соглашения об API-интерфейсы. Это ситуация где с помощью соглашения для модели — это единственный вариант.

Пример того, как использовать соглашение на основе модели для настройки создаваемых столбцов рекомендуется именовать способ названы дискриминатора. Ниже приведен пример простой модели на основе соглашения, которое изменяет имя каждого столбца в модели с именем «Дискриминатор» на «EntityType». Сюда входят столбцы, что разработчик просто с именем «Дискриминатор». Так как столбец «Дискриминатор» — это созданный столбец это нужно выполнить в пространстве S.

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

class DiscriminatorRenamingConvention : IStoreModelConvention<EdmProperty>
{
    public void Apply(EdmProperty property, DbModel model)
    {
        if (property.Name == "Discriminator")
        {
            property.Name = "EntityType";
        }
    }
}
```

Пример: Общие IA переименование соглашение

Еще один более сложный пример модели на основе соглашений в действии — настроить так, что именуются независимых сопоставлений (IAs). Это ситуация, где применяются из-за создания IAs в EF соглашения для модели, а не присутствуют в модели, можно получить доступ к API-интерфейса DbModelBuilder.

Когда EF создает IA, он создает столбец с именем EntityType_KeyName. Например для связи с именем Customer с ключевым столбцом с именем CustomerId, столбец с именем Customer_CustomerId будут сформированы. Следующие ленты соглашение "_" символа из имени столбца, который создается для IA.

```

using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

// Provides a convention for fixing the independent association (IA) foreign key column names.
public class ForeignKeyNamingConvention : IStoreModelConvention<AssociationType>
{
    public void Apply(AssociationType association, DbModel model)
    {
        // Identify ForeignKey properties (including IAs)
        if (association.IsForeignKey)
        {
            // rename FK columns
            var constraint = association.Constraint;
            if (DoPropertiesHaveDefaultNames(constraint.FromProperties, constraint.ToRole.Name,
                constraint.ToProperties))
            {
                NormalizeForeignKeyProperties(constraint.FromProperties);
            }
            if (DoPropertiesHaveDefaultNames(constraint.ToProperties, constraint.FromRole.Name,
                constraint.FromProperties))
            {
                NormalizeForeignKeyProperties(constraint.ToProperties);
            }
        }
    }

    private bool DoPropertiesHaveDefaultNames(ReadOnlyMetadataCollection<EdmProperty> properties, string
        roleName, ReadOnlyMetadataCollection<EdmProperty> otherEndProperties)
    {
        if (properties.Count != otherEndProperties.Count)
        {
            return false;
        }

        for (int i = 0; i < properties.Count; ++i)
        {
            if (!properties[i].Name.EndsWith("_" + otherEndProperties[i].Name))
            {
                return false;
            }
        }
        return true;
    }

    private void NormalizeForeignKeyProperties(ReadOnlyMetadataCollection<EdmProperty> properties)
    {
        for (int i = 0; i < properties.Count; ++i)
        {
            int underscoreIndex = properties[i].Name.IndexOf('_');
            if (underscoreIndex > 0)
            {
                properties[i].Name = properties[i].Name.Remove(underscoreIndex, 1);
            }
        }
    }
}

```

Расширение существующего соглашения

Если вам нужно написать соглашением, похожее на одно из соглашений, которые Entity Framework уже применяется к модели всегда можно расширить соглашение, чтобы избежать необходимости переписывать его заново. Примером этого является замените существующие сопоставления соглашение о на новый

идентификатор. Дополнительное преимущество для переопределения в соответствии с соглашением ключа — что переопределенный метод будет вызван только в том случае, если отсутствует ключ уже определено или задано явно. Список соглашений, используемых Entity Framework можно найти здесь:
<http://msdn.microsoft.com/library/system.data.entity.modelconfiguration.conventions.aspx>.

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;

// Convention to detect primary key properties.
// Recognized naming patterns in order of precedence are:
// 1. 'Key'
// 2. [type name]Key
// Primary key detection is case insensitive.
public class CustomKeyDiscoveryConvention : KeyDiscoveryConvention
{
    private const string Id = "Key";

    protected override IEnumerable<EdmProperty> MatchKeyProperty(
        EntityType entityType, IEnumerable<EdmProperty> primitiveProperties)
    {
        Debug.Assert(entityType != null);
        Debug.Assert(primitiveProperties != null);

        var matches = primitiveProperties
            .Where(p => Id.Equals(p.Name, StringComparison.OrdinalIgnoreCase));

        if (!matches.Any())
        {
            matches = primitiveProperties
                .Where(p => (entityType.Name + Id).Equals(p.Name, StringComparison.OrdinalIgnoreCase));
        }

        // If the number of matches is more than one, then multiple properties matched differing only by
        // case--for example, "Key" and "key".
        if (matches.Count() > 1)
        {
            throw new InvalidOperationException("Multiple properties match the key convention");
        }

        return matches;
    }
}
```

Затем необходимо добавить наш новое соглашение перед Конвенции существующего ключа. После того как мы добавлен CustomKeyDiscoveryConvention, мы можем удалить IdKeyDiscoveryConvention. Если мы не удалим существующий IdKeyDiscoveryConvention, это соглашение будет по-прежнему иметь приоритет над конвенцией идентификатор обнаружения, так как он выполняется, во-первых, но в случае, когда свойство «key» не найдено, будет выполняться в соответствии с соглашением «id». Мы видим это поведение, поскольку каждый соглашение видит модели, при обновлении с предыдущей соглашением (а не на управлении его независимо друг от друга и все объединяемых друг с другом), чтобы если, например, предыдущий соглашение обновлены так, то имя столбца могут представлять интерес для вашего настраиваемого соглашения (если до этого имени не интерес), то оно будет применяться к этому столбцу.

```
public class BlogContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.AddBefore<IdKeyDiscoveryConvention>(new CustomKeyDiscoveryConvention());
        modelBuilder.Conventions.Remove<IdKeyDiscoveryConvention>();
    }
}
```

Примечания

Список соглашений, применяемых в настоящее время платформа Entity Framework можно найти в документации MSDN: <http://msdn.microsoft.com/library/system.data.entity.modelconfiguration.conventions.aspx>. Этот список извлекаются непосредственно из исходного кода. Исходный код для платформы Entity Framework 6 можно найти в [GitHub](#) и хорошо подходят многие из соглашения, используемые платформой Entity Framework отправных точек для пользовательской модели на основе соглашений.

Fluent API - связи

13.09.2018 • 10 minutes to read • [Edit Online](#)

NOTE

Эта страница содержит сведения о настройке связи в модели Code First с помощью текущего API. Общие сведения о связях в EF и способах доступа к данным и управления ими с помощью связей см. в разделе [связей и свойств навигации](#).

При работе в режиме Code First, можно определить модель, определение классов CLR вашего домена. По умолчанию Entity Framework использует соглашения Code First для сопоставления классов со схемой базы данных. При использовании Code First соглашения об именовании, в большинстве случаев можно положиться на Code First, устанавливать отношения между таблицами на основе внешних ключей и свойства навигации, которые определяются в классах. Если не использовать соглашения при определении классов, если вы хотите изменить способ правила работают, можно использовать fluent API или заметки к данным для настройки классов, поэтому Code First можно сопоставить связи между таблицами.

Вступление

При настройке связи с помощью текущего API, сначала с экземпляром EntityTypeConfiguration и затем используется метод HasRequired, HasOptional или HasMany для указания типа отношения, в которых участвует эта сущность. Методы HasRequired и HasOptional принимают лямбда-выражение, представляющее свойство навигации ссылки. Метод HasMany принимает лямбда-выражение, представляющее свойство навигации по коллекции. Свойство обратной навигации, которое затем можно настроить с помощью методов WithRequired WithOptional и WithMany. Эти методы имеют перегрузки, которые не принимают аргументы и может использоваться для указания количества элементов с односторонним переходом.

Затем можно настроить свойства внешнего ключа с помощью метода HasForeignKey. Этот метод принимает лямбда-выражение, представляющее свойство для использования в качестве внешнего ключа.

Настройка необходимых к дополнительную связь (к нулю или одному)

В следующем примере настраивается отношение один к нулю или одному. OfficeAssignment имеет InstructorID свойства, которое является первичным ключом и внешнего ключа, так как имя свойства не соответствует соглашение, которое метод HasKey используется для настройки первичного ключа.

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

Настройка связи, где требуются (один к одному) обоих концах

В большинстве случаев Entity Framework может определить, какой тип является зависимым и который является основным для связи. Тем не менее при оба конца связи являются обязательными или обеих сторон необязательно Entity Framework не может идентифицировать зависимым и участника. Если требуются оба конца связи, используйте WithRequiredPrincipal или WithRequiredDependent после метода HasRequired. Когда оба конца связи являются необязательными, используйте WithOptionalPrincipal или WithOptionalDependent после метода HasOptional.

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

Настройка связи Многие ко многим

Приведенный ниже код настраивает отношение "многие ко многим" между типами Course и Instructor. В следующем примере Чтобы создать таблицу соединения используются соглашения Code First по умолчанию. В результате создается таблица CourseInstructor со столбцами Course_CourseID и Instructor_InstructorID.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

Если вы хотите указать имя таблицы соединения и имена столбцов в таблице, необходимо выполнить дополнительную настройку с помощью метода карты. Следующий код приводит к возникновению ошибки CourseInstructor таблицу со столбцами CourseID и InstructorID.

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
{
    m.ToTable("CourseInstructor");
    m.MapLeftKey("CourseID");
    m.MapRightKey("InstructorID");
});
```

Настройка связи с одним свойством навигации

Однонаправленное (также называемый однонаправленные) связи выполняется во время определения только на одном из конечных точках, а не на оба свойства навигации. По соглашению Code First всегда интерпретирует Однонаправленная связь как один ко многим. Например если вы хотите, чтобы однозначное соответствие между Instructor и OfficeAssignment, при наличии свойства навигации на только тип данных Instructor, необходимо использовать текущий API для настройки этой связи.

```
// Configure the primary Key for the OfficeAssignment  
modelBuilder.Entity<OfficeAssignment>()  
    .HasKey(t => t.InstructorID);  
  
modelBuilder.Entity<Instructor>()  
    .IsRequired(t => t.OfficeAssignment)  
    .WithRequiredPrincipal();
```

Включение каскадное удаление

Можно настроить каскадное удаление для связи с помощью метода WillCascadeOnDelete. Если внешний ключ для зависимой сущности не допускает значения NULL, то Code First устанавливает каскадное удаление в отношении. Если внешний ключ для зависимой сущности допускает значения NULL, Code First не устанавливает каскадное удаление для связи, и при удалении участника внешний ключ устанавливается в значение null.

Эти правила cascade delete можно удалить с помощью:

```
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()  
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>()
```

Следующий код настраивает требуемую связь, а затем отключает каскадное удаление.

```
modelBuilder.Entity<Course>()  
    .IsRequired(t => t.Department)  
    .WithMany(t => t.Courses)  
    .HasForeignKey(d => d.DepartmentID)  
    .WillCascadeOnDelete(false);
```

Настройка составного внешнего ключа

Если первичный ключ типа отдела состоял из свойств DepartmentID и Name, вы бы настроили первичный ключ для подразделения и внешнего ключа типов курсов следующим образом:

```
// Composite primary key  
modelBuilder.Entity<Department>()  
    .HasKey(d => new { d.DepartmentID, d.Name });  
  
// Composite foreign key  
modelBuilder.Entity<Course>()  
    .IsRequired(c => c.Department)  
    .WithMany(d => d.Courses)  
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

Переименование внешний ключ, который не определен в модели

Если вы не захотите определения внешнего ключа в типе CLR, но, чтобы задать имя, она должна иметь в базе данных, сделайте следующее:

```
modelBuilder.Entity<Course>()  
    .IsRequired(c => c.Department)  
    .WithMany(t => t.Courses)  
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Настройка имени внешнего ключа, который является исключением из соглашения первого кода

Если свойства внешнего ключа в классе курс называется SomeDepartmentID вместо DepartmentID, необходимо выполнить следующую команду, чтобы указать, что SomeDepartmentID в качестве внешнего ключа:

```
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(c => c.SomeDepartmentID);
```

Модели, используемые в примерах

Для примеров на этой странице используется следующая модель Code First.

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
// add a reference to System.ComponentModel.DataAnnotations DLL
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System;

public class SchoolEntities : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention then the generated tables will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public System.DateTime StartDate { get; set; }
    public int? Administrator { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; private set; }
}

public class Course
{
    public Course()
    {
        this.Instructors = new HashSet<Instructor>();
    }
    // Primary key
    public int CourseID { get; set; }
```

```

public string Title { get; set; }
public int Credits { get; set; }

// Foreign key
public int DepartmentID { get; set; }

// Navigation properties
public virtual Department Department { get; set; }
public virtual ICollection<Instructor> Instructors { get; private set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}

public class Instructor
{
    public Instructor()
    {
        this.Courses = new List<Course>();
    }

    // Primary key
    public int InstructorID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public System.DateTime HireDate { get; set; }

    // Navigation properties
    public virtual ICollection<Course> Courses { get; private set; }
}

public class OfficeAssignment
{
    // Specifying InstructorID as a primary
    [Key()]
    public Int32 InstructorID { get; set; }

    public string Location { get; set; }

    // When Entity Framework sees Timestamp attribute
    // it configures ConcurrencyCheck and DatabaseGeneratedPattern=Computed.
    [Timestamp]
    public Byte[] Timestamp { get; set; }

    // Navigation property
    public virtual Instructor Instructor { get; set; }
}

```

Fluent API — Настройка и сопоставление типов и свойств

13.09.2018 • 18 minutes to read • [Edit Online](#)

При работе с Entity Framework Code First поведение по умолчанию — классы РОСО сопоставляются с таблицами, используя набор соглашений, помещенного в EF. Иногда тем не менее, нельзя или нежелательно выполнить эти соглашения и нужно сопоставить сущности с отличные от контекстом.

Существует два основных способа, можно настроить EF использовать отличные от соглашений, а именно [заметки](#) или текущего API файловой системы EFs. Заметки только охватывает часть возможностей, текущего API, поэтому существуют сценарии сопоставления, которые невозможно сделать с помощью заметок. Эта статья предназначена для демонстрации использовать текущий API для настройки свойств.

Fluent API для code first чаще всего осуществляется путем переопределения [OnModelCreating](#) метод в производном [DbContext](#). Приведенные ниже примеры предназначены для показано, как выполнять различные задачи с помощью текущего api и скопируйте код и настроить его в соответствии с вашей модели, если вы хотите увидеть модель, они могут использоваться в качестве-то он предоставляемся в конце этой статьи.

Параметры на уровне модели

Схема по умолчанию (для EF6 и выше)

Начиная с EF6 можно использовать [HasDefaultSchema](#) метод на [DbModelBuilder](#) для указания схемы базы данных для таблиц, хранимых процедур, и тд. Этот параметр по умолчанию будет переопределен для любых объектов, которые явным образом настроить другую схему для.

```
modelBuilder.HasDefaultSchema("sales");
```

Соглашения об именовании (для EF6 и выше)

Начиная с EF6, можно создать собственные правила, чтобы дополнить тех, которые включены в Code First. Дополнительные сведения см. в разделе [первый соглашения о написании пользовательского кода](#).

Сопоставление свойств

[Свойство](#) метод используется для настройки атрибутов для каждого свойства, относящиеся к сущности или сложного типа. Метод [свойства](#) используется для получения объекта конфигурации для заданного свойства. Параметры в объекте конфигурации используются только в настраиваемый; тип Например [IsUnicode](#) доступны только для свойства строки.

Настройка первичного ключа

Действует соглашение, Entity Framework для первичных ключей:

1. Ваш класс определяет свойство, имя которого является «ID» или «Id»
2. или имя класса, за которым следует «ID» или «Id»

Чтобы явно задать свойство в качестве первичного ключа, можно использовать метод [HasKey](#). В следующем примере метод [HasKey](#) используется для настройки [InstructorID](#) первичный ключ типа [OfficeAssignment](#).

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

Настройка составного первичного ключа

В следующем примере настраивается свойство DepartmentID и имя составного первичного ключа типа отдела.

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

Выключение идентификации для числовых первичных ключей

В следующем примере свойство DepartmentID для System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None, чтобы указать, что значение не создается в базе данных.

```
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

Указание Максимальная длина свойства

В следующем примере свойство Name должно быть не более 50 символов. Если значение длиннее 50 символов, вы получите [DbEntityValidationException](#) исключение. Если Code First создает базу данных из этой модели она будет также задан Максимальная длина имени столбца до 50 символов.

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

Настройка свойства обязательным

В следующем примере свойство Name является обязательным. Если вы не укажете имя, вы получите исключение [DbEntityValidationException](#). Если Code First создает базу данных из этой модели выберите столбец, используемый для хранения этого свойства обычно будет не допускающие значения NULL.

NOTE

В некоторых случаях может оказаться невозможным для столбца в базе данных и не допускало, несмотря на то, что свойство является обязательным. Например, когда с помощью данных стратегии наследование TPH для нескольких типов хранится в одной таблице. Если производный тип содержит свойство столбца невозможно не допускающие значения NULL, так как не все типы в иерархии, это свойство будет задано.

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

Настройка индекса для одного или нескольких свойств

NOTE

EF6.1 и более поздних версий только -индекс, представляющий атрибут появился в Entity Framework 6.1. При использовании более ранней версии сведения в этом разделе не применяется.

Создание индексов не поддерживается в собственном коде Fluent API-интерфейсом, но чтобы использовать поддержку **IndexAttribute** через Fluent API. Атрибуты индекса обрабатываются путем включения заметки в виде модели на модель, которая затем преобразуется в индекс в базе данных далее в конвейере. Можно вручную добавить эти же заметок с помощью Fluent API.

Для этого проще всего создать экземпляр **IndexAttribute**, содержащий все параметры для нового индекса. Затем можно создать экземпляр **IndexAnnotation** это определенного типа EF, которое преобразует **IndexAttribute** параметры в заметку в виде модели, которые могут быть сохранены на модели EF. Это может быть передан **HasColumnAnnotation** метод Fluent API-интерфейсы, указав имя **индекс** для заметки.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute()));
```

Полный список параметров, доступных в **IndexAttribute**, см. в разделе **индекс** раздел [заметки данных Code First](#). Сюда входят Настройка имя индекса, создание уникальных индексов и создания индексы с несколькими столбцами.

Можно указать несколько заметок индекса по одному свойству, передавая ему массив **IndexAttribute** конструктору **IndexAnnotation**.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation(
        "Index",
        new IndexAnnotation(new[]
        {
            new IndexAttribute("Index1"),
            new IndexAttribute("Index2") { IsUnique = true }
        }));
    ));;
```

Указание не для сопоставления свойства CLR со столбцом в базе данных

Приведенный ниже показано, как указать, что свойство на основе типа CLR не сопоставлен со столбцом в базе данных.

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

Сопоставление свойства CLR для заданного столбца в базе данных

Следующий пример сопоставляет свойство CLR имя столбца базы данных «отдел».

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

Переименование внешний ключ, который не определен в модели

Если вы не захотите определения внешнего ключа на основе типа CLR, но, чтобы задать имя, она должна иметь в базе данных, сделайте следующее:

```
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Настройка ли строковое свойство поддерживать содержимое в Юникоде

По умолчанию строки являются Юникода (nvarchar в SQL Server). Чтобы указать, что строка должна иметь

типа varchar, можно использовать метод IsUnicode.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .IsUnicode(false);
```

Настройка типа данных столбца базы данных

[HasColumnType](#) метод, который позволяет сопоставление с различными представлениями один и тот же базовый тип. С помощью этого метода не позволяют выполнять любые преобразования данных во время выполнения. Обратите внимание на то, что IsUnicode является предпочтительным способом настройки столбцов varchar, так как он зависит от базы данных.

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

Настройка свойств для сложного типа

Существует два способа настройки скалярные свойства сложного типа.

ComplexTypeConfiguration можно вызвать свойство.

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

Можно также использовать нотацию для доступа к свойству сложного типа.

```
modelBuilder.Entity<OnsiteCourse>()
    .Property(t => t.Details.Location)
    .HasMaxLength(20);
```

Настройка свойств для использования в качестве маркера оптимистичного параллелизма

Чтобы указать, что свойство в сущности представляет маркер параллелизма, можно использовать атрибут ConcurrencyCheck или метод IsConcurrencyToken.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

Можно также использовать метод IsRowVersion свойство может быть версию строки в базе данных.

Задание свойства следует убедиться, что версия строки автоматически настраивает его, чтобы быть маркера оптимистичного параллелизма.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsRowVersion();
```

Сопоставление типов

Указывается, что класс является сложным типом

По соглашению типом, который не имеет первичного ключа указан рассматривается как сложный тип.

Существуют некоторые сценарии, где Code First не обнаружит сложного типа (например, если у вас есть свойство с именем идентификатор, но не значит, для того, чтобы быть первичным ключом). В таких случаях используется fluent API для явного указания, что тип является сложным типом.

```
modelBuilder.ComplexType<Details>();
```

Указание не для того, чтобы сопоставить тип CLR сущности в таблицу в базе данных

Приведенный ниже показано, как исключение из таблицы в базе данных, сопоставляемого типа CLR.

```
modelBuilder.Ignore<OnlineCourse>();
```

Сопоставление типа сущности в конкретную таблицу в базе данных

Все свойства отдела будет сопоставлено со столбцами в таблице, именуемой t_ отдела.

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

Можно также указать имя схемы следующим образом:

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

Сопоставление наследования одна таблица на иерархию (TPH)

В случае сопоставления TPH всех типов в иерархии наследования сопоставляются с одной таблицей. Столбец дискриминатора используется для определения типа каждой строки. При создании модели с помощью Code First, TPH — это стратегия по умолчанию для типов, которые участвуют в иерархии наследования. По умолчанию столбец дискриминатора добавляется в таблицу с именем «Дискриминатор» и имя типа CLR каждого типа в иерархии используется для значения дискриминатора. Поведение по умолчанию можно изменить с помощью текущего API.

```
modelBuilder.Entity<Course>()
    .Map<Course>(m => m.Requires("Type").HasValue("Course"))
    .Map<OnsiteCourse>(m => m.Requires("Type").HasValue("OnsiteCourse"));
```

Сопоставление наследования одна таблица на тип (TPT)

В сценарии сопоставления TPT все типы сопоставляются с отдельными таблицами. Как свойства базового типа, так и свойства производного типа хранятся в таблице, сопоставленной с этим типом. Таблицы, которые сопоставляются производные типы также хранить внешний ключ, который соединяет производной таблицы с базовой таблицей.

```
modelBuilder.Entity<Course>().ToTable("Course");
modelBuilder.Entity<OnsiteCourse>().ToTable("OnsiteCourse");
```

Сопоставление наследования таблица на конкретный класс (TPC)

В сценарии сопоставления TPC все не являющиеся абстрактными типы в иерархии сопоставляются с отдельными таблицами. Таблицы, которые сопоставляются производные классы имеют связана ни таблицей, которая сопоставляется с базовым классом в базе данных. Все свойства класса, включая унаследованные, сопоставляются столбцы соответствующей таблицы.

Вызовите метод MapInheritedProperties для настройки каждого производного типа. MapInheritedProperties

повторно отображает все свойства, унаследованные от базового класса в новые столбцы в таблице для производного класса.

NOTE

Обратите внимание, что так как не используют совместно таблицы, участвующие в иерархии наследования TPC первичного ключа ключи повторяющиеся сущности при вставке в таблицах, сопоставленных с подклассами, если сформированный базой данных значения с тем же начальное значение идентификатора. Чтобы решить эту проблему, можно указать разные начальное значение для каждой таблицы или отключить удостоверения на свойство первичного ключа. Удостоверение значение по умолчанию для ключевых свойств целое число, при работе в режиме Code First.

```
modelBuilder.Entity<Course>()
    .Property(c => c.CourseID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

modelBuilder.Entity<OnsiteCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnsiteCourse");
});

modelBuilder.Entity<OnlineCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnlineCourse");
});
```

Сопоставление свойств типа сущности с несколькими таблицами в базе данных (разбиение сущностей)

Разбиение сущностей позволяет свойства типа сущности и распределяются по нескольким таблицам. В следующем примере сущность Department разбивается на две таблицы: отдела и DepartmentDetails. Разделение сущностей использует несколько вызовов метода карты для сопоставления подмножества свойств в конкретную таблицу.

```
modelBuilder.Entity<Department>()
    .Map(m =>
{
    m.Properties(t => new { t.DepartmentID, t.Name });
    m.ToTable("Department");
})
    .Map(m =>
{
    m.Properties(t => new { t.DepartmentID, t.Administrator, t.StartDate, t.Budget });
    m.ToTable("DepartmentDetails");
});
```

Сопоставление нескольких типов сущностей к одной таблице в базе данных (разбиение таблиц)

Следующий пример сопоставляет двумя типами сущностей, которые совместно используют первичный ключ для одной таблицы.

```

modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .IsRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");

```

Сопоставление типа сущности с хранимые процедуры Insert/Update/Delete (для EF6 и выше)

Начиная с EF6 можно сопоставить сущность использовать хранимые процедуры для вставки обновления и удаления. Дополнительные сведения см. в разделе, [код первого Insert/Update/Delete хранимые процедуры](#).

Модели, используемые в примерах

Для примеров на этой странице используется следующая модель Code First.

```

using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
// add a reference to System.ComponentModel.DataAnnotations DLL
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System;

public class SchoolEntities : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention then the generated tables will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public System.DateTime StartDate { get; set; }
    public int? Administrator { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; private set; }
}

public class Course
{
    public Course()
    {
        this.Instructors = new HashSet<Instructor>();
    }
}

```

```

// Primary key
public int CourseID { get; set; }

public string Title { get; set; }
public int Credits { get; set; }

// Foreign key
public int DepartmentID { get; set; }

// Navigation properties
public virtual Department Department { get; set; }
public virtual ICollection<Instructor> Instructors { get; private set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}

public class Instructor
{
    public Instructor()
    {
        this.Courses = new List<Course>();
    }

    // Primary key
    public int InstructorID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public System.DateTime HireDate { get; set; }

    // Navigation properties
    public virtual ICollection<Course> Courses { get; private set; }
}

public class OfficeAssignment
{
    // Specifying InstructorID as a primary
    [Key()]
    public Int32 InstructorID { get; set; }

    public string Location { get; set; }

    // When Entity Framework sees Timestamp attribute
    // it configures ConcurrencyCheck and DatabaseGeneratedPattern=Computed.
    [Timestamp]
    public Byte[] Timestamp { get; set; }

    // Navigation property
    public virtual Instructor Instructor { get; set; }
}

```


Fluent API с использованием VB.NET

27.09.2018 • 11 minutes to read • [Edit Online](#)

Во-первых, код позволяет определить модель с помощью c## или классам VB.NET. Дополнительная настройка при необходимости выполняются с помощью атрибутов для классов и свойств или с помощью текущего API. В этом пошаговом руководстве показано, как выполнять с помощью VB.NET конфигурация текущего API.

Здесь предполагается, что у вас есть базовое представление о Code First. См. Дополнительные сведения о Code First следующие пошаговые руководства.

- [Code First в новой базе данных](#)
- [Code First для существующей базы данных](#)

Предварительные требования

Для выполнения этого пошагового руководства необходимо иметь по крайней мере Visual Studio 2010 или Visual Studio 2012.

Если вы используете Visual Studio 2010, также необходимо будет иметь [NuGet](#) установлен

Создание приложения

Для простоты мы собираемся создать простое консольное приложение, которое использует Code First для осуществления доступа к данным.

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в меню слева и **консольного приложения**
- Введите **CodeFirstVBSSample** как имя
- Нажмите кнопку **OK**

Определение модели

На этом шаге будут определены VB.NET POCO типов сущностей, которые представляют концептуальную модель. Классы не обязательно являются производными от базовых классов или реализовывать интерфейсы.

- Добавьте новый класс в проект, введите **SchoolModel** имени класса
- Замените содержимое нового класса следующим кодом

```
Public Class Department
    Public Sub New()
        Me.Courses = New List(Of Course)()
    End Sub

    ' Primary key
    Public Property DepartmentID() As Integer
    Public Property Name() As String
    Public Property Budget() As Decimal
    Public Property StartDate() As Date
    Public Property Administrator() As Integer?
    Public Overridable Property Courses() As ICollection(Of Course)
End Class
```

```

Public Class Course
    Public Sub New()
        Me.Instructors = New HashSet(Of Instructor)()
    End Sub

    ' Primary key
    Public Property CourseID() As Integer
    Public Property Title() As String
    Public Property Credits() As Integer

    ' Foreign key that does not follow the Code First convention.
    ' The fluent API will be used to configure DepartmentID_FK to be the foreign key for this entity.
    Public Property DepartmentID_FK() As Integer

    ' Navigation properties
    Public Overridable Property Department() As Department
    Public Overridable Property Instructors() As ICollection(Of Instructor)
End Class

Public Class OnlineCourse
    Inherits Course

    Public Property URL() As String
End Class

Partial Public Class OnsiteCourse
    Inherits Course

    Public Sub New()
        Details = New OnsiteCourseDetails()
    End Sub

    Public Property Details() As OnsiteCourseDetails
End Class

' Complex type
Public Class OnsiteCourseDetails
    Public Property Time() As Date
    Public Property Location() As String
    Public Property Days() As String
End Class

Public Class Person
    ' Primary key
    Public Property PersonID() As Integer
    Public Property LastName() As String
    Public Property FirstName() As String
End Class

Public Class Instructor
    Inherits Person

    Public Sub New()
        Me.Courses = New List(Of Course)()
    End Sub

    Public Property HireDate() As Date

    ' Navigation properties
    Private privateCourses As ICollection(Of Course)
    Public Overridable Property Courses() As ICollection(Of Course)
    Public Overridable Property OfficeAssignment() As OfficeAssignment
End Class

Public Class OfficeAssignment
    ' Primary key that does not follow the Code First convention.
    ' The HasKey method is used later to configure the primary key for the entity.
    Public Property InstructorID() As Integer

```

```
Public Property Location() As String  
Public Property Timestamp() As Byte()  
  
' Navigation property  
Public Overridable Property Instructor() As Instructor  
End Class
```

Определение производного контекста

Мы можем начать использование типов из Entity Framework, поэтому нам нужно добавить пакет EntityFramework NuGet.

- ** Проект —> **управление пакетами NuGet...**

NOTE

Если у вас нет **управление пакетами NuGet...** параметр, необходимо установить [последнюю версию NuGet](#)

- Выберите **Online** вкладку
- Выберите **EntityFramework** пакета
- Нажмите кнопку **установки**

Пришло время для определения производного контекста, который представляет сеанс подключения с базой данных, позволяет запрашивать и сохранять данные. Мы определяем контекст, который является производным от System.Data.Entity.DbContext и предоставляет типизированный DbSet< TEntity > для каждого класса в нашей модели.

- Добавьте новый класс в проект, введите **SchoolContext** имени класса
- Замените содержимое нового класса следующим кодом

```
Imports System.Data.Entity  
Imports System.Data.Entity.Infrastructure  
Imports System.Data.Entity.ModelConfiguration.Conventions  
Imports System.ComponentModel.DataAnnotations  
Imports System.ComponentModel.DataAnnotations.Schema  
  
Public Class SchoolContext  
    Inherits DbContext  
  
    Public Property OfficeAssignments() As DbSet(Of OfficeAssignment)  
    Public Property Instructors() As DbSet(Of Instructor)  
    Public Property Courses() As DbSet(Of Course)  
    Public Property Departments() As DbSet(Of Department)  
  
    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)  
        End Sub  
    End Class
```

Настройка с помощью Fluent API

В этом разделе показано, как использовать текущий API-интерфейсы для настройки типов в таблицах, сопоставление, свойства, чтобы сопоставление столбцов и связей между таблицами\тип в модели. Текущий API предоставляется с помощью **DbModelBuilder** введите и чаще всего осуществляется путем переопределения **OnModelCreating** метод **DbContext**.

- Скопируйте следующий код и добавьте его в **OnModelCreating** метода, определенного в **SchoolContext** класс комментарии объясняют, что делает каждое сопоставление

```

' Configure Code First to ignore PluralizingTableName convention
' If you keep this convention then the generated tables
' will have pluralized names.
modelBuilder.Conventions.Remove(Of PluralizingTableNameConvention)()

' Specifying that a Class is a Complex Type

' The model defined in this topic defines a type OnsiteCourseDetails.
' By convention, a type that has no primary key specified
' is treated as a complex type.
' There are some scenarios where Code First will not
' detect a complex type (for example, if you do have a property
' called ID, but you do not mean for it to be a primary key).
' In such cases, you would use the fluent API to
' explicitly specify that a type is a complex type.
modelBuilder.ComplexType(Of OnsiteCourseDetails)()

' Mapping a CLR Entity Type to a Specific Table in the Database.

' All properties of OfficeAssignment will be mapped
' to columns in a table called t_OfficeAssignment.
modelBuilder.Entity(Of OfficeAssignment)().ToTable("t_OfficeAssignment")

' Mapping the Table-Per-Hierarchy (TPH) Inheritance

' In the TPH mapping scenario, all types in an inheritance hierarchy
' are mapped to a single table.
' A discriminator column is used to identify the type of each row.
' When creating your model with Code First,
' TPH is the default strategy for the types that
' participate in the inheritance hierarchy.
' By default, the discriminator column is added
' to the table with the name "Discriminator"
' and the CLR type name of each type in the hierarchy
' is used for the discriminator values.
' You can modify the default behavior by using the fluent API.
modelBuilder.Entity(Of Person)().
    Map(Of Person)(Function(t) t.Requires("Type").
        HasValue("Person")).
    Map(Of Instructor)(Function(t) t.Requires("Type").
        HasValue("Instructor"))

' Mapping the Table-Per-Type (TPT) Inheritance

' In the TPT mapping scenario, all types are mapped to individual tables.
' Properties that belong solely to a base type or derived type are stored
' in a table that maps to that type. Tables that map to derived types
' also store a foreign key that joins the derived table with the base table.
modelBuilder.Entity(Of Course)().ToTable("Course")
modelBuilder.Entity(Of OnsiteCourse)().ToTable("OnsiteCourse")
modelBuilder.Entity(Of OnlineCourse)().ToTable("OnlineCourse")

' Configuring a Primary Key

' If your class defines a property whose name is "ID" or "Id",
' or a class name followed by "ID" or "Id",
' the Entity Framework treats this property as a primary key by convention.
' If your property name does not follow this pattern, use the HasKey method
' to configure the primary key for the entity.
modelBuilder.Entity(Of OfficeAssignment)().
    HasKey(Function(t) t.InstructorID)

```

```

' Specifying the Maximum Length on a Property

' In the following example, the Name property
' should be no longer than 50 characters.
' If you make the value longer than 50 characters,
' you will get a DbEntityValidationException exception.
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    HasMaxLength(60)

' Configuring the Property to be Required

' In the following example, the Name property is required.
' If you do not specify the Name,
' you will get a DbEntityValidationException exception.
' The database column used to store this property will be non-nullable.
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    IsRequired()

' Switching off Identity for Numeric Primary Keys

' The following example sets the DepartmentID property to
' System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None to indicate that
' the value will not be generated by the database.
modelBuilder.Entity(Of Course)().Property(Function(t) t.CourseID).
    HasDatabaseGeneratedOption(DatabaseGeneratedOption.None)

'Specifying NOT to Map a CLR Property to a Column in the Database
modelBuilder.Entity(Of Department)().
    Ignore(Function(t) t.Administrator)

'Mapping a CLR Property to a Specific Column in the Database
modelBuilder.Entity(Of Department)().Property(Function(t) t.Budget).
    HasColumnName("DepartmentBudget")

'Configuring the Data Type of a Database Column
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    HasColumnType("varchar")

'Configuring Properties on a Complex Type
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Days).
    HasColumnName("Days")
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Location).
    HasColumnName("Location")
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Time).
    HasColumnName("Time")

' Map one-to-zero or one relationship

' The OfficeAssignment has the InstructorID
' property that is a primary key and a foreign key.
modelBuilder.Entity(Of OfficeAssignment)().
    HasRequired(Function(t) t.Instructor).
    WithOptional(Function(t) t.OfficeAssignment)

' Configuring a Many-to-Many Relationship

' The following code configures a many-to-many relationship
' between the Course and Instructor types.
' In the following example, the default Code First conventions
' are used to create a join table.
' As a result the CourseInstructor table is created with
' Course_CourseID and Instructor_InstructorID columns.
modelBuilder.Entity(Of Course)().
    HasMany(Function(t) t.Instructors).

```

```

WithMany(Function(t) t.Courses)

' Configuring a Many-to-Many Relationship and specifying the names
' of the columns in the join table

' If you want to specify the join table name
' and the names of the columns in the table
' you need to do additional configuration by using the Map method.
' The following code generates the CourseInstructor
' table with CourseID and InstructorID columns.
modelBuilder.Entity(Of Course)().
    HasMany(Function(t) t.Instructors).
    WithMany(Function(t) t.Courses).
    Map(Sub(m)
        m.ToTable("CourseInstructor")
        m.MapLeftKey("CourseID")
        m.MapRightKey("InstructorID")
    End Sub)

' Configuring a foreign key name that does not follow the Code First convention

' The foreign key property on the Course class is called DepartmentID_FK
' since that does not follow Code First conventions you need to explicitly specify
' that you want DepartmentID_FK to be the foreign key.
modelBuilder.Entity(Of Course)().
    HasRequired(Function(t) t.Department).
    WithMany(Function(t) t.Courses).
    HasForeignKey(Function(t) t.DepartmentID_FK)

' Enabling Cascade Delete

' By default, if a foreign key on the dependent entity is not nullable,
' then Code First sets cascade delete on the relationship.
' If a foreign key on the dependent entity is nullable,
' Code First does not set cascade delete on the relationship,
' and when the principal is deleted the foreign key will be set to null.
' The following code configures cascade delete on the relationship.

' You can also remove the cascade delete conventions by using:
' modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()
' and modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>().
modelBuilder.Entity(Of Course)().
    HasRequired(Function(t) t.Department).
    WithMany(Function(t) t.Courses).
    HasForeignKey(Function(d) d.DepartmentID_FK).
    WillCascadeOnDelete(False)

```

С помощью модели

Выполним некоторые доступ к данным с помощью **SchoolContext** для просмотра модели в действии.

- Откройте файл Module1.vb, где определена функция Main
- Скопируйте и вставьте следующее определение Module1

```
Imports System.Data.Entity

Module Module1

Sub Main()

    Using context As New SchoolContext()

        ' Create and save a new Department.
        Console.Write("Enter a name for a new Department: ")
        Dim name = Console.ReadLine()

        Dim department = New Department With { .Name = name, .StartDate = DateTime.Now }
        context.Departments.Add(department)
        context.SaveChanges()

        ' Display all Departments from the database ordered by name
        Dim departments =
            From d In context.Departments
            Order By d.Name
            Select d

        Console.WriteLine("All Departments in the database:")
        For Each department In departments
            Console.WriteLine(department.Name)
        Next

    End Using

    Console.WriteLine("Press any key to exit...")
    Console.ReadKey()

End Sub

End Module
```

Теперь можно запустить приложение и протестировать его.

```
Enter a name for a new Department: Computing
All Departments in the database:
Computing
Press any key to exit...
```

Код первой вставки, обновления и удаления хранимых процедур

13.09.2018 • 12 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

По умолчанию Code First настроит все сущности, чтобы выполнить операцию вставки, обновления и удаления команд с помощью прямого доступа к таблицам. Начиная с EF6 можно настроить модель данных Code First с помощью хранимых процедур для некоторых или всех сущностей в модели.

Основные сущности сопоставления

Можно явно задано использование хранимых процедур для вставки, обновления и удаления с помощью Fluent API.

```
modelBuilder  
    .Entity<Blog>()  
    .MapToStoredProcedures();
```

Это приведет к неспособности Code First, использование некоторых соглашений для создания Прогнозируемая форма хранимые процедуры в базе данных.

- Три хранимые процедуры с именами **<type_name>_вставить**, **<type_name>_обновить** и **<type_name>_Удалить** (например, Blog_Insert, Blog_Update и Blog_Delete).
- Имена параметров соответствуют именам свойств.

NOTE

Если вы используете HasColumnName() или атрибут столбца для переименования столбца для заданного свойства, это имя используется для параметров вместо имени свойства.

- **Хранимая процедура insert** будет иметь параметр для каждого свойства, за исключением отмечается как хранения созданных (идентификаторов или вычисляемых). Хранимая процедура должна возвращать результирующий набор со столбцом для каждого свойства хранения созданных.
- **Обновление хранимой процедуры** будет иметь параметр для каждого свойства, за исключением тех, отмеченные хранения созданных шаблон «Computed». Некоторые маркеры параллелизма требует параметра для исходного значения, см. в разделе *маркеры параллелизма* сведения в приведенном ниже разделе. Хранимая процедура должна возвращать результирующий набор со столбцом для каждого вычисляемого свойства.
- **Удаление хранимой процедуры** должен иметь параметр для значения ключа сущности (или несколько параметров, если сущность имеет составной ключ). Кроме того процедура delete также иметь параметры для любых внешних ключей в независимом сопоставлении для целевой таблицы (связей, у которых нет соответствующих свойств внешнего ключа объявлено в сущности). Некоторые маркеры параллелизма требует параметра для исходного значения, см. в разделе *маркеры параллелизма* сведения

в приведенном ниже разделе.

Используя в качестве примера следующий класс:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
}
```

По умолчанию, будет хранимых процедур:

```
CREATE PROCEDURE [dbo].[Blog_Insert]
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
BEGIN
    INSERT INTO [dbo].[Blogs] ([Name], [Url])
    VALUES (@Name, @Url)

    SELECT SCOPE_IDENTITY() AS BlogId
END
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
UPDATE [dbo].[Blogs]
SET [Name] = @Name, [Url] = @Url
WHERE BlogId = @BlogId;
CREATE PROCEDURE [dbo].[Blog_Delete]
    @BlogId int
AS
DELETE FROM [dbo].[Blogs]
WHERE BlogId = @BlogId
```

Переопределение параметров по умолчанию

Вы можете переопределить частично или полностью тем, что было настроено по умолчанию.

Можно изменить имя одной или нескольких хранимых процедур. В этом примере переименовывает хранимую процедуру обновления только.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog")));

```

В этом примере переименовывает всех трех хранимых процедур.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog"))
        .Delete(d => d.HasName("delete_blog"))
        .Insert(i => i.HasName("insert_blog")));

```

В этих примерах вызовы соединяются друг с другом, но можно также использовать синтаксис блока лямбда-выражения.

```

modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
    {
        s.Update(u => u.HasName("modify_blog"));
        s.Delete(d => d.HasName("delete_blog"));
        s.Insert(i => i.HasName("insert_blog"));
    });

```

В этом примере переименовывает параметр для свойства BlogId хранимая процедура update.

```

modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.Parameter(b => b.BlogId, "blog_id")));

```

Эти вызовы являются все к цепочкам и композицию. Ниже приведен пример, который переименовывает все три хранимые процедуры и их параметрах.

```

modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog")
            .Parameter(b => b.BlogId, "blog_id")
            .Parameter(b => b.Name, "blog_name")
            .Parameter(b => b.Url, "blog_url"))
        .Delete(d => d.HasName("delete_blog")
            .Parameter(b => b.BlogId, "blog_id"))
        .Insert(i => i.HasName("insert_blog")
            .Parameter(b => b.Name, "blog_name")
            .Parameter(b => b.Url, "blog_url")));

```

Можно также изменить имена столбцов в результирующем наборе, который содержит значения, сформированные базой данных.

```

modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Result(b => b.BlogId, "generated_blog_identity")));

```

```

CREATE PROCEDURE [dbo].[Blog_Insert]
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
BEGIN
    INSERT INTO [dbo].[Blogs] ([Name], [Url])
    VALUES (@Name, @Url)

    SELECT SCOPE_IDENTITY() AS generated_blog_id
END

```

Связи без внешнего ключа в классе (независимых сопоставлений)

Если свойство внешнего ключа включен в определение класса, соответствующего параметра могут быть переименованы в так же, как любое другое свойство. Если связь существует без свойство внешнего ключа в классе, имя параметра по умолчанию — **<navigation_property_name>_<primary_key_name>**.

Например следующие определения класса может привести параметр Blog_BlogId ожидаемая в хранимые процедуры для вставки и обновления записей.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

Переопределение параметров по умолчанию

Вы можете изменить параметры для внешних ключей, которые не находятся в классе, указав путь к свойство первичного ключа в метод параметр.

```
modelBuilder
    .Entity<Post>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Parameter(p => p.Blog.BlogId, "blog_id")));
```

Если у вас нет свойства навигации на зависимой сущности (например) свойство не Post.Blog) методу связи можно использовать для идентификации на другом конце связи, а затем настройте параметры, соответствующие каждому из свойств ключа.

```
modelBuilder
    .Entity<Post>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Navigation<Blog>(
            b => b.Posts,
            c => c.Parameter(b => b.BlogId, "blog_id"))));
```

Маркеры параллелизма

Обновления и удаления хранимых процедур может понадобиться иметь дело с параллелизмом:

- Если сущность содержит маркеры параллелизма, хранимая процедура может быть Необязательный выходной параметр, который возвращает количество обновленные или удаленные строки (обработано строк). Такие параметры должны быть настроены с помощью метода RowsAffectedParameter.
По умолчанию EF использует возвращаемое значение из ExecuteNonQuery, чтобы определить, сколько строк затронуто. Указание параметра выходные данные строк, затронутых полезно, если выполнить какую-либо логику в sproc, приведет к возвращаемому значению ExecuteNonQuery, неверный (с точки зрения платформы EF) в конце выполнения.
- Для каждого параллелизма существует токен будет параметр с именем **<property_name>_Original** (например, Timestamp_Original). Он будет передаваться исходное значение этого свойства — значение, при запросе из базы данных.
 - Маркеры параллелизма, которые вычисляются по базе данных — например отметки времени —

будет иметь только параметром исходное значение.

- Вычисленный отличные от свойств, заданных как маркеры параллелизма также получат новое значение параметра в процедуре обновления. При этом используется соглашения об именовании, описанные для новых значений. Примером такого маркера объект с помощью URL-адреса блога как маркер параллелизма, новое значение является обязательной, поскольку это могут быть обновлены на новое значение в коде (в отличие от маркер отметки времени, который обновляется только в базе данных).

Ниже приведен пример класса и хранимая процедура update с маркером параллелизма отметки времени.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

```
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max),
    @Timestamp_Original rowversion
AS
    UPDATE [dbo].[Blogs]
    SET [Name] = @Name, [Url] = @Url
    WHERE BlogId = @BlogId AND [Timestamp] = @Timestamp_Original
```

Ниже приведен пример класса и хранимая процедура update с маркером невычисляемый параллелизма.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    [ConcurrencyCheck]
    public string Url { get; set; }
}
```

```
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max),
    @Url_Original nvarchar(max),
AS
    UPDATE [dbo].[Blogs]
    SET [Name] = @Name, [Url] = @Url
    WHERE BlogId = @BlogId AND [Url] = @Url_Original
```

Переопределение параметров по умолчанию

При необходимости можно ввести параметр числа затронутых строк.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.RowsAffectedParameter("rows_affected")));
```

Для базы данных, рассчитанное параллелизма маркеры — где передается только исходное значение — можно просто использовать стандартный параметр переименование механизм для переименования параметра для исходного значения.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.Parameter(b => b.Timestamp, "blog_timestamp")));
```

Для маркеров параллелизма невычисляемый — где как исходное и новое значение передаются — можно использовать перегрузку параметр, который позволяет указать имя для каждого параметра.

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s => s.Update(u => u.Parameter(b => b.Url, "blog_url", "blog_original_url")));
```

Отношения "многие ко многим"

Мы будем использовать следующие классы в качестве примера в этом разделе.

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<Tag> Tags { get; set; }
}

public class Tag
{
    public int TagId { get; set; }
    public string TagName { get; set; }

    public List<Post> Posts { get; set; }
}
```

Многие к много связей могут сопоставляться с хранимыми процедурами, используя следующий синтаксис.

```
modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(t => t.Posts)
    .MapToStoredProcedures();
```

Если указано полностью выполнили настройку по умолчанию используется хранимая процедура следующего вида.

- Две хранимые процедуры с именами **<type_one><type_two>_вставить** и **<type_one><type_two>_удалить** (например, PostTag_Insert и PostTag_Delete).
- Параметры будут иметь значения ключа для каждого типа. Имя каждого параметра **<type_name>_<property_name>** (например, Post_PostId и Tag_TagId).

Вот пример вставки и обновления хранимых процедур.

```
CREATE PROCEDURE [dbo].[PostTag_Insert]
    @Post_PostId int,
    @Tag_TagId int
AS
    INSERT INTO [dbo].[Post_Tags] (Post_PostId, Tag_TagId)
    VALUES (@Post_PostId, @Tag_TagId)
CREATE PROCEDURE [dbo].[PostTag_Delete]
    @Post_PostId int,
    @Tag_TagId int
AS
    DELETE FROM [dbo].[Post_Tags]
    WHERE Post_PostId = @Post_PostId AND Tag_TagId = @Tag_TagId
```

Переопределение параметров по умолчанию

Имена процедур и параметров можно настроить в так же, как сущности хранимой процедуры.

```
modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(t => t.Posts)
    .MapToStoredProcedures(s =>
        s.Insert(i => i.HasName("add_post_tag")
            .LeftKeyParameter(p => p.PostId, "post_id")
            .RightKeyParameter(t => t.TagId, "tag_id"))
        .Delete(d => d.HasName("remove_post_tag")
            .LeftKeyParameter(p => p.PostId, "post_id")
            .RightKeyParameter(t => t.TagId, "tag_id")));
    );
```

Code First Migrations

12.10.2018 • 18 minutes to read • [Edit Online](#)

Code First Migrations — это рекомендуемый способ разработки структуры базы данных в приложении при использовании рабочего процесса Code First. Code First Migrations предоставляет набор средств со следующими функциями:

1. Создание исходной базы данных, которая работает с моделью EF
2. Создание миграций для отслеживания изменений в модели EF
3. Поддержание актуальности базы данных с учетом этих изменений

Следующее пошаговое руководство содержит обзор Code First Migrations в Entity Framework. Вы можете выполнить все пошаговое руководство или сразу перейти в нужный раздел. Здесь представлены следующие разделы.

Привязка начальной модели и базы данных

Прежде чем мы начнем использовать миграции, нам потребуется проект и модель Code First, с которой мы будем работать. В этом пошаговом руководстве мы собираемся использовать каноническую модель **Blog** и **Post**.

- Создайте новое консольное приложение **MigrationsDemo**.
- Добавьте последнюю версию пакета NuGet **EntityFramework** в проект.
 - Средства → Диспетчер пакетов библиотеки → Консоль диспетчера пакетов
 - Запустите команду **EntityFramework Install-Package**
- Добавьте файл **Model.cs** с кодом, показанным ниже. Этот код определяет один класс **Blog**, составляющий нашу модель предметной области, и класс **BlogContext**, который будет нашим контекстом EF Code First.

```
using System.Data.Entity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.Infrastructure;

namespace MigrationsDemo
{
    public class BlogContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
    }
}
```

- Теперь, когда у нас есть модель, пора использовать ее для доступа к данным. Внесите в файл **Program.cs** приведенный ниже код.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

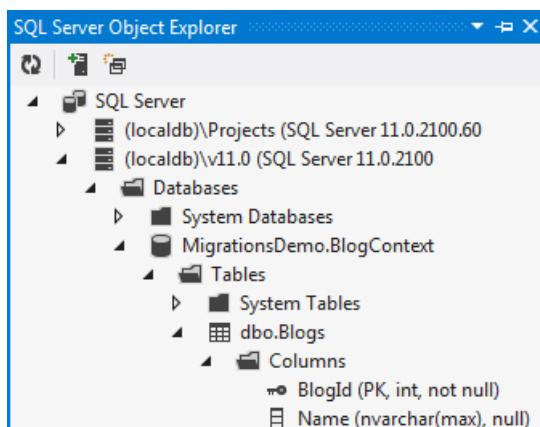
namespace MigrationsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog" });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

- Запустите приложение, и вы увидите, что база данных **.MigrationsCodeDemo.BlogContext** создается автоматически.



Включение миграций

Пора внести дополнительные изменения в нашу модель.

- Давайте внесем в класс Blog свойство Url.

```
public string Url { get; set; }
```

Если бы вы запустили приложение снова, оно выдало бы исключение `InvalidOperationException` с сообщением: *Модель, поддерживающая контекст `BlogContext`, изменилась с момента создания базы данных. Попробуйте обновить базу данных с помощью `Code First Migrations` (<http://go.microsoft.com/fwlink/?LinkId=238269>).*

Как видно из исключения, пора приступить к использованию `Code First Migrations`. Сначала нужно включить миграции для нашего контекста.

- Запустите команду **Enable-Migrations** в консоли диспетчера пакетов.

В результате этой команды в проект будет добавлена папка **Migrations**. В новой папке два файла:

- **Класс конфигурации.** Этот класс позволяет настраивать поведение миграций для контекста. В этом пошаговом руководстве мы будем просто использовать конфигурацию по умолчанию. Поскольку имеется только один контекст *Code First* в проекте, **Enable-Migrations** автоматически заполняет тип контекста, к которому относится эта конфигурация.
- **Миграция InitialCreate.** Эта миграция создана, так как мы уже использовали *Code First* для создания базы данных, прежде чем включили миграции. Код в этой созданной по шаблону миграции представляет объекты, которые уже были созданы в базе данных. В нашем случае это таблица **Blog** со столбцами **BlogId** и **Name**. Имя файла содержит метку времени для удобства упорядочения. Если бы база данных еще не была создана, миграция *InitialCreate* не была бы добавлена в проект. Вместо этого, когда мы впервые вызвали бы *Add-Migration*, код для создания этих таблиц был бы перенесен в новую миграцию.

Несколько моделей для одной целевой базы данных

При использовании версий до EF6 можно было использовать только одну модель *Code First* для создания схемы базы данных и управления ей. Это было связано с наличием всего одной таблицы **_MigrationsHistory** на базу данных, когда невозможно было определить, какая запись какой модели принадлежит.

Начиная с EF6, класс **Configuration** включает свойство **ContextKey**. Это уникальный идентификатор для каждой модели *Code First*. В соответствующий столбец в таблице **_MigrationsHistory** вносятся записи из нескольких моделей, которые используют таблицу совместно. По умолчанию этому свойству присвоено полное имя контекста.

Создание и запуск миграций

Вам нужно знать о двух основных командах *Code First Migrations*.

- **Add-Migration** будет автоматически формировать следующую миграцию на основе изменений, внесенных в модель с момента создания последней миграции.
- **Update-Database** будет применять ожидающие обработки миграции к базе данных.

Нам нужно сформировать шаблон миграции для нового свойства **Url**. Команда **Add-Migration** позволяет нам давать этим миграциям имя. Давайте назовем нашу **AddBlogUrl**.

- Запустите команду **Add-Migration AddBlogUrl** в консоли диспетчера пакетов.
- В папке **Migrations** у нас теперь есть новая миграция **AddBlogUrl**. Имя файла миграции имеет метку времени в целях упорядочения.

```

namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddBlogUrl : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Blogs", "Url", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Blogs", "Url");
        }
    }
}

```

Теперь мы могли бы изменить или дополнить эту миграцию, но все и так в порядке. Мы используем **Update-Database**, чтобы применить эту миграцию к базе данных.

- Запустите команду **Update-Database** в консоли диспетчера пакетов.
- Code First Migrations сравнил миграции из папки **Migrations** с теми, которые были применены к базе данных. Вы увидите, что необходимо применить и запустить миграцию **AddBlogUrl**.

База данных **MigrationsDemo.BlogContext** обновлена и включает столбец **Url** в таблице **Blog**.

Настройка миграции

Пока мы создавали и запускали миграции без изменений. Теперь давайте рассмотрим редактирование кода, который создается по умолчанию.

- Пора внести некоторые изменения в нашу модель. Давайте добавим новое свойство **Rating** в класс **Blog**.

```
public int Rating { get; set; }
```

- Также добавим новый класс **Post**.

```

public class Post
{
    public int PostId { get; set; }
    [MaxLength(200)]
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}

```

- Еще мы добавим коллекцию **Posts** в класс **Blog**, чтобы создать второй конец связи между **Blog** и **Post**.

```
public virtual List<Post> Posts { get; set; }
```

Мы будем использовать команду **Add-Migration**, чтобы с помощью Code First Migrations автоматически создать миграцию. Мы назовем эту миграцию **AddPostClass**.

- Запустите команду **Add-Migration AddPostClass** в консоли диспетчера пакетов.

С помощью Code First Migrations прекрасно удалось создать эти изменения по шаблону, но мы хотим изменить кое-что еще.

1. Сначала давайте добавим уникальный индекс в столбец **Posts.Title** (строки 22 и 29 в приведенном ниже коде).
2. Мы также добавим столбец **Blogs.Rating**, не допускающий значение null. Если в таблице есть данные, новому столбцу будет назначено значение CLR по умолчанию в зависимости от типа данных (оценка — это целое число, так что это будет **0**). Но нам нужно указать значение по умолчанию **3**, чтобы существующие строки в таблице **Blogs** начинались с неплохой оценки. (Значение по умолчанию присваивается в строке 24 приведенного ниже кода.)

```
namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPostClass : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Posts",
                c => new
                {
                    PostId = c.Int(nullable: false, identity: true),
                    Title = c.String(maxLength: 200),
                    Content = c.String(),
                    BlogId = c.Int(nullable: false),
                })
                .PrimaryKey(t => t.PostId)
                .ForeignKey("dbo.Blogs", t => t.BlogId, cascadeDelete: true)
                .Index(t => t.BlogId)
                .Index(p => p.Title, unique: true);

            AddColumn("dbo.Blogs", "Rating", c => c.Int(nullable: false, defaultValue: 3));
        }

        public override void Down()
        {
            DropIndex("dbo.Posts", new[] { "Title" });
            DropIndex("dbo.Posts", new[] { "BlogId" });
            DropForeignKey("dbo.Posts", "BlogId", "dbo.Blogs");
            DropColumn("dbo.Blogs", "Rating");
            DropTable("dbo.Posts");
        }
    }
}
```

Измененная миграция готова к запуску, так что давайте обновим ее с помощью команды **Update-Database**. Давайте укажем флаг **-Verbose**, чтобы вы могли видеть SQL, который выполняет Code First Migrations.

- Запустите команду **Update-Database -Verbose** в консоли диспетчера пакетов.

Перемещение данных или настраиваемый SQL

До сих пор мы рассматривали операции миграции, которые не меняют и не перемещают данные. Теперь давайте посмотрим на примеры, где мы перемещаем данные. Для перемещения данных пока нет собственной поддержки, но мы можем запустить произвольные команды SQL в любой момент в нашем

сценарии.

- Давайте добавим свойство **Post.Abstract** в нашу модель. Позже мы заполним свойство **Abstract** для существующих записей текстом из начала столбца **Content**.

```
public string Abstract { get; set; }
```

Мы будем использовать команду **Add-Migration**, чтобы с помощью Code First Migrations автоматически создать миграцию.

- Запустите команду **Add-Migration AddPostAbstract** в консоли диспетчера пакетов.
- Созданная миграция обрабатывает изменения схемы, но мы хотим предварительно заполнить столбец **Abstract** первыми 100 символами содержимого для каждой записи. Для этого мы можем перейти к SQL и выполнить оператор **UPDATE** после добавления столбца. (Добавление в строке 12 в приведенном ниже коде)

```
namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPostAbstract : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Posts", "Abstract", c => c.String());
            Sql("UPDATE dbo.Posts SET Abstract = LEFT(Content, 100) WHERE Abstract IS NULL");
        }

        public override void Down()
        {
            DropColumn("dbo.Posts", "Abstract");
        }
    }
}
```

Измененная миграция выглядит прекрасно, так что давайте обновим ее с помощью команды **Update-Database**. Необходимо указать флаг **-Verbose**, чтобы видеть выполнение SQL в базе данных.

- Запустите команду **Update-Database -Verbose** в консоли диспетчера пакетов.

Перенос в определенную версию (в том числе более раннюю)

Пока мы всегда выполняли обновление до последней миграции, но иногда могут возникнуть ситуации, когда вы хотите перейти к более поздней или более ранней версии миграции.

Предположим, мы хотим перенести базу данных в состояние до запуска миграции **AddBlogUrl**. Мы можем использовать переключатель **-TargetMigration**, чтобы перейти на эту более раннюю версию миграции.

- Запустите команду **Update-Database -TargetMigration: AddBlogUrl** в консоли диспетчера пакетов.

Эта команда запустит скрипт понижения версии для миграций **AddBlogAbstract** и **AddPostClass**.

Если вы хотите выполнить откат до пустой базы данных, используйте команду **Update-Database -TargetMigration: \$InitialDatabase**.

Получение скрипта SQL

Если другой разработчик хочет использовать эти изменения на своем компьютере, он может просто синхронизироваться, когда мы вернем наши изменения в систему управления версиями. Когда он получит наши новые миграции, он может выполнить команду `Update-Database`, чтобы применить эти изменения локально. Но если мы хотим передать эти изменения на тестовый сервер, а затем и в рабочую среду, возможно, нам потребуется скрипт SQL, который мы сможем передать нашему администратору базы данных.

- Запустите команду **Update-Database**, но на этот раз укажите флаг **-Script**, чтобы записать изменения в скрипт, а не применить. Мы также укажем исходную и целевую миграцию, для которой необходимо создать скрипт. Мы хотим, чтобы скрипт переходил от пустой базы данных (`$InitialDatabase`) до последней версии (миграция **AddPostAbstract**). Если вы не укажете целевую миграцию, *Code First Migrations* будет использовать последнюю версию миграции в качестве целевого объекта. Если вы не укажете исходную миграцию, *Code First Migrations* будет использовать текущее состояние базы данных.
- Запустите команду **Update-Database -Script -SourceMigration: \$InitialDatabase -TargetMigration: AddPostAbstract** в консоли диспетчера пакетов.

Code First Migrations будет выполнять конвейер миграций, но будет не применять изменения, а записывать их в SQL-файл. Созданный скрипт автоматически открывается в Visual Studio, готовый для просмотра или сохранения.

Создание идемпотентных скриптов

Начиная с EF6, если вы укажете **-SourceMigration \$InitialDatabase**, создаваемый скрипт будет идемпотентным. Идемпотентные скрипты могут обновить базу данных любой версии до последней версии (или указанной версии, если вы используете **-TargetMigration**). Созданный скрипт содержит логику для проверки таблицы `__MigrationsHistory` и применяет только изменения, которые еще не применены.

Автоматическое обновление при запуске приложения (инициализатор `MigrateDatabaseToLatestVersion`)

Когда вы развертываете приложение, вы можете настроить автоматическое обновление базы данных (путем применения ожидающих миграций) при запуске приложения. Это можно сделать с помощью регистрации инициализатора базы данных **MigrateDatabaseToLatestVersion**. Инициализатор базы данных просто содержит определенную логику, которая используется для контроля правильной настройки базы данных. Эта логика выполняется при первом использовании контекста в процессе приложения (**AppDomain**).

Мы можем обновить файл **Program.cs**, как показано ниже, чтобы настроить инициализатор **MigrateDatabaseToLatestVersion** для `BlogContext`, прежде чем мы используем контекст (строка 14). Обратите внимание, что необходимо также добавить оператор `using` для пространства имен **System.Data.Entity** (строка 5).

При создании экземпляра этого инициализатора нам нужно указать тип контекста (**BlogContext**) и конфигурацию миграций (**Configuration**). Конфигурация миграций — это класс, который был добавлен в нашу папку **Migrations** при включении *Migrations*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using MigrationsDemo.Migrations;

namespace MigrationsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<BlogContext, Configuration>());
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog" });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Теперь при каждом запуске приложение сначала будет проверять актуальность целевой базы данных и применять незавершенные миграции, если это необходимо.

Автоматическое Code First Migrations

27.09.2018 • 10 minutes to read • [Edit Online](#)

Автоматический перенос позволяет использовать Code First Migrations без файл кода в проекте для каждого изменения, внесенные. Не все изменения, которые могут применяться автоматически — например переименование столбцов требуют использования миграции на основе кода.

NOTE

В этой статье предполагается, что вы знаете, как использовать Code First Migrations в основных сценариях. Если этого не сделать, то нужно будет прочесть [Code First Migrations](#) перед продолжением.

Рекомендации для сред Team

Вы можете перемежать миграций автоматическое и на основе кода, но это не рекомендуется при коллективной разработки. Если вы работаете в команде разработчиков, использующих систему управления версиями следует использовать исключительно автоматической миграции или миграции исключительно на основе кода. Учитывая ограничения автоматический перенос рекомендуется использовать миграция на базе кода в командных средах.

Привязка начальной модели и базы данных

Прежде чем мы начнем использовать миграции, нам потребуется проект и модель Code First, с которой мы будем работать. В этом пошаговом руководстве мы собираемся использовать каноническую модель **Blog** и **Post**.

- Создайте новый **MigrationsAutomaticDemo** консольное приложение
- Добавьте последнюю версию пакета NuGet **EntityFramework** в проект.
 - Средства —> Диспетчер пакетов библиотеки —> Консоль диспетчера пакетов
 - Запустите команду **EntityFramework Install-Package**
- Добавьте файл **Model.cs** с кодом, показанным ниже. Этот код определяет один класс **Blog**, составляющий нашу модель предметной области, и класс **BlogContext**, который будет нашим контекстом EF Code First.

```

using System.Data.Entity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.Infrastructure;

namespace MigrationsAutomaticDemo
{
    public class BlogContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
    }
}

```

- Теперь, когда у нас есть модель, пора использовать ее для доступа к данным. Внесите в файл **Program.cs** приведенный ниже код.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

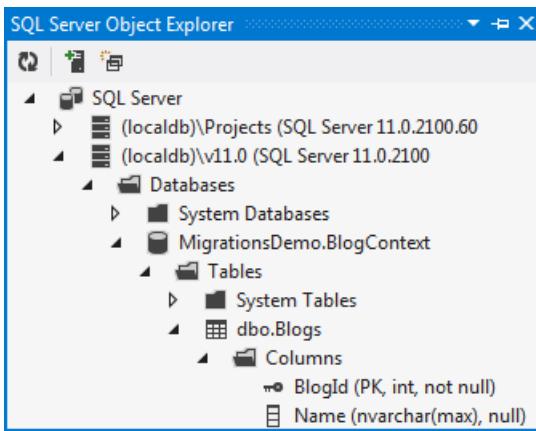
namespace MigrationsAutomaticDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog" });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

- Запустите приложение и вы увидите, что **MigrationsAutomaticCodeDemo.BlogContext** база данных создается автоматически.



Включение миграций

Пора внести дополнительные изменения в нашу модель.

- Давайте внесем в класс Blog свойство Url.

```
public string Url { get; set; }
```

Если бы вы запустили приложение снова, оно выдало бы исключение `InvalidOperationException` с сообщением: *Модель, поддерживающая контекст BlogContext, изменилась с момента создания базы данных. Попробуйте обновить базу данных с помощью Code First Migrations (<http://go.microsoft.com/fwlink/?LinkId=238269>).*

Как видно из исключения, пора приступить к использованию Code First Migrations. Поскольку мы хотим использовать автоматический перенос мы собираемся укажите — **EnableAutomaticMigrations** переключения.

- Запустите **Enable-Migrations** — **EnableAutomaticMigrations** добавленные команды в консоли диспетчера пакетов эту команду **миграций** папку для нашего проекта. В этой новой папке содержит один файл:
- **Класс конфигурации.** Этот класс позволяет настраивать поведение миграций для контекста. В этом пошаговом руководстве мы будем просто использовать конфигурацию по умолчанию. *Поскольку имеется только один контекст Code First в проекте, Enable-Migrations автоматически заполняет тип контекста, к которому относится эта конфигурация.*

Первый автоматического переход

Вам нужно знать о двух основных командах Code First Migrations.

- **Add-Migration** будет автоматически формировать следующую миграцию на основе изменений, внесенных в модель с момента создания последней миграции.
- **Update-Database** будет применять ожидающие обработки миграции к базе данных.

Мы хотим избежать с помощью Add-Migration (если не нужно в действительности) и сосредоточиться на возможности Code First Migrations автоматически вычислить и применить изменения. Мы используем **Update-Database** для получения Code First Migrations для отправьте изменения в нашей модели (новый **Blog.Url** свойство!) к базе данных.

- Запустите **Update-Database** команду в консоли диспетчера пакетов.

MigrationsAutomaticDemo.BlogContext базы данных теперь обновляется для включения **URL-адрес** столбца в **блоги** таблицы.

Второй автоматического переход

Давайте создадим другой изменения и позволить Code First Migrations автоматически отправьте изменения в базе данных для нас.

- Также добавим новый класс **Post**.

```
public class Post
{
    public int PostId { get; set; }
    [MaxLength(200)]
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

- Еще мы добавим коллекцию **Posts** в класс **Blog**, чтобы создать второй конец связи между **Blog** и **Post**.

```
public virtual List<Post> Posts { get; set; }
```

Теперь с помощью **Update-Database** перевести базу данных в актуальном состоянии. Давайте укажем флаг **-Verbose**, чтобы вы могли видеть SQL, который выполняет Code First Migrations.

- Запустите команду **Update-Database -Verbose** в консоли диспетчера пакетов.

Добавление кода на основе миграции

Теперь давайте взглянем на что-то, возможно, мы хотим использовать миграции на основе кода для.

- Давайте добавим **Оценка** свойства **блог** класса

```
public int Rating { get; set; }
```

Мы просто выполнить **Update-Database** отправлять эти изменения к базе данных. Тем не менее, мы добавляем не допускающим **Blogs.Rating** столбца, если все существующие данные в таблице он будет назначаются по умолчанию CLR типа данных для нового столбца (оценка — целое число, это будет **0**). Но нам нужно указать значение по умолчанию **3**, чтобы существующие строки в таблице **Blogs** начинались с неплохой оценки. Для записи этого изменения *out* для миграции на основе кода, таким образом, чтобы его можно изменить мы используем команду Add-Migration. **Add-Migration** команда позволяет нам назовите эти переносы, давайте просто вызовите наши **AddBlogRating**.

- Запустите **AddBlogRating Add-Migration** команду в консоли диспетчера пакетов.
- В **миграций** папки теперь у нас есть новый **AddBlogRating** миграции. Имя файла миграции предварительно исправлена с меткой времени для облегчения упорядочения. Давайте изменить созданный код, чтобы указать значение по умолчанию 3 для **Blog.Rating** (строка 10 в приведенном ниже коде)

Миграции также содержит файл кода, в котором регистрируются некоторые метаданные. Эти метаданные позволят Code First Migrations для репликации автоматический перенос, которые мы выполнили перед этой миграции на основе кода. Это важно в том случае, если другой разработчик хочет выполните наши миграции, или когда придет время, чтобы развернуть наше приложение.

```
namespace MigrationsAutomaticDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddBlogRating : DbMigration
    {
        public override void Up()
        {
            AddColumn("Blogs", "Rating", c => c.Int(nullable: false, defaultValue: 3));
        }

        public override void Down()
        {
            DropColumn("Blogs", "Rating");
        }
    }
}
```

Измененная миграция выглядит прекрасно, так что давайте обновим ее с помощью команды **Update-Database**.

- Запустите **Update-Database** команду в консоли диспетчера пакетов.

К автоматический перенос

Теперь мы бесплатно переключиться обратно на автоматический перенос для нашей простой изменения. Code First Migrations позаботится о выполнении миграции автоматическое и на основе кода в правильном порядке на основе метаданных, хранящиеся в файле кода для каждого миграции на основе кода.

- Давайте добавим свойство Post.Abstract для нашей модели

```
public string Abstract { get; set; }
```

Теперь мы можем использовать **Update-Database** для получения Code First Migrations, чтобы передать изменения в базу данных, с помощью автоматической миграции.

- Запустите **Update-Database** команду в консоли диспетчера пакетов.

Сводка

В этом пошаговом руководстве вы узнали, как использовать автоматический перенос для передачи в базу данных изменения модели. Вы также узнали, как сформировать шаблон и выполнения миграции на основе кода между автоматический перенос в том случае, когда требуется более строгий контроль.

Code First Migrations с существующей базы данных

13.09.2018 • 14 minutes to read • [Edit Online](#)

NOTE

EF4.3 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в версии 4.1 платформы Entity Framework. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

В этой статье рассматривается использование с существующей базы данных, которая не была создана с Entity Framework Code First Migrations.

NOTE

В этой статье предполагается, что вы знаете, как использовать Code First Migrations в основных сценариях. Если этого не сделать, то нужно будет прочесть [Code First Migrations](#) перед продолжением.

Демонстрационные ролики

Если вместо этого будет смотреть презентация, чем, в этой статье, следующие два видео рассматриваются тоже содержимое, что в этой статье.

Один видео: «Миграции — взгляд изнутри»

Этот демонстрационный ролик рассматривается как отслеживает миграций и использует сведения о модели для обнаружения изменений в модели.

Два видео: «Миграции — существующих баз данных»

Основываясь на концепциях из предыдущих видео, этой трансляции описывается, как включить и использовать миграции с существующей базы данных.

Шаг 1: Создание модели

Первым шагом является создание модели Code First, предназначенного для существующей базы данных. [Code First для существующей базы данных](#) разделе содержатся подробные указания о том, как это сделать.

NOTE

Очень важно выполните остальные действия, описанные в этом разделе перед внесением любых изменений в модель, могут потребовать изменений в схему базы данных. Для следующих действий требуется модель должна быть синхронизированной со схемой базы данных.

Шаг 2: Включение миграции

Следующим шагом является включение migrations. Это можно сделать, выполнив **Enable-Migrations** команду в консоли диспетчера пакетов.

Эта команда будет создайте папку с названием миграций и помещать один класс в его конфигурацию. Класс конфигурации здесь настраивается миграции для вашего приложения, можно найти дополнительные сведения см. в [Code First Migrations](#) раздела.

Шаг 3: Добавление начального переноса

После создания и применения к миграции локальной базы данных, можно также применить эти изменения к другим базам данных. Например локальную базу данных могут быть тестовую базу данных и в конечном итоге можно также применить изменения к рабочей базе данных или другим разработчикам тестирования базы данных. Существует два варианта для этого шага, а также то, что следует выбрать от ли схемы из других баз данных пуст, или в настоящее время соответствуют схеме локальной базы данных.

- **Вариант один: Использование существующей схемы как начальная точка.** Этот подход следует использовать при других баз данных, которые миграции будет применяться к будущему будет иметь ту же схему, как в настоящее время имеет локальную базу данных. Например можно использовать это если локальной тестовой базе данных в настоящее время соответствует версии 1 из рабочей базы данных и будет применен эти миграции для обновления рабочей базы данных до версии 2.
- **Второй вариант: Используйте пустую базу данных как начальная точка.** Этот подход следует использовать при других баз данных, которые миграции будет применяться к будущему являются пустыми (или еще не существуют). Например это может использовать, если к разработке приложения с помощью тестовой базы данных, но без использования миграции и вы позднее необходимо будет создавать рабочей базы данных с нуля.

Вариант один: Используйте существующую схему в качестве отправной точки

Code First Migrations использует моментальный снимок модели, сохраненной в самой последней миграции для обнаружения изменений в модель (можно найти подробные сведения об этом в [Code First Migrations в командных средах](#)). Так как мы собираемся предполагается, что базы данных уже имеют схему текущей модели, будет создан пустой миграции (нет-ор), с текущей модели как моментальный снимок.

1. Запустите **это добавить миграцию InitialCreate — IgnoreChanges** команду в консоли диспетчера пакетов. Будет создана пустая Миграция с текущей моделью, как моментальный снимок.
2. Запустите **Update-Database** команду в консоли диспетчера пакетов. Это InitialCreate миграции будут применены к базе данных. Так как фактический перенос не содержит никаких изменений, он просто добавлять строки `_MigrationsHistory` таблицы, указывающее, что уже был применен такой миграции.

Второй вариант: Используйте пустую базу данных в качестве отправной точки

В этом сценарии мы должны миграций, чтобы иметь возможность создать всю базу данных с нуля — включая таблицы, которые уже присутствуют в локальной базе данных. Мы собираемся создать это InitialCreate миграции, включающий логику для создания существующей схемы. Затем мы сделаем нашу существующую базу данных похож на уже был применен такой миграции.

1. Запустите **это добавить миграцию InitialCreate** команду в консоли диспетчера пакетов. При этом создается миграции для создания существующей схемы.
2. Закомментируйте весь код в методе Up только что созданный миграции. Это позволит нам «apply» миграции в локальную базу данных без попытки повторного создания всех таблиц и т.д., которые уже существуют.
3. Запустите **Update-Database** команду в консоли диспетчера пакетов. Это InitialCreate миграции будут применены к базе данных. Так как не содержит фактического переноса всех изменений (так как мы временно комментариями их), он будет просто добавлять строки `_MigrationsHistory` таблицы, указывающее, что уже был применен такой миграции.
4. Отменить преобразование в комментарий код в метод. Это означает, что при применении этой миграции в будущих базах данных, схемы, которая уже существует в локальной базе данных будут создаваться с помощью миграций.

Моментов, которые следует учитывать

Существует несколько вещей, которые необходимо учитывать при использовании миграции для

существующей базы данных.

Имена по умолчанию/вычисляться может не соответствовать существующей схемы

Миграции явно задает имена столбцов и таблиц, когда он формирует шаблоны миграций. Однако существуют другие объекты базы данных, миграция вычисляет имя по умолчанию при применении для миграции. Сюда входят индексы и ограничения внешнего ключа. При нацеливании на существующую схему, эти имена вычисляемых может не совпадать, что действительно существует в базе данных.

Ниже приведены некоторые примеры, при необходимости учитывать это.

Если вы использовали "один параметр: использовать существующую схему в качестве отправной точки" на шаге 3:

- Если будущие изменения в модели требуется изменение или удаление одного из объектов базы данных, называется по-разному, необходимо будет изменить сформированный миграции, чтобы указать правильное имя. Миграция API-интерфейсы имеют необязательный параметр имени, который позволяет это сделать. Например, существующей схемы могут иметь таблицу Post с BlogId внешний ключевой столбец, имеющий индекс с именем IndexFk_BlogId. Однако по умолчанию миграции будет ожидать, что этот индекс, чтобы называться IX_BlogId. При внесении изменений в модель, которая происходит удаление этого индекса, необходимо будет изменить сформированный DropIndex вызов для указания IndexFk_BlogId имени.

Если вы использовали "два параметра: пустую базу данных в качестве отправной точки используйте" на шаге 3:

- При попытке запустить метод вниз первоначальной миграции (, восстановление пустую базу данных) в локальной базе данных может завершиться ошибкой, так как миграция будет пытаться удалить индексы и ограничения внешнего ключа, с использованием неверные имена. Это только повлияет на локальную базу данных так, как другие базы данных будет создан с нуля, используя метод вверх первоначальной миграции. Если понизить существующей локальной базы данных пустым, проще всего сделать это вручную, либо путем удаления базы данных или удаления всех таблиц. После этого начального перехода на предыдущий, все объекты базы данных будут созданы заново с именами по умолчанию поэтому эта проблема не представляют собой еще раз.
- Если будущие изменения в модели требуется изменение или удаление одного из объектов базы данных, называется по-разному, это не будет работать для существующей локальной базы данных — так как имена не совпадают значения по умолчанию. Тем не менее он будет работать для баз данных, которые были созданы «с нуля», так как они будут использоваться имена по умолчанию, выбранный с помощью миграций. Можно либо вручную внести эти изменения на локальную базу данных существующего или рассмотрите возможность установки миграций повторного создания базы данных с нуля — как будет происходить на других компьютерах.
- Баз данных, созданных с помощью метода вверх первоначальной миграции может немного отличаться от локальной базы данных, так как имена вычисленное значение по умолчанию для индексов и ограничений внешнего ключа будет использоваться. Может также оказаться с дополнительные индексы как миграции создаст индексы во внешних ключевых столбцах по умолчанию — это не так в локальной исходной базы данных.

Не все объекты базы данных представлены в модели

Объекты базы данных, которые не являются частью модели не будет обрабатываться с помощью миграций. Они могут включать представления, хранимые процедуры, разрешения, таблицы, которые не являются частью вашей модели, дополнительные индексы, и т.д.

Ниже приведены некоторые примеры, при необходимости учитывать это.

- Независимо от параметра выбранное на шаге 3, если требуется будущие изменения в модели, изменение или удаление эти дополнительные объекты, миграция не будут знать для внесения этих изменений.

Например если удалить столбец, содержащий указатель на нем, миграция не будет удалить индекс. Необходимо вручную добавить код для формирования шаблонов миграции.

- Если вы использовали "два параметра: используйте пустую базу данных в качестве отправной точки", эти дополнительные объекты не будут создаваться по метод первоначальной миграции. Вы можете изменить вверх и вниз методы для обеспечения проверки этих дополнительных объектов, если вы хотите. Для объектов, которые изначально не поддерживаются в API миграции — например представления — можно использовать [Sql](#) необработанных SQL на создание или удаление их выполнение метода.

Настройка таблицы журнала миграции

27.09.2018 • 5 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

NOTE

В этой статье предполагается, что вы знаете, как использовать Code First Migrations в основных сценариях. Если этого не сделать, то нужно будет прочесть [Code First Migrations](#) перед продолжением.

Что такое таблица журнала миграции?

Таблица журнала миграции — это таблица, используемая Code First Migrations для хранения сведений о миграции, применить к базе данных. По умолчанию является имя таблицы в базе данных `_MigrationHistory` и он создается при применении первой до миграции базы данных. В Entity Framework 5 эта таблица была системную таблицу, если в приложении используется база данных Microsoft Sql Server. Это изменилось в Entity Framework 6, но и таблице журнала миграции больше не помечена системную таблицу.

Зачем настраивать таблицы журнала миграции?

Таблица журнала миграции должен использоваться исключительно с Code First Migrations и менять ее вручную можно нарушить миграций. Тем не менее иногда конфигурация по умолчанию не подходит, и таблицы необходимо выполнить настройку, например:

- Необходимо изменить имена и/или аспекты столбцов, чтобы включить `3` к удаленному рабочему столу миграций стороннего поставщика
- Чтобы изменить имя таблицы
- Необходимо использовать схему не по умолчанию для `_MigrationHistory` таблицы
- Вам необходимо сохранить дополнительные данные для данной версии контекста, и поэтому необходимо добавить дополнительный столбец в таблицу

Меры предосторожности слов

Изменение таблицы журнала миграции обладает широкими возможностями, но вам нужно соблюдать осторожность, чтобы не злоупотреблять. В настоящее время среда выполнения EF не проверяет, совместим ли таблицы журнала настроенный миграции в среде выполнения. Если приложение не может прервать во время выполнения или ведут себя непредсказуемым образом. Это еще более важно, если вы используете несколько контекстов на одну базу данных в этом случае несколько контекстов можно использовать одну прежнюю таблицу миграции для хранения сведений о миграции.

Как настроить таблицу журнала миграции?

Перед началом необходимо знать, в таблицу журнала миграции можно настраивать только в том случае, прежде чем применить первую миграцию. Теперь чтобы код.

Во-первых необходимо создать класс, производный от класса `System.Data.Entity.Migrations.History.HistoryContext`. Класс `HistoryContext` является производным от класса `DbContext`, поэтому Настройка таблицы журнала миграции не очень похожа на настройку модели EF с помощью текущего API. Нужно переопределить метод `OnModelCreating` и используйте fluent API для настройки в таблице.

NOTE

Обычно при настройке модели EF не нужно вызывать базовый `OnModelCreating()` из переопределенного метода `OnModelCreating`, так как `DbContext.OnModelCreating()` имеет пустой текст. Это условие не выполняется, при настройке миграции таблицы журнала. В данном случае первое, что необходимо сделать в переопределении `OnModelCreating()` является фактически вызов базового `OnModelCreating()`. Это настроит таблице журнала миграции в метод по умолчанию, что вы примените в переопределяющем методе.

Предположим, что вы хотите переименовать таблицу журнала миграции и поместите его на пользовательских схема с именем «`admin`». В дополнение к этому Администратору просят переименуйте столбец `MigrationId` миграции_идентификатор. Это достигается путем создания следующий класс, производный от `HistoryContext`:

```
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Migrations.History;

namespace CustomizableMigrationsHistoryTableSample
{
    public class MyHistoryContext : HistoryContext
    {
        public MyHistoryContext(DbConnection dbConnection, string defaultSchema)
            : base(dbConnection, defaultSchema)
        {
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<HistoryRow>().ToTable(tableName: "MigrationHistory", schemaName: "admin");
            modelBuilder.Entity<HistoryRow>().Property(p => p.MigrationId).HasColumnName("Migration_ID");
        }
    }
}
```

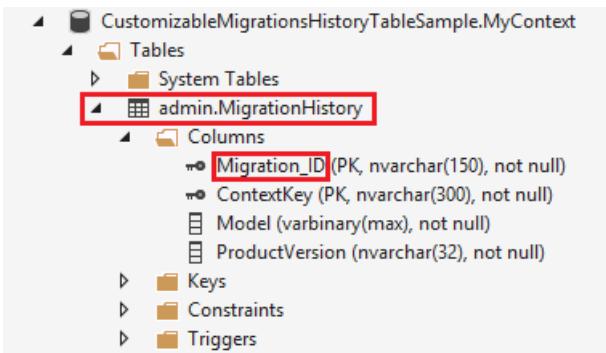
После подготовки вашего пользовательского `HistoryContext` необходимо сообщить EF его, зарегистрировав его с помощью [конфигурация на основе кода](#):

```
using System.Data.Entity;

namespace CustomizableMigrationsHistoryTableSample
{
    public class ModelConfiguration : DbConfiguration
    {
        public ModelConfiguration()
        {
            this.SetHistoryContext("System.Data.SqlClient",
                (connection, defaultSchema) => new MyHistoryContext(connection, defaultSchema));
        }
    }
}
```

Это равносильно использованию его. Теперь вы можете перейти в консоль диспетчера пакетов, Enable-

Migrations, Add-Migration и, наконец, обновления базы данных. Это должно привести к появлению добавления в базу данных в таблицу журнала миграции настраиваются в соответствии с данные, которые вы указали в HistoryContext производного класса.



С помощью migrate.exe

01.10.2018 • 8 minutes to read • [Edit Online](#)

Code First Migrations может использоваться для обновления базы данных в visual studio, но также можно выполнить с помощью migrate.exe средство командной строки. Эта страница будет дать краткий обзор по использованию migrate.exe для выполнения операций базы данных миграции.

NOTE

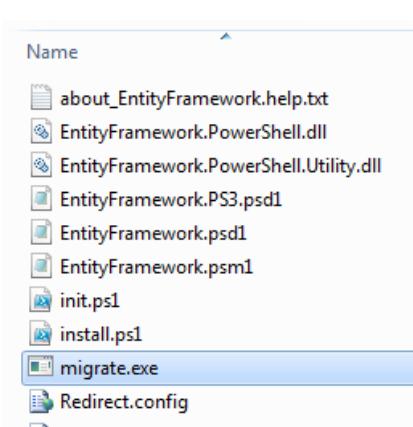
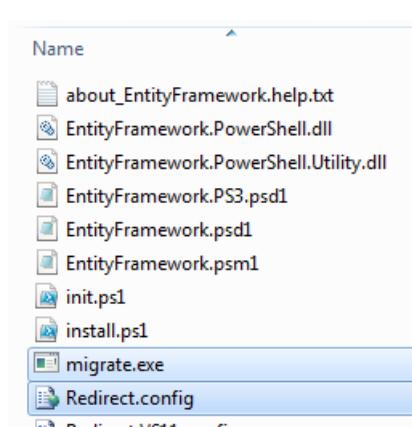
В этой статье предполагается, что вы знаете, как использовать Code First Migrations в основных сценариях. Если этого не сделать, то нужно будет прочесть [Code First Migrations](#) перед продолжением.

Скопируйте migrate.exe

При установке платформы Entity Framework с помощью NuGet migrate.exe будет в папке tools загруженного пакета. В <папку проекта>\пакетов\EntityFramework.< версия>\средства

При наличии migrate.exe необходимо скопировать его в расположение сборки, содержащей миграции.

Если приложение предназначено для .NET 4, а не 4.5, затем необходимо будет скопировать **Redirect.config** в место, а также и переименуйте его **migrate.exe.config**. Это так, что migrate.exe получит правильные перенаправления привязок чтобы иметь возможность найти сборку платформы Entity Framework.

.NET 4.5	.NET 4.0
	

NOTE

migrate.exe не поддерживает x64 сборок.

После перемещения migrate.exe к папке должен быть использует ее для выполнения миграции в базе данных. Программа призвана сделать всего лишь выполнении миграций. Не может создать миграций или создать скрипт SQL.

См. в разделе параметров

```
Migrate.exe /?
```

Оно будет отображаться на страницу справки, связанный с этой программой, обратите внимание, что необходимо будет иметь EntityFramework.dll в том же расположении, migrate.exe выполняются в порядке, чтобы это работало.

Миграция на последнюю миграции

```
Migrate.exe MyMvcApplication.dll /startupConfigurationFile="..\web.config"
```

При под управлением migrate.exe только обязательный параметр — это сборки, которая находится на сборку, содержащую миграций, которые вы пытаетесь запустить, но он использует все соглашения на основе параметров, если не указать файл конфигурации.

Перенос в конкретных миграции

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /targetMigration="AddTitle"
```

Если вы хотите запустить миграции до конкретных миграции, можно указать имя миграции. Будет запущен всех предыдущих операций перемещения при необходимости до начала миграции указано.

Укажите рабочий каталог

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /startupDirectory="c:\\MyApp"
```

Если вы сборка имеет зависимости или считывает файлы относительно рабочего каталога вам потребуется задать startupDirectory.

Укажите конфигурацию для переноса для использования

```
Migrate.exe MyAssembly CustomConfig /startupConfigurationFile="..\web.config"
```

Если у вас есть несколько классов конфигурации миграции, классы, наследующие от DbMigrationConfiguration, затем необходимо указать, который будет использоваться при данном выполнении. Это указано, предоставляя необязательный второй параметр без параметра как выше.

Укажите строку подключения

```
Migrate.exe BlogDemo.dll /connectionString="Data Source=localhost;Initial Catalog=BlogDemo;Integrated Security=SSPI" /connectionProviderName="System.Data.SqlClient"
```

Если вы хотите указать строку подключения в командной строке, то необходимо также указать имя поставщика. Имя поставщика не указано, вызовет исключение.

Распространенные проблемы

СООБЩЕНИЕ О БИЗОНКЕ	РЕШЕНИЕ
Необработанное исключение: System.IO.FileLoadException: не удалось загрузить файл или сборку "EntityFramework, Version = 5.0.0.0, язык и региональные параметры = neutral, PublicKeyToken = b77a5c561934e089" или одну из ее зависимостей. Определение манифеста расположены сборки не соответствует ссылки на сборку. (Исключение из HRESULT: 0x80131040)	Как правило, это означает, что при запуске приложения .NET 4 без файла Redirect.config. Необходимо скопировать в то же расположение, что migrate.exe Redirect.config и переименуйте его в migrate.exe.config.
Необработанное исключение: System.IO.FileLoadException: не удалось загрузить файл или сборку "EntityFramework, Version = 4.4.0.0, язык и региональные параметры = neutral, PublicKeyToken = b77a5c561934e089" или одну из ее зависимостей. Определение манифеста расположены сборки не соответствует ссылки на сборку. (Исключение из HRESULT: 0x80131040)	Это исключение означает, что вы используете .NET 4.5, приложение с Redirect.config скопированное в расположение migrate.exe. Если приложение является .NET 4.5 файл конфигурации с помощью перенаправления внутри не нужно. Удалите файл migrate.exe.config.
Ошибка: Не удалось обновить базу данных для согласования текущей модели, так как имеются ожидающие изменения и отключить автоматический перенос. Включение автоматического переноса или запись изменений ожидающие модели миграции на основе кода. DbMigrationsConfiguration.AutomaticMigrationsEnabled, чтобы задайте значение true, чтобы включить автоматическую миграцию.	Эта ошибка возникает, если выполняется миграция, при миграции, чтобы справиться с изменениями, внесенными в модель еще не создана, а базы данных совпадает с моделью. Добавление свойства в класс модели, запустив migrate.exe без создания миграции для обновления базы данных является примером.
Ошибка: Тип не разрешается для члена "System.Data.Entity.Migrations.Design.ToolingFacade+UpdateRunner, EntityFramework, Version = 5.0.0.0, язык и региональные параметры = neutral, PublicKeyToken = b77a5c561934e089".	Эта ошибка может возникать, указав каталог при запуске. Это должно быть расположение migrate.exe
Необработанное исключение: System.NullReferenceException: ссылка на объект не указывает на экземпляр объекта. в System.Data.Entity.Migrations.Console.Program.Main (String [] args)	Может быть причиной, не указывая обязательный параметр для сценария, который используется. Например, указывающий строку подключения без указания имени поставщика.
Ошибка: более одного типа миграции конфигурации найден в сборке «ClassLibrary1». Укажите имя любой из них.	Как говорится в ошибку, имеется более одного класса конфигурации в заданной сборке. Чтобы указать, какие использовать, необходимо использовать параметр /configurationType.
Ошибка: Не удалось загрузить файл или сборку "<assemblyName>" или одну из ее зависимостей. Данную сборку имя или базы кода был недопустимым. (Исключение из HRESULT: 0x80131047)	Это может быть вызвано неправильно Указание имени сборки или отсутствия
Ошибка: Не удалось загрузить файл или сборку "<assemblyName>" или одну из ее зависимостей. Была сделана попытка загрузить программу, имеющую неверный формат.	Это происходит, если вы пытаетесь запустить migrate.exe от x64 приложения. EF 5.0 и более ранних версий будет работать только на x86.

Code First Migrations в командных средах

13.09.2018 • 26 minutes to read • [Edit Online](#)

NOTE

В этой статье предполагается, что вы знаете, как использовать Code First Migrations в основных сценариях. Если этого не сделать, то нужно будет прочесть [Code First Migrations](#) перед продолжением.

Перерыв на кофе, необходимо прочитать эту статью

Данные о проблемах в командных средах являются главным образом вокруг слияния миграций, когда два разработчика создали миграций, в свою базу кода на локальном. Хотя необходимые действия для решения этих довольно просты, они требуют понимания принципа работы миграций. Пожалуйста не просто сразу перейти к концу — уделить время читать эту статью, чтобы убедиться, что вы все.

Некоторые общие рекомендации

Прежде чем мы углубимся в управлении слияния миграции, созданных несколькими разработчиками, ниже приведены некоторые общие рекомендации, чтобы установить для успешного.

Каждый член команды должен иметь локальной разработки базы данных

Использует миграций `_MigrationsHistory` таблицу для хранения, какие виды миграции были применены к базе данных. При наличии нескольких разработчиков, создание миграцию по отдельности при попытке предназначенных для той же базе данных (и тем самым совместно использовать `_MigrationsHistory` таблицы) миграции будет крайне запутанным.

Само собой при наличии участников команды, которые не создаются миграций, не возникало необходимости их совместное использование базы данных центра разработки.

Избегайте автоматический перенос

Суть в том, что автоматический перенос изначально красиво в командных средах, но на самом деле они просто не работают. Если вы хотите узнать почему, читайте дальше, — в противном случае перейдите к следующему разделу.

Автоматический перенос позволяет иметь схему базы данных, обновляется в соответствии с текущей моделью без необходимости создания файлов кода (миграция на базе кода). Автоматический перенос будет очень хорошо работать в среде группы, если вы только использовали их и никогда не создаются все миграции на основе кода. Проблема заключается в том, что автоматический перенос ограничены и не обрабатывают несколько операций – свойство столбец переименование, перемещение данных в другую таблицу, и т.д. Для обработки таких ситуаций, вы в итоге создание миграции на основе кода (и редактирования шаблонный код), которые используются переменные типов между изменениями, которые обрабатываются автоматический перенос. Это позволяет практически на невозможно выполнить слияние изменений, когда два разработчика проверять при миграции.

Демонстрационные ролики

Если вместо этого будет смотреть презентация, чем, в этой статье, следующие два видео рассматриваются то же содержимое, что в этой статье.

Один видео: «Миграции — взгляд изнутри»

Этот демонстрационный ролик рассматривается как отслеживает миграций и использует сведения о модели для обнаружения изменений в модели.

Два видео: «Миграции — командные среды»

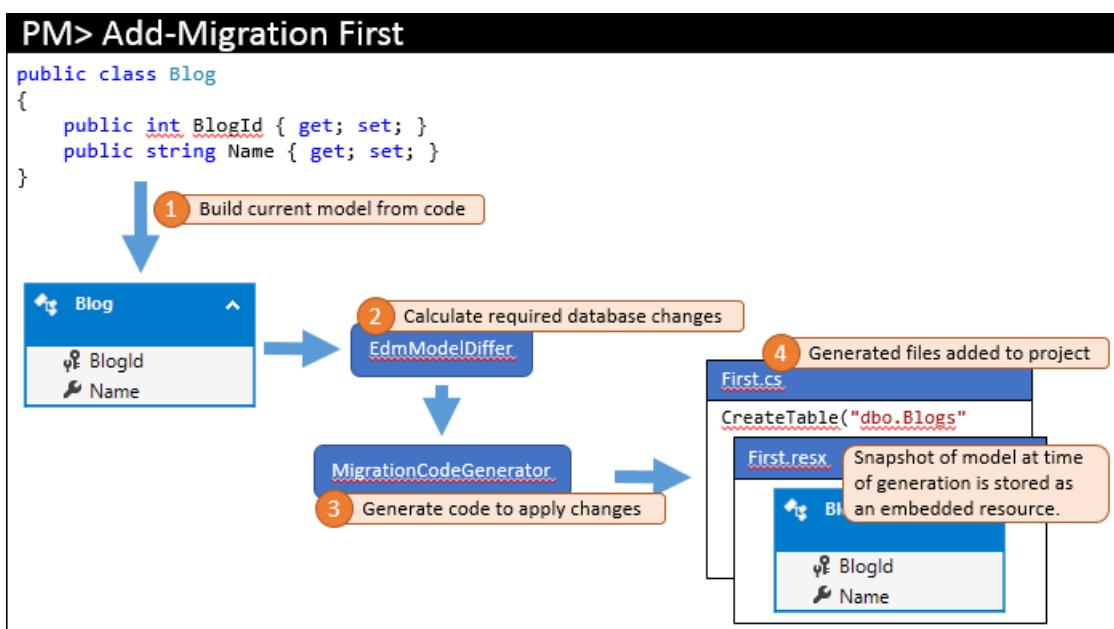
Основываясь на концепциях из предыдущих видео, [этой трансляции](#) рассматриваются проблемы, возникающие в среде team и способы их устранения.

Основные сведения о работе миграции

Ключ для успешного использования миграции в среде группы представляет собой простую, основные сведения о том, как миграция отслеживает и использует сведения о модели для обнаружения изменений в модели.

Первой миграции

При добавлении первой миграции в проект, выполните примерно **Add-Migration первый** в консоли диспетчера пакетов. Описаны основные шаги, которые выполняет эту команду показанные на рисунке ниже.



Текущая модель вычисляется на основе кода (1). Объекты базы данных, необходимых затем вычисляются в ходе различия модели (2) — так как это первой миграции модели различаются только использует пустой модели для сравнения. Необходимые изменения передаются в генератор кода для сборки кода требуется миграции (3), который затем добавляется в решение Visual Studio (4).

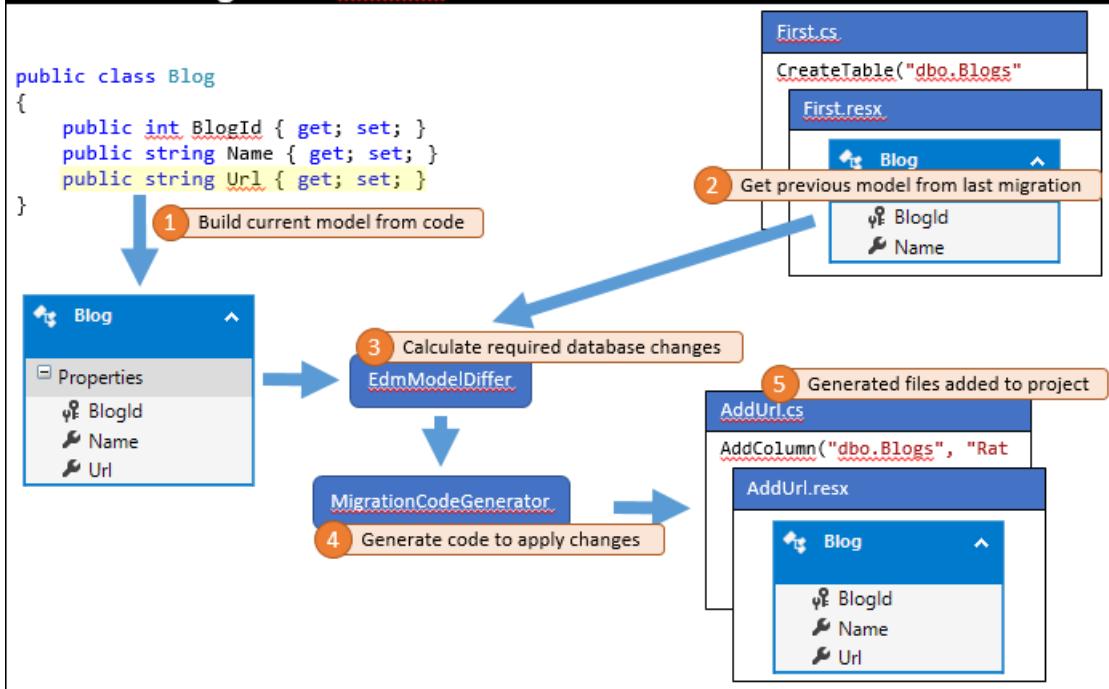
Помимо фактического переноса кода, который хранится в основной файл кода миграций также создает некоторые дополнительные файлы кода. Эти файлы метаданных, который используется с помощью миграций и не то, что следует изменять. Один из этих файлов является файлом ресурсов (.resx), с моментальным снимком модели во время миграции был создан. Вы увидите, как он используется на следующем шаге.

На этом этапе, скорее всего, нужно выполнить **Update-Database**. Чтобы применить изменения к базе данных, а затем перейдите о реализации других частях приложения.

Последующей миграции

Позже вы сможете вернуться и внести некоторые изменения в модель, — в нашем примере мы добавим **URL-адрес** свойства **блог**. Можно затем выдать команду например **AddUrl Add-Migration** сформировать шаблон миграции для применения соответствующей базы данных изменений. Описаны основные шаги, которые выполняет эту команду показанные на рисунке ниже.

PM > Add-Migration AddUrl



Так же, как в последний раз текущей модели вычисляется на основе кода (1). Тем не менее это время существует имеющихся миграций, Предыдущая модель извлекается из последней миграции (2). Эти две модели представляют собой diffed базы данных, необходимых изменений (3), а затем процесс завершает работу, как и раньше.

Этот же процесс используется для всех дополнительных миграций, добавляемые в проект.

А зачем возиться с моментального снимка модели?

Может возникнуть вопрос, почему остановились EF с моментальным снимком модели — почему не только взгляд на базе данных. Если это так, читайте дальше. Если вы не хотите можно пропустить этот раздел.

Существует ряд причин, по которым EF сохраняет моментальный снимок модели решения.

- Это позволяет базе данных для отклонения от модели EF. Эти изменения можно внести непосредственно в базе данных, или шаблонный код можно изменить в миграции для внесения изменений. Ниже приведены несколько примеров этого на практике.
 - Вы хотите добавить столбец, чтобы один или несколько таблиц `Inserted` и `Updated`, но вы не хотите включать эти столбцы в модели EF. Если оно постоянно попытается удалить эти столбцы, каждый раз, сформированного миграции базы данных миграции. С помощью моментального снимка модели, EF только когда-либо обнаружит законные возможности внесения изменений в модель.
 - Вы хотите изменить текст хранимую процедуру, используемую для обновления для включения некоторых ведения журнала. Если миграций рассмотрели этой хранимой процедуры из базы данных она будет постоянно попробуйте и восстановить его определение, которое ожидает, что EF. С помощью моментального снимка модели, EF будет только сформировать код для изменения хранимой процедуры, при изменении вида процедуру, описанную в модель EF.
 - Эти же принципы применимы к добавлению дополнительных индексов, включая дополнительные таблицы в базе данных, сопоставление EF представление базы данных, которая размещается на таблицы и т. д.
- Модель EF содержит не только фигуры базы данных. Наличие всей модели позволяет миграций просмотреть сведения о свойствах и классы в модели и их сопоставлении столбцов и таблиц. Эта информация позволяет миграции на более разумно, в коде, который она формирует шаблоны. Например при изменении имени столбца, который сопоставляет свойство миграции может обнаруживать переименования, зная, что это то же свойство — то, что невозможно при наличии только схему базы данных.

Что вызывает проблемы в средах рабочих групп

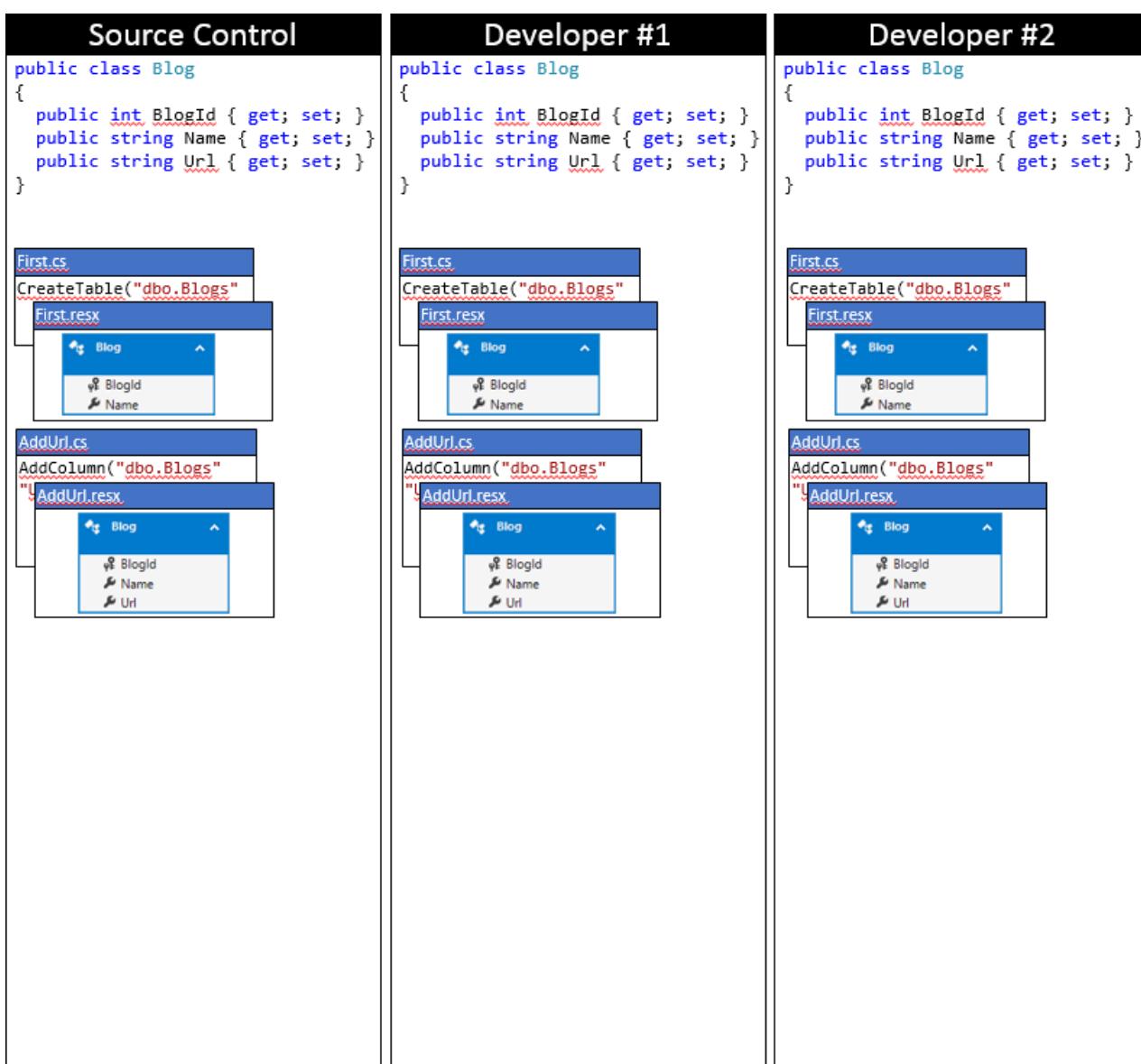
Рабочий процесс рассматривается в предыдущем разделе отлично работает, когда вы работаете в одной в приложении. Он также хорошо работает в среде группы, если вы являетесь единственным лицом, внесение изменений в модели. В этом сценарии можно внесение изменений модели, создания миграции и их отправки в системе управления версиями. Другие разработчики могут синхронизировать изменения и запуска **Update-Database** для применения изменений схемы.

Проблемы начинают возникать при наличии нескольких разработчиков, внесение изменений в модель EF и отправки в систему управления версиями, в то же время. EF отсутствует способ первого класса для слияния локальной миграции с миграциями, другой разработчик отправил в систему управления версиями, с момента последней синхронизации.

Пример конфликта слияния

Первый Рассмотрим конкретный пример такой конфликт слияния. Мы продолжим на примере, рассмотренном ранее. Давайте качестве отправной точки предполагается изменения в предыдущем разделе были возвращены исходном разработчиком. Мы будем отслеживать два разработчика, они внесения изменений в код базового.

Мы будем отслеживать модель EF и миграции через ряд изменений. Для начала разработчиков синхронизированы репозиторием системы управления версиями, как показано на следующем рисунке.

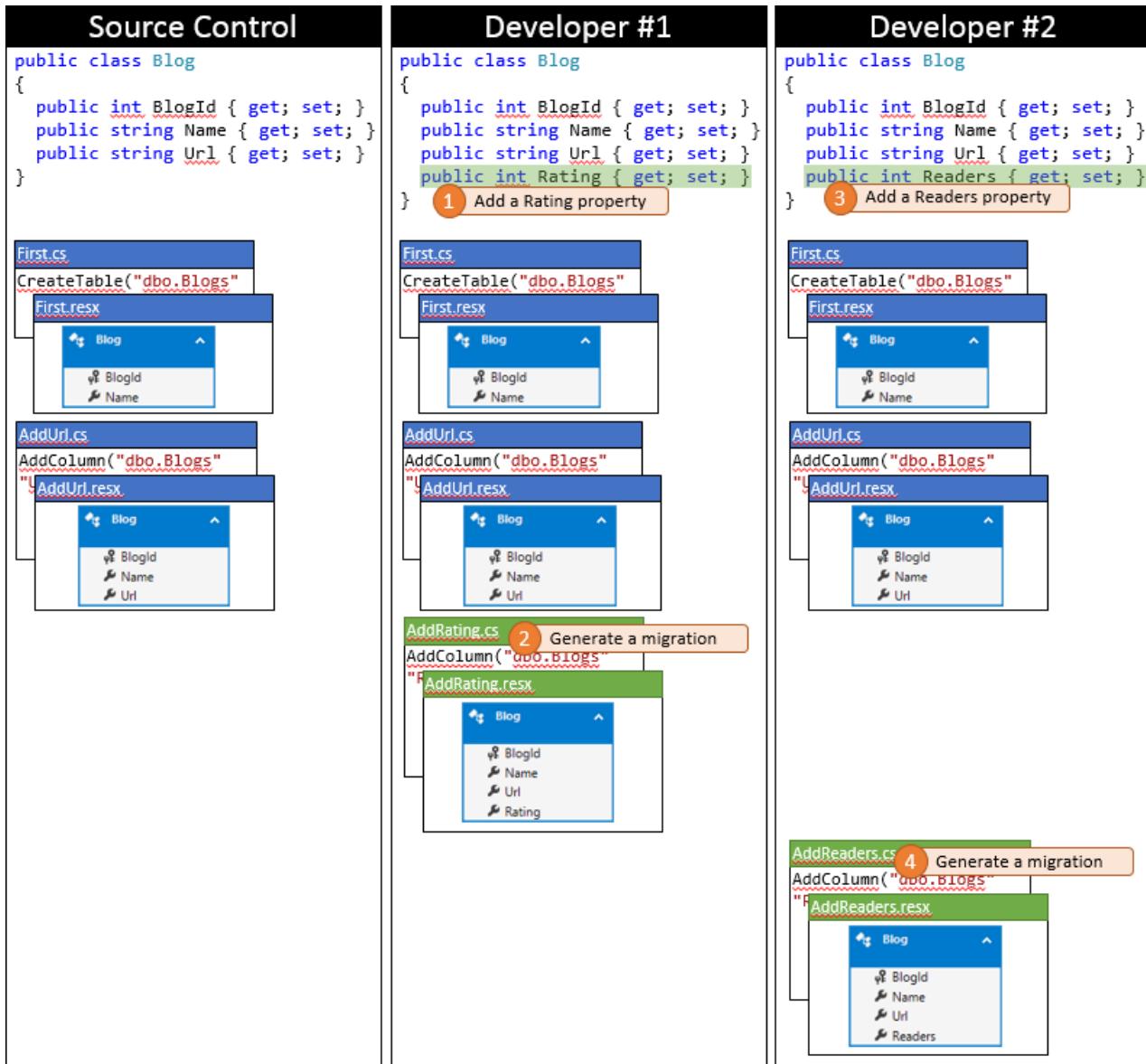


Разработчик #1 и developer #2 теперь вносит изменения в модель EF в своем локальном коде базовый.

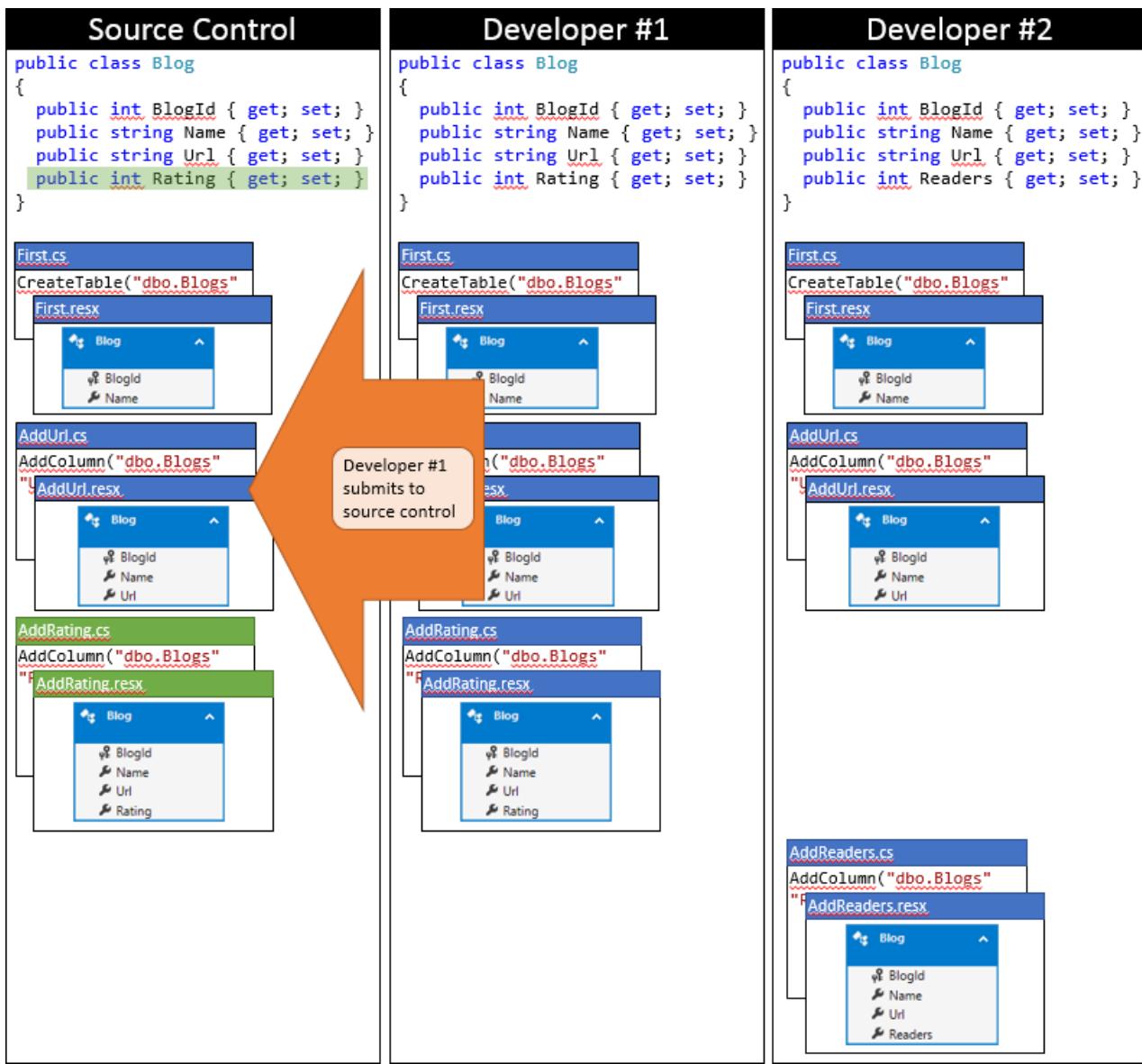
Разработчик #1 добавляет 1 **Оценка** свойства **блог** — и создает **AddRating** миграции, чтобы применить изменения к базе данных. Разработчик #2 добавляет **читатели** свойства **блог** — и создает соответствующий **AddReaders** миграции. Запустите разработчиков **Update-Database**, чтобы применить изменения к их локальным базам данных, а затем продолжить разработку приложения.

NOTE

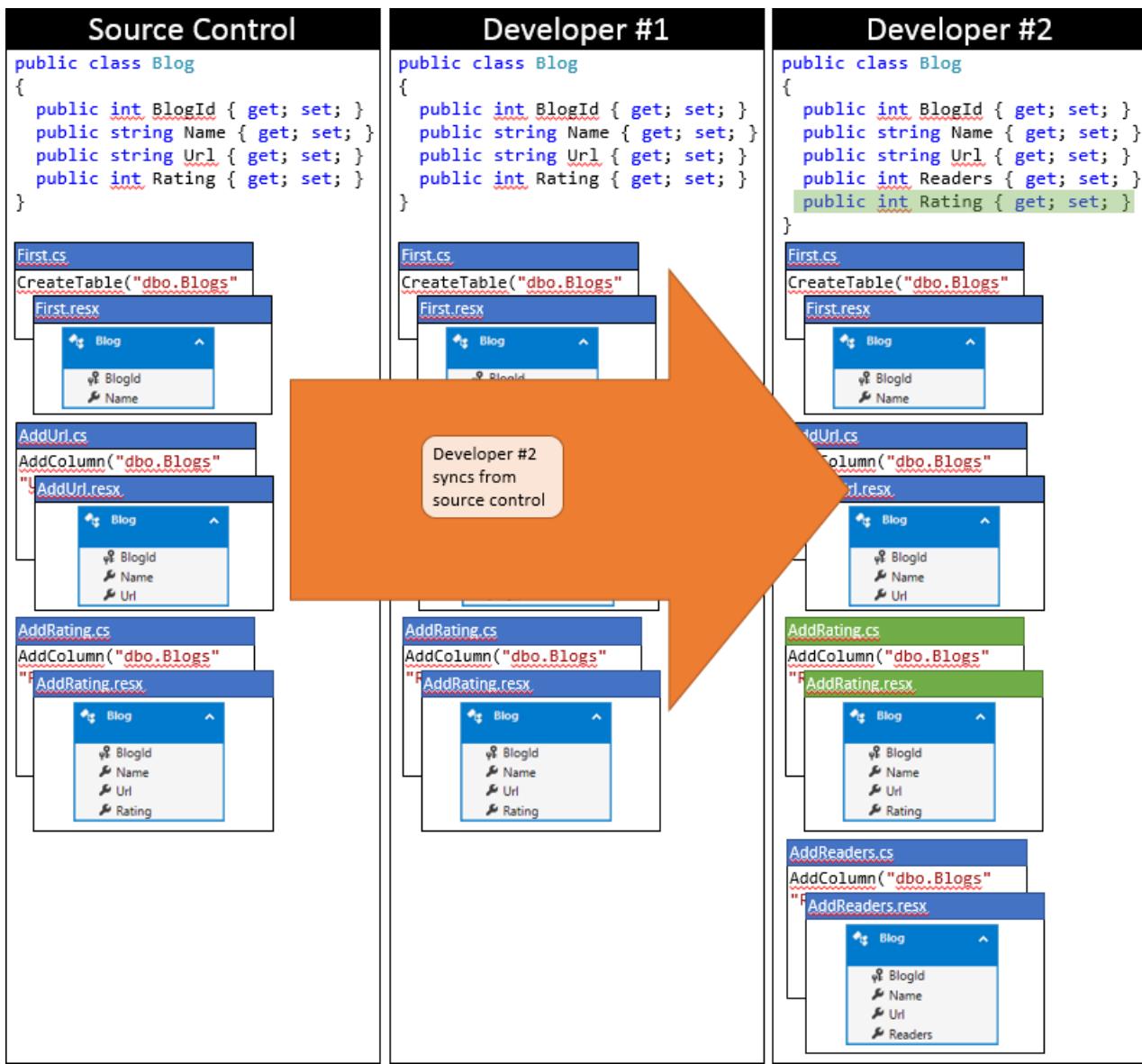
Миграции начинаются с префикса метку времени, поэтому графика представляет AddReaders миграции от разработчика #2 переходит после миграции AddRating от разработчика #1. Ли разработчик #1 или #2 созданный делает первой миграции не влияет на проблемы в работе в команде или процесс объединения, которые мы рассмотрим в следующем разделе.



Это Счастливое день для разработчиков #1 они происходят сначала отправить их изменения. Поскольку никто не проверен, поскольку они синхронизированы их репозитории, их просто предоставлять свои изменения без выполнения любой слияния.



Теперь настало время для разработчиков #2 для отправки. Они не повезло. Так как кто-то другой отправил изменения, поскольку они синхронизированы, им необходимо будет извлечь изменения и слияния. Система управления версиями будет автоматически слияния изменений на уровне кода, поскольку они очень просты. Состояние разработчика #2 в локальном репозитории после синхронизации показан на следующем рисунке.



На этот этап разработчика #2 можно запустить **обновления базы данных** обнаружит, что новый **AddRating** миграции (который не был применен для разработчика #2 базы данных) и применить его. Теперь **Оценка** столбец будет добавлен **блоги** таблицы и базы данных в соответствии с моделью.

Однако существует несколько проблем:

1. Несмотря на то что **Update-Database** применит **AddRating** миграции также приводит к формированию предупреждения: *не удалось обновить базу данных для соответствия текущей модели, так как имеются ожидающие изменения и отключается автоматическая миграция...* Проблема в том, что моментальный снимок модели хранятся в последней миграции (**AddReader**) отсутствует **Оценка** свойство **блог** (так как он не был частью модели при миграции был создан). Код сначала обнаруживает, что модель в последней миграции не совпадает с текущей моделью и выдает предупреждение.
2. Запуск приложения приведет к появлению исключения `InvalidOperationException` о том, что "модель, поддерживающая контекст «`BloggingContext`» изменилось с момента создания базы данных. Рассмотрите возможность использования `Code First Migrations` для обновления базы данных..." Опять же проблема заключается в моментальный снимок модели, хранящихся в последней миграции не соответствует текущей модели.
3. Наконец, ожидается, что выполнение **Add-Migration** теперь создаст пустой миграции (так как нет изменений для применения к базе данных). Но так как миграция сравнивает текущую модель к одному из последней миграции (которой отсутствует **Оценка** свойство) он фактически будет формировать другой **AddColumn** вызов для добавления в **Оценка** столбца. Конечно, этот вид миграции произойдет во время **Update-Database** поскольку **Оценка** столбец уже существует.

Разрешение конфликта слияния

Хорошой новостью является то, что это не слишком трудно справиться со слиянием вручную — при этом требуется понимание того, как работает миграция. Таким образом, если пропущены вперед к этому разделу... к сожалению вам нужно вернуться назад и сначала прочитать в оставшейся части статьи!

Существует два варианта, лучше всего создайте пустой перенос с правильного текущей модели как моментальный снимок. Второй вариант — для обновления моментального снимка в последней миграции на правильный моментальный снимок модели. Второй вариант немного сложнее и не может использоваться в каждом сценарии, но это также чище, поскольку он не включает добавление дополнительных миграций.

Вариант 1: Добавление миграции пустой «слияние»

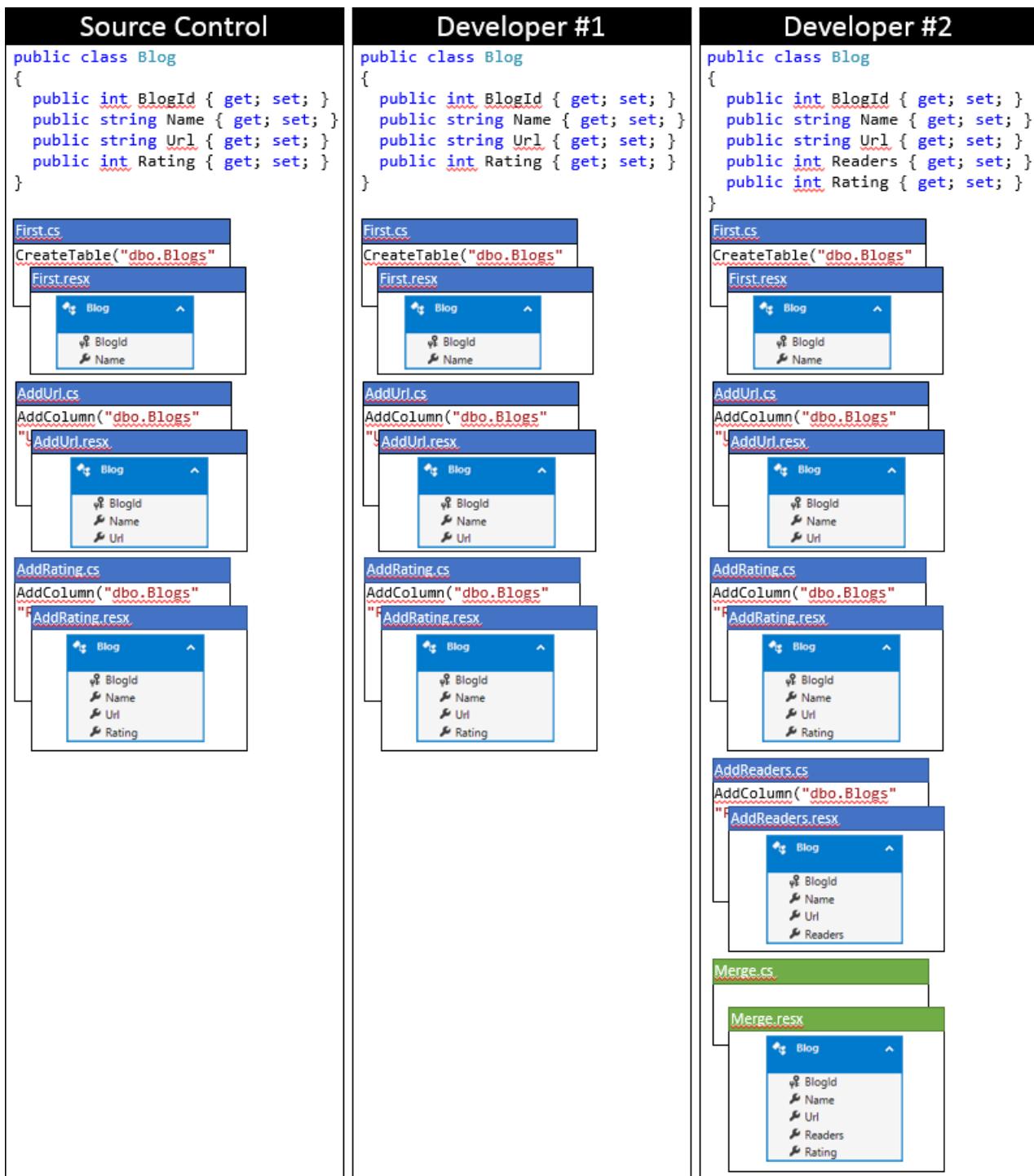
В этом параметре, мы создаем пустой миграцию только с целью убедиться, что для последней переноса правильную модель моментального снимка, сохраненного в них.

Этот параметр можно использовать независимо от создавшего последней миграции. В примере, мы выполняли разработчика #2 принципиально слияния, и они возникли для создания последней миграции. Но эти же действия можно использовать, если разработчик #1 создан последней миграции. Эти действия также применимы, если задействованы несколько миграций — мы только что рассмотрели два для простоты.

Для этого подхода, начиная с момента вы поймете, что у вас есть изменения, которые должны быть синхронизированы из системы управления версиями можно использовать следующий процесс.

1. Убедитесь, что любые изменения ожидающие модели в базе данных локального кода были записаны для миграции. Этот шаг гарантирует, что не пропустите любой законные возможности внесения изменений, когда придет время для создания пустого миграции.
2. Синхронизация с системой управления версиями.
3. Запустите **Update-Database** применить все новые миграции, которые возвращены в других разработчиков. *** Примечание: *** Если вы не получаете предупреждения на наличие с помощью команды `Update-Database`, а затем было ни одной новой миграции от других разработчиков, и нет необходимости для выполнения дальнейшей слияние.*
4. Запустите **Add-Migration <выбрать_имя> — IgnoreChanges** (например, **Add-Migration слияния — IgnoreChanges**). Это приводит к возникновению ошибки миграции с все метаданные (включая моментальный снимок текущей модели), но игнорирует любые изменения, он определяет при сравнении текущей модели к снимку в последней миграции (то есть вы получаете пустой **вверх** и **вниз** метод).
5. Продолжить разработку, или отправьте в систему управления версиями (после запуска модульных тестов конечно).

Здесь находится в состоянии разработчика #2 локальный кода после использования этого подхода.



Вариант 2: Обновление моментального снимка модели в последней миграции

Этот параметр очень похоже на вариант 1, но удаляет лишние пустые миграции — поскольку посмотрим правде в глаза, желающий файлы дополнительного кода в свое решение.

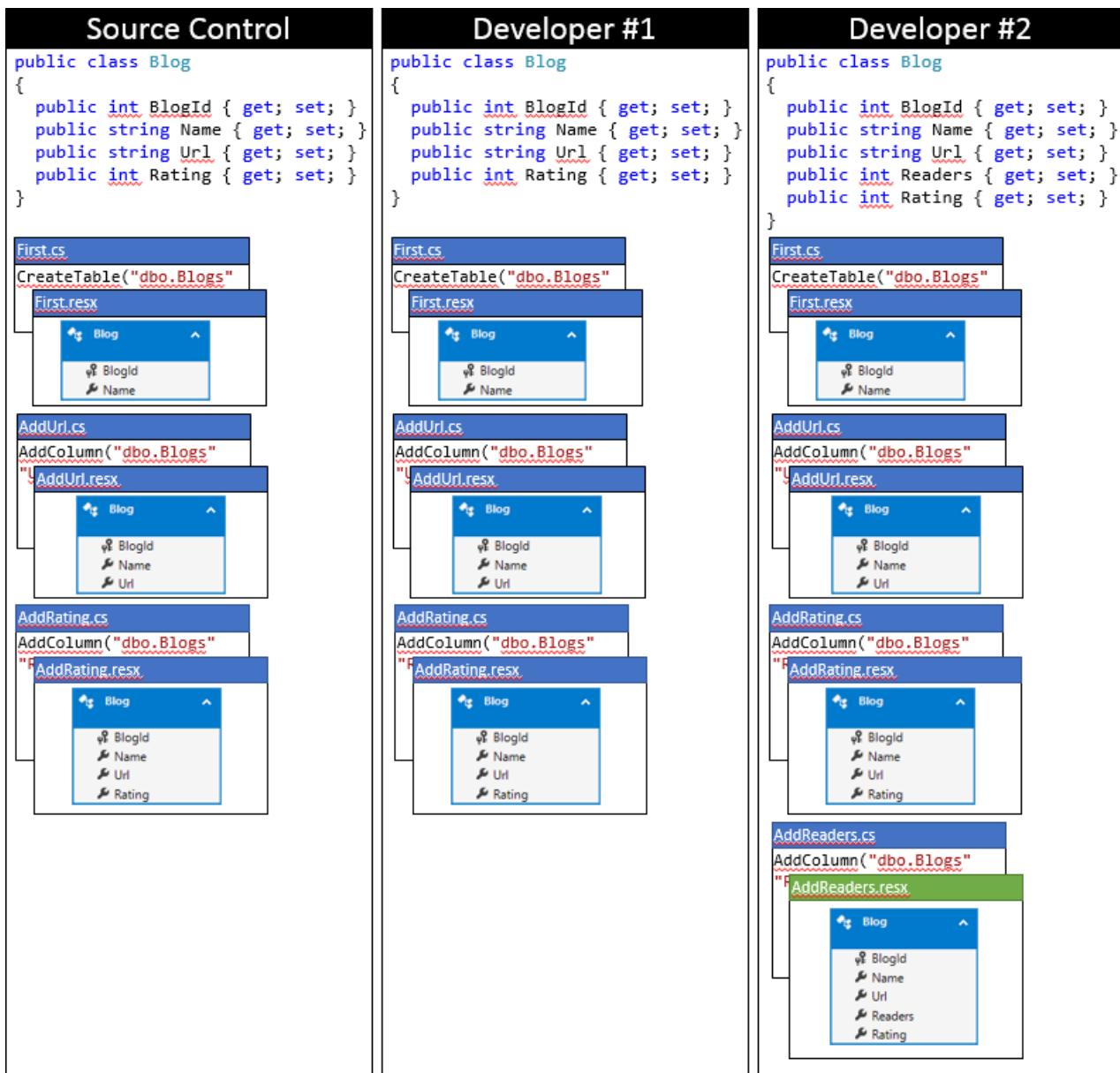
Этот подход возможно только в том случае, если существует только в базе данных локального кода последней миграции и еще не был отправлен в систему управления версиями (например, если последняя миграция была создана пользователем, выполняющим слияния). Изменение метаданных миграций, которые другие разработчики уже применяется к их разработке базы данных — или даже хуже, применяется к производственной базы данных — может привести непредвиденные побочные эффекты. Во время процесса мы собираемся откат последней миграции в локальной базе данных и повторно применить с помощью обновленных метаданных.

Во время последней миграции необходимо просто быть в локальной базе нет никаких ограничений на количество или порядок операций миграции, продолжить его кода. Может существовать несколько миграций из нескольких различных разработчиков и те же действия применяются — мы только что рассмотрели два для простоты.

Для этого подхода, начиная с момента вы поймете, что у вас есть изменения, которые должны быть синхронизированы из системы управления версиями можно использовать следующий процесс.

1. Убедитесь, что любые изменения ожидающие модели в базе данных локального кода были записаны для миграции. Этот шаг гарантирует, что не пропустите любой законные возможности внесения изменений, когда придет время для создания пустого миграции.
2. Синхронизация с системой управления версиями.
3. Запустите **Update-Database** применить все новые миграции, которые возвращены в других разработчиков. *** Примечание: *** Если вы не получаете предупреждения на наличие с помощью команды Update-Database, а затем было ни одной новой миграции от других разработчиков, и нет необходимости для выполнения дальнейшей слияние.*
4. Запустите **обновления базы данных — TargetMigration <второй_последнего_миграции>** (в примере, мы выполняли это было бы **обновления базы данных — TargetMigration AddRating**). Этой роли базы данных обратно в состояние второй последнего миграции — фактически «без применения» последней миграции из базы данных. *** Примечание: *** этот шаг необходим, чтобы сделать безопасно изменить метаданные миграции, так как метаданные также хранятся в _MigrationsHistoryTable базы данных. Именно поэтому этот параметр следует использовать только в случае, если последняя миграция только в локальной базе. Если других баз данных было последней миграцией, применяемой также пришлось бы их откат и повторное применение последней миграции для обновления метаданных.*
5. Запустите **Add-Migration <полный_имя_включая_timestamp_из_последнего_миграции >** (в примере мы выполняли это было бы что-нибудь вроде **201311062215252 Add-Migration_AddReaders**). *** Примечание: *** необходимо включить метку времени, чтобы миграция знал, нужно изменить существующие миграции, а не формирование шаблонов новый. Будут обновлены метаданные для последней миграции в соответствии с текущей моделью. Вы получите следующее предупреждение, когда команда будет выполнена, но это именно то, что нужно. "Только код конструктора для миграции" 201311062215252_AddReaders была повторно шаблонный. Чтобы повторно сформировать шаблон миграции, используйте параметр - Force.»*
6. Запустите **Update-Database** повторно применить последнюю миграцию с обновленными метаданными.
7. Продолжить разработку, или отправьте в систему управления версиями (после запуска модульных тестов конечно).

Здесь находится в состоянии разработчика #2 локальный кода после использования этого подхода.



Сводка

Существуют некоторые проблемы при использовании Code First Migrations в среде группы. Тем не менее базовое представление о принципах работы миграций и некоторые простые подходы для разрешения конфликтов слияния позволяют легко преодолеть эти трудности.

Фундаментальная проблема заключается в неверные метаданные, сохраненные в последней миграции. В результате Code First неправильно обнаружить, что текущая модель и схемы базы данных не совпадают и сформировать шаблон неверный код в следующей миграции. Такой ситуации можно устраниТЬ путем создания пустой миграцию с подходящей модели или обновления метаданных в последней миграции.

Model First

27.09.2018 • 14 minutes to read • [Edit Online](#)

В этом пошаговом руководстве видео и пошаговые познакомят вас с первой модели разработки, использующий Entity Framework. Во-первых, модель позволяет создать новую модель с помощью Entity Framework Designer, а затем создать схему базы данных из модели. Модель хранится в EDMX-файла (расширение EDMX) и их можно просмотреть и изменить в конструкторе Entity Framework. Классы, которые взаимодействуют с в приложении автоматически создаются из файла EDMX.

Просмотреть видео

В этом пошаговом руководстве видео и пошаговые познакомят вас с первой модели разработки, использующий Entity Framework. Во-первых, модель позволяет создать новую модель с помощью Entity Framework Designer, а затем создать схему базы данных из модели. Модель хранится в EDMX-файла (расширение EDMX) и их можно просмотреть и изменить в конструкторе Entity Framework. Классы, которые взаимодействуют с в приложении автоматически создаются из файла EDMX.

Представляет: Роэн Миллер (Rowan Miller)

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Для выполнения этого пошагового руководства вам потребуется Visual Studio 2010 или Visual Studio 2012.

Если вы используете Visual Studio 2010, также необходимо будет иметь [NuGet](#) установлен.

1. Создание приложения

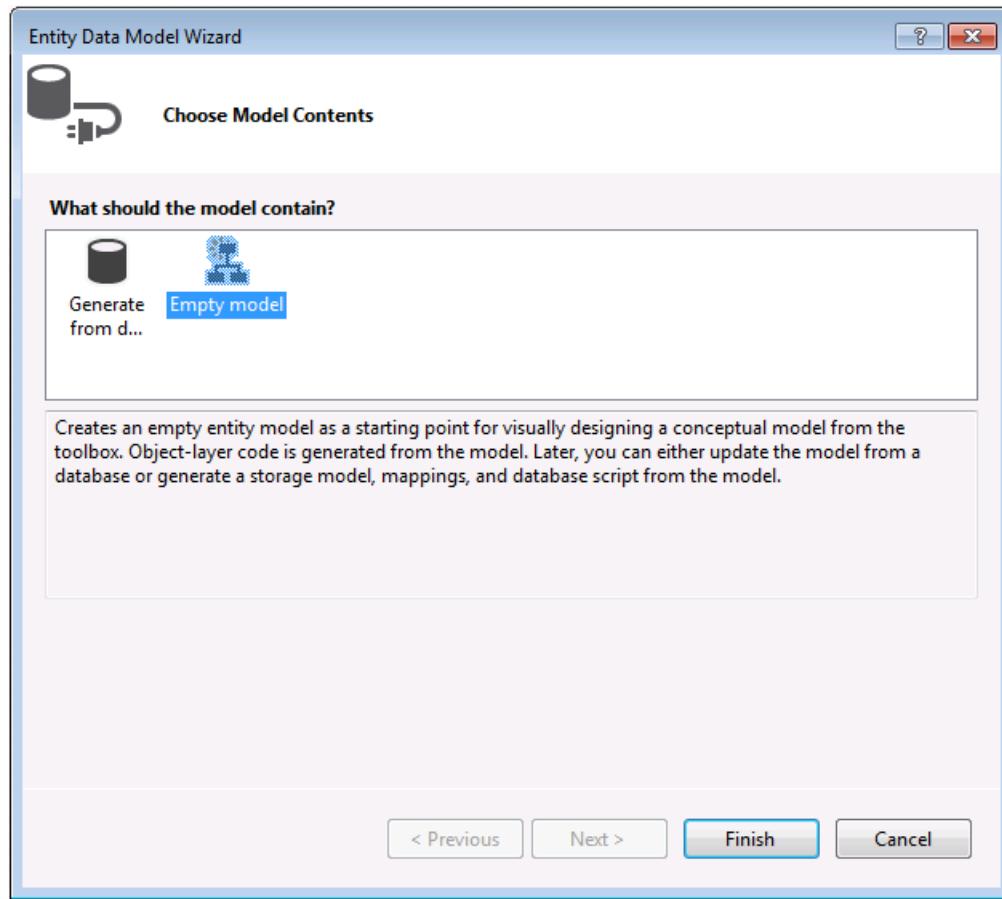
Для простоты мы создадим простое консольное приложение, использующего Model First для осуществления доступа к данным:

- Открытие Visual Studio
- **Файл —> Новинка —> проекта...**
- Выберите **Windows** в меню слева и **консольного приложения**
- Введите **ModelFirstSample** как имя
- Нажмите кнопку **OK**

2. Создание модели

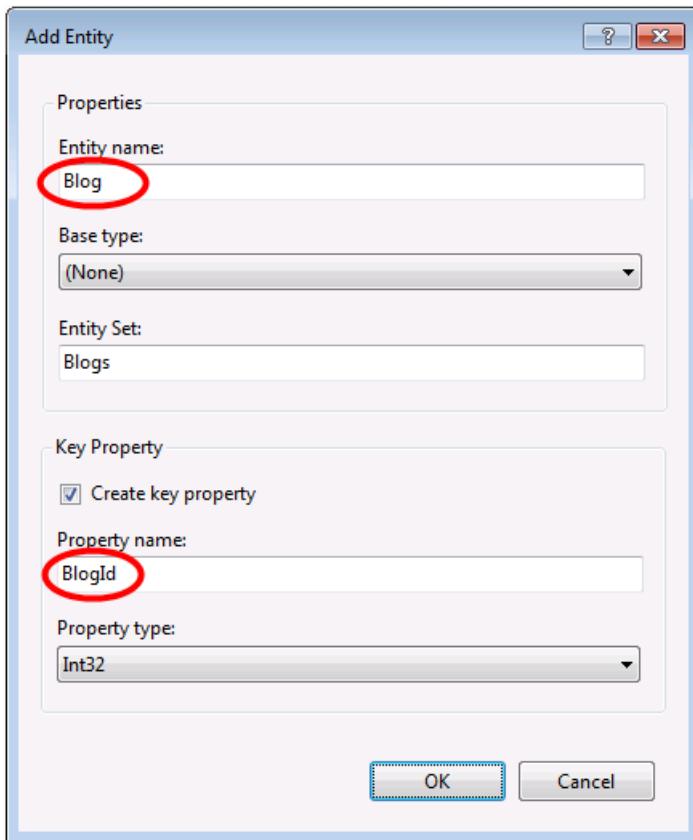
Мы собираемся использовать Entity Framework Designer, который входит в состав Visual Studio, для создания нашей модели.

- **Проект -> добавить новый элемент...**
- Выберите **данных** в меню слева и затем **модель EDM ADO.NET**
- Введите **BloggingModel** имя и нажмите кнопку **OK**, запустится мастер моделей EDM
- Выберите **пустую модель** и нажмите кнопку **Готово**



Entity Framework Designer открывается с пустой модели. Теперь мы сможем добавлять сущности, свойства и ассоциации в модели.

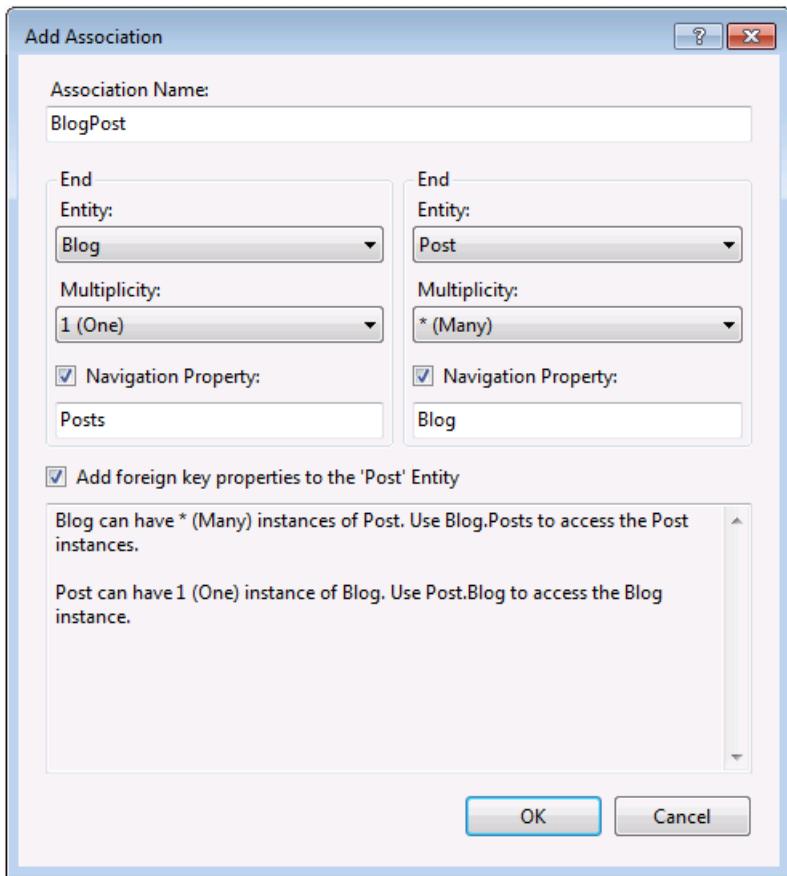
- Щелкните правой кнопкой мыши на область конструктора и выберите **свойства**
- В окне изменения свойства **имени контейнера сущностей** для **BloggingContext** имя производного контекста, который будет создан автоматически, контекст Представляет сеанс с базой данных, позволяет запрашивать и сохранять данные
- Щелкните правой кнопкой мыши на область конструктора и выберите **Add New -> сущности...**
- Введите **блог** как имя сущности и **BlogId** имя ключа и нажмите кнопку **OK**



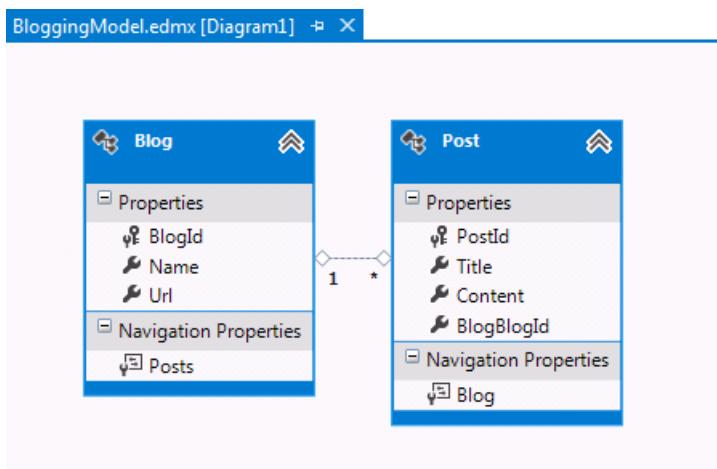
- Щелкните правой кнопкой мыши на новую сущность в область конструктора и выберите **Add New -> скалярное свойство**, введите **имя** как имя свойства.
- Повторите эту процедуру, чтобы добавить **URL-адрес** свойство.
- Щелкните правой кнопкой мыши **URL-адрес** свойство в область конструктора и выберите **свойства**, в окне изменения свойства **Nullable** присвоить **True** Это позволяет сохранить блог в базе данных, не назначая его URL-адрес
- С помощью методик, вы только что ознакомились предварительными сведениями, добавьте **Post** сущность с **PostId** ключевое свойство
- Добавить **Title** и **содержимого** скалярные свойства **Post** сущности

Теперь, когда у нас есть несколько сущностей, пришло время добавить сопоставление (или связь) между ними.

- Щелкните правой кнопкой мыши на область конструктора и выберите **Add New -> связи...**
- Сделать один конец связи пункты **блог** с кратность **один** и других конечную точку для **Post** с кратность **многих** Это означает, что блог записях и Post принадлежит один блог
- Убедитесь, **добавить свойства внешнего ключа для сущности «Post»** установлен флагок и нажмите кнопку **OK**



Теперь у нас есть простой модели, которую можно создать базу данных из и использовать для чтения и записи данных.



Дополнительные действия в Visual Studio 2010

Если вы работаете в Visual Studio 2010, существуют некоторые дополнительные действия, которые необходимо выполнить для обновления до последней версии платформы Entity Framework. Обновление важно, так как он предоставляет вам доступ к Улучшенная поверхность API, то есть гораздо проще в использовании, а также последние исправления ошибок.

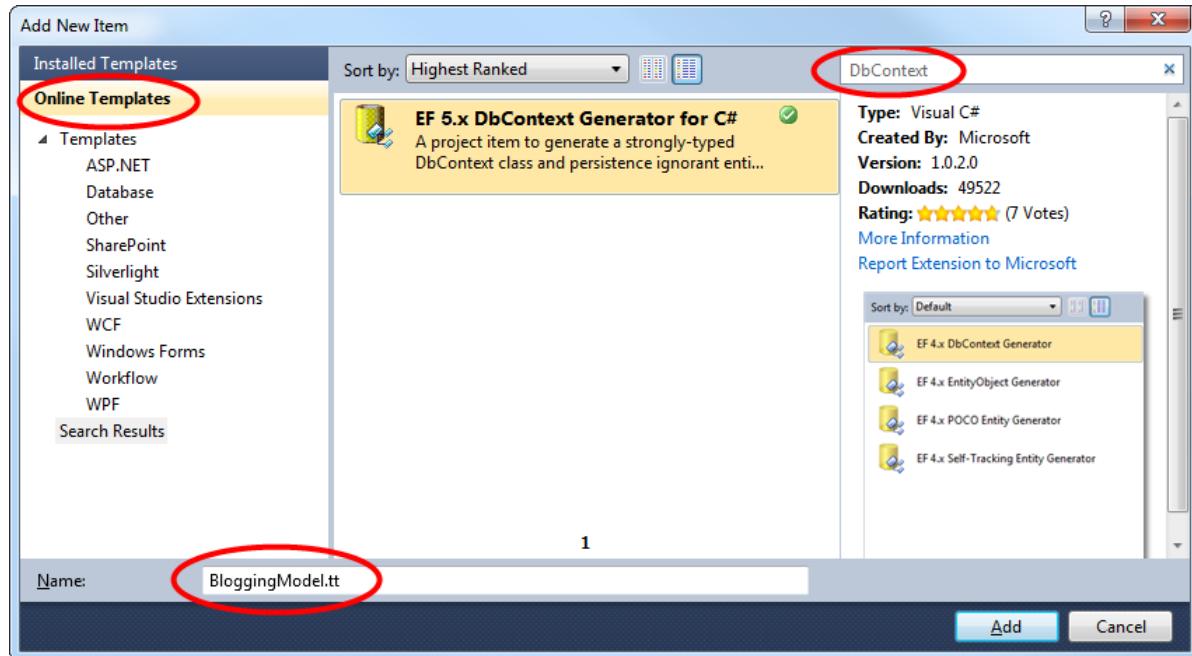
Во-первых, нам нужно получить последнюю версию Entity Framework из NuGet.

- **Проект —> управление пакетами NuGet... ** *При отсутствии *управление пакетами NuGet... ** следует установить параметр [последнюю версию NuGet](#)
- Выберите **Online** вкладку
- Выберите **EntityFramework** пакета
- Нажмите кнопку **установки**

Далее нам нужно переключить нашей модели, для которого создается код, который использует API

DbContext, который появился в более поздних версиях Entity Framework.

- Щелкните правой кнопкой мыши пустое место модели в конструкторе EF и выберите **добавить элемент формирования кода...**
- Выберите **шаблоны в Интернете** из меню слева и выполните поиск **DbContext**
- Выберите **EF 5.x генератор DbContext для C#**, введите **BloggingModel** имя и нажмите кнопку **добавить**



3. Создание базы данных

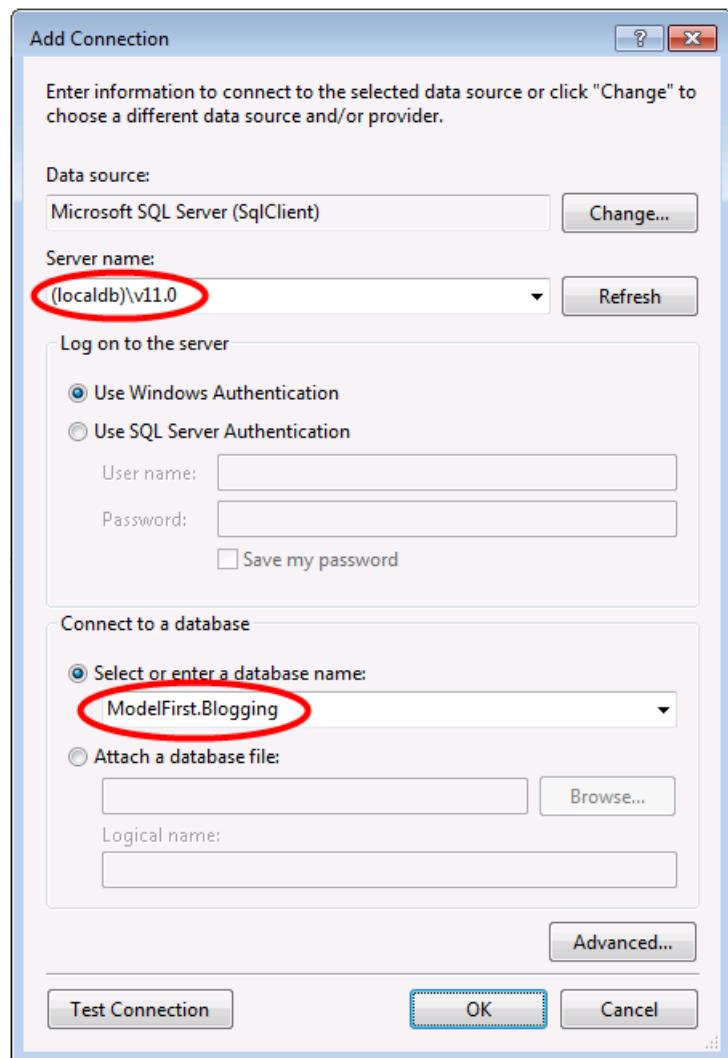
Учитывая нашу модель, Entity Framework можно вычислить схему базы данных, которые позволят нам для хранения и извлечения данных с помощью модели.

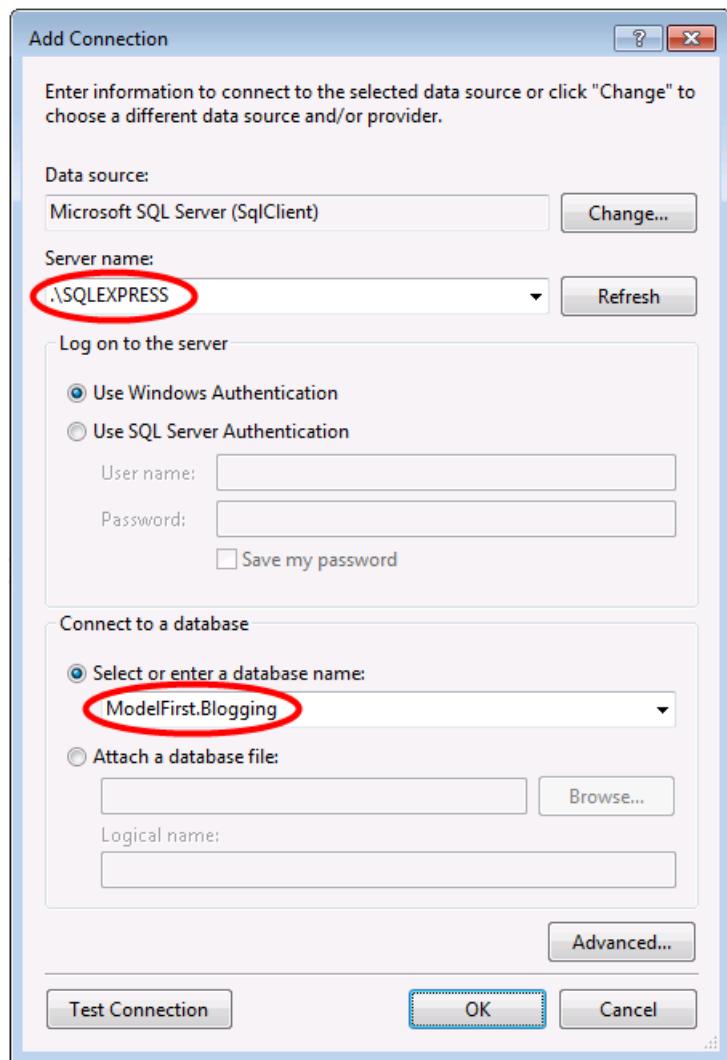
Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, вы установили:

- Если вы используете Visual Studio 2010 вы создадите базу данных SQL Express.
- Если вы используете Visual Studio 2012, а затем вы создадите **LocalDB** базы данных.

Перейдем дальше и создать базу данных.

- Щелкните правой кнопкой мыши на область конструктора и выберите **создать базу данных из модели...**
- Нажмите кнопку **новое соединение...** и укажите LocalDB или SQL Express, в зависимости от версии Visual Studio вы используете, введите **ModelFirst.Blogging** имя базы данных.



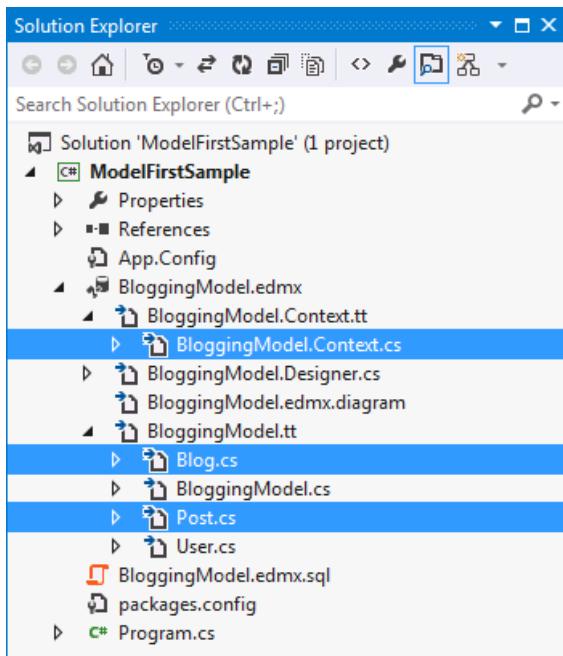


- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**
- Выберите **Далее** и Entity Framework Designer вычислит скрипт для создания схемы базы данных
- Если он отображается, нажмите кнопку **Готово**, сценарий будет добавлен в проект и открывается
- Щелкните правой кнопкой мыши сценарий, а затем выберите **Execute**, вам будет предложено указать базу данных для подключения, укажите LocalDB или SQL Server Express, в зависимости от версии Visual Studio вы используете

4. Чтение и запись данных

Теперь, когда у нас есть модель, настала пора использовать ее для доступа к некоторые данные. Классы мы будем использовать для доступа к данным автоматически создаются зависимости в EDMX-файл.

Этот снимок экрана из Visual Studio 2012, если вы используете Visual Studio 2010 BloggingModel.tt и BloggingModel.Context.tt файлов будет непосредственно в проекте, а не вложен в узел EDMX-файла.



Реализуйте метод Main в файле Program.cs, как показано ниже. Этот код создает новый экземпляр класса наш контекст, а затем использует его для вставки нового блога. Затем он использует запрос LINQ для извлечения из базы данных, представлены в алфавитном порядке по названию все блоги.

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Теперь можно запустить приложение и протестировать его.

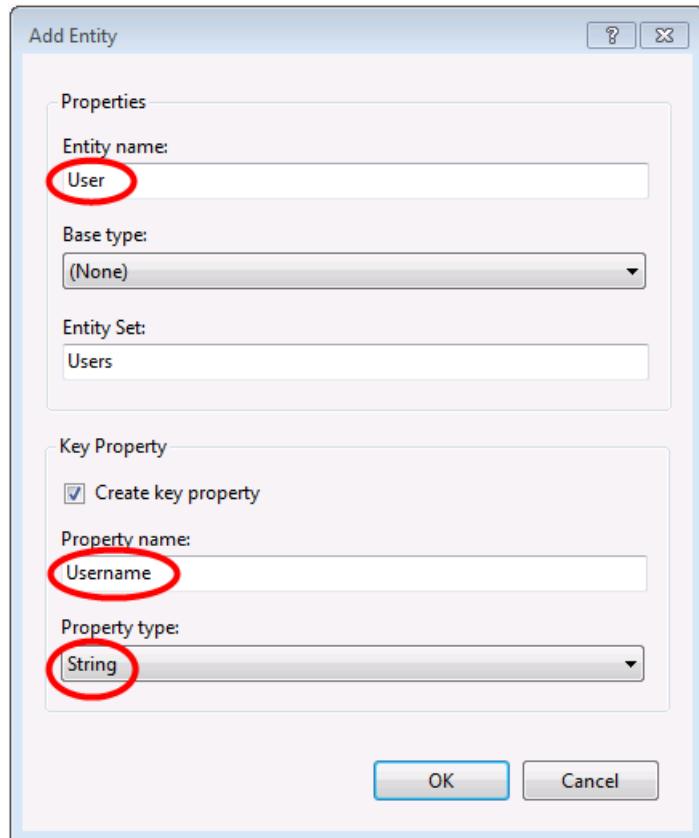
```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

5. Изменения модели

Теперь пора внести некоторые изменения в нашей модели, когда мы внести эти изменения, необходимо также обновить схему базы данных.

Мы начнем с добавления новой сущности пользователя для нашей модели.

- Добавьте новый **пользователя** имя сущности с **Username** как имя ключа и **строка** как тип свойства для ключа



- Щелкните правой кнопкой мыши **Username** свойство в область конструктора и выберите **свойства**, изменение свойства окна **MaxLength** присвоить значение **50 ** Это ограничивает данные, которые могут быть сохранены в имени пользователя до 50 символов
- Добавить **DisplayName** скалярного свойства к **пользователя** сущности

Теперь у нас есть обновленной модели, и мы готовы обновить базу данных в соответствии с наш новый тип сущности пользователя.

- Щелкните правой кнопкой мыши на область конструктора и выберите **создать базу данных из модели...**, Entity Framework будет вычислять скрипт для повторного создания схемы на основе обновленной модели.
- Нажмите кнопку **Готово**
- Можно получать предупреждения о перезаписи существующего скрипта DDL и сопоставления и хранения части модели, щелкните **Да** для обоих этих предупреждений
- Обновленный сценарий SQL, чтобы создать базу данных открывается при
Созданный скрипт будет удалять все существующие таблицы и заново создать схему с нуля. Это может работать для локальной разработки, но не является приемлемым для принудительной отправки изменений в базе данных, который уже был развернут. Если вам нужно опубликовать изменения в уже развернутой базе данных, необходимо будет изменить сценарий или использовать средство сравнения схем для вычисления скрипта переноса.
- Щелкните правой кнопкой мыши сценарий, а затем выберите **Execute**, вам будет предложено указать

базу данных для подключения, укажите LocalDB или SQL Server Express, в зависимости от версии Visual Studio вы используете

Сводка

В этом пошаговом руководстве мы рассмотрели сначала модель разработки, которое позволяет создать модель в конструкторе EF, а затем создать базу данных из этой модели. Затем мы использовали модель для чтения и записи некоторых данных из базы данных. Наконец мы обновившего модель и создать повторно схему базы данных для соответствия модели.

База данных как основа

16.10.2018 • 11 minutes to read • [Edit Online](#)

Следующее видео и пошаговое руководство познакомят вас с разработкой на основе базы данных в Entity Framework. Такой подход дает возможность реконструировать модель по существующей базе. Модель хранится в EDMX-файле (расширение .emdx), и её можно просмотреть и изменить в Entity Framework Designer. Классы, с которыми вы взаимодействуете в приложении, автоматически создаются из файла EDMX.

Просмотрите видео

Это видео предоставляет общие сведения о разработке на основе базы данных в Entity Framework. Такой подход дает возможность реконструировать модель по существующей базе. Модель хранится в EDMX-файле (расширение .emdx), и её можно просмотреть и изменить в Entity Framework Designer. Классы, с которыми вы взаимодействуете в приложении, автоматически создаются из файла EDMX.

Представляет: Роэн Миллер (Rowan Miller)

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Для выполнения этого пошагового руководства необходимо иметь по крайней мере Visual Studio 2010 или Visual Studio 2012.

Если вы используете Visual Studio 2010, также необходимо будет установить [NuGet](#).

1. Создание базы данных

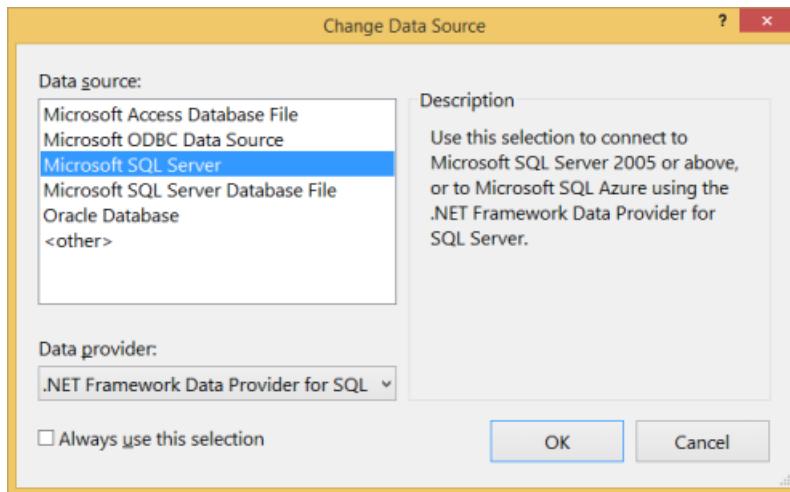
Обычно при ориентировании на существующую базу данных она уже будет создана, но в этом пошаговом руководстве нам нужно создать базу данных для доступа.

Сервер базы данных, который устанавливается вместе с Visual Studio, отличается в зависимости от версии Visual Studio, которую вы установили:

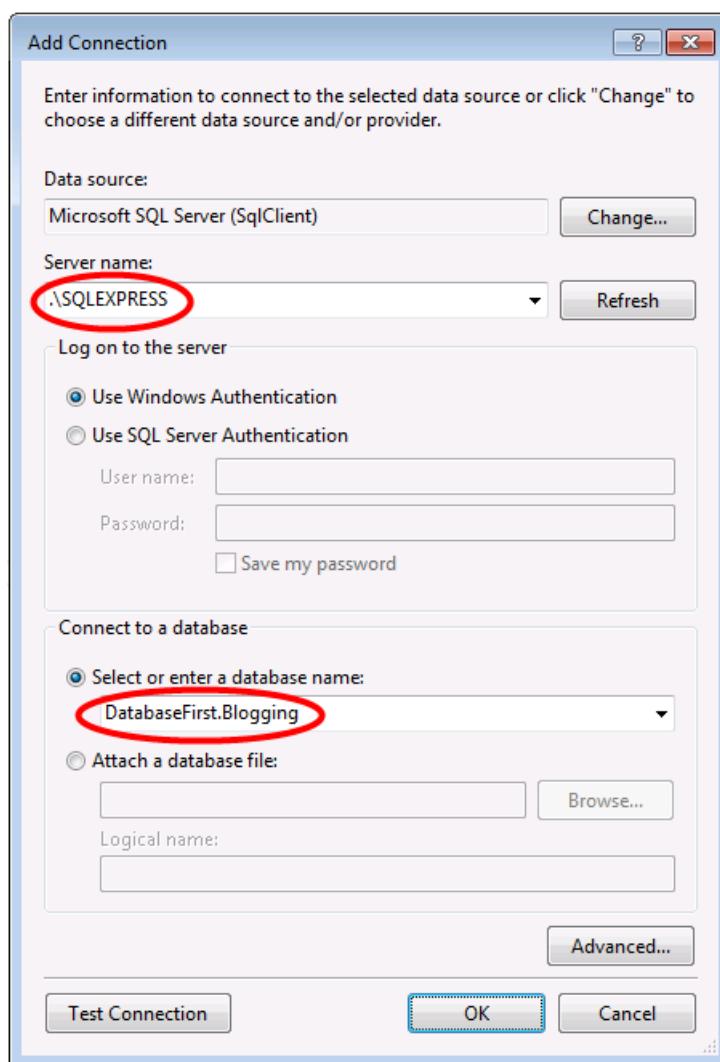
- Если вы используете Visual Studio 2010, вы создадите базу данных SQL Express.
- Если вы используете Visual Studio 2012, вы создадите базу данных [LocalDB](#).

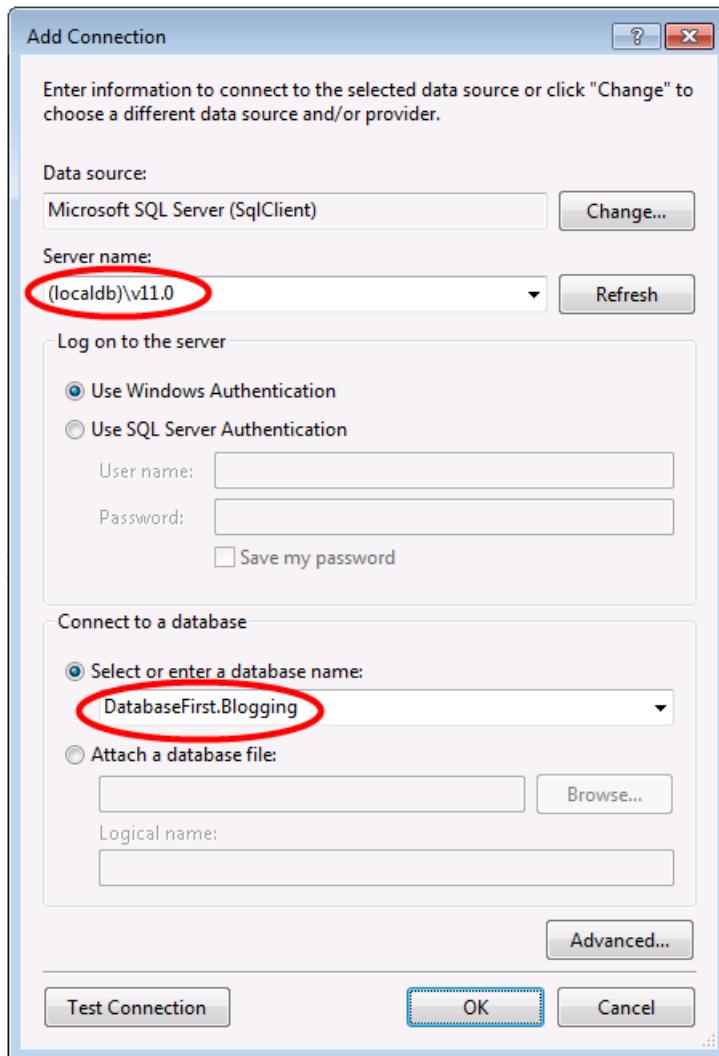
Перейдем дальше и создадим базу данных.

- Откройте Visual Studio
- **Вид** —> **"Обозреватель сервера"**
- Щелкните правой кнопкой мыши на **Подключения к данным** -> **Добавить подключение...**
- Если вы не подключались к базе данных с помощью "Обозревателя сервера" ранее, потребуется выбрать Microsoft SQL Server в качестве источника данных



- Подключитесь к LocalDB или SQL Express, в зависимости от того, какую из них вы установили, и введите имя базы данных **DatabaseFirst.Blogging**





- Выберите **OK**, и вам будет задан вопрос, хотите ли вы создать новую базу данных. Выберите **Да**



- Новая база данных появится в Обозревателе сервера. Щелкните ее правой кнопкой мыши и выберите **Новый запрос**
- Скопируйте следующий код SQL в новый запрос, а затем щелкните запрос правой кнопкой мыши и выберите **Выполнить**

```
CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId]) ON
DELETE CASCADE
);
```

2. Создайте приложение

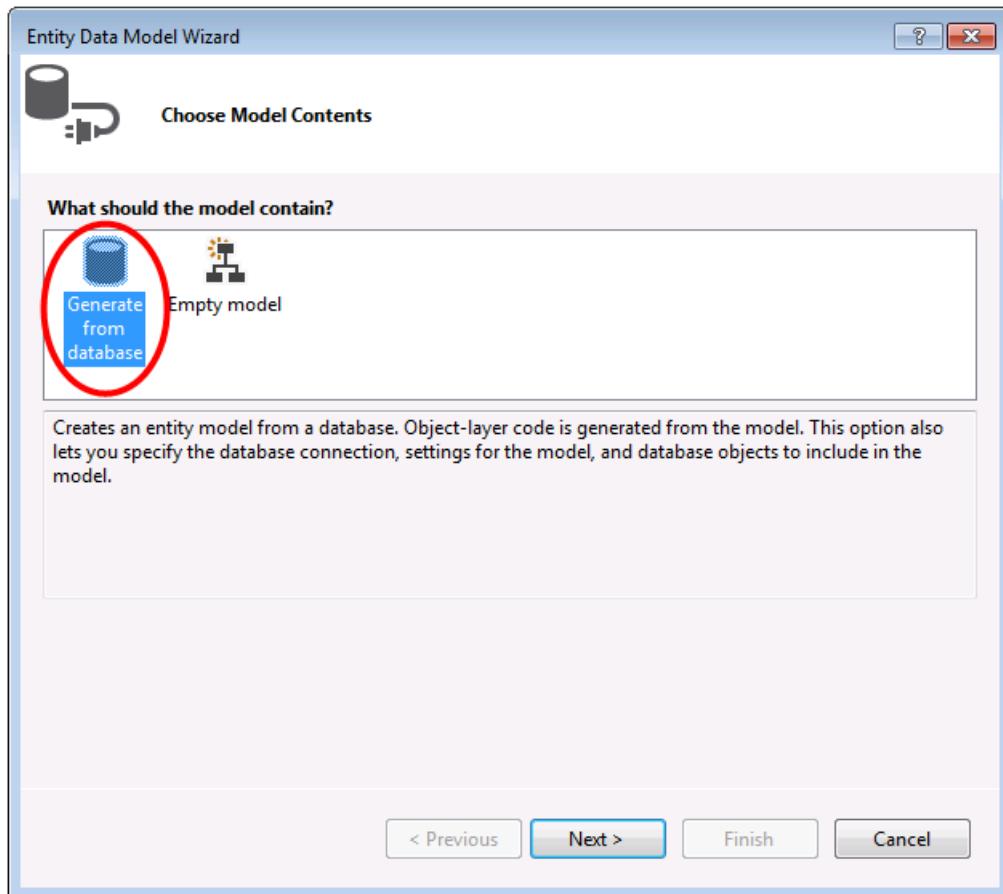
Для простоты мы создадим простое консольное приложение, использующее подход, основывающийся на базе данных, для выполнения доступа к данным:

- Откройте Visual Studio
- **Файл —> Создать —> Проект...**
- В меню слева выберите **Windows и Консольное приложение**
- Введите **DatabaseFirstSample** в поле "Имя"
- Нажмите кнопку **OK**

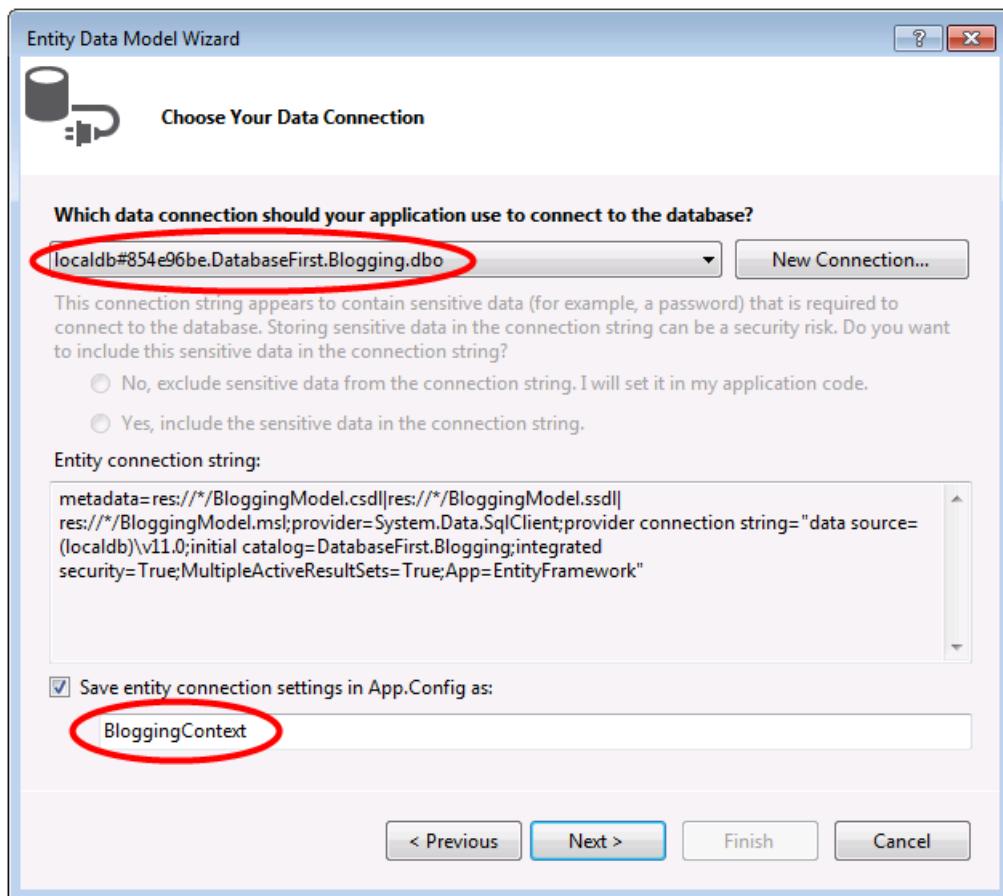
3. Реконструируйте модель

Для создания нашей модели мы собираемся использовать конструктор Entity Framework Designer, который входит в состав Visual Studio.

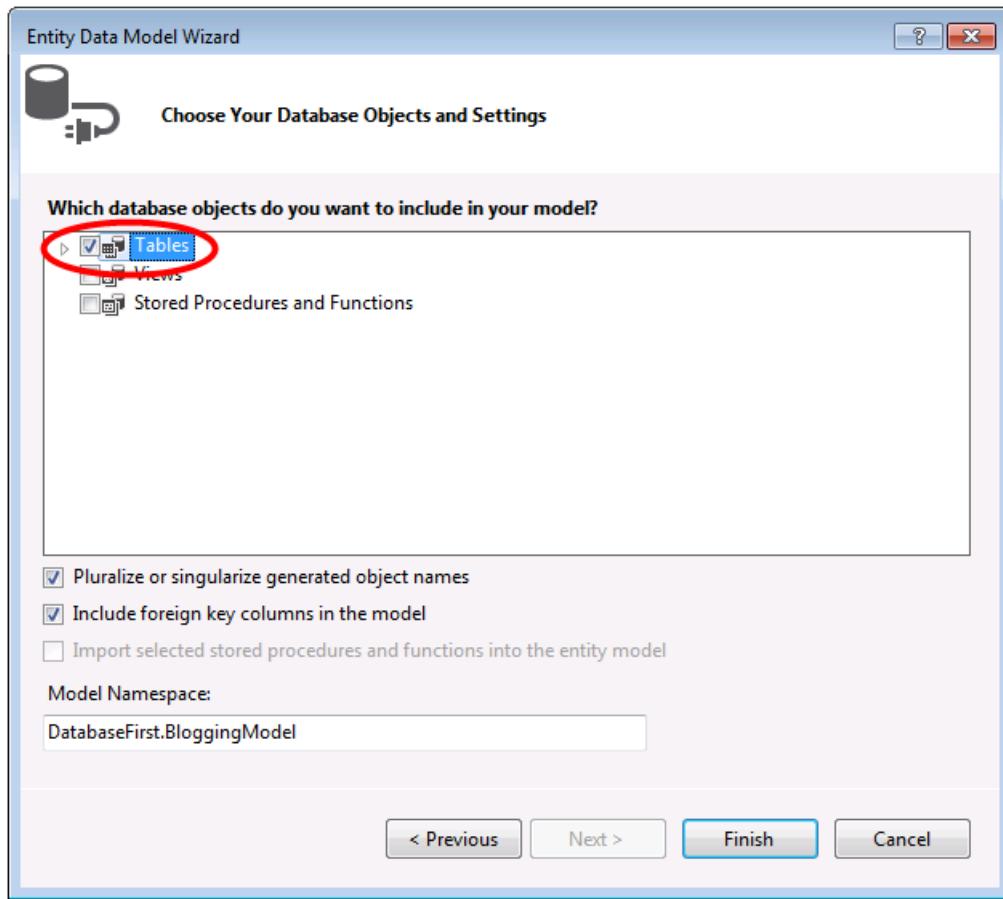
- **Проект -> Добавить новый элемент...**
- Выберите **Данные** в меню слева и затем **Модель ADO.NET EDM**
- Введите имя **BloggingModel** и нажмите кнопку **OK**
- Это откроет **Мастер моделей EDM**
- Выберите **Создать из базы данных** и нажмите кнопку **Далее**



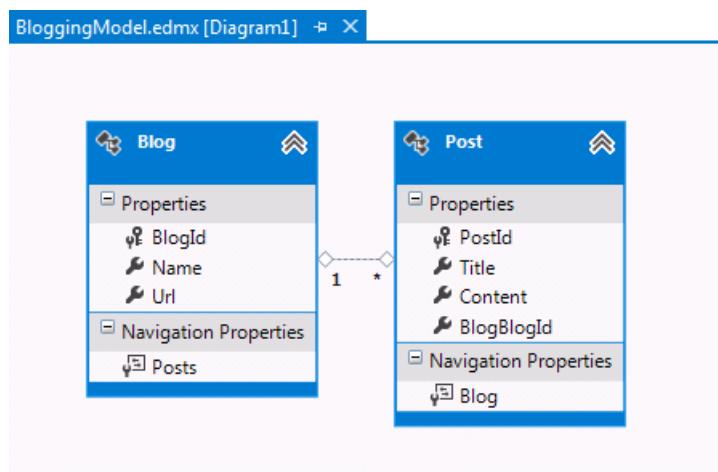
- Выберите соединение с базой данных, созданной в первом разделе, введите **BloggingContext** как имя строки подключения и нажмите кнопку **Далее**



- Установите флажок рядом с «Таблицы», чтобы импортировать все таблицы и нажмите кнопку «Готово»



После завершения процесса реконструирования новая модель будет добавлена в проект и откроется для просмотра в Entity Framework Designer. В проект также будет добавлен файл App.config со сведениями о подключении к базе данных.



Дополнительные действия в Visual Studio 2010

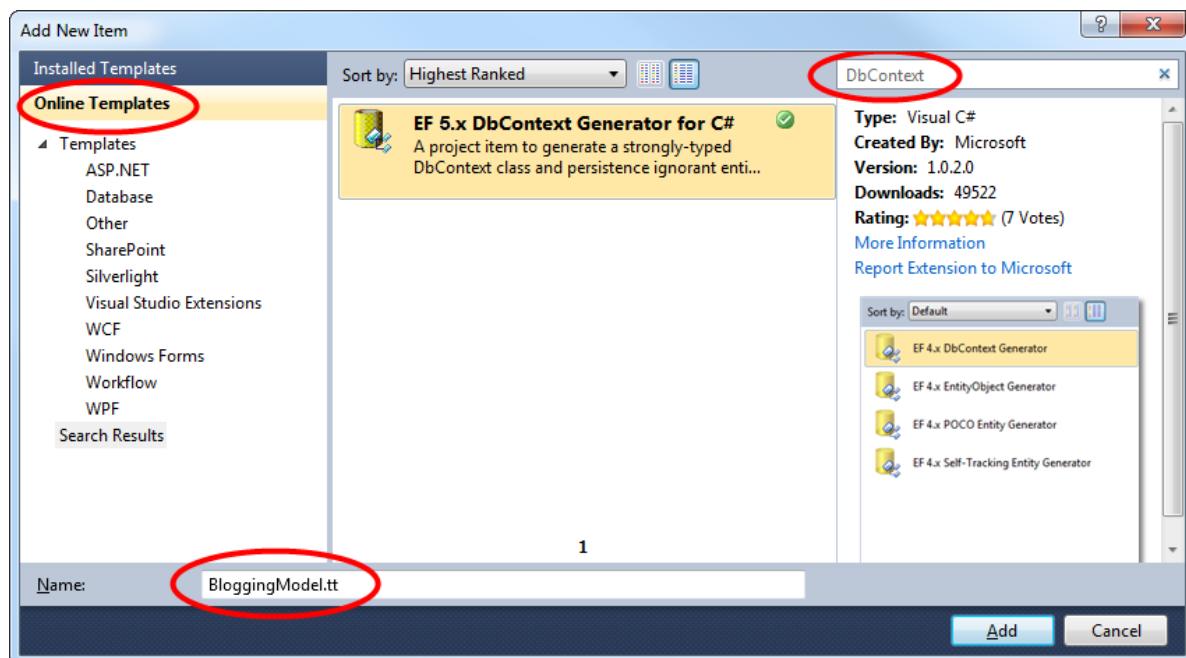
Если вы работаете в Visual Studio 2010, существуют некоторые дополнительные действия, которые необходимо выполнить для обновления до последней версии платформы Entity Framework. Обновление важно, так как оно предоставляет вам доступ к улучшенным API, которые гораздо проще в использовании, а также последние исправления ошибок.

Во-первых, нам нужно получить последнюю версию Entity Framework из NuGet.

- **Проект —> Управление пакетами NuGet...** *При отсутствии пункта *Управление пакетами NuGet...* следует установить [последнюю версию NuGet](#)
- Выберите вкладку **В сети**
- Выберите пакет **EntityFramework**
- Нажмите кнопку **Установить**

Далее нам нужно переключить нашу модель, чтобы сгенерировать код, который использует API DbContext, появившийся в более поздних версиях Entity Framework.

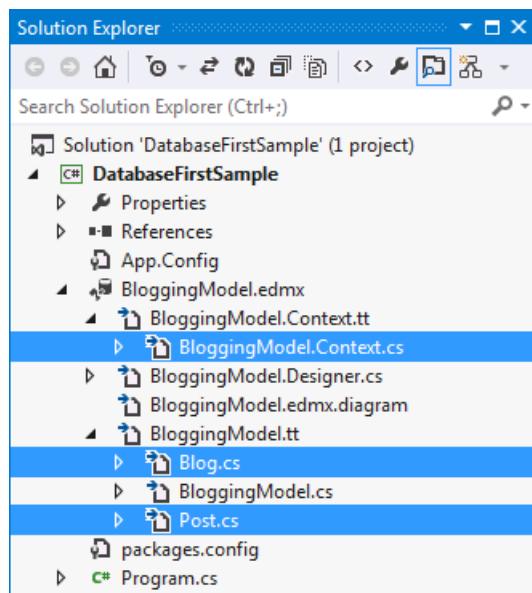
- Щелкните правой кнопкой мыши на пустом месте модели в EF Designer и выберите **Добавить элемент создания кода...**
- Выберите **Шаблоны в Интернете** из меню слева и выполните поиск **DbContext**
- Выберите **Генератор EF 5.x DbContext для C#**, введите имя **BloggingModel** и нажмите кнопку **Добавить**



4. Чтение и запись данных

Теперь, когда у нас есть модель, настала пора использовать ее для доступа к каким-нибудь данным. Классы, которые мы будем использовать для доступа к данным, автоматически создаются в зависимости от EDMX-файла.

Этот снимок экрана из Visual Studio 2012. Если вы используете Visual Studio 2010, файлы *BloggingModel.tt* и *BloggingModel.Context.tt* будут лежать непосредственно в проекте, а не вложены в узел EDMX-файла.



Реализуйте метод Main в файле Program.cs так, как показано ниже. Этот код создает новый экземпляр

нашего контекста, а затем использует его для вставки новой записи блога. Затем он использует запрос LINQ для извлечения из базы данных всех записей блога, упорядоченных в алфавитном порядке по названию.

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

Теперь можно запустить приложение и протестировать его.

```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

5. Изменения базы данных

Теперь пора внести некоторые изменения в схему нашей базы данных. Когда мы внесём эти изменения, необходимо также обновить нашу модель, чтобы отразить эти изменения.

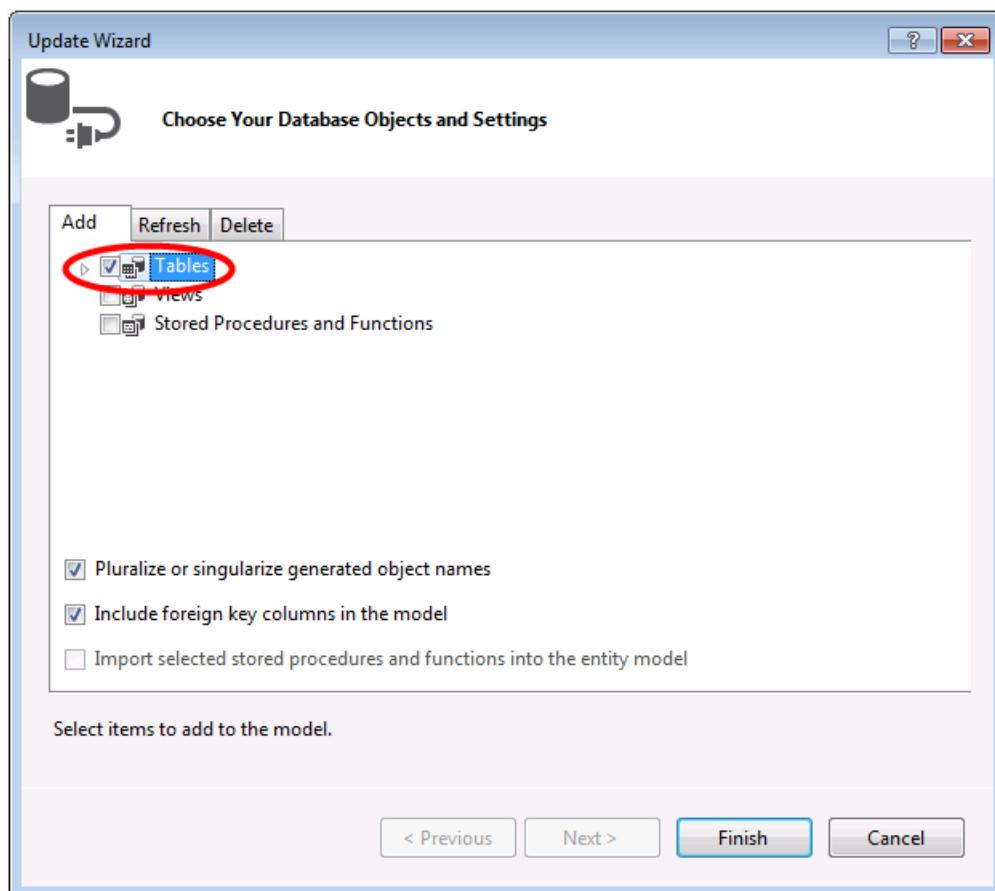
Первый шаг — это внесение некоторых изменений в схему базы данных. Мы собираемся добавить таблицу пользователей в схему.

- Щелкните правой кнопкой мыши базу данных **DatabaseFirst.Blogging** в Обозревателе сервера и выберите **Создать запрос**
- Скопируйте следующий код SQL в новый запрос, а затем щелкните запрос правой кнопкой мыши и выберите **Выполнить**

```
CREATE TABLE [dbo].[Users]
(
    [Username] NVARCHAR(50) NOT NULL PRIMARY KEY,
    [DisplayName] NVARCHAR(MAX) NULL
)
```

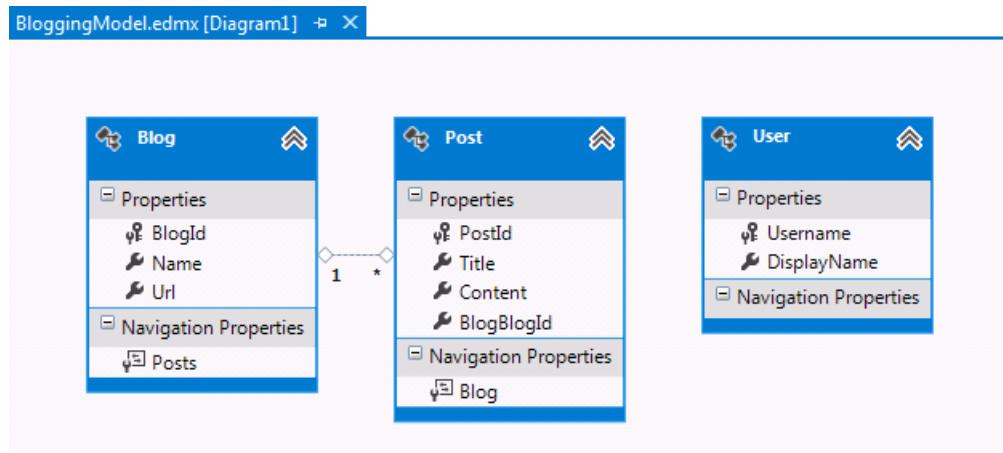
Теперь, когда схема обновлена, пришло время внести изменения в модель.

- Щелкните правой кнопкой мыши пустое место модели в конструкторе EF и выберите «Обновить модель из базы данных...», это приведет к запуску мастера обновления
- На вкладке "Добавление" мастера обновления установите флажок рядом с таблицами. Это означает, что мы хотим добавить новые таблицы из схемы. На вкладке "Обновление" отображаются все имеющиеся таблицы в модели, которые будут проверены на изменения во время обновления. На вкладке "Удаление" отображаются все таблицы, которые были удалены из схемы и которые будут удалены из модели как часть обновления. Сведения на этих двух вкладках формируются автоматически и предоставляются только в ознакомительных целях. Изменить какие-либо параметры невозможно.



- Щелкните "Готово" в мастере обновления

Теперь модель обновлена. Добавлена новая сущность "Пользователь", которая сопоставляется с таблицей пользователей, которую мы добавили в базу данных.



Сводка

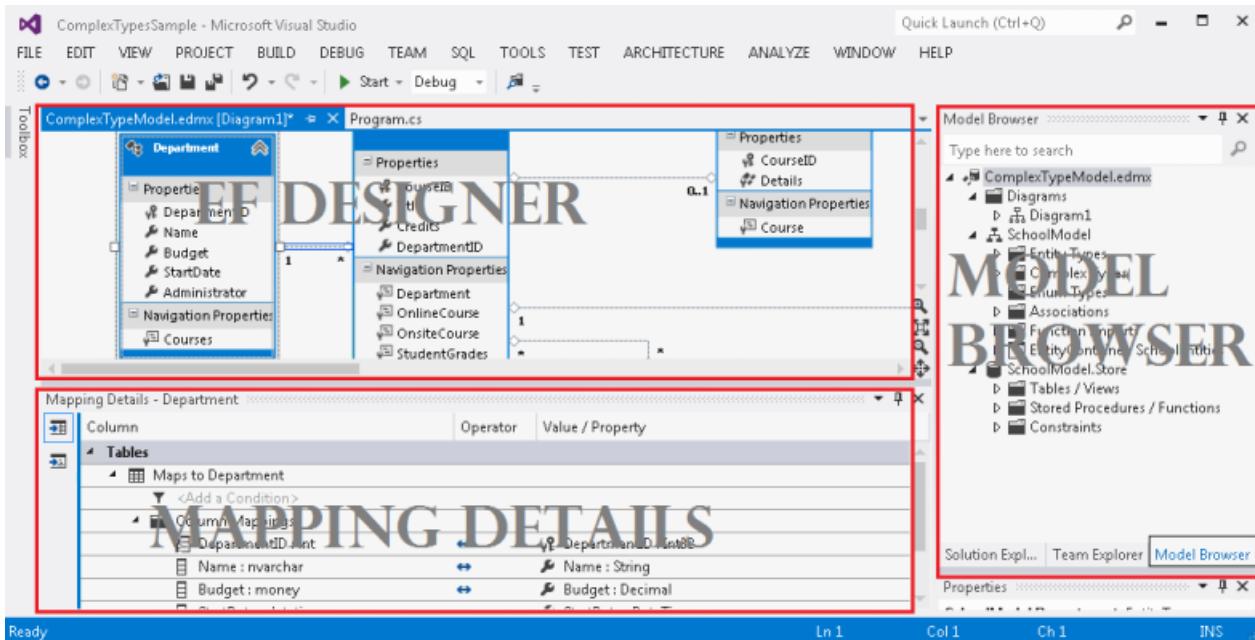
В этом пошаговом руководстве мы рассмотрели разработку на основе базы данных, которая позволяет создать в EF Designer модель, основываясь на существующей базе. Мы использовали эту модель для чтения и записи некоторых данных в базе. Наконец, мы обновили модель для отражения изменений, внесенных в схему базы данных.

Сложные типы – конструктор EF

13.09.2018 • 11 minutes to read • [Edit Online](#)

В этом разделе показано, как сопоставить сложных типов с помощью Entity Framework Designer (конструктор EF) и как выполнять запросы для сущностей, которые содержат свойства сложного типа.

На следующем рисунке показана основные окна, которые используются при работе с конструктором EF.



NOTE

При создании концептуальной модели, предупреждения о несопоставленных сущностях и ассоциациях, появляются в списке ошибок. Эти предупреждения можно игнорировать, поскольку после нажатия кнопки для создания базы данных из модели, пропадут.

Что такое сложный тип

Сложные типы — это нескалярные свойства типов сущности, которые позволяют организовать в сущностях скалярные свойства. Подобно сущностям, сложные типы состоят из скалярных свойств или свойств других сложных типов.

При работе с объектами, представляющими сложные типы, следует учитывать следующее:

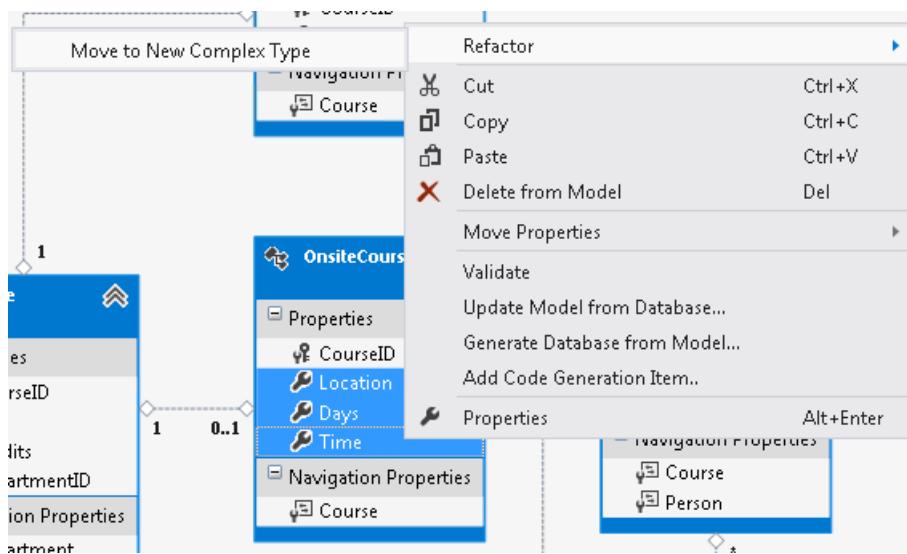
- Сложные типы не имеют ключей и поэтому не могут существовать независимо друг от друга. Сложные типы могут существовать только как свойства типов сущностей или других сложных типов.
- Сложные типы не могут участвовать в ассоциации и не может содержать свойства навигации.
- Свойства сложного типа не может быть **null**. ** InvalidOperationException ** возникает, когда **DbContext.SaveChanges** вызывается и обнаруживается неопределенный сложный объект. Скалярные свойства сложных объектов могут принимать **null**.
- Сложные типы не могут наследоваться от других сложных типов.
- Необходимо определить сложный тип как **класс**.
- EF определяет изменения в члены на объект сложного типа при **DbContext.DetectChanges** вызывается. Платформа Entity Framework вызывает **DetectChanges** автоматически, когда вызываются следующие

члены: **DbSet.Find**, **DbSet.Local**, **DbSet.Remove**, **DbSet.Add**, **DbSet.Attach**, **DbContext.SaveChanges**, **DbContext.GetValidationErrors**, **DbContext.Entry**, **DbChangeTracker.Entries**.

Оптимизация свойств сущности в новый сложный тип

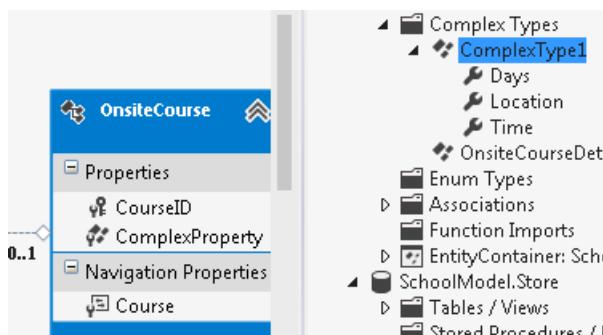
При наличии сущности в концептуальной модели может потребоваться реструктурировать некоторые свойства в свойство сложного типа.

В рабочей области конструктора, выберите один или несколько свойств (кроме свойств навигации) сущность, затем щелкните правой кнопкой мыши и выберите **рефакторинг -> переместить в новый сложный тип**.



Добавляется новый сложный тип с выбранными свойствами **браузер моделей**. Сложному типу присваивается имя по умолчанию.

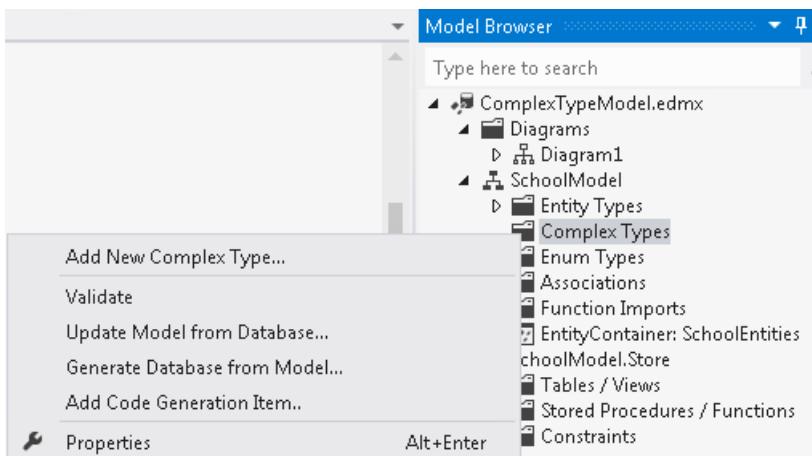
Сложное свойство только что созданного типа заменит выбранные свойства. Все сопоставления свойств будут сохранены.



Создать новый сложный тип

Можно также создать новый сложный тип, который не содержит свойства существующей сущности.

Щелкните правой кнопкой мыши **сложные типы** папки в браузере моделей, пункты **AddNew сложный тип...**. Кроме того, можно выбрать **сложные типы** папку и нажмите клавишу **вставить** на клавиатуре.



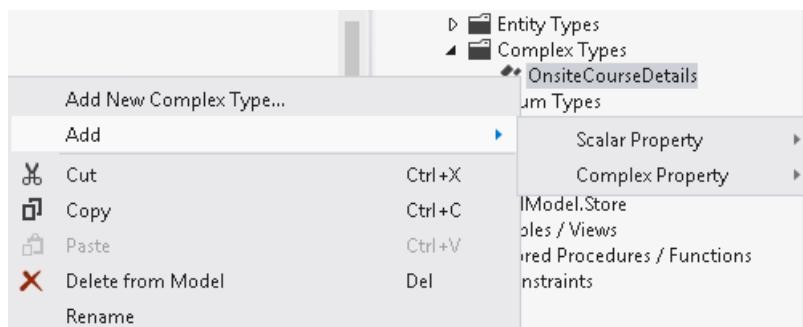
Новый сложный тип будет добавлен в папку с именем по умолчанию. Теперь можно добавить свойства к типу.

Добавлять свойства к сложному типу

Свойства сложного типа могут иметь как скалярные, так и существующие сложные типы. Однако свойства сложного типа не могут иметь циклических ссылок. Например, сложный тип **OnsiteCourseDetails** свойство сложного типа не может быть **OnsiteCourseDetails**.

Добавить свойство к сложному типу можно любым из следующих способов.

- Щелкните правой кнопкой мыши сложный тип в браузере моделей, выберите пункт **добавить**, наведите указатель на **скалярное свойство** или **сложное свойство**, затем выберите необходимый тип свойства. Кроме того, можно выбрать сложный тип и нажмите клавишу **вставить** на клавиатуре.



Новое свойство будет добавлено к сложному типу с именем по умолчанию.

- OR-
- Щелкните правой кнопкой мыши свойство сущности **конструктор EF** и выберите пункт **копирования**, щелкните правой кнопкой мыши сложный тип в **браузер моделей** и выберите **Вставить**.

Переименование сложного типа

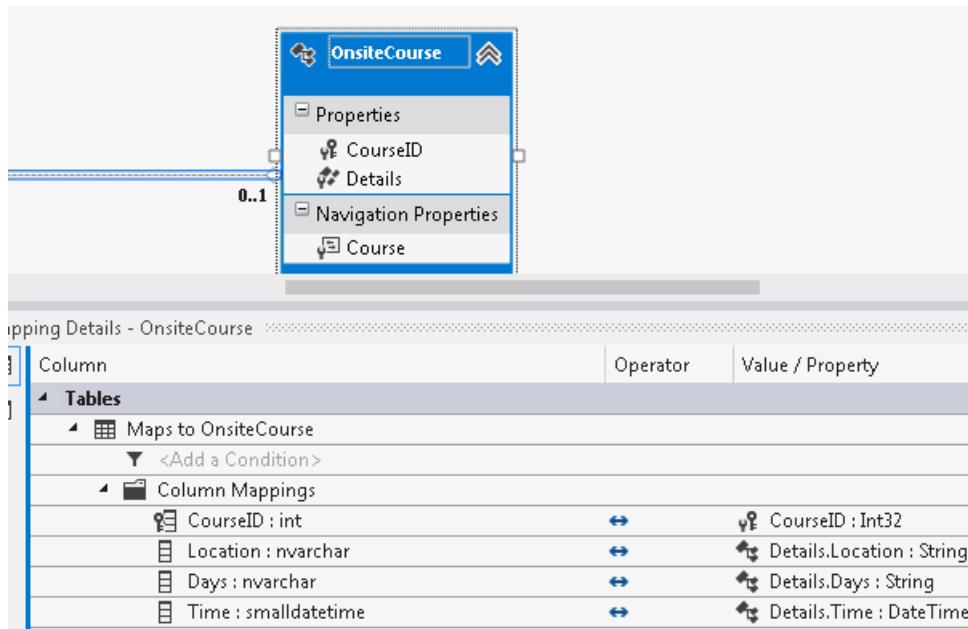
При переименовании сложного типа все ссылки на тип обновляются по всему проекту.

- Дважды щелкните сложный тип в **браузер моделей**. Имя будет выбрано в режиме редактирования.
- OR-
- Щелкните правой кнопкой мыши сложный тип в **браузер моделей** и выберите **Переименовать**.
- OR-
- Выберите сложный тип в браузере моделей и нажмите клавишу F2.

- OR-
- Щелкните правой кнопкой мыши сложный тип в **браузер моделей** и выберите **свойства**. Измените имя в **свойства** окна.

Добавление существующего сложного типа к сущности и сопоставить ее свойства со столбцами таблицы

1. Щелкните правой кнопкой мыши сущность, выберите пункт **Add New** и выберите **сложное свойство**. Свойство сложного типа с именем по умолчанию будет добавлено к сущности. Свойству назначается тип по умолчанию (выбранный из имеющихся сложных типов).
2. Назначьте свойству в требуемый тип **свойства** окна. После добавления свойства сложного типа к сущности необходимо сопоставить ее свойства со столбцами таблицы.
3. Щелкните правой кнопкой мыши тип сущности в области конструктора или в **браузер моделей** и выберите **сопоставления таблиц**. Сопоставления таблицы отображаются в **сведения о сопоставлении** окна.
4. Разверните **сопоставляется <имя таблицы>** узла. Объект **сопоставления столбцов** появится узел.
5. Разверните **сопоставления столбцов** узла. Появится список всех столбцов таблицы. Свойства по умолчанию (если такие имеются) с которыми сопоставляются столбцы отображаются в категории **значение/свойство** заголовок.
6. Выберите столбец, необходимо сопоставить и щелкните правой кнопкой мыши соответствующий **значение/свойство** поля. Отобразится раскрывающийся список всех скалярных свойств.
7. Выберите соответствующее свойство.



8. Повторите шаги 6 и 7 для каждого столбца таблицы.

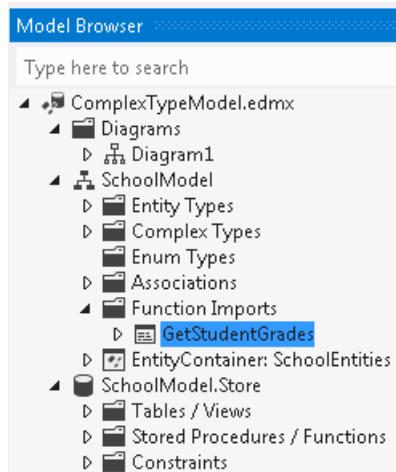
NOTE

Чтобы удалить сопоставление столбцов, выберите столбец, который необходимо сопоставить, а затем нажмите кнопку **значение/свойство** поля. Выберите **удалить** из раскрывающегося списка.

Сопоставить импорт функции со сложным типом

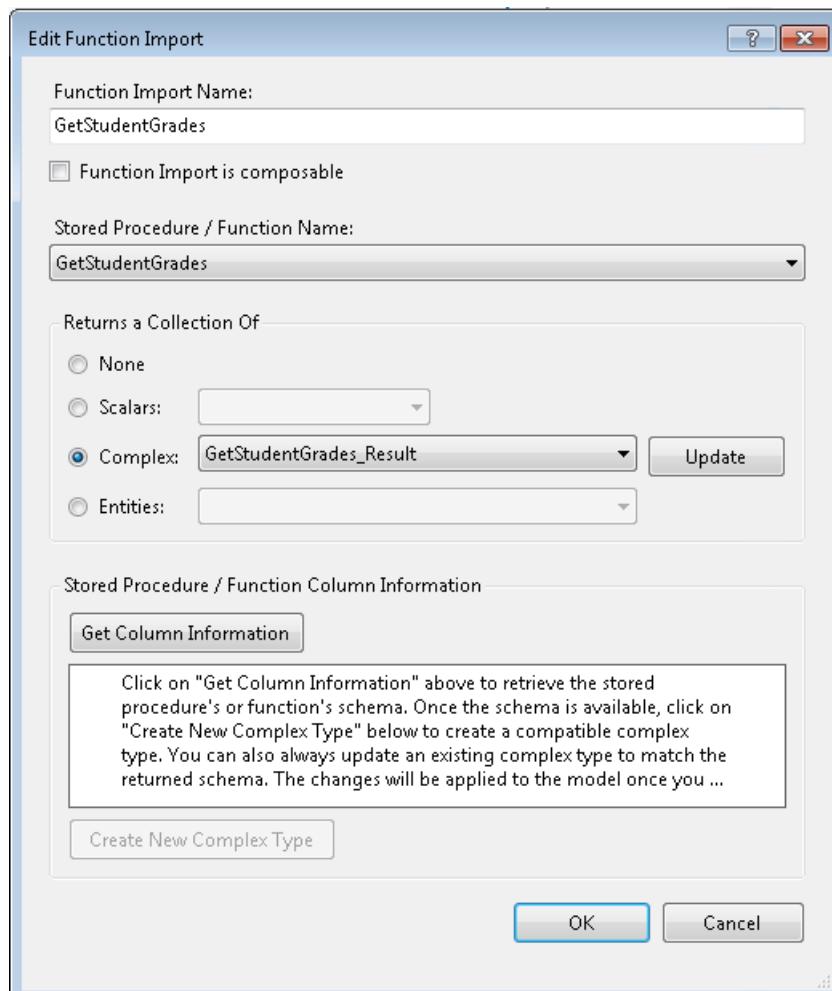
Импорт функций основан на хранимых процедурах. Чтобы сопоставить импорт со сложным типом, столбцы, возвращаемые соответствующей хранимой процедурой, должны соответствовать свойствам сложного типа по числу и должны иметь типы хранения, совместимые с типами свойств.

- Дважды щелкните импортированной функции, которую необходимо сопоставить со сложным типом.



- Задайте параметры для нового импорта функции следующим образом.

- Укажите хранимую процедуру, для которой создается импорт функции в **имя хранимой процедуры** поля. Это поле представляет собой раскрывающийся список, содержащий все хранимые процедуры, которые имеются в модели хранения.
- Укажите имя импорта функции в **имя импорта функции** поля.
- Выберите **сложных** как возвращаемый тип, а затем укажите конкретный сложный возвращаемый тип, выбрав соответствующий тип из раскрывающегося списка.



- Нажмите кнопку **OK**. В концептуальной модели создается запись импорта функции.

Настроить сопоставление для импорта функции столбцов

- Щелкните правой кнопкой мыши импорт функции в браузере моделей и выберите **сопоставление импорта функций. Сведения о сопоставлении** окно отобразится сопоставление по умолчанию для импорта функции. Стрелки указывают сопоставления между значениями столбцов и значениями свойств. По умолчанию предполагается, что имена столбцов совпадают с именами свойств сложного типа. Имена столбцов по умолчанию отображаются серым текстом.
- Если необходимо, измените имена столбцов таким образом, чтобы они совпадали с именами столбцов, возвращаемых хранимой процедурой, соответствующей импорту функции.

Удаление сложного типа

При удалении сложного типа он удаляется из концептуальной модели. Кроме того, удаляются сопоставления для всех экземпляров типа. Однако ссылки на тип не обновляются. Например, если сущность имеет свойство сложного типа типа ComplexType1, и ComplexType1 был удален в **браузер моделей**, то соответствующее свойство сущности не обновляется. Модель не пройдет проверку, так как он содержит сущность, которая ссылается на удаленный сложный тип. Обновить или удалить ссылки на удаленные сложные типы можно с помощью конструктора сущностей.

- Щелкните правой кнопкой мыши сложный тип в браузере моделей и выберите **удалить**.
- OR-
- В браузере моделей выберите сложный тип и нажмите на клавиатуре клавишу DELETE.

Запрос для сущностей, содержащий свойства сложного типа

Ниже показано, как выполнить запрос, возвращающий коллекцию сущностей типа объектов, содержащих свойства сложного типа.

```
using (SchoolEntities context = new SchoolEntities())
{
    var courses =
        from c in context.OnsiteCourses
        order by c.Details.Time
        select c;

    foreach (var c in courses)
    {
        Console.WriteLine("Time: " + c.Details.Time);
        Console.WriteLine("Days: " + c.Details.Days);
        Console.WriteLine("Location: " + c.Details.Location);
    }
}
```

Поддержка перечислений — конструктор EF

27.09.2018 • 10 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Это видео и пошаговое руководство показано, как использовать типы перечисления с Entity Framework Designer. Также демонстрируется использование перечислений в запросе LINQ.

В этом пошаговом руководстве будет использоваться для создания новой базы данных Model First, но конструктор EF может также использоваться с [Database First](#) рабочий процесс для сопоставления существующей базы данных.

В Entity Framework 5 добавлена поддержка перечисления. Чтобы использовать новые функции, например перечисления, Пространственные типы данных и функции, возвращающие табличные значения, необходимо ориентироваться .NET Framework 4.5. Visual Studio 2012 предназначенного для .NET 4.5 по умолчанию.

В Entity Framework, что перечисление может иметь следующие базовые типы: **байтов**, **Int16**, **Int32**, **Int64**, или **SByte**.

Просмотреть видео

В этом видео показано, как использовать типы перечисления с Entity Framework Designer. Также демонстрируется использование перечислений в запросе LINQ.

Представленный: Юлия Корнич

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Необходимо будет установлен выпуск Visual Studio 2012 Ultimate, Premium, Professional и Web Express для выполнения этого пошагового руководства.

Настройка проекта

1. Откройте Visual Studio 2012
2. На **файл** последовательно выберите пункты **New**, а затем нажмите кнопку **проекта**
3. В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона
4. Введите **EnumEFDesigner** как имя проекта и нажмите кнопку **OK**

Создание новой модели в конструкторе EF

1. Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**
2. Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов
3. Введите **EnumTestModel.edmx** имя файла, а затем нажмите кнопку **добавить**
4. На странице мастера моделей EDM, выберите **пустую модель** в диалоговом окне Выбор содержимого

модели

5. Нажмите кнопку **Готово**

Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается.

Мастер выполняет следующие действия.

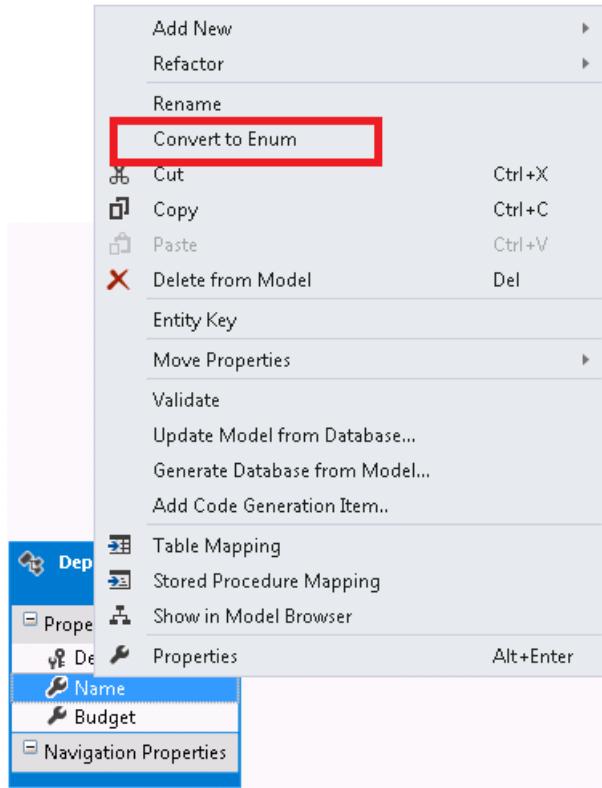
- Создает файл EnumTestModel.edmx, который определяет концептуальную модель, модель хранения и сопоставления между ними. Задает свойство обработка артефактов метаданных из EDMX-файл для внедрения в выходной сборки, поэтому созданные метаданные файлы внедряются в сборку.
- Добавляет ссылку на следующие сборки: EntityFramework, System.ComponentModel.DataAnnotations и System.Data.Entity.
- Создает файлы EnumTestModel.tt и EnumTestModel.Context.tt и добавляет их в EDMX-файл. Эти файлы шаблонов T4 создавать код, который определяет тип производным DbContext и типов РОСО, которые сопоставляются с сущностями в модели .edmx.

Добавить новый тип сущности

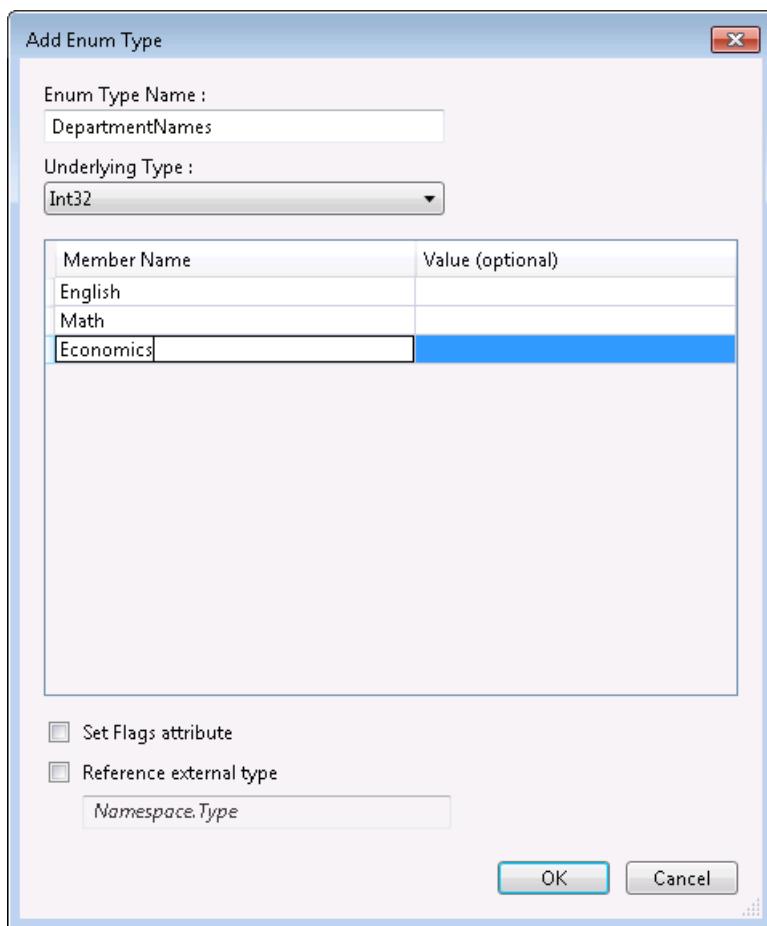
1. Щелкните правой кнопкой мыши пустой участок области конструктора, выберите **Add -> сущности**, появится диалоговое окно новой сущности
2. Укажите **отдел** для типа имя и указать **DepartmentID** оставьте тот тип, что имя ключевого свойства **Int32**
3. Нажмите кнопку **OK**.
4. Щелкните правой кнопкой мыши сущность и выберите **Add New -> скалярное свойство**
5. Переименовать новое свойство **имя**
6. Измените тип нового свойства для **Int32** (по умолчанию свойство имеет строковый тип) чтобы изменить тип, откройте окно свойств и измените свойство Type для **Int32**
7. Добавить еще одно скалярное свойство и переименуйте его в **бюджета**, измените тип на **Decimal**

Добавьте тип перечисления

1. В конструкторе Entity Framework, щелкните правой кнопкой мыши свойство Name, выберите **преобразовать перечисление**



2. В **перечислить** тип диалогового окна **DepartmentNames** для именем типа перечисления, изменить базовый тип для **Int32**, а затем добавьте следующие элементы в тип: английского языка, Математические и экономики



3. Нажмите клавишу **OK**

4. Сохранение модели и построение проекта

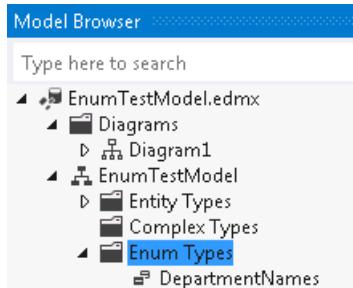
NOTE

При создании, предупреждения о несопоставленных сущностях и ассоциациях, появляются в списке ошибок. Эти предупреждения можно игнорировать, поскольку после мы решили создать базу данных из модели, пропадут.

Если взглянуть на окно «Свойства», можно заметить, что тип свойства имя было изменено на

DepartmentNames и новые перечисляемого типа была добавлена в список типов.

При переключении в окно браузера модели, вы увидите сообщение о том, что тип также был добавлен к узлу для перечислимых типов.



NOTE

Можно также добавить новые типы перечисления из этого окна, щелкнув правой кнопкой мыши и выбрав **добавить тип перечисления**. После создания типа, он будет отображаться в списке типов, и вы смогли бы связать со свойством

Создание базы данных на основе модели

Теперь мы сможем создать базу данных, основанный на модели.

1. Щелкните правой кнопкой мыши пустое место в области конструктора сущностей и выберите пункт **создать базу данных из модели**
2. Диалогового окна соединения данных из мастера создания базы данных является выбор отображаемых щелкните **новое подключение** кнопку укажите **(localdb)\mssqllocaldb** для имени сервера и **EnumTest** базы данных и нажмите кнопку **OK**
3. Диалоговое окно с запросом, если вы хотите создать новую базу данных будет всплывающее окно, нажмите кнопку **Да**.
4. Нажмите кнопку **Далее** и мастер создания базы данных приводит к возникновению ошибки языка описания данных (DDL) для создания базы данных, сформированный код DDL отображается сводка и параметры диалогового окна поле Обратите внимание, что, DDL не содержит определение для таблицу, которая сопоставляется с типом перечисления
5. Нажмите кнопку **Готово** щелкнув Готово не выполняет скрипта DDL.
6. Мастер создания базы данных выполняет следующие: открывает **EnumTest.edmx.sql** в редакторе T-SQL формирует разделы схемы и сопоставления хранилища EDMX-файла файл строку подключения Adds в файл App.config
7. Щелкните правой кнопкой мыши в редакторе T-SQL и выберите **Execute The Server** диалоговое окно подключения к откроетсяся, введите сведения о соединении из шага 2 и нажмите кнопку **Connect**
8. Чтобы просмотреть созданную схему, щелкните правой кнопкой мыши имя базы данных в обозревателе объектов SQL Server и выберите **обновления**

Сохранения и извлечения данных

Откройте файл Program.cs, в котором определен метод Main. Добавьте следующий код в функцию Main.

Код добавляет новый объект отдела в контекст. Затем она сохраняет данные. Код также выполняется запрос LINQ, возвращающий подразделение, где имя задается DepartmentNames.English.

```
using (var context = new EnumTestModelContainer())
{
    context.Departments.Add(new Department{ Name = DepartmentNames.English });

    context.SaveChanges();

    var department = (from d in context.Departments
                      where d.Name == DepartmentNames.English
                      select d).FirstOrDefault();

    Console.WriteLine(
        "DepartmentID: {0} and Name: {1}",
        department.DepartmentID,
        department.Name);
}
```

Скомпилируйте и запустите приложение. Программа выдаст следующие результаты.

```
DepartmentID: 1 Name: English
```

Чтобы просмотреть данные в базе данных, щелкните правой кнопкой мыши имя базы данных в обозревателе объектов SQL Server и выберите **обновить**. Щелкните правой кнопкой мыши таблицу и выберите **данные представления**.

Сводка

В этом пошаговом руководстве мы рассмотрели сведения о сопоставлении типов перечисления, с помощью Entity Framework Designer и как использовать перечисления в коде.

Пространственные – конструктор EF

27.09.2018 • 9 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Видео и пошаговые руководства показано, как сопоставить пространственных типов с помощью Entity Framework Designer. Также показано, как использовать LINQ-запрос для поиска расстояние между двумя расположениями.

В этом пошаговом руководстве будет использоваться для создания новой базы данных Model First, но конструктор EF может также использоваться с Database First рабочий процесс для сопоставления существующей базы данных.

В Entity Framework 5 появилась поддержка пространственных типов. Обратите внимание на то, что чтобы использовать новые функции, например пространственного типа, перечислимые типы и функции, возвращающие табличные значения, необходимо ориентироваться .NET Framework 4.5. Visual Studio 2012 предназначенног для .NET 4.5 по умолчанию.

Для использования пространственных типов данных необходимо также использовать с поставщиком Entity Framework с поддержкой пространственных. См. в разделе [поддержка пространственных типов](#) Дополнительные сведения.

Существует два типа основных пространственных данных: geography и geometry. Тип данных сохраняет эллиптические данные (например, GPS координаты широты и долготы). Тип данных geometry представляет Евклидовой (плоской) системе координат.

Просмотреть видео

В этом видео показано, как сопоставить пространственных типов с помощью Entity Framework Designer. Также показано, как использовать LINQ-запрос для поиска расстояние между двумя расположениями.

Представленный: Юлия Корнич

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Необходимо будет установлен выпуск Visual Studio 2012 Ultimate, Premium, Professional и Web Express для выполнения этого пошагового руководства.

Настройка проекта

1. Откройте Visual Studio 2012
2. На **файл** последовательно выберите пункты **New**, а затем нажмите кнопку **проекта**
3. В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона
4. Введите **SpatialEFDesigner** как имя проекта и нажмите кнопку **OK**

Создание новой модели в конструкторе EF

1. Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**
2. Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов
3. Введите **UniversityModel.edmx** имя файла, а затем нажмите кнопку **добавить**
4. На странице мастера моделей EDM, выберите **пустую модель** в диалоговом окне Выбор содержимого модели
5. Нажмите кнопку **Готово**

Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается.

Мастер выполняет следующие действия.

- Создает файл EnumTestModel.edmx, который определяет концептуальную модель, модель хранения и сопоставления между ними. Задает свойство обработка артефактов метаданных из EDMX-файл для внедрения в выходной сборки, поэтому созданные метаданные файлы внедряются в сборку.
- Добавляет ссылку на следующие сборки: EntityFramework, System.ComponentModel.DataAnnotations и System.Data.Entity.
- Создает файлы UniversityModel.tt и UniversityModel.Context.tt и добавляет их в EDMX-файл. Эти файлы шаблонов T4 создания кода, который определяет тип производным DbContext и типов РОСО, которые сопоставляются с сущностями в модели .edmx

Добавить новый тип сущности

1. Щелкните правой кнопкой мыши пустой участок области конструктора, выберите **Add -> сущности**, появится диалоговое окно новой сущности
2. Укажите **университета** для типа имени и укажите **UniversityID** оставьте тот тип, что имя ключевого свойства **Int32**
3. Нажмите кнопку **OK**.
4. Щелкните правой кнопкой мыши сущность и выберите **Add New -> скалярное свойство**
5. Переименовать новое свойство **имя**
6. Добавить еще одно скалярное свойство и переименуйте его в **расположение** откройте окно свойств и измените тип нового свойства для **Geography**
7. Сохранение модели и построение проекта

NOTE

При создании, предупреждения о несопоставленных сущностях и ассоциациях, появляются в списке ошибок. Эти предупреждения можно игнорировать, поскольку после мы решили создать базу данных из модели, пропадут.

Создание базы данных на основе модели

Теперь мы сможем создать базу данных, основанный на модели.

1. Щелкните правой кнопкой мыши пустое место в области конструктора сущностей и выберите пункт **создать базу данных из модели**
2. Диалогового окна соединения данных из мастера создания базы данных является выбор отображаемых щелкните **новое подключение** кнопку укажите **(localdb)\mssqllocaldb** для имени сервера и **Университет** базы данных и нажмите кнопку **OK**

3. Диалоговое окно с запросом, если вы хотите создать новую базу данных будет всплывающее окно, нажмите кнопку **Да**.
4. Нажмите кнопку **Далее** и мастер создания базы данных приводит к возникновению ошибки языка описания данных (DDL) для создания базы данных, сформированный код DDL отображается сводка и параметры диалогового окна поле Обратите внимание, что, DDL не содержит определение для таблицы, которая сопоставляется с типом перечисления
5. Нажмите кнопку **Готово** щелкнув Готово не выполняет скрипт DDL.
6. Мастер создания базы данных выполняет следующие: открывает **UniversityModel.edmx.sql** в редакторе T-SQL формирует разделы схемы и сопоставления хранилища EDMX-файла файл строку подключения Adds в файл App.config
7. Щелкните правой кнопкой мыши в редакторе T-SQL и выберите **Execute The Server** диалоговое окно подключения к откроется, введите сведения о соединении из шага 2 и нажмите кнопку **Connect**
8. Чтобы просмотреть созданную схему, щелкните правой кнопкой мыши имя базы данных в обозревателе объектов SQL Server и выберите **обновления**

Сохранения и извлечения данных

Откройте файл Program.cs, в котором определен метод Main. Добавьте следующий код в функцию Main.

Код добавляет два новых университета объектов к контексту. Пространственные свойства инициализируются с помощью метода DbGeography.FromText. Географическая точка представить в виде WellKnownText передается в метод. Затем код сохраняет данные. Затем запрос LINQ, который возвращает объект университета, где она будет находиться в указанное расположение, ближайшее составлением и выполнением.

```
using (var context = new UniversityModelContainer())
{
    context.Universities.Add(new University()
    {
        Name = "Graphic Design Institute",
        Location = DbGeography.FromText("POINT(-122.336106 47.605049)"),
    });

    context.Universities.Add(new University()
    {
        Name = "School of Fine Art",
        Location = DbGeography.FromText("POINT(-122.335197 47.646711)"),
    });

    context.SaveChanges();

    var myLocation = DbGeography.FromText("POINT(-122.296623 47.640405)");

    var university = (from u in context.Universities
                      orderby u.Location.Distance(myLocation)
                      select u).FirstOrDefault();

    Console.WriteLine(
        "The closest University to you is: {0}.",
        university.Name);
}
```

Скомпилируйте и запустите приложение. Программа выдает следующие результаты.

```
The closest University to you is: School of Fine Art.
```

Чтобы просмотреть данные в базе данных, щелкните правой кнопкой мыши имя базы данных в

обозревателе объектов SQL Server и выберите **обновить**. Щелкните правой кнопкой мыши таблицу и выберите **данные представления**.

Сводка

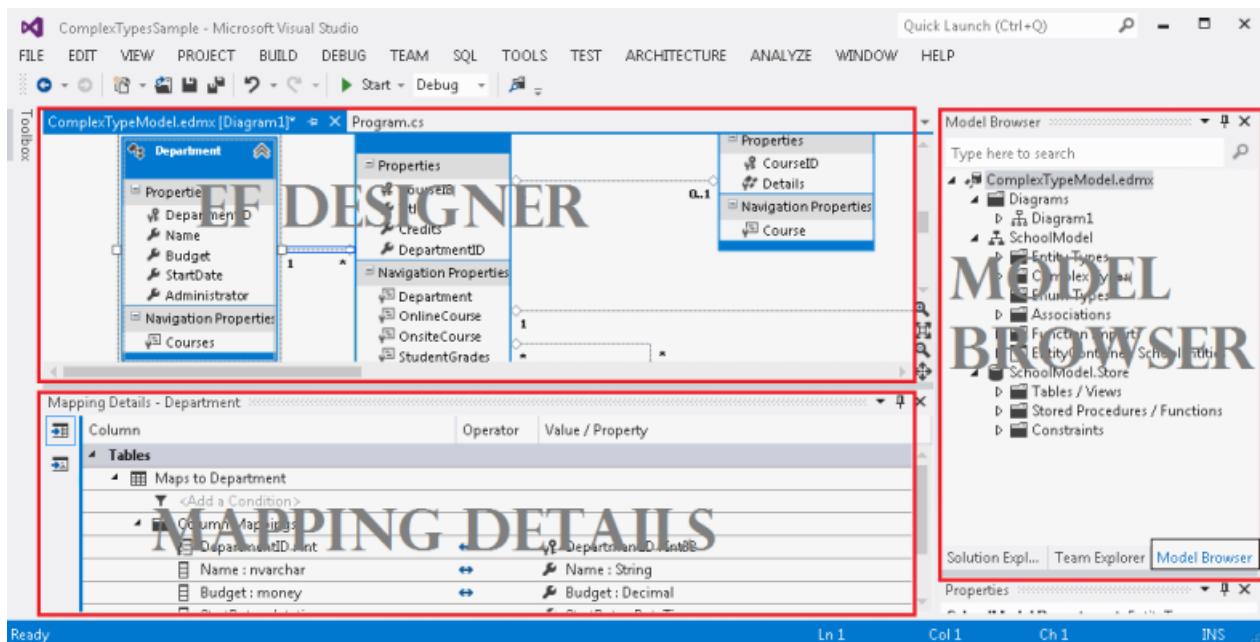
В этом пошаговом руководстве мы рассмотрели способ сопоставления пространственных типов, с помощью Entity Framework Designer и использовании пространственных типов в коде.

Разбиение конструктора сущностей

13.09.2018 • 7 minutes to read • [Edit Online](#)

В этом пошаговом руководстве показано, как сопоставить тип сущности с двумя таблицами путем изменения модели с Entity Framework Designer (конструктор EF). Сущность можно сопоставить с несколькими таблицами, если в таблицах имеется общий ключ. Основные понятия, которые применяются для сопоставления типа сущности с двумя таблицами, легко распространить на сопоставление типа сущности более чем с двумя таблицами.

На следующем рисунке показана основные окна, которые используются при работе с конструктором EF.



Предварительные требования

Visual Studio 2012 или Visual Studio 2010 Ultimate, Premium, Professional и Web Express edition.

Создание базы данных

Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, вы установили:

- Если вы используете Visual Studio 2012, а затем вы создадите базу данных LocalDB.
- Если вы используете Visual Studio 2010 вы создадите базу данных SQL Express.

Сначала мы создадим базу данных с двумя таблицами, которые мы хотим объединить в одну сущность.

- Открытие Visual Studio
- **Представление —> обозревателя серверов**
- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**
- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам нужно будет выбрать **Microsoft SQL Server** как источник данных
- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них установки
- Введите **EntitySplitting** имя базы данных
- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**

- Новая база данных появится в обозревателе серверов
- Если вы используете Visual Studio 2012
 - Щелкните правой кнопкой мыши, в базе данных в обозревателе серверов и выберите **новый запрос**
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**
- Если вы используете Visual Studio 2010
 - Выберите **тарифный план** —> **Transact редактор SQL** -> **новое соединение запроса...**
 - Введите **.\\SQLEXPRESS** имя сервера и нажмите кнопку **OK**
 - Выберите **EntitySplitting** базы данных в раскрывающемся вниз в верхней части редактора запросов
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Выполнение SQL**

```

CREATE TABLE [dbo].[Person] (
[PersonId] INT IDENTITY (1, 1) NOT NULL,
[FirstName] NVARCHAR (200) NULL,
[LastName] NVARCHAR (200) NULL,
CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED ([PersonId] ASC)
);

CREATE TABLE [dbo].[PersonInfo] (
[PersonId] INT NOT NULL,
[Email] NVARCHAR (200) NULL,
[Phone] NVARCHAR (50) NULL,
CONSTRAINT [PK_PersonInfo] PRIMARY KEY CLUSTERED ([PersonId] ASC),
CONSTRAINT [FK_Person_PersonInfo] FOREIGN KEY ([PersonId]) REFERENCES [dbo].[Person] ([PersonId]) ON DELETE CASCADE
);

```

Создание проекта

- В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**.
- В левой области щелкните **Visual C#**, а затем выберите **консольное приложение** шаблона.
- Введите **MapEntityToTablesSample** как имя проекта и нажмите кнопку **OK**.
- Нажмите кнопку **нет** Если будет предложено сохранить запрос SQL, созданный в первом разделе.

Создать модель на основе базы данных

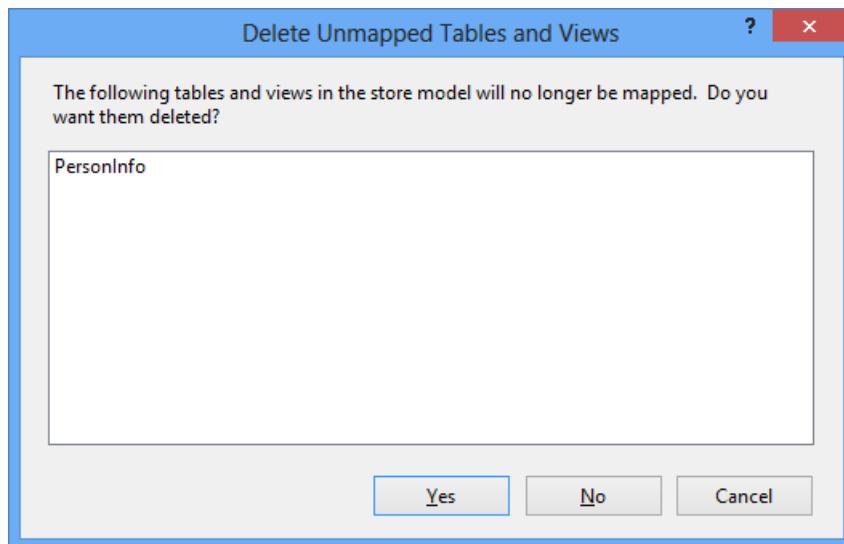
- Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **MapEntityToTablesModel.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.
- Выберите **EntitySplitting** подключения из раскрывающегося списка и нажмите кнопку **Далее**.
- В диалоговом окне Выбор объектов базы данных, установите флажок рядом с полем **таблиц** узла. Это добавит все таблицы из **EntitySplitting** базы данных в модель.
- Нажмите кнопку **Готово**.

Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается.

Сопоставить сущности с двумя таблицами

На этом шаге мы обновим **Person** тип сущности для объединения данных из **Person** и **PersonInfo** таблиц.

- Выберите **электронной почты** и **Phone** свойства ** PersonInfo ** сущности и нажмите клавишу **Ctrl + X** клавиши.
- Выберите ** Person ** сущности и нажмите клавишу **Ctrl + V** клавиши.
- В рабочей области конструирования выберите **PersonInfo** сущности и нажмите клавишу **удалить** кнопку на клавиатуре.
- Нажмите кнопку **нет** при запросе, если вы хотите удалить **PersonInfo** таблицы из модели, мы сейчас будет сопоставить с **Person** сущности.



Для следующего шага требуется **сведения о сопоставлении** окна. Если это окно не отображается, щелкните правой кнопкой мыши область конструктора и выберите **сведения о сопоставлении**.

- Выберите **Person** типа сущности и нажмите кнопку **<добавить таблицу или представление>** в **сведения о сопоставлении** окна.
- Выберите ** PersonInfo ** из раскрывающегося списка. **Сведения о сопоставлении** окно обновляется с помощью сопоставления столбцов по умолчанию, но это нормально для нашего сценария.

Person типа сущности теперь сопоставлена с **Person** и **PersonInfo** таблиц.

A screenshot of the "Mapping Details - Person" dialog box. On the left is a tree view showing "Tables" expanded, with "Maps to Person" and "Maps to PersonInfo" selected. Under "Maps to Person", there is a "Column Mappings" section with mappings for PersonId (int) to PersonId (Int32), FirstName (nvarchar) to FirstName (String), and LastName (nvarchar) to LastName (String). Under "Maps to PersonInfo", there is another "Column Mappings" section with mappings for PersonId (int) to PersonId (Int32), Email (nvarchar) to Email (String), and Phone (nvarchar) to Phone (String). A blue horizontal bar highlights the "Maps to PersonInfo" section.

Использование модели

- Вставьте следующий код в методе Main.

```

using (var context = new EntitySplittingEntities())
{
    var person = new Person
    {
        FirstName = "John",
        LastName = "Doe",
        Email = "john@example.com",
        Phone = "555-555-5555"
    };

    context.People.Add(person);
    context.SaveChanges();

    foreach (var item in context.People)
    {
        Console.WriteLine(item.FirstName);
    }
}

```

- Скомпилируйте и запустите приложение.

Следующие инструкции T-SQL были выполнены в базе данных в результате выполнения этого приложения.

- Следующие два **вставить** инструкции были выполнены в результате выполнения контекста. SaveChanges(). Они принимают данные из **Person** сущности и разделения между **Person** и **PersonInfo** таблиц.

⚡ ADO.NET: Execute Reader "insert [dbo].[Person]([FirstName], [L
The command text "insert [dbo].[Person]([FirstName],
[LastName])
values (@0, @1)
select [PersonId]
from [dbo].[Person]
where @@ROWCOUNT > 0 and [PersonId] = scope_identity()
was executed on connection "data source=(localdb)
\v11.0;initial catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", building a SqlDataReader.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

⚡ ADO.NET: Execute NonQuery "insert [dbo].[PersonInfo]([Person
The command text "insert [dbo].[PersonInfo]([PersonId],
[Email], [Phone])
values (@0, @1, @2)
" was executed on connection "data source=(localdb)
\v11.0;initial catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", returning the number of rows affected.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

- Следующие **выберите** был выполнен в результате перечисление людей в базе данных. Он объединяет данные из **Person** и **PersonInfo** таблицы.

⚡ ADO.NET: Execute Reader "SELECT [Extent1].[PersonId] AS [Per
The command text "SELECT
[Extent1].[PersonId] AS [PersonId],
[Extent2].[FirstName] AS [FirstName],
[Extent2].[LastName] AS [LastName],
[Extent1].[Email] AS [Email],
[Extent1].[Phone] AS [Phone]
FROM [dbo].[PersonInfo] AS [Extent1]
INNER JOIN [dbo].[Person] AS [Extent2] ON [Extent1].[PersonId] = [Extent2].[PersonId]" was executed on connection
"data source=(localdb)\v11.0;initial
catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", building a SqlDataReader.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

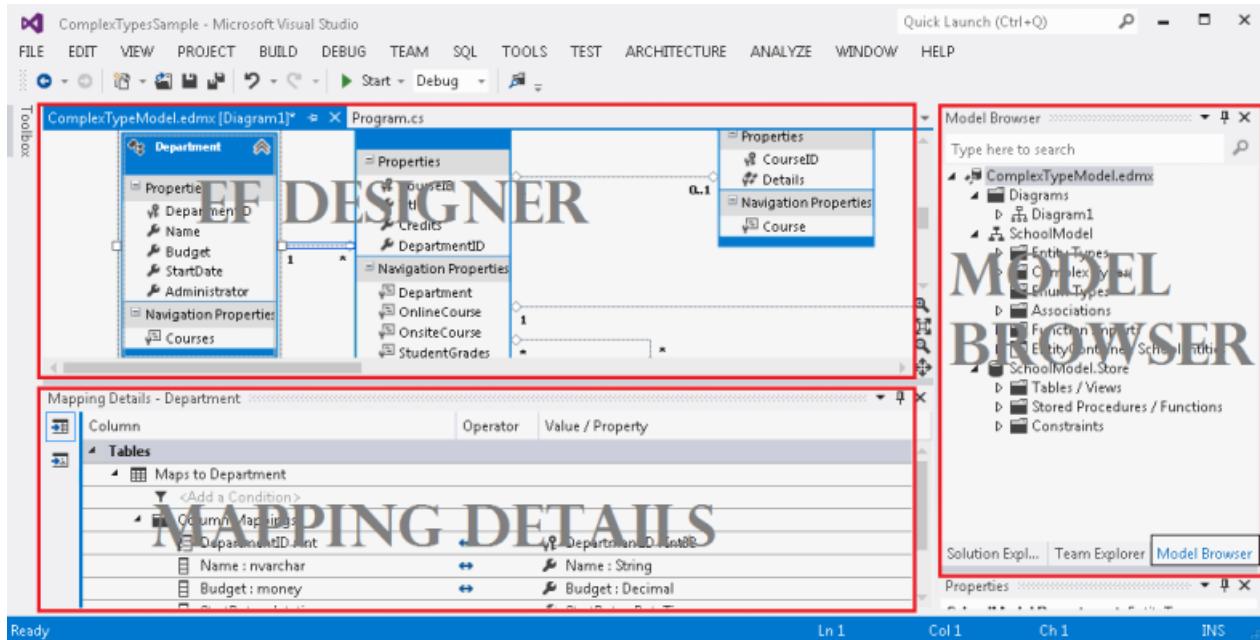
Разбиение конструктора таблиц

13.09.2018 • 7 minutes to read • [Edit Online](#)

В этом пошаговом руководстве показано, как сопоставить несколько типов сущностей с одной таблицей путем изменения модели с Entity Framework Designer (конструктор EF).

Одна из причин, которые можно использовать разбиение таблиц задерживает загрузки некоторых свойств при использовании отложенной загрузки для загрузки объектов. Свойства, которые могут содержать очень большой объем данных в отдельную сущность и загружать только его при необходимости можно разделить.

На следующем рисунке показана основные окна, которые используются при работе с конструктором EF.



Предварительные требования

Для выполнения данного пошагового руководства требуется:

- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

В этом пошаговом руководстве используется Visual Studio 2012.

- Откройте Visual Studio 2012.
- В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**.
- В левой области выберите Visual C#, а затем выберите шаблон консольного приложения.
- Введите **TableSplittingSample** как имя проекта и нажмите кнопку **OK**.

Создать модель, основанную на базе данных School

- Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.

- Введите **TableSplittingModel.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.
- Щелкните новое подключение. В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**. В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В диалоговом окне Выбор объектов базы данных можно раскрыть **таблиц** узел и проверьте **Person** таблицы. Это добавит к указанной таблице **School** модели.
- Нажмите кнопку **Готово**.

Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается. Все объекты, которые выбраны в **Choose Your Database Objects** диалоговое окно добавляются в модель.

Сопоставить две сущности с одной таблицей

В этом разделе вы разделите **Person** сущность на две сущности и сопоставьте их с одной таблицей.

NOTE

Person сущность не содержит все свойства, которые могут содержать большой объем данных; он используется только в качестве примера.

- Щелкните правой кнопкой мыши пустой участок области конструктора, выберите пункт **Add New** и нажмите кнопку **сущности. Новую сущность** откроется диалоговое окно.
- Тип **HireInfo** для **имя сущности** и **PersonID** для **свойство Key** имя.
- Нажмите кнопку **OK**.
- Новый тип сущности будет создан и отобразится в области конструктора.
- Выберите **HireDate** свойство **Person** типа сущности и нажмите клавишу **Ctrl + X** клавиши.
- Выберите **HireInfo** сущности и нажмите клавишу **Ctrl + V** клавиши.
- Создание ассоциации между **Person** и **HireInfo**. Чтобы сделать это, щелкните правой кнопкой мыши пустой участок области конструктора, выберите пункт **Add New** и нажмите кнопку **ассоциации**.
- **Добавить сопоставление** откроется диалоговое окно. **PersonHireInfo** присвоено имя по умолчанию.
- Укажите кратность **1(One)** на обеих сторонах связи.
- Нажмите клавишу **OK**.

Для следующего шага требуется **сведения о сопоставлении** окна. Если это окно не отображается, щелкните правой кнопкой мыши область конструктора и выберите **сведения о сопоставлении**.

- Выберите **HireInfo** типа сущности и нажмите кнопку **<добавить таблицу или представление>** в **сведения о сопоставлении** окна.
- Выберите **Person** из **<добавить таблицу или представление>** поле с раскрывающимся списком. Список содержит таблицы или представления, который может быть сопоставлен выбранной сущности. Необходимо сопоставить соответствующие свойства по умолчанию.

Column	Ope...	Value / Property
PersonID : int	\leftrightarrow	PersonID : Int32
LastName : nvarchar	\leftrightarrow	
FirstName : nvarchar	\leftrightarrow	
HireDate : datetime	\leftrightarrow	HireDate : DateTime
EnrollmentDate : datetime	\leftrightarrow	
Discriminator : nvarchar	\leftrightarrow	

- Выберите **PersonHireInfo** ассоциации в рабочей области конструирования.
- Щелкните правой кнопкой мыши связь на область конструктора и выберите **свойства**.
- В **свойства** выберите **ссылочные ограничения** свойство и нажмите кнопку с многоточием.
- Выберите **Person** из **участника** стрелку раскрывающегося списка.
- Нажмите клавишу **OK**.

Использование модели

- Вставьте следующий код в методе Main.

```
using (var context = new SchoolEntities())
{
    Person person = new Person()
    {
        FirstName = "Kimberly",
        LastName = "Morgan",
        Discriminator = "Instructor",
    };

    person.HireInfo = new HireInfo()
    {
        HireDate = DateTime.Now
    };

    // Add the new person to the context.
    context.People.Add(person);

    // Insert a row into the Person table.
    context.SaveChanges();

    // Execute a query against the Person table.
    // The query returns columns that map to the Person entity.
    var existingPerson = context.People.FirstOrDefault();

    // Execute a query against the Person table.
    // The query returns columns that map to the Instructor entity.
    var hireInfo = existingPerson.HireInfo;

    Console.WriteLine("{0} was hired on {1}",
        existingPerson.LastName, hireInfo.HireDate);
}
```

- Скомпилируйте и запустите приложение.

Следующие инструкции T-SQL были выполнены с **School** базы данных в результате запуска этого приложения.

- Следующие **вставить** был выполнен в результате выполнения контекста SaveChanges() и объединяет данные из **Person** и **HireInfo** сущностей

```
⚡ ADO.NET: Execute Reader "insert [dbo].[Person]([LastName], [FirstName], [HireDate], [EnrollmentDate], [Discriminator]) values (@0, @1, @2, null, @3) select [PersonID] from [dbo].[Person] where @@ROWCOUNT > 0 and [PersonID] = scope_identity()" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.
```

- Следующие **выберите** был выполнен в результате выполнения контекста People.FirstOrDefault() и выбирает только столбцы сопоставляются **Person**

```
⚡ ADO.NET: Execute Reader "SELECT TOP (1) [c].[PersonID], [c].[LastName] AS [LastName], [c].[FirstName] AS [FirstName], [c].[EnrollmentDate] AS [EnrollmentDate], [c].[Discriminator] AS [Discriminator] FROM [dbo].[Person] AS [c]" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.
```

- Следующие **выберите** была выполнена из-за доступа к existingPerson.Instructor свойство навигации и выбирает только нужные столбцы, которые сопоставляются с **HireInfo**

```
⚡ ADO.NET: Execute Reader "SELECT [Extent1].[PersonID], [Extent1].[HireDate] AS [HireDate] FROM [dbo].[Person] AS [Extent1] WHERE [Extent1].[PersonID] = @EntityKeyValue1" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.
```

Конструктора МОДЕЛИ наследования

13.09.2018 • 10 minutes to read • [Edit Online](#)

Это пошаговое руководство показывает, как реализовать наследование таблица на иерархию (ТРН) в концептуальной модели с помощью Entity Framework Designer (конструктор EF). Наследование ТРН использует одну таблицу базы данных для сопровождения данных всех типов сущностей в иерархии наследования.

В этом пошаговом руководстве мы сопоставим таблицы Person с тремя типами сущностей: человек (базовый тип), Student (является производным от пользователя) и Instructor (наследуется от Person). Мы создадим концептуальной модели из базы данных (Database First) и затем изменить модель для реализации МОДЕЛИ наследования, в конструкторе EF.

Можно сопоставить МОДЕЛИ наследования, с помощью Model First, но вам пришлось бы писать собственные рабочего процесса создания базы данных, который является сложным. Можно затем назначить этот рабочий процесс для **рабочего процесса создания базы данных** свойства в конструкторе EF. Простой альтернативой является использование Code First.

Другие параметры наследования

Одна таблица на тип (ТРТ) – еще один тип наследования, в котором отдельные таблицы в базе данных, сопоставляются сущности, участвующие в наследовании. Сведения о сопоставлении таблица на тип наследования в конструкторе EF, см. в разделе [наследование ТРТ конструктор EF](#).

Таблица на конкретный тип наследования (ТРС) и модели смешанного наследования поддерживаются средой выполнения Entity Framework, но не поддерживаются в конструкторе EF. Если вы хотите использовать ТРС или смешанных наследования, можно двумя способами: использовать Code First, или вручную изменить EDMX-файла. Если вы предпочтете работать с файлом EDMX, окно сведений о сопоставлении будет помещен в «безопасный режим», и вы не сможете использовать конструктор для изменения сопоставлений.

Предварительные требования

Для выполнения данного пошагового руководства требуется:

- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

- Откройте Visual Studio 2012.
- Выберите **файл -> Новинка —> проекта**
- В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона.
- Введите **TPHDBFirstSample** как имя.
- Нажмите кнопку **OK**.

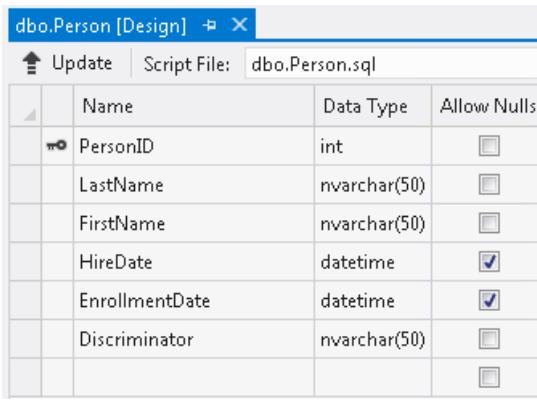
Создание модели

- Щелкните правой кнопкой мыши имя проекта в обозревателе решений и выберите **Add -> новый элемент**.

- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **TPHModel.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.
- Нажмите кнопку **новое подключение**. В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**. В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В диалоговом окне Выбор объектов базы данных в узле таблицы выберите **Person** таблицы.
- Нажмите кнопку **Готово**.

Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается. Все объекты, которые выбраны в диалоговом окне Выбор объектов базы данных добавляются в модель.

То есть как **Person** таблица выглядит в базе данных.



The screenshot shows the EntityDataSource configuration window for the 'Person' table. At the top, there's a toolbar with 'Update', 'Script File:', and 'dbo.Person.sql'. Below it is a table with columns: Name, Data Type, and Allow Nulls. The table rows are:

	Name	Data Type	Allow Nulls
PersonID	int	<input type="checkbox"/>	
LastName	nvarchar(50)	<input type="checkbox"/>	
FirstName	nvarchar(50)	<input type="checkbox"/>	
HireDate	datetime	<input checked="" type="checkbox"/>	
EnrollmentDate	datetime	<input checked="" type="checkbox"/>	
Discriminator	nvarchar(50)	<input type="checkbox"/>	
		<input type="checkbox"/>	

Реализация наследования таблица на иерархию

Person таблица имеет **дискриминатора** столбец, который может иметь одно из двух значений: «Student» и «Instructor». В зависимости от значения **Person** таблицы будут сопоставлены с **учащихся** сущности или **преподавателя** сущности. **Person** таблица также содержит два столбца, **HireDate** и **EnrollmentDate**, который должен быть **допускает значения NULL** так, как пользователь не может быть Student и instructor, в то же время (по крайней мере не в этом пошаговом руководстве).

Добавление новых сущностей

- Добавьте новую сущность. Чтобы сделать это, щелкните правой кнопкой мыши на пустую часть области конструктора в конструкторе Entity Framework и выберите **Add ->сущности**.
- Тип **преподавателя** для **имя сущности** и выберите **Person** из раскрывающегося списка для **базовый тип**.
- Нажмите кнопку **OK**.
- Добавьте другой новой сущности. Тип **учащихся** для **имя сущности** и выберите **Person** из раскрывающегося списка для **базовый тип**.

Были добавлены два новых типа сущности в область конструктора. Стрелка идет от новых типов сущности к **Person** типа сущности; это означает, что **Person** является базовым типом для новых типов сущности.

- Щелкните правой кнопкой мыши **HireDate** свойство **Person** сущности. Выберите **Вырезать** (или с помощью клавиши Ctrl + X).
- Щелкните правой кнопкой мыши **преподавателя** сущности и выберите **вставить** (или с помощью клавиши Ctrl + V).
- Щелкните правой кнопкой мыши **HireDate** свойство и выберите **свойства**.

- В **свойства** окне **Nullable** свойства **false**.
- Щелкните правой кнопкой мыши **EnrollmentDate** свойство **Person** сущности. Выберите **Вырезать** (или с помощью клавиши **Ctrl + X**).
- Щелкните правой кнопкой мыши **учащихся** сущности и выберите **вставки** (или **раздела используйте Ctrl + V**).
- Выберите **EnrollmentDate** и установите **Nullable** свойства **false**.
- Выберите **Person** типа сущности. В **свойства** окна, задайте его **абстрактный** свойства **true**.
- Удалить **дискриминатора** свойства из **Person**. В следующем разделе объясняется причина, по которой ее следует удалить.

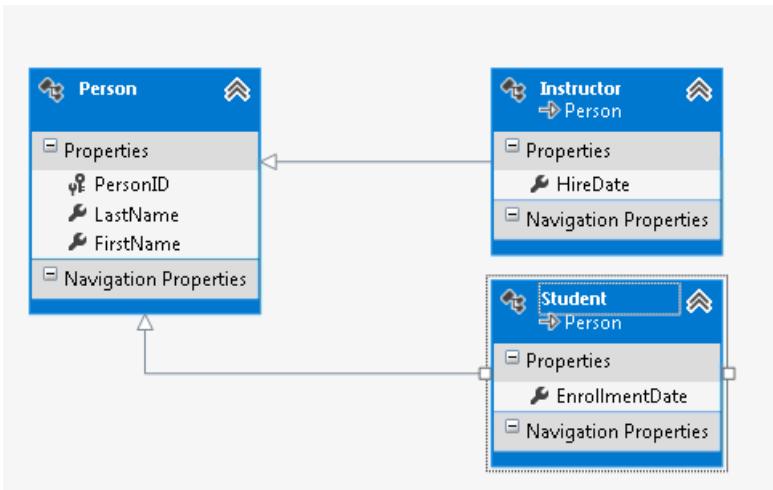
Сопоставьте сущности

- Щелкните правой кнопкой мыши **преподавателя** и выберите **сопоставление таблицы**. Сущности **Instructor** выбран в окне «сведения о сопоставлении».
- Нажмите кнопку **<добавить таблицу или представление>** в **сведения о сопоставлении** окна. **<Добавить таблицу или представление>** поле становится стрелку раскрывающегося списка таблиц и представлений, с которой могут быть сопоставлены выбранной сущности.
- Выберите **Person** из раскрывающегося списка.
- **Сведения о сопоставлении** окно обновляется с помощью сопоставления столбцов по умолчанию и параметр для добавления условия.
- Щелкните **<добавить условие>**. **<Добавить условие>** поле становится стрелку раскрывающегося списка столбцов, для которых можно задать условия.
- Выберите **дискриминатора** из раскрывающегося списка.
- В **оператор** столбец **сведения о сопоставлении** выберите **=** из раскрывающегося списка.
- В **значение/свойство** столбец, тип **преподавателя**. Конечный результат должен выглядеть следующим образом:

Column	Operator	Value / Property
Tables		
Maps to Person		
When Discriminator	=	Instructor
<Add a Condition>		
Column Mappings		
HireDate : datetime	↔	HireDate : DateTime
EnrollmentDate : datetime	↔	
Discriminator : nvarchar	↔	

- Повторите эти шаги для **учащихся** тип сущности, но создать условие равно **участников** значение. Причина, мы хотели бы удалить **дискриминатора** свойство, является, поскольку невозможно сопоставить столбец таблицы более одного раза. Этот столбец будет использоваться для условного сопоставления, поэтому его нельзя использовать для сопоставления свойства, а также. Единственным способом, он может использоваться в обоих случаях если условие использует **Is Null** или **Is Not Null** сравнения.

Теперь наследование типа «одна таблица на иерархию» успешно реализовано.



Использование модели

Откройте **Program.cs** файл, в котором **Main** определен метод. Вставьте следующий код в **Main** функции. Код выполняет три запроса. Первый запрос возвращает все **Person** объектов. Второй запрос использует **OfType** метод для возврата **преподавателя** объектов. Третий запрос использует **OfType** метод для возврата **учащихся** объектов.

```

using (var context = new SchoolEntities())
{
    Console.WriteLine("All people:");
    foreach (var person in context.People)
    {
        Console.WriteLine("    {0} {1}", person.FirstName, person.LastName);
    }

    Console.WriteLine("Instructors only: ");
    foreach (var person in context.People.OfType<Instructor>())
    {
        Console.WriteLine("    {0} {1}", person.FirstName, person.LastName);
    }

    Console.WriteLine("Students only: ");
    foreach (var person in context.People.OfType<Student>())
    {
        Console.WriteLine("    {0} {1}", person.FirstName, person.LastName);
    }
}

```

Наследование TPT конструктора

13.09.2018 • 6 minutes to read • [Edit Online](#)

Это пошаговое руководство показывает, как реализовать наследование таблицы на тип (TPT) в вашей модели, с помощью Entity Framework Designer (конструктор EF). В наследовании типа «одна таблица на тип» используется отдельная таблица в базе данных для сопровождения данных, относящихся к ненаследуемым свойствам и ключевым свойствам для каждого типа в иерархии наследования.

В этом пошаговом руководстве мы сопоставим **курс** (базовый тип), **OnlineCourse** (наследует от курса), и **OnsiteCourse** (является производным от **курс**) сущности для таблиц с одинаковыми именами. Мы создадим модель из базы данных и затем изменить модель для реализации наследования TPT.

Можно также начать работу с первой модели и затем создать базу данных из модели. Конструктор EF по умолчанию использует стратегию TPT и поэтому все наследования в модели будут сопоставлены с несколькими таблицами.

Другие параметры наследования

Одна таблица на иерархию (TPH) – еще один тип наследования, в которых одна база данных таблица используется для сопровождения данных всех типов сущностей в иерархии наследования. Сведения о сопоставлении таблицы на иерархию наследования с использованием конструктора сущностей см. в разделе [EF конструктор МОДЕЛИ наследования](#).

Обратите внимание, что таблица на конкретный тип наследования (TPC) и смешанных наследование моделей поддерживаются средой выполнения Entity Framework, но не поддерживаются в конструкторе EF. Если вы хотите использовать TPC или смешанных наследования, можно двумя способами: использовать **Code First**, или вручную изменить EDMX-файла. Если вы предпочтете работать с файлом EDMX, окно сведений о сопоставлении будет помещен в «безопасный режим», и вы не сможете использовать конструктор для изменения сопоставлений.

Предварительные требования

Для выполнения данного пошагового руководства требуется:

- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

- Откройте Visual Studio 2012.
- Выберите **файл -> Новинка —> проекта**
- В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона.
- Введите **TPTDBFirstSample** как имя.
- Нажмите кнопку **OK**.

Создание модели

- Щелкните правой кнопкой мыши проект в обозревателе решений и выберите **Add -> новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **TPTModel.edmx** имя файла, а затем нажмите кнопку **добавить**.

- В диалоговом окне Выбор содержимого модели, выберите ** Создать из базы данных ** и нажмите кнопку **Далее**.
- Нажмите кнопку **новое подключение**. В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**. В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В диалоговом окне Выбор объектов базы данных в узле таблицы выберите **отдел, курс, OnlineCourse и OnsiteCourse** таблиц.
- Нажмите кнопку **Готово**.

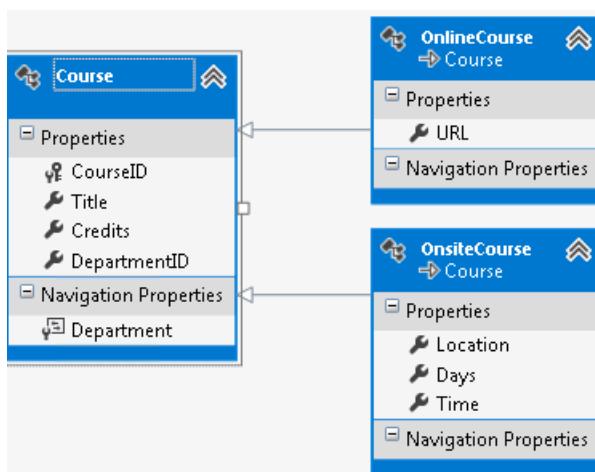
Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается. Все объекты, которые выбраны в диалоговом окне Выбор объектов базы данных добавляются в модель.

Реализация наследования таблица на тип

- В области конструктора щелкните правой кнопкой мыши **OnlineCourse** типа сущности и выберите **свойства**.
- В **свойства** окна, задайте для свойства Type базы **курс**.
- Щелкните правой кнопкой мыши **OnsiteCourse** типа сущности и выберите **свойства**.
- В **свойства** окна, задайте для свойства Type базы **курс**.
- Щелкните правой кнопкой мыши ассоциацию (линию) между **OnlineCourse** и **курс** типов сущностей. Выберите **удалить из модели**.
- Щелкните правой кнопкой мыши связь между **OnsiteCourse** и **курс** типов сущностей. Выберите **удалить из модели**.

Теперь удалим **CourseID** свойства из **OnlineCourse** и **OnsiteCourse** так, как эти классы наследуют **CourseID** из **курс** базовый тип.

- Щелкните правой кнопкой мыши **CourseID** свойство **OnlineCourse** тип сущности, а затем выберите **удалить из модели**.
- Щелкните правой кнопкой мыши **CourseID** свойство **OnsiteCourse** тип сущности, а затем выберите **удалить из модели**
- Теперь наследование типа «одна таблица на тип» реализовано.



Использование модели

Откройте **Program.cs** файл, в котором **Main** определен метод. Вставьте следующий код в **Main** функции. Код выполняет три запроса. Первый запрос возвращает все **курсы** связанные с указанное подразделение. Второй запрос использует **OfType** метод для возврата **OnlineCourses** связанные с указанное

подразделение. Возвращает третий запрос **OnsiteCourses**.

```
using (var context = new SchoolEntities())
{
    foreach (var department in context.Departments)
    {
        Console.WriteLine("The {0} department has the following courses:",
                          department.Name);

        Console.WriteLine("    All courses");
        foreach (var course in department.Courses )
        {
            Console.WriteLine("        {0}", course.Title);
        }

        foreach (var course in department.Courses.
                  OfType<OnlineCourse>())
        {
            Console.WriteLine("        Online - {0}", course.Title);
        }

        foreach (var course in department.Courses.
                  OfType<OnsiteCourse>())
        {
            Console.WriteLine("        Onsite - {0}", course.Title);
        }
    }
}
```

Конструктора запроса хранимых процедур

27.09.2018 • 5 minutes to read • [Edit Online](#)

Это пошаговое руководство показано, как используется Entity Framework Designer (конструктор EF) для импорта хранимой процедуры в модели, а затем вызвать импортированных хранимые процедуры для получения результатов.

Обратите внимание, что Code First не поддерживает сопоставление хранимых процедур или функций. Тем не менее можно вызвать хранимые процедуры или функции с помощью метода System.Data.Entity.DbSet.SqlQuery. Пример:

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]");
```

Предварительные требования

Для выполнения данного пошагового руководства требуется:

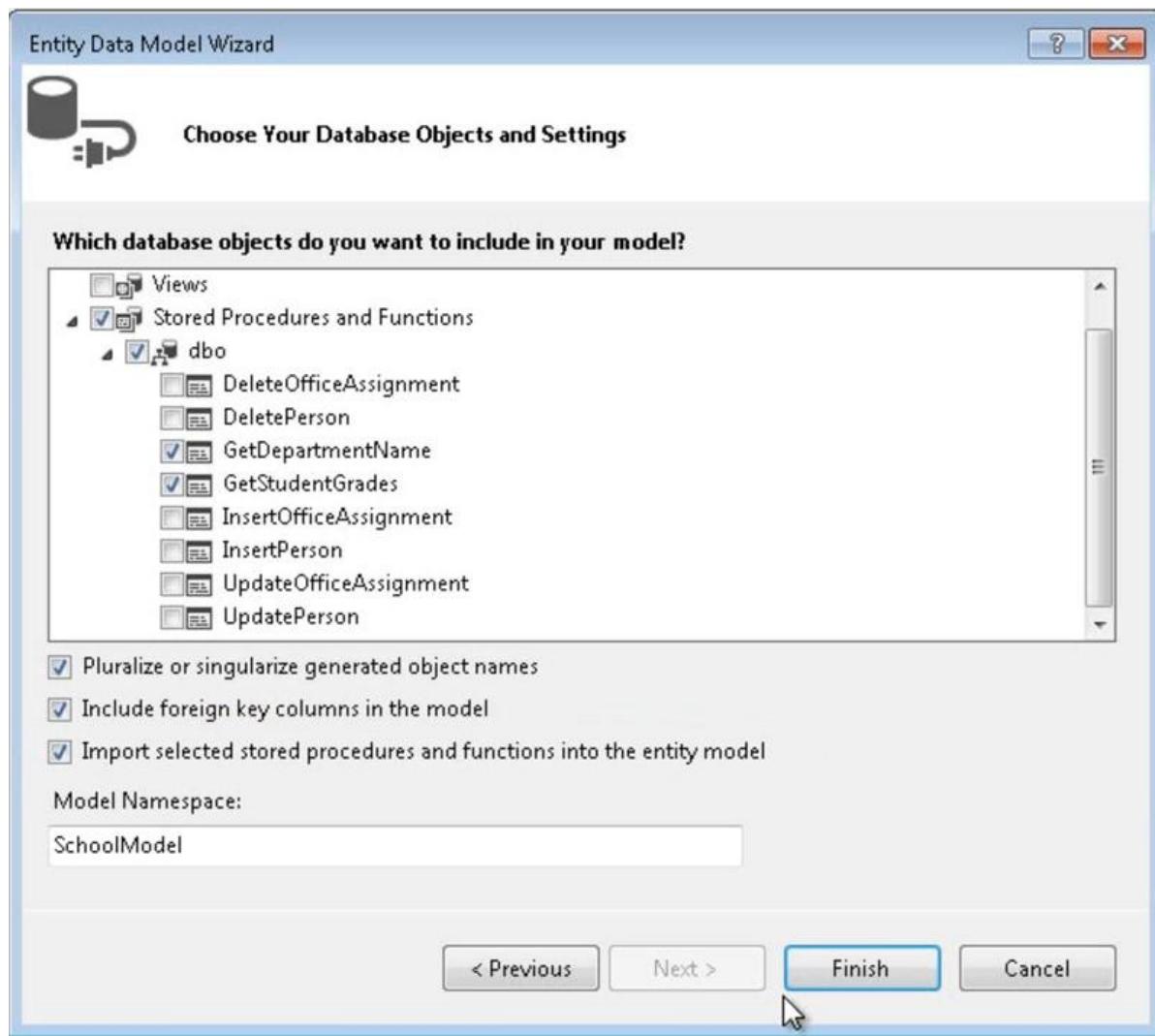
- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

- Откройте Visual Studio 2012.
- Выберите **файл -> Новинка —> проекта**
- В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона.
- Введите **EFwithSProcsSample** как имя.
- Нажмите кнопку **OK**.

Создание модели

- Щелкните правой кнопкой мыши проект в обозревателе решений и выберите **Add -> новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **EFwithSProcsModel.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.
- Нажмите кнопку **новое подключение**.
В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**.
В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В диалоговом окне Выбор объектов базы данных проверьте **таблиц** флажок, чтобы выбрать все таблицы.
Кроме того, выберите следующие хранимые процедуры в разделе **хранимые процедуры и функции** узла: **GetStudentGrades** и **GetDepartmentName**.



Начиная с Visual Studio 2012 конструкторе EF поддерживает массовый импорт хранимых процедур. **Импортировать выбранные хранимые процедуры и функции в модели theentity** установлен по умолчанию.

- Нажмите кнопку **Готово**.

По умолчанию результирующая форма каждого импортированного хранимую процедуру или функцию, которая возвращает более одного столбца автоматически становится новый сложный тип. В этом примере требуется сопоставить результаты **GetStudentGrades** функцию **StudentGrade** сущности и результаты **GetDepartmentName** для **none** (**none** значение по умолчанию).

Импорт функции для возврата типа сущности столбцы, возвращенные соответствующей хранимой процедуре должно совпадать скалярных свойств типа возвращаемой сущности. Импорт функции также может возвращать коллекции простых типов, сложных типов или нет значения.

- Щелкните правой кнопкой мыши область конструктора и выберите **браузер моделей**.
- В **браузер моделей** выберите **импортируемые функции**, а затем дважды щелкните **GetStudentGrades** функции.
- В диалоговом окне Изменение импорта функции выберите **сущностей** и выберите **StudentGrade**. **Импорт функции составляема** флагок в верхней части **импортируемые функции** диалогового окна можно сопоставить с поддержкой композиции функции. Если этот флагок установлен, только **составную функциях** (функции, возвращающие табличные значения) будет отображаться в **хранимой процедуре или имя функции** стрелку раскрывающегося списка. Если вы не установите этот флагок, в списке отобразятся только функции, не допускающие композицию.

Использование модели

Откройте **Program.cs** файл, в котором **Main** определен метод. Добавьте следующий код в функцию Main.

Код вызывает две хранимые процедуры: **GetStudentGrades** (возвращает **StudentGrades** для указанного **StudentId**) и **GetDepartmentName** (Возвращает имя подразделения в выходном параметре).

```
using (SchoolEntities context = new SchoolEntities())
{
    // Specify the Student ID.
    int studentId = 2;

    // Call GetStudentGrades and iterate through the returned collection.
    foreach (StudentGrade grade in context.GetStudentGrades(studentId))
    {
        Console.WriteLine("StudentID: {0}\tSubject={1}", studentId, grade.Subject);
        Console.WriteLine("Student grade: " + grade.Grade);
    }

    // Call GetDepartmentName.
    // Declare the name variable that will contain the value returned by the output parameter.
    ObjectParameter name = new ObjectParameter("Name", typeof(String));
    context.GetDepartmentName(1, name);
    Console.WriteLine("The department name is {0}", name.Value);

}
```

Скомпилируйте и запустите приложение. Программа выдает следующие результаты.

```
StudentID: 2
Student grade: 4.00
StudentID: 2
Student grade: 3.50
The department name is Engineering
```

Выходные параметры

Если используются выходные параметры, их значения не будут доступны, пока не полностью считать результаты. Это происходит из-за особенностей **DbDataReader**, см. в разделе [получение данных с помощью объекта **DataReader**](#) для получения дополнительных сведений.

Конструктора CUD хранимых процедур

13.09.2018 • 9 minutes to read • [Edit Online](#)

Это пошаговое руководство показано, как сопоставить создания\вставки, обновления и удаления (CUD) типа сущности с хранимыми процедурами, с помощью Entity Framework Designer (конструктор EF). По умолчанию Entity Framework автоматически создает инструкции SQL для операций CUD, но также можно сопоставить хранимые процедуры для этих операций.

Обратите внимание, что Code First не поддерживает сопоставление хранимых процедур или функций. Тем не менее можно вызвать хранимые процедуры или функции с помощью метода System.Data.Entity.DbSet.SqlQuery. Пример:

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]");
```

Рекомендации при сопоставлении операций CUD для хранимых процедур

При сопоставлении операций CUD с хранимыми процедурами, следующие соображения.

- Если вы осуществляете сопоставление одной из операций CUD в хранимую процедуру, сопоставьте их все. Если не выполнено сопоставление всех трех, то несопоставленные операции завершится ошибкой при выполнении и **UpdateException** будет создано.
- Свойства сущности необходимо сопоставить каждый параметр хранимой процедуры.
- Если сервер создает значение первичного ключа для вставленной строки, необходимо сопоставить это значение обратно к свойству ключа сущности. В следующем примере **InsertPerson** хранимая процедура возвращает вновь созданный первичный ключ как часть результирующего набора хранимой процедуры. Первичный ключ сопоставляется с ключом сущности (**PersonID**) с помощью <Добавление привязок результатов> функции конструктора Entity FRAMEWORK.
- Вызовы хранимых процедур, сопоставленных 1:1 с сущностями в концептуальной модели. Например, если вы реализуете иерархии наследования в концептуальной модели и затем карты CUD хранимых процедур для **родительского** (базовый) и **дочерних** (производные) сущности, сохранение **Дочерних** изменений будет вызывать только **дочерних** хранимые процедуры, она не сможет запустить **родительского** хранимые процедуры вызовов.

Предварительные требования

Для выполнения данного пошагового руководства требуется:

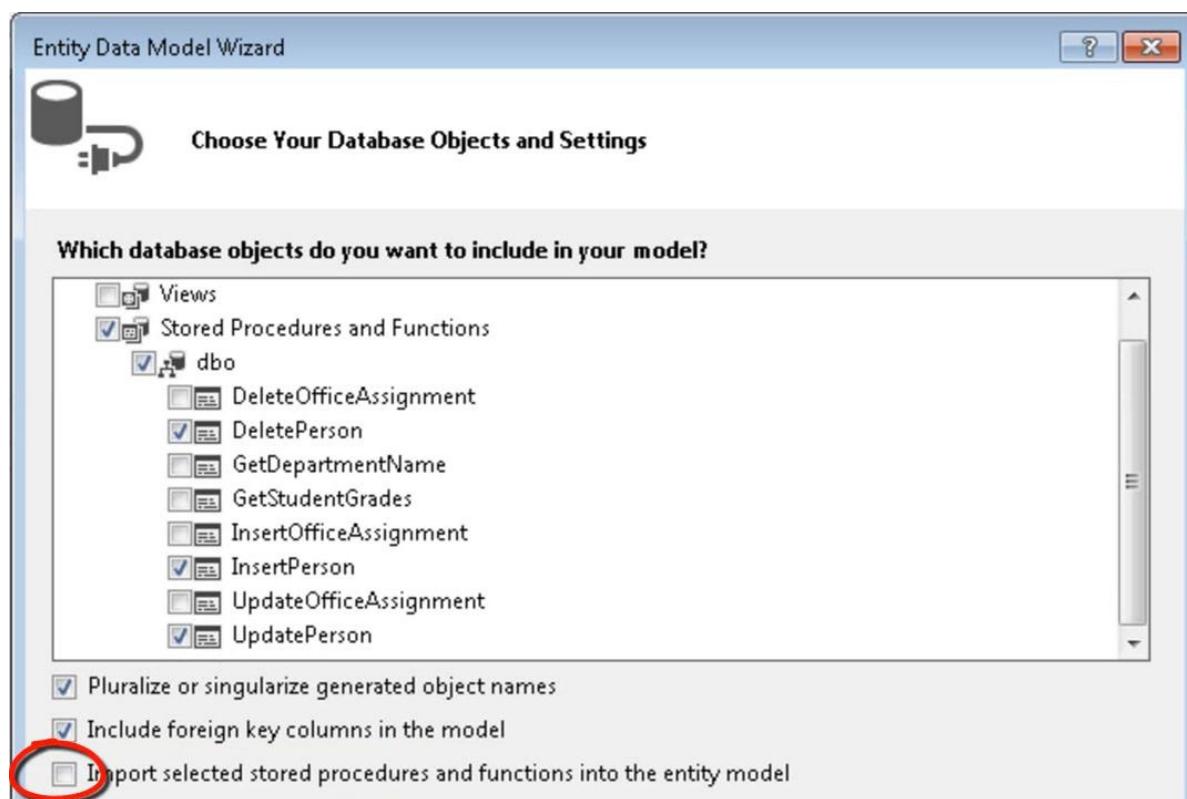
- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

- Откройте Visual Studio 2012.
- Выберите **файл -> Новинка —> проекта**
- В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона.
- Введите **CUDSProcsSample** как имя.
- Нажмите кнопку **OK**.

Создание модели

- Щелкните правой кнопкой мыши имя проекта в обозревателе решений и выберите **Add -> новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **CUDSProcs.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.
- Нажмите кнопку **новое подключение**. В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**. В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В поле выберите ваши объекты базы данных диалогового **таблиц** выберите **Person** таблицы.
- Кроме того, выберите следующие хранимые процедуры в разделе **хранимые процедуры и функции** узла: **DeletePerson**, **InsertPerson**, и **UpdatePerson**.
- Начиная с Visual Studio 2012 конструкторе EF поддерживает массовый импорт хранимых процедур. **Импортировать выбранные хранимые процедуры и функции в модели сущности** установлен по умолчанию. Так как в этом примере мы сохранили процедуры, вставки, обновления и удаления типов сущностей, мы не требуется импортировать их и будет снимите этот флагок.



- Нажмите кнопку **Готово**. Конструктор EF, который предоставляет область конструктора для изменения модели, отображается.

Сопоставление сущности Person с хранимыми процедурами

- Щелкните правой кнопкой мыши **Person** типа сущности и выберите **сопоставление хранимых процедур**.
- Сопоставления хранимых процедур, появятся в **сведения о сопоставлении** окна.

- Нажмите кнопку <функция Insert Select>. Поле становится раскрывающимся списком хранимых процедур в модели хранения, которые могут быть сопоставлены с типами сущностей в концептуальной модели. Выберите **InsertPerson** из раскрывающегося списка.
- Появятся применяемые по умолчанию сопоставления параметров хранимой процедуры со свойствами сущности. Обратите внимание, что стрелки указывают направление сопоставления: «Значения свойств передаются параметрам хранимой процедуры».
- Нажмите кнопку <Добавление привязки к результату>.
- Тип **NewPersonID**, имя параметра, возвращенного **InsertPerson** хранимой процедуры. Убедитесь, что не введите начальные или конечные пробелы.
- Нажмите клавишу **ВВОД**.
- По умолчанию **NewPersonID** сопоставляется с ключом сущности **PersonID**. Обратите внимание, что стрелка указывает направление сопоставления — значение столбца результата передается свойству.

Mapping Details - Person					
Parameter / Column	Operator	Property	Use Original...	Rows Affected Parameter	
Functions					
Insert Using InsertPerson					
Parameters					
@ LastName : nvarchar	←	LastName : String	<input type="checkbox"/>		
@ FirstName : nvarchar	←	FirstName : String	<input type="checkbox"/>		
@ HireDate : datetime	←	HireDate : DateTime	<input type="checkbox"/>		
@ EnrollmentDate : datetime	←	EnrollmentDate : DateTime	<input type="checkbox"/>		
@ Discriminator : nvarchar	←	Discriminator : String	<input type="checkbox"/>		
Result Column Bindings					
NewPersonID	→	PersonID : Int32			

- Нажмите кнопку <Выбор функции обновления> и выберите **UpdatePerson** из результирующего раскрывающегося списка.
- Появятся применяемые по умолчанию сопоставления параметров хранимой процедуры со свойствами сущности.
- Нажмите кнопку <Выбор функции удаления> и выберите **DeletePerson** из результирующего раскрывающегося списка.
- Появятся применяемые по умолчанию сопоставления параметров хранимой процедуры со свойствами сущности.

Вставки, обновления и удаления операций **Person** типа сущности теперь сопоставлены с хранимыми процедурами.

Если вы хотите включить проверку при обновлении или удалении сущности с хранимыми процедурами параллелизма, используйте один из следующих вариантов:

- Используйте **ВЫВОДА** параметр для возврата числа обработанных строк из хранимой процедуры и проверка **параметр числа затронутых строк** флагок рядом с именем параметра. Если возвращенное значение равно нулю, при вызове операции **OptimisticConcurrencyException** будет создано.
- Проверьте **использовать исходное значение** флагок рядом со свойством, которое вы хотите использовать для проверки параллелизма. При попытке обновления значение свойства, которое было первоначально считано из базы данных будет использоваться при записи данных обратно в базу данных. Если значение не соответствует значению в базе данных, **OptimisticConcurrencyException** будет создано.

Использование модели

Откройте **Program.cs** файл, в котором **Main** определен метод. Добавьте следующий код в функцию **Main**.

Код создает новый **Person** объекта, затем обновляет объект и наконец удаляет объект.

```
using (var context = new SchoolEntities())
{
    var newInstructor = new Person
    {
        FirstName = "Robyn",
        LastName = "Martin",
        HireDate = DateTime.Now,
        Discriminator = "Instructor"
    }

    // Add the new object to the context.
    context.People.Add(newInstructor);

    Console.WriteLine("Added {0} {1} to the context.",
        newInstructor.FirstName, newInstructor.LastName);

    Console.WriteLine("Before SaveChanges, the PersonID is: {0}",
        newInstructor.PersonID);

    // SaveChanges will call the InsertPerson sproc.
    // The PersonID property will be assigned the value
    // returned by the sproc.
    context.SaveChanges();

    Console.WriteLine("After SaveChanges, the PersonID is: {0}",
        newInstructor.PersonID);

    // Modify the object and call SaveChanges.
    // This time, the UpdatePerson will be called.
    newInstructor.FirstName = "Rachel";
    context.SaveChanges();

    // Remove the object from the context and call SaveChanges.
    // The DeletePerson sproc will be called.
    context.People.Remove(newInstructor);
    context.SaveChanges();

    Person deletedInstructor = context.People.
        Where(p => p.PersonID == newInstructor.PersonID).
        FirstOrDefault();

    if (deletedInstructor == null)
        Console.WriteLine("A person with PersonID {0} was deleted.",
            newInstructor.PersonID);
}
```

- Скомпилируйте и запустите приложение. Программа создает следующие выходные данные *

NOTE

PersonID формируется автоматически сервером, поэтому вы скорее всего увидите другой номер *

```
Added Robyn Martin to the context.
Before SaveChanges, the PersonID is: 0
After SaveChanges, the PersonID is: 51
A person with PersonID 51 was deleted.
```

Если вы работаете с версией Visual Studio Ultimate, Intellitrace можно использовать с помощью отладчика для просмотра инструкций SQL, которые выполняются.

IntelliTrace

Video: How to use Intellitrace

All Categories All Threads

Search

Debugger: Beginning of Application: Main, Program.cs

XML: XmlDocument Loaded

Debugger: Breakpoint Hit: Main, Program.cs line 33

ADO.NET: Execute Reader "[dbo].[InsertPerson]"

Debugger: Step Recorded: Main, Program.cs line 35

ADO.NET: Execute NonQuery "[dbo].[UpdatePerson]"

ADO.NET: Execute NonQuery "[dbo].[DeletePerson]"

ADO.NET: Execute Reader "SELECT TOP (1) [Extent1].[I

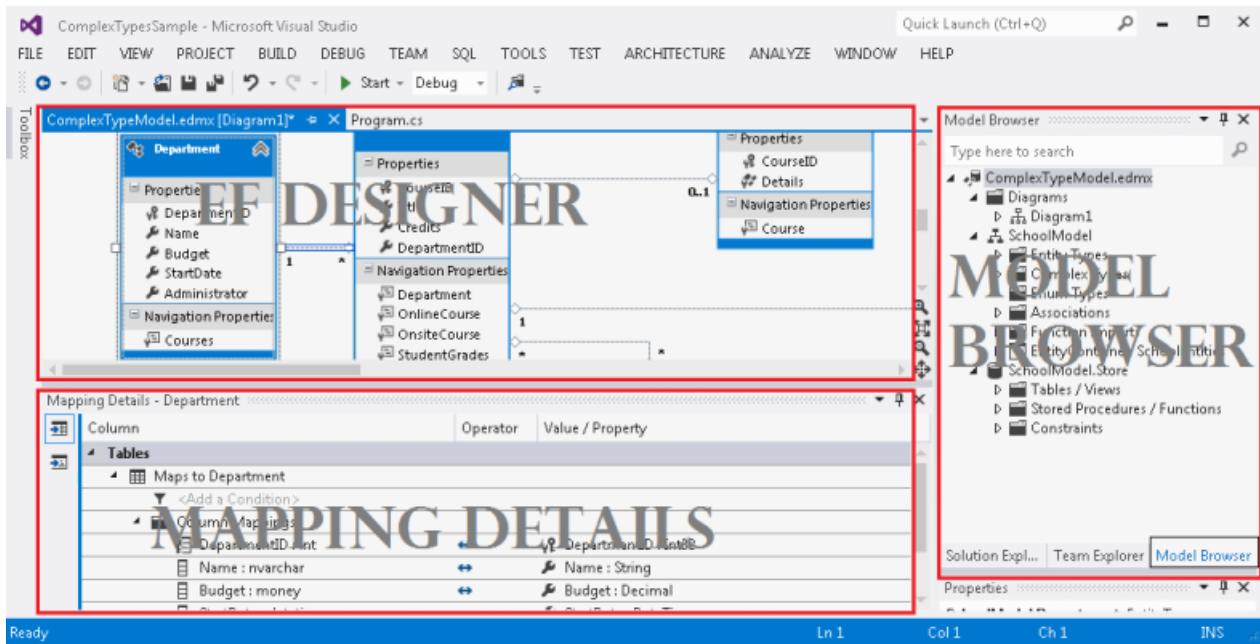
Связей - конструктор EF

13.09.2018 • 9 minutes to read • [Edit Online](#)

NOTE

Эта страница содержит сведения о настройке связи в модели в конструкторе EF. Общие сведения о связях в EF и способах доступа к данных и управления ими с помощью связей см. в разделе [связей и свойств навигации](#).

Ассоциации определяют связи между типами сущностей в модели. В этом разделе показано, как сопоставить ассоциации с Entity Framework Designer (конструктор EF). На следующем рисунке показана основные окна, которые используются при работе с конструктором EF.



NOTE

При создании концептуальной модели, предупреждения о несопоставленных сущностях и ассоциациях, появляются в списке ошибок. Эти предупреждения можно игнорировать, поскольку после нажатия кнопки для создания базы данных из модели, пропадут.

Общие сведения о связи

При разработке модели в конструкторе EF, EDMX-файл представляет модель. В файле EDMX **ассоциации** элемент определяет связь между двумя типами сущностей. Ассоциация должна указывать типы сущностей, которые участвуют в связи, и возможное количество типов сущностей на каждом конце связи, которое называется кратностью. Кратность элемента ассоциации может иметь значение (1), ноль или один (0.. 1) или многие (*). Эта информация указывается в двух дочерних **окончания** элементов.

Во время выполнения экземпляров типа сущности на одном конце ассоциации может осуществляться через свойства навигации или внешних ключей (Если вы решили предоставить внешние ключи в сущностях). Внешние ключи представлены, связи между сущностями, управляются с помощью **ReferentialConstraint** элемент (дочерний элемент элемента **ассоциации** элемент). Рекомендуется всегда предоставлять внешние ключи для связи в сущностях.

NOTE

В многие ко многим (*:*) не удается добавить внешних ключей в сущности. В *:* связи, данные сопоставления управляется с помощью независимый объект.

Дополнительные сведения об элементах языка CSDL (**ReferentialConstraint**, **ассоциаций** и т. д.) см. в разделе [спецификация языка CSDL](#).

Создание и удаление ассоциации

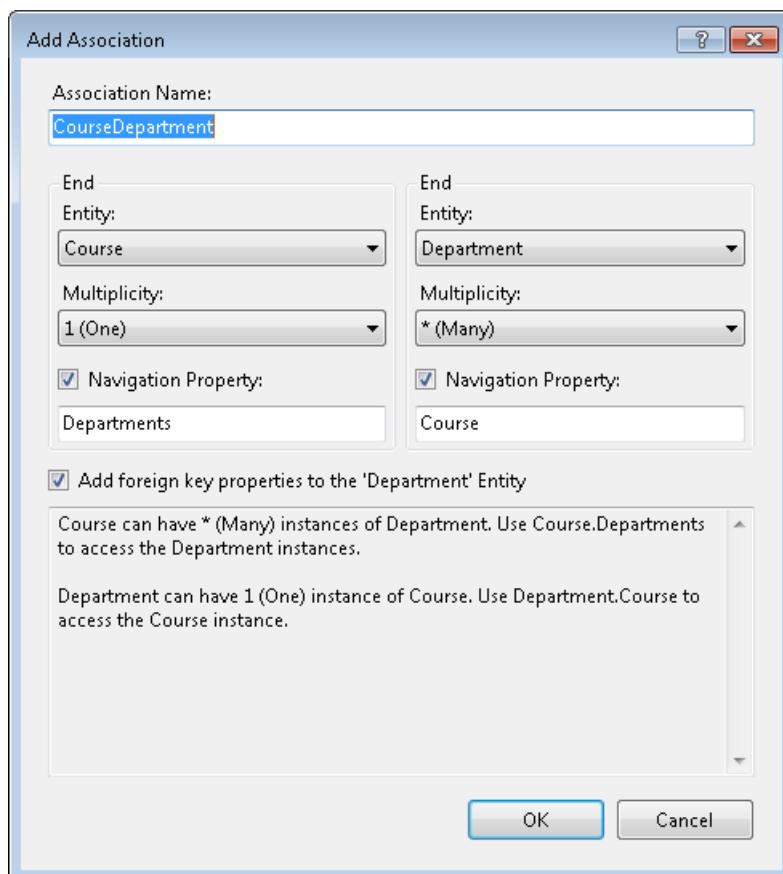
Создание ассоциации с обновлениями конструктор EF содержимое модели в EDMX-файле. После создания ассоциации необходимо создать сопоставления для ассоциации (обсуждается далее в этом разделе).

NOTE

В этом разделе предполагается, что вы уже добавили сущностей, которые вы хотите создать ассоциацию между для вашей модели.

Создание ассоциации

- Щелкните правой кнопкой мыши пустой участок области конструктора, выберите пункт **Add New** и выберите **Связи...**.
- Задайте параметры для ассоциации в **добавить сопоставление** диалоговое окно.



NOTE

Вы можете не добавлять свойства навигации или свойства внешнего ключа для сущностей в элементах ассоциации, сбросив ** свойство навигации ** и ** добавить свойства внешнего ключа для <имя типа сущности> сущности ** флажки. При добавлении одного свойства навигации ассоциацию можно будет перемещать только в одном направлении. При добавлении свойств навигации необходимо добавить свойства внешнего ключа, чтобы обеспечить доступ к сущностям в элементах ассоциации.

3. Нажмите кнопку **OK**.

Удаление ассоциации

Чтобы удалить связь выполните одно из следующих:

- Щелкните правой кнопкой мыши связь в конструкторе EF и выберите пункт **удалить**.
- OR-
- Выберите одну или несколько ассоциаций и нажмите клавишу **DELETE**.

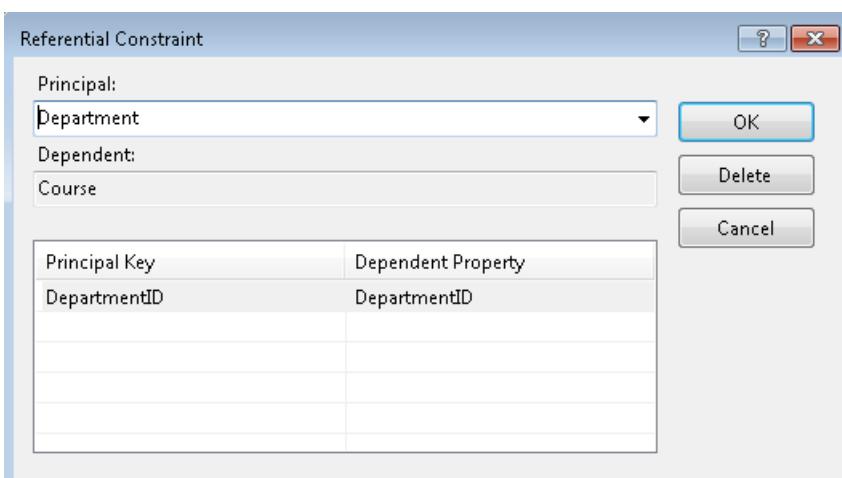
Включить свойства внешнего ключа в сущности (ссылочные ограничения)

Рекомендуется всегда предоставлять внешние ключи для связи в сущностях. Entity Framework использует ссылочное ограничение, чтобы определить, что свойство действует как внешний ключ для связи.

Если вы установили флажок **добавить свойства внешнего ключа для <имя типа сущности> сущности** флажок был добавлен при создании связи этого справочного ограничения для вас.

При использовании для добавления или изменения справочных ограничений конструкторе EF, EF добавляет или изменяет **ReferentialConstraint** элемент в содержимом CSDL EDMX-файла.

- Дважды щелкните ассоциацию, которую необходимо изменить. **Справочного ограничения** откроется диалоговое окно.
- Из **участника** выберите в раскрывающемся списке, выберите основную сущность в ссылочном ограничении. Свойства ключа сущности добавляются к **ключ субъекта-службы** списка в диалоговом окне.
- Из **зависимых** выберите в раскрывающемся списке, выберите зависимую сущность в ссылочном ограничении.
- Для каждого основного ключа, который имеет зависимый ключ, выберите соответствующий зависимый ключ из раскрывающихся списков в **зависящий от него ключ** столбца.



- Нажмите кнопку **OK**.

Создавать и изменять сопоставления ассоциаций

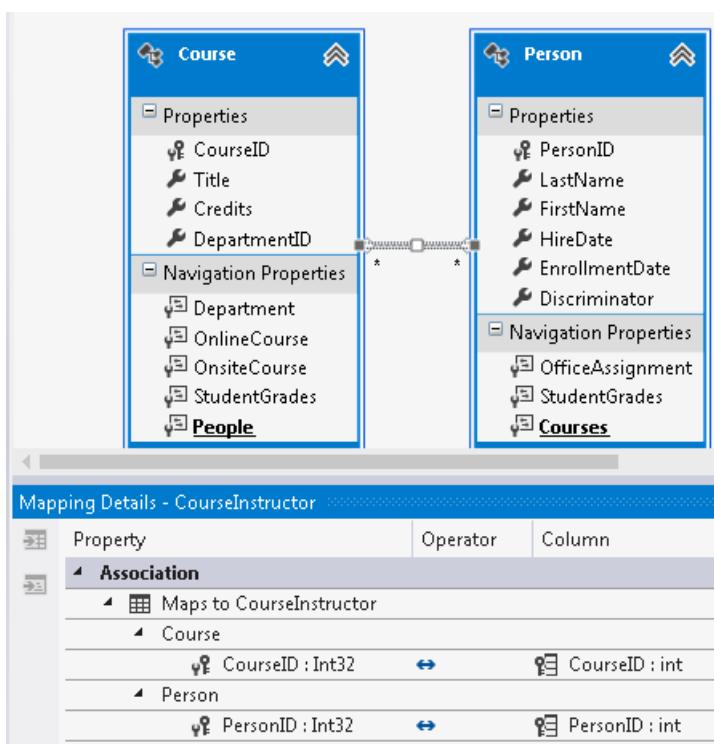
Можно указать сопоставление ассоциации в базу данных в **сведения о сопоставлении** окно конструктора Entity FRAMEWORK.

NOTE

Можно сопоставить только сведения для связей, которые не имеют ссылочные ограничения, указанные. Если указано справочного ограничения свойство внешнего ключа входит в сущности, и можно использовать сведения о сопоставлении сущности, к элементу управления, какой столбец сопоставляется внешний ключ.

Создание сопоставления ассоциаций

- Щелкните правой кнопкой мыши ассоциацию в области конструктора и выберите **сопоставление таблицы**. Это появится сопоставление ассоциации в **сведения о сопоставлении** окна.
- Нажмите кнопку **добавить таблицу или представление**. Появится раскрывающийся список, содержащий все таблицы в модели хранения.
- Выберите таблицу, с которой будет сопоставлена ассоциация. **Сведения о сопоставлении** окне отображаются оба элемента ассоциации и свойства ключа для типа сущности на каждом **окончании**.
- Для каждого свойства ключа щелкните **столбец** поле и выберите столбец, к которому будет сопоставлено это свойство.



Изменение сопоставления ассоциаций

- Щелкните правой кнопкой мыши ассоциацию в области конструктора и выберите **сопоставление таблицы**. Это появится сопоставление ассоциации в **сведения о сопоставлении** окна.
- Нажмите кнопку **сопоставляется <имя таблицы>**. Появится раскрывающийся список, содержащий все таблицы в модели хранения.
- Выберите таблицу, с которой будет сопоставлена ассоциация. **Сведения о сопоставлении** окне отображаются оба элемента ассоциации и свойства ключа для типа сущности на концах.
- Для каждого свойства ключа щелкните **столбец** поле и выберите столбец, к которому будет

сопоставлено это свойство.

Изменение и удаление свойства навигации

Свойства навигации — это свойства быстрого доступа, используемые для нахождения сущностей в элементах ассоциации в модели. Свойства навигации могут быть созданы при создании ассоциации между двумя типами сущностей.

Чтобы изменить свойства навигации

- Выберите свойство навигации на поверхности конструктора EF. Сведения о свойстве навигации отображаются в Visual Studio **свойства** окна.
- Измените параметры свойств в **свойства** окна.

Удалить свойства навигации

- Если внешние ключи не представлены в типах сущностей в концептуальной модели, то удаление свойства навигации может обеспечить перемещение соответствующей ассоциации только в одном направлении или сделать ее неперемещаемой.
- Щелкните правой кнопкой мыши свойство навигации в конструкторе EF и выберите пункт **удалить**.

Несколько диаграмм на модель

27.09.2018 • 7 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Это видео и странице показано, как разделить модель на несколько схем, с помощью Entity Framework Designer (конструктор EF). Можно использовать эту функцию, когда ваша модель становится слишком большим, чтобы просмотреть или изменить.

В более ранних версиях конструктор EF одной схеме может иметь только каждого EDMX-файла. Начиная с Visual Studio 2012, можно использовать конструктор EF для разбиения в EDMX-файла на несколько схем.

Просмотреть видео

В этом видео показано, как разделить модель на несколько схем, с помощью Entity Framework Designer (конструктор EF). Можно использовать эту функцию, когда ваша модель становится слишком большим, чтобы просмотреть или изменить.

Представленный: Юлия Корнич

Видео: [WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Общие сведения о конструкторе EF

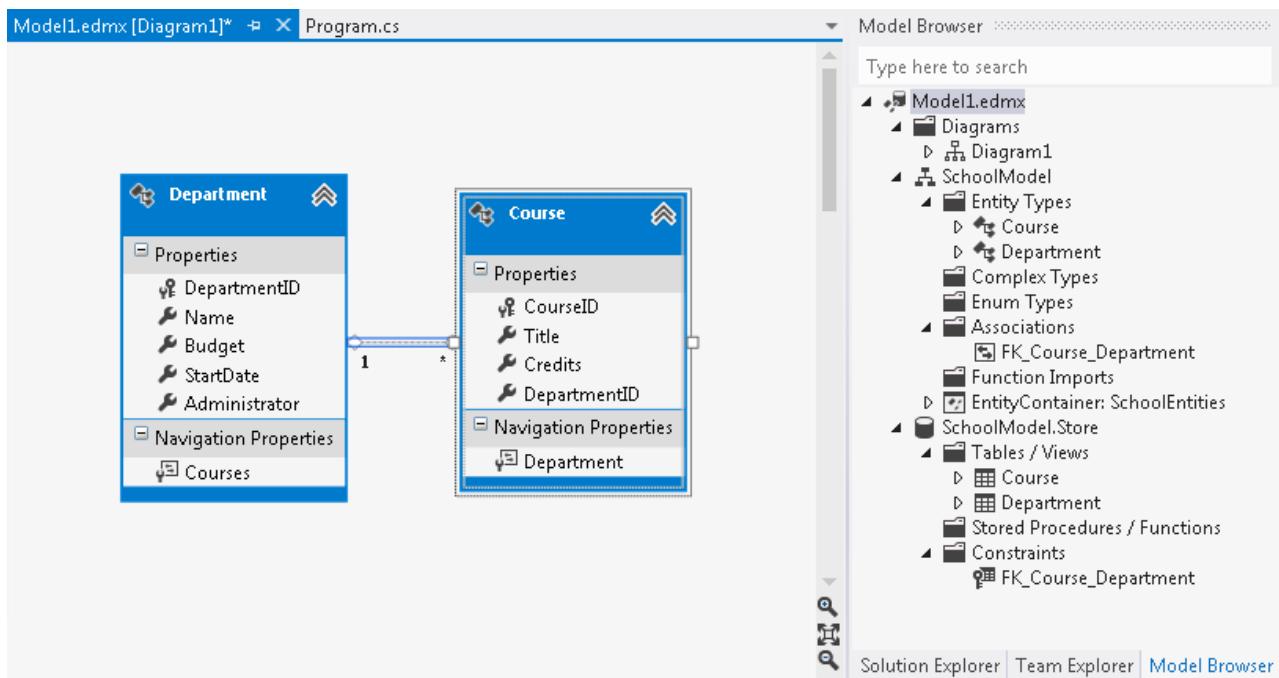
При создании модели с помощью мастера моделей EDM конструкторе EF, EDMX-файл создается и добавляется в решение. Этот файл определяет форму сущностей и их сопоставлении в базу данных.

Конструктор EF состоит из следующих компонентов:

- Визуальную область конструктора для редактирования модели. Сущности и взаимосвязи можно создавать, изменять или удалять.
- Объект **браузер моделей** окно, обеспечивающее три представления модели. Сущности и связи расположены под *[ModelName]* папки. В таблицах базы данных и ограничениях, находятся в разделе *[ModelName]*. Папка Store.
- Объект **сведения о сопоставлении** окно для просмотра и редактирования сопоставлений. Типы сущностей и взаимосвязи можно сопоставить с таблицами базы данных, столбцами и хранимыми процедурами.

Окно поверхности визуальной разработки автоматически открывается после завершения работы мастера модели EDM. Если браузер моделей не отображается, щелкните правой кнопкой мыши главную область конструктора и выберите **браузер моделей**.

На следующем снимке экрана показаны EDMX-файл открывается в конструкторе EF. На снимке экрана показаны область визуального конструирования (слева) и **браузер моделей** окна (справа).



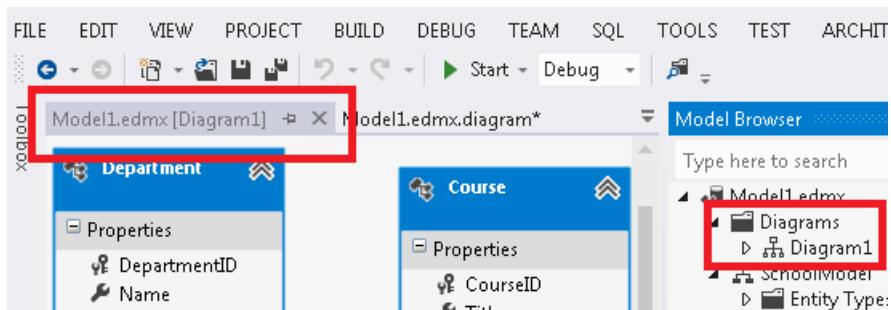
Чтобы отменить операцию в конструкторе EF, нажмите сочетание клавиш Ctrl-Z.

Работа со схемами

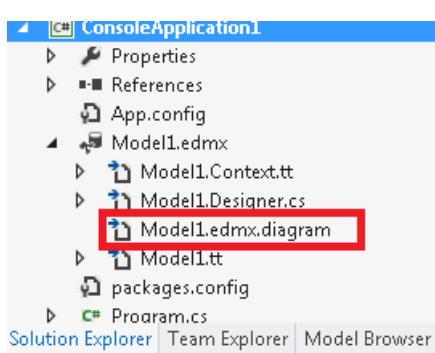
По умолчанию конструктор EF создает вызывается Diagram1 одной схеме. Если у вас есть диаграмму с большим числом сущностей и ассоциаций, будет наиболее как логически разделить их. Начиная с Visual Studio 2012, можно просмотреть концептуальной модели на нескольких схемах.

По мере добавления новых схем, они отображаются в папке схемы в окне браузера модели. Переименование диаграммы: выберите схему в окне браузера модели, щелкните один раз имя и введите новое имя. Можно также правой кнопкой мыши имя диаграммы и выберите **Переименовать**.

Имя диаграммы отображается рядом с именем файла .edmx, в редакторе Visual Studio. Например Model1.edmx[Diagram1].



Содержимое схемы (форма и цвет сущностей и ассоциаций) хранится в файле edmx.diagram. Чтобы просмотреть этот файл, выберите обозреватель решений и раскрыть EDMX-файл.



Не следует изменять .edmx.diagram файл вручную, содержимое этого файла может быть перезаписан в конструкторе EF.

Разделение сущностей и ассоциаций в новую схему

Вы можете выбрать сущности на существующую диаграмму (удерживайте клавишу Shift, чтобы выбрать несколько сущностей). Щелкните правой кнопкой мыши и выберите **переместить в новую схему**. Новая схема, и выбранные сущности и их связи, перемещаются в схему.

Кроме того, щелкните правой кнопкой мыши папку схемы в браузере моделей и выберите **добавить новую схему**. Затем можно перетащить и перетаскивать сущности из папки типы сущностей в браузере моделей, в область конструктора.

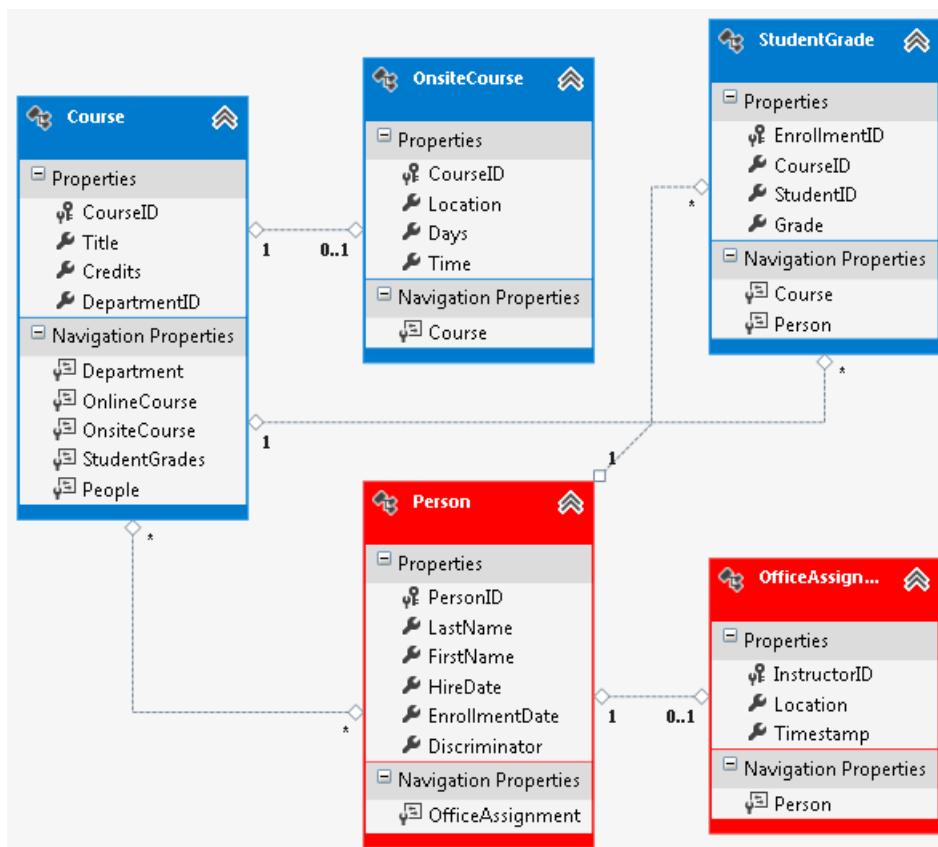
Также можно вырезать или копировать из одной схемы сущностей (с помощью клавиш Ctrl + X или Ctrl + C) и вставить (с помощью клавиши Ctrl + V) на другой. Если схема, в который копируются сущности уже содержит сущность с тем же именем, новую сущность создана и добавлена в модель. Например: схемы 2 содержит сущность. Затем вы вставьте другой отдел на схемы 2. Сущность Department1 создается и добавляется в концептуальную модель.

Чтобы включить связанные сущности в схеме, щелкните правой кнопкой сущность и выберите **включать связанные**. Это будет выполнено копирование связанных сущностей и ассоциаций в указанной схемы.

Изменение цвета сущностей

В дополнение к модели разделения на несколько схем, можно также изменить цвета в одной сущности.

Чтобы изменить цвет, выберите сущность (или несколько сущностей) в области конструктора. Затем щелкните правой кнопкой мыши и выберите **свойства**. В окне «Свойства» выберите **цвет заливки** свойство. Укажите цвет, с помощью допустимым именем цвета (например, красный цвет) или допустимый RGB (например, 255, 128, 128).



Сводка

В этом разделе мы рассмотрели разбиение модели на несколько схем, а также как указать другой цвет для сущности с помощью Entity Framework Designer.

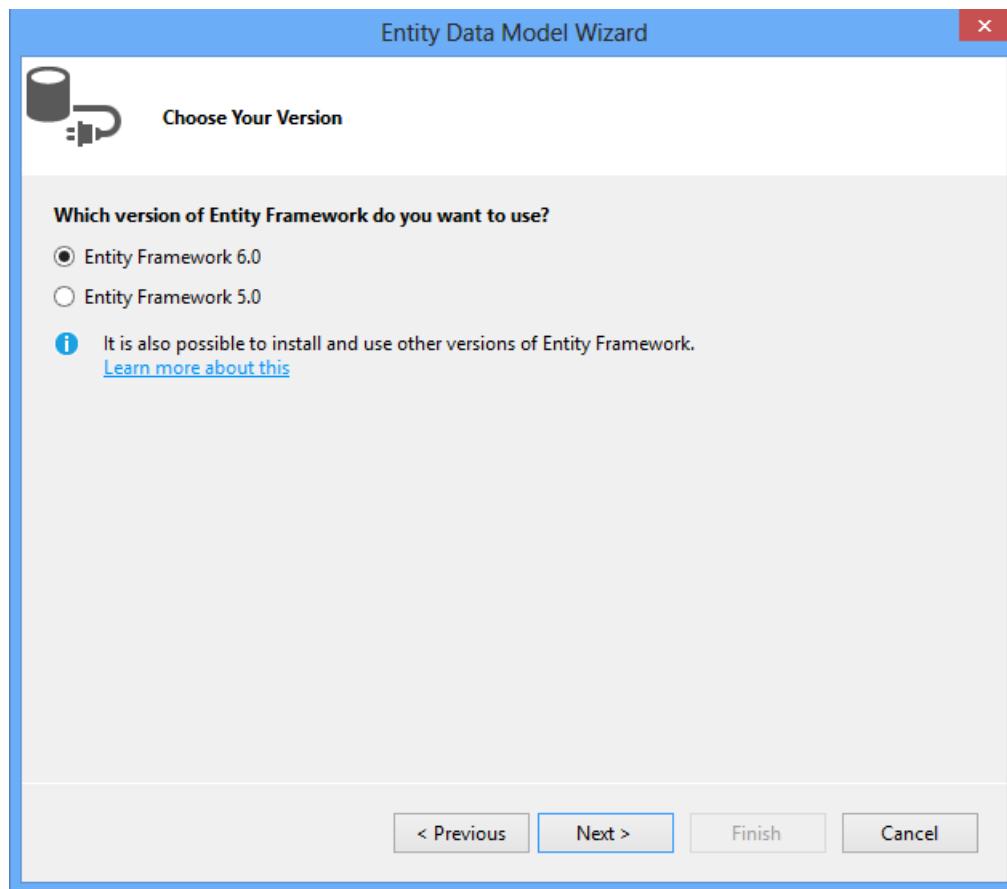
Выбрав версия среды выполнения Entity Framework для конструктора моделей EF

13.09.2018 • 2 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Начиная с EF6 был добавлен следующий экран конструкторе EF, позволяющее выбрать версию среды выполнения, которые вы хотите применить при создании модели. Экран появляется, если последнюю версию Entity Framework не установлена в проекте. Если уже установлена последняя версия просто будет использоваться по умолчанию.



Нацеливание EF6.x

Вы можете EF6 на экране «Выбор версии» для добавления среды выполнения EF6 в проект. После добавления EF6, будет остановлено, отображается этот экран в текущем проекте.

EF6 будет отключен, если у вас уже есть более старая версия EF, установлена (поскольку невозможно указать несколько версий среды выполнения, в том же проекте). Если параметр EF6 здесь не включен, выполните следующие действия для обновления проекта к EF6.

1. Щелкните правой кнопкой мыши на проекте в обозревателе решений и выберите **управление пакетами**

NuGet...

2. Выберите **обновлений**
3. Выберите **EntityFramework** (Убедитесь, что нужно обновить его до версии, требуется)
4. Нажмите кнопку **обновления**

Нацеливание EF5.x

Вы можете EF5 на экране «Выбор версии» в среду выполнения EF5 добавить в проект. После добавления EF5, по-прежнему появится экран с параметром EF6 отключена.

Если у вас есть более EF4.x версии уже установлена среда выполнения вы увидите этой версии платформы EF, перечисленные в экран, а не EF5. В этой ситуации, можно обновить до EF5, выполнив следующие действия:

1. Выберите **средства —> диспетчер пакетов библиотеки -> консоль диспетчера пакетов**
2. Запустите **EntityFramework Install-Package-версии 5.0.0**

Нацеливание EF4.x

Среда выполнения EF4.x можно установить в проект, выполнив следующие действия:

1. Выберите **средства —> диспетчер пакетов библиотеки -> консоль диспетчера пакетов**
2. Запустите **EntityFramework Install-Package-версии 4.3.0**

Шаблоны создания кода конструктора

18.09.2018 • 13 minutes to read • [Edit Online](#)

При создании модели с помощью Entity Framework Designer классы и производный контекст создаются автоматически. Помимо создания кода по умолчанию, вам доступны несколько шаблонов, которые можно использовать для настройки генерируемого кода. Эти шаблоны представляют собой текстовые шаблоны T4, позволяющие при необходимости настраивать шаблоны.

Создаваемый по умолчанию код зависит от версии Visual Studio, в которой создается модель.

- Модели, созданные в Visual Studio 2012 и 2013, будут создавать классы простых сущностей РОСО и контекст, который является производным от упрощенного контекста DbContext.
- Модели, созданные в Visual Studio 2010, будут создавать классы сущностей, которые являются производными от класса EntityObject, и контекст, который является производным от ObjectContext.

NOTE

После добавления модели рекомендуется переключиться на шаблон генератора DbContext.

На этой странице рассматриваются доступные шаблоны, а затем приводятся инструкции по добавлению шаблона в модель.

Доступные шаблоны

Группа разработки Entity Framework предоставляет перечисленные ниже шаблоны.

Генератор DbContext

Этот шаблон предназначен для создания классов простых сущностей РОСО и контекста, который является производным от DbContext, с помощью EF6. Это рекомендуемый шаблон, если только у вас нет причин для использования одного из других шаблонов, перечисленных ниже. Этот шаблон создания кода предоставляется по умолчанию при использовании последних версий Visual Studio (Visual Studio 2013 и более поздние версии): при создании модели этот шаблон применяется по умолчанию, а файлы T4 (TT-файлы) вложены в EDMX-файл.

Предыдущие версии Visual Studio

- **Visual Studio 2012.** Для получения шаблонов **DbContextGenerator для EF 6.x** потребуется установить последнюю версию **инструментов Entity Framework для Visual Studio**. Дополнительные сведения см. на странице о [получении Entity Framework](#).
- **Visual Studio 2010.** Шаблоны **DbContextGenerator для EF 6.x** недоступны для Visual Studio 2010.

Генератор DbContext для EF 5.x

При работе в более старой версии пакета NuGet для EntityFramework (с номером основной версии 5) необходимо использовать шаблон **генератора DbContext для EF 5.x**.

Если вы используете Visual Studio 2013 или 2012, этот шаблон уже установлен.

Если вы используете Visual Studio 2010, при добавлении шаблона необходимо выбрать вкладку **Online** (В Интернете), чтобы скачать его из коллекции Visual Studio. Или шаблон можно установить заранее непосредственно из коллекции Visual Studio. Так как шаблоны входят в состав более поздних версий Visual Studio, версии из коллекции устанавливаются только в Visual Studio 2010.

- [Генератор DbContext для EF 5.x для C#](#)

- Генератор DbContext для EF 5.x для веб-сайтов на C#
- Генератор DbContext для EF 5.x для VB.NET
- Генератор DbContext для EF 5.x для веб-сайтов на VB.NET

Генератор DbContext для EF 4.x

Применяя более старую версию пакета NuGet для EntityFramework (с номером основной версии 4), необходимо использовать шаблон **Генератор DbContext для EF 4.x**. Его можно найти на вкладке **Online** (В сети) при добавлении шаблона, либо шаблон можно установить заранее непосредственно из коллекции Visual Studio.

- Генератор DbContext для EF 4.x для C#
- Генератор DbContext для EF 4.x для веб-сайтов на C#
- Генератор DbContext для EF 4.x для VB.NET
- Генератор DbContext для EF 4.x для веб-сайтов на VB.NET

Генератор EntityObject

Этот шаблон используется для создания классов сущностей, которые являются производными от класса EntityObject, и контекста, который является производным от ObjectContext.

NOTE

Возможность использования шаблона "Генератор DbContext"

Сейчас генератор DbContext представляет собой рекомендуемый шаблон для новых приложений. Генератор DbContext использует преимущества более простого API DbContext. Генератор EntityObject по-прежнему доступен для поддержки существующих приложений.

Visual Studio 2010, 2012 & 2013

При добавлении шаблона необходимо выбрать вкладку **Online** (В сети), чтобы скачать его из коллекции Visual Studio. Или шаблон можно установить заранее непосредственно из коллекции Visual Studio.

- Генератор EntityObject для EF 6.x для C#
- Генератор EntityObject для EF 6.x для веб-сайтов на C#
- Генератор EntityObject для EF 6.x для VB.NET
- Генератор EntityObject для EF 6.x для веб-сайтов на VB.NET

Генератор EntityObject для EF 5.x

Если вы используете Visual Studio 2012 или 2013, при добавлении шаблона необходимо выбрать вкладку **Online** (В сети), чтобы скачать его из коллекции Visual Studio. Или шаблон можно установить заранее непосредственно из коллекции Visual Studio. Так как шаблоны входят в состав Visual Studio 2010, версии из коллекции устанавливаются только в Visual Studio 2012 & 2013.

- Генератор EntityObject для EF 5.x для C#
- Генератор EntityObject для EF 5.x для веб-сайтов на C#
- Генератор EntityObject для EF 5.x для VB.NET
- Генератор EntityObject для EF 5.x для веб-сайтов на VB.NET

Если требуется просто создать код ObjectContext без изменения шаблона, вернитесь к [созданию кода EntityObject](#).

Если вы используете Visual Studio 2010, этот шаблон уже установлен. При создании модели в Visual Studio 2010 этот шаблон используется по умолчанию, но ТТ-файлы не включены в проект. Чтобы настроить шаблон, его следует добавить в проект.

Генератор сущностей с самостоятельным отслеживанием (STE)

Этот шаблон предназначен для создания классов сущностей с самостоятельным отслеживанием и контекста, который является производным от `ObjectContext`. В приложении EF контекст объекта отвечает за отслеживание изменений в сущностях. Однако в многоуровневых сценариях на уровне, изменяющем сущности, контекст может быть недоступен. Сущности с самостоятельным отслеживанием позволяют отслеживать изменения на любом уровне. Дополнительные сведения см. в разделе [Сущности с самостоятельным отслеживанием](#).

NOTE

Шаблон STE является нерекомендуемым

Больше не рекомендуется использовать шаблон STE в новых приложениях, но он по-прежнему будет доступен для поддержки существующих приложений. Другие варианты, применимые в многоуровневых сценариях, см. в статье об [отключенных сущностях](#).

NOTE

У шаблона STE нет версии для EF 6.x.

NOTE

У шаблона STE нет версии для Visual Studio 2013.

Visual Studio 2012

Если вы используете Visual Studio 2012, при добавлении шаблона необходимо выбрать вкладку **Online** (В Интернете), чтобы скачать его из коллекции Visual Studio. Или шаблон можно установить заранее непосредственно из коллекции Visual Studio. Так как шаблоны входят в состав Visual Studio 2010, версии из коллекции устанавливаются только в Visual Studio 2012.

- [Генератор STE для EF 5.x для C#](#)
- [Генератор STE для EF 5.x для веб-сайтов на C#](#)
- [Генератор STE для EF 5.x для VB.NET](#)
- [Генератор STE для EF 5.x для веб-сайтов на VB.NET](#)

Visual Studio 2010**

Если вы используете Visual Studio 2010, этот шаблон уже установлен.

Генератор сущностей POCO

Этот шаблон предназначен для создания классов сущностей POCO и контекста, который является производным от `ObjectContext`.

NOTE

Возможность использования шаблона "Генератор DbContext"

Сейчас "Генератор DbContext" представляет собой рекомендуемый шаблон для создания классов POCO в новых приложениях. "Генератор DbContext" использует новый API `DbContext` и может создавать упрощенные классы POCO. "Генератор сущностей POCO" по-прежнему доступен для поддержки существующих приложений.

NOTE

У шаблона STE нет версии для EF 5.x или EF 6.x.

NOTE

У шаблона РОСО нет версии для Visual Studio 2013.

Visual Studio 2012 & Visual Studio 2010

При добавлении шаблона необходимо выбрать вкладку **Online** (В сети), чтобы скачать его из коллекции Visual Studio. Или шаблон можно установить заранее непосредственно из коллекции Visual Studio.

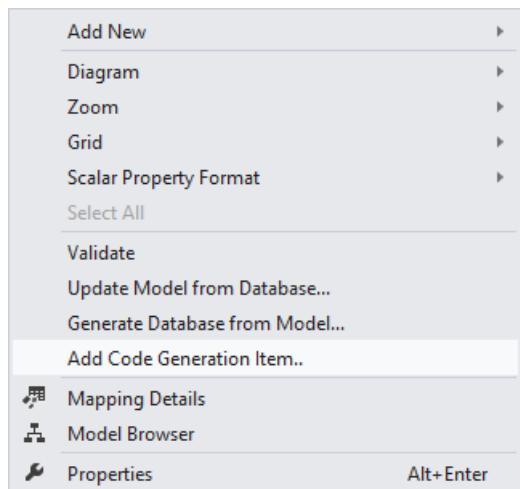
- [Генератор РОСО для EF 4.x для C#](#)
- [Генератор РОСО для EF 4.x для веб-сайтов на C#](#)
- [Генератор РОСО для EF 4.x для VB.NET](#)
- [Генератор РОСО для EF 4.x для веб-сайтов на VB.NET](#)

Что такое шаблоны "Веб-сайты"

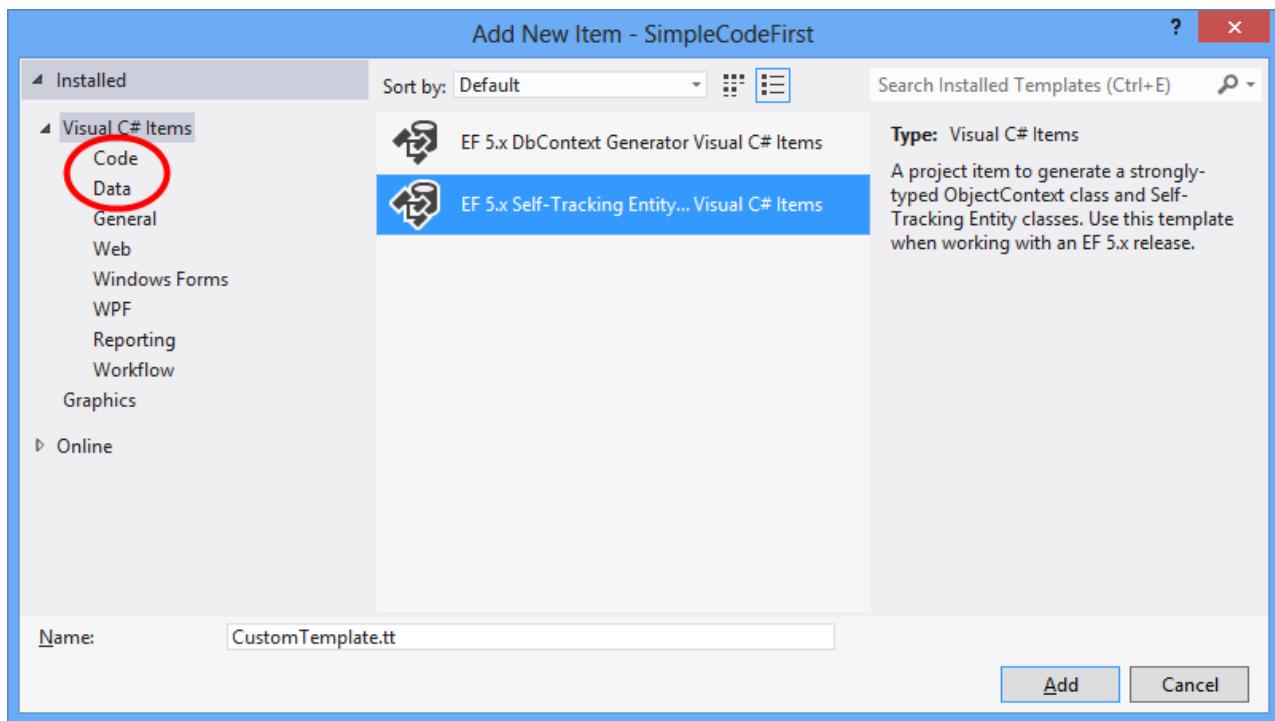
Шаблоны "Веб-сайты", например **EF 5.x DbContext Generator for C# Web Sites** (Генератор DbContext для веб-сайтов C# в EF 5.x), предназначены для проектов веб-сайтов, создаваемых путем выбора **Файл —> Создать —> Веб-сайт....** Они отличаются от веб-приложений, создаваемых при выборе **Файл —> Создать —>Проект** и использующих стандартные шаблоны. Для этих целей предоставляются отдельные шаблоны, которые требуются системе шаблонов элементов в Visual Studio.

Использование шаблона

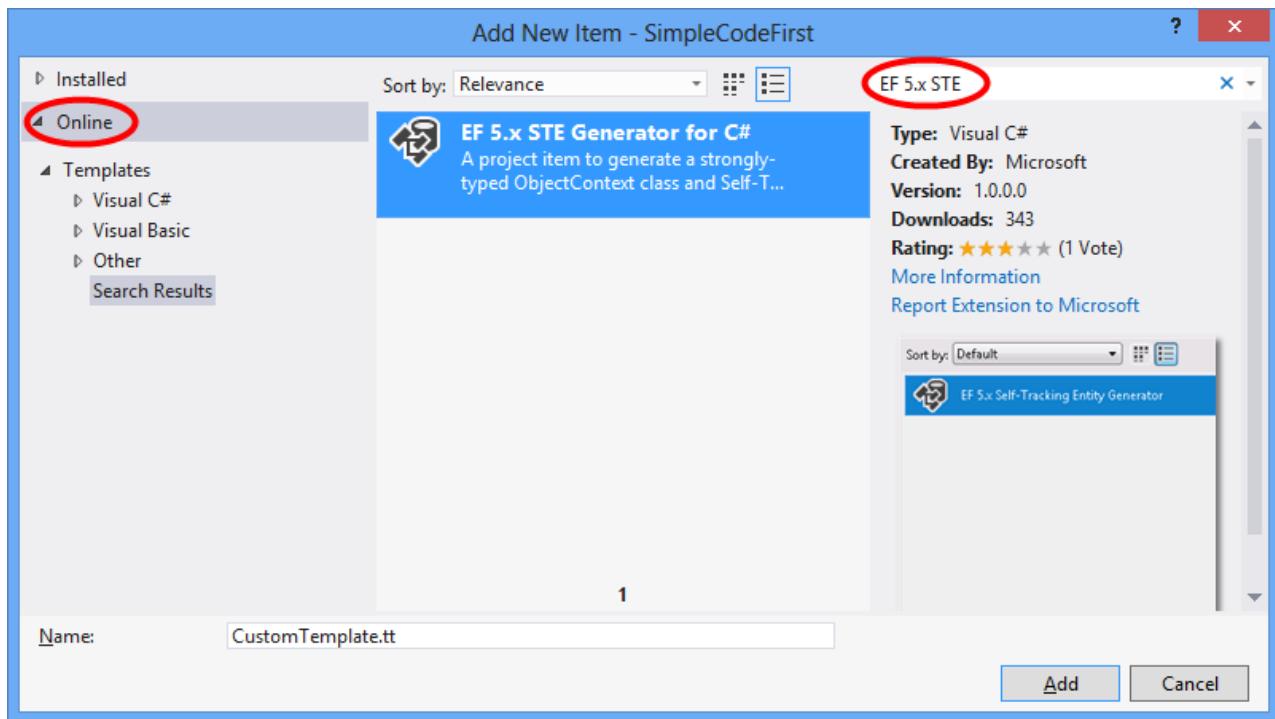
Чтобы приступить к использованию шаблона создания кода, щелкните правой кнопкой мыши пустое место в области конструктора в конструкторе EF Designer и выберите **Добавить элемент создания кода....**



Если вы уже установили нужный шаблон (или он был включен в состав Visual Studio), он будет доступен в разделе **Код** или **Данные** в меню слева.



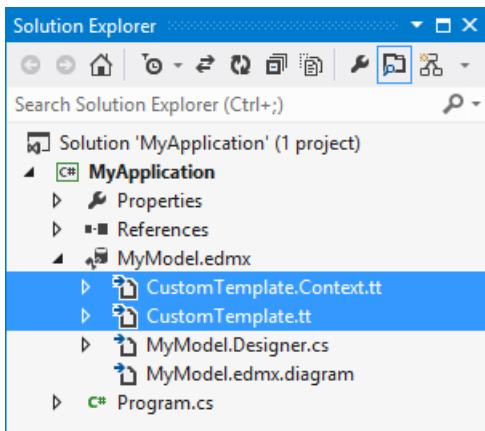
Если шаблон еще не установлен, в левом меню выберите пункт **Online** (В Интернете) и найдите нужный шаблон.



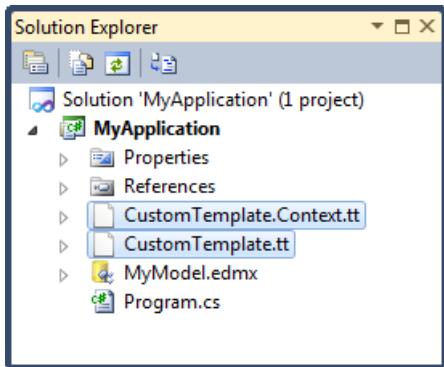
Если вы используете Visual Studio 2012, новые TT-файлы будут вложены в EDMX-файл.*

NOTE

Для моделей, созданных в Visual Studio 2012, потребуется удалить шаблоны, используемые для создания кода по умолчанию. В противном случае будут созданы повторяющиеся классы и контекст. Файлы по умолчанию: `<имя_модели>.tt` и `<имя_модели>.context.tt`.



Если вы используете Visual Studio 2010, ТТ-файлы добавляются непосредственно в проект.



Возврат к ObjectContext в Entity Framework Designer

13.09.2018 • 2 minutes to read • [Edit Online](#)

С помощью предыдущей версии Entity Framework модели, созданной с помощью конструктора EF вызовет контекст, производный от ObjectContext и классов сущностей, производные от класса EntityObject.

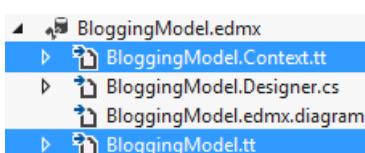
Начиная с EF4.1 рекомендуется обращений шаблон генерации кода, который создает контекст, наследование от классов сущностей DbContext и POCO.

В Visual Studio 2012 вы получаете DbContext код, созданный по умолчанию для всех новых моделей, созданных с помощью конструктора EF. Существующие модели будут создавать ObjectContext на основе кода, если только вы сами замены в генератор кода на основе DbContext.

Возвращение обратно к генерации кода ObjectContext

1. Отключите формирование кода DbContext

Создание производных классов DbContext и POCO обрабатывается двумя tt-файлов в проекте, если развернуть узел EDMX-файл в обозревателе решений появится эти файлы. Удалите оба файла из проекта.



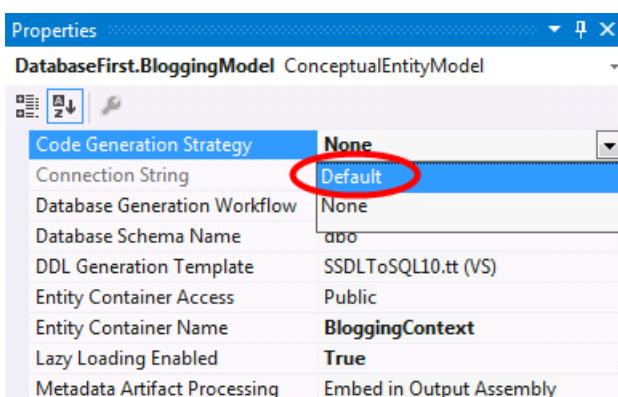
При использовании VB.NET необходимо выбрать **Показать все файлы** кнопку, чтобы увидеть вложенные файлы.



2. Повторно включите создание кода ObjectContext

Открыть модель в конструкторе EF, щелкните правой кнопкой мыши пустую часть области конструктора и выберите **свойства**.

В окне изменения свойства **стратегия создания кода** из **None** для **по умолчанию**.



Спецификация CSDL

02.10.2018 • 106 minutes to read • [Edit Online](#)

Язык CSDL — это язык на основе XML, описывающий сущности, связи и функции, составляющие концептуальную модель управляемого данными приложения. Эту концептуальную модель можно использовать с Entity Framework или службы WCF Data Services. Метаданные, описываемые на языке CSDL используется платформой Entity Framework для сопоставления сущностей и связей, которые определены в концептуальной модели к источнику данных. Дополнительные сведения см. в разделе [спецификация SSDL](#) и [спецификация MSL](#).

CSDL — это реализация модели EDM Entity Framework.

В приложении Entity Framework метаданные концептуальной модели, загружается из CSDL-файл (написанный на языке CSDL) в экземпляр System.Data.Metadata.Edm.EdmItemCollection и доступен с помощью методов в Класс System.Data.Metadata.Edm.MetadataWorkspace. Метаданные концептуальной модели Entity Framework используются для преобразования запросов к концептуальной модели в команды, зависящие от источника данных.

Конструктор EF хранит сведения о концептуальной модели в EDMX-файла во время разработки. Во время сборки, EF конструктор на основе сведений в EDMX-файла, создаваемого CSDL-файл, необходимый Entity Framework во время выполнения.

Версии языка CSDL различаются по пространствам имен XML.

ВЕРСИЯ ЯЗЫКА CSDL	ПРОСТРАНСТВО ИМЕН XML
CSDL версии 1	http://schemas.microsoft.com/ado/2006/04/edm
CSDL версии 2	http://schemas.microsoft.com/ado/2008/09/edm
CSDL версии 3	http://schemas.microsoft.com/ado/2009/11/edm

Элемент Association (CSDL)

Ассоциации элемент определяет связь между двумя типами сущностей. Ассоциация должна указывать типы сущностей, которые участвуют в связи, и возможное количество типов сущностей на каждом конце связи, которое называется кратностью. Кратность элемента ассоциации может иметь значение (1), ноль или один (0.. 1) или многие (*). Эта информация указывается в два дочерних элемента.

К экземплярам типов сущностей в одном элементе ассоциации можно обращаться посредством свойств навигации или внешних ключей, если они предоставлены в типе сущности.

В приложении экземпляр ассоциации представляет конкретную ассоциацию между экземплярами типов сущностей. Экземпляры ассоциации логически сгруппированы в набор ассоциаций.

Ассоциации элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- End (ровно 2 элемента)
- ReferentialConstraint (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **ассоциации** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя ассоциации.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ассоциации** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, определяющий **CustomerOrders** ассоциации, если внешние ключи не представлены в **клиента** и **Порядок** типов сущностей. **Кратность** значения для каждого **конечных** ассоциации указывают что многие **заказы** может быть связано с **клиента**, но только один **клиента** может быть связано с **порядок**. Кроме того **OnDelete** элемент указывает, что все **заказы**, относящихся к конкретному **клиента** и были загружены в контекст ObjectContext будут удалены Если **клиента** удаляется.

```
<Association Name="CustomerOrders">
    <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" >
        <OnDelete Action="Cascade" />
    </End>
    <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

В следующем примере показан **ассоциации** элемент, определяющий **CustomerOrders** ассоциации, если внешние ключи представлены в **клиента** и **Порядок** типов сущностей. Внешние ключи представлены, связи между сущностями, управляются с помощью **ReferentialConstraint** элемент. Соответствующий элемент AssociationSetMapping необязателен для сопоставления этой ассоциации с источником данных.

```
<Association Name="CustomerOrders">
    <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" >
        <OnDelete Action="Cascade" />
    </End>
    <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
    <ReferentialConstraint>
        <Principal Role="Customer">
            <PropertyRef Name="Id" />
        </Principal>
        <Dependent Role="Order">
            <PropertyRef Name="CustomerId" />
        </Dependent>
    </ReferentialConstraint>
</Association>
```

Элемент AssociationSet (CSDL)

AssociationSet элемента на языке определения концептуальной схемы (CSDL) — это логический контейнер для экземпляров ассоциаций одного типа. Набор ассоциаций предоставляет определение группы экземпляров ассоциаций, чтобы их можно было сопоставить с источником данных.

AssociationSet элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один допустимое количество элементов)
- End (требуется ровно 2 элемента)
- Элементы annotation (ноль или более допустимое количество элементов)

Ассоциации атрибут задает тип ассоциации, содержащейся в наборе ассоциаций. Наборы сущностей, которые представляют конечные набора ассоциаций, указываются ровно двумя дочерними **окончания** элементов.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **AssociationSet** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя набора сущностей. Значение имя атрибут не может быть таким же, как значение ассоциации атрибута.
Ассоциации	Да	Полное имя ассоциации, экземпляры которой содержатся в наборе ассоциаций. Ассоциация должна находиться в том же пространстве имен, что и набор ассоциаций.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **AssociationSet** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент с двумя **AssociationSet** элементов:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

Элемент CollectionType (язык CSDL)

CollectionType элемента на языке определения концептуальной схемы (CSDL) указывает, что параметр функции или функции возвращает тип является коллекцией. **CollectionType** элемент может быть дочерним элементом параметра или элемент Return Type (функция). Тип коллекции можно указать с помощью **тип** атрибут или одно из следующих дочерних элементов:

- **CollectionType**
- **ReferenceType**

- RowType
- TypeRef

NOTE

Модель не пройдет проверку, если тип коллекции будет указан и с **тип** атрибут и дочерний элемент.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **CollectionType** элемент. Обратите внимание, что **DefaultValue**, **MaxLength**, **FixedLength**, **точности**, **масштабирования**, **Юникод**, и **параметры сортировки** атрибуты применимы только к коллекциям **EDMSimpleTypes**.

Имя атрибута	Необходимо	Значение
Type	Нет	Тип коллекции.
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли свойство иметь значение null. [!NOTE]
> В языке CSDL v1, должен иметь свойство сложного типа Nullable="False".		
DefaultValue	Нет	Значение свойства по умолчанию.
MaxLength	Нет	Максимальная длина значения свойства.
FixedLength	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка фиксированной длины.
Точность	Нет	Точность значения свойства.
Масштаб	Нет	Масштаб значения свойства.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server)
Юникод	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка Юникода.
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **CollectionType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано определяемой моделью функции, в которой используется **CollectionType** элемента, чтобы указать, что функция возвращает коллекцию **Person** типы сущностей (как указано с **ElementType** атрибут).

```
<Function Name="LastNamesAfter">
    <Parameter Name="someString" Type="Edm.String"/>
    <ReturnType>
        <CollectionType ElementType="SchoolModel.Person"/>
    </ReturnType>
    <DefiningExpression>
        SELECT VALUE p
        FROM SchoolEntities.People AS p
        WHERE p.LastName >= someString
    </DefiningExpression>
</Function>
```

В следующем примере показано определяемой моделью функции, который использует **CollectionType** элемента, чтобы указать, что функция возвращает коллекцию строк (как указано в **RowType** элемент).

```
<Function Name="LastNamesAfter">
    <Parameter Name="someString" Type="Edm.String" />
    <ReturnType>
        <CollectionType>
            <RowType>
                <Property Name="FirstName" Type="Edm.String" Nullable="false" />
                <Property Name="LastName" Type="Edm.String" Nullable="false" />
            </RowType>
        </CollectionType>
    </ReturnType>
    <DefiningExpression>
        SELECT VALUE ROW(p.FirstName, p.LastName)
        FROM SchoolEntities.People AS p
        WHERE p.LastName &gt;= somestring
    </DefiningExpression>
</Function>
```

В следующем примере показано определяемой моделью функции, который использует **CollectionType** элемента, чтобы указать, что функция принимает в качестве параметра коллекцию **отдел** типов сущностей.

```
<Function Name="GetAvgBudget">
    <Parameter Name="Departments">
        <CollectionType>
            <TypeRef Type="SchoolModel.Department"/>
        </CollectionType>
    </Parameter>
    <ReturnType Type="Collection(Edm.Decimal)"/>
    <DefiningExpression>
        SELECT VALUE AVG(d.Budget) FROM Departments AS d
    </DefiningExpression>
</Function>
```

Элемент **ComplexType** (CSDL)

Объект **ComplexType** элемент определяет структуру данных, состоящих из **EdmSimpleType** свойств или других сложных типов. Сложный тип может быть свойством типа сущности или другого сложного типа. Сложный тип аналогичен типу сущности, поскольку также определяет данные. Однако между сложными типами и типами сущности существуют некоторые ключевые различия.

- Сложные типы не имеют идентификаторов (или ключей) и поэтому не могут существовать независимо. Сложные типы могут существовать только как свойства типов сущностей или других сложных типов.
- Сложные типы не могут участвовать в связях. Ни конечной точки ассоциации может быть сложным типом, и поэтому свойства навигации нельзя определить для сложных типов.
- Свойство сложного типа не может иметь значение null, хотя каждое скалярное свойство сложного типа может быть установлено в это значение.

Объект **ComplexType** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Свойство (ноль или более элементов)
- Элементы annotation (ноль или более элементов)

В следующей таблице описаны атрибуты, которые могут применяться к **ComplexType** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
name	Да	Имя сложного типа. Имя сложного типа не может совпадать с именем другого сложного типа, типа сущности или сопоставления, которые находятся в области модели.
BaseType	Нет	Имя другого сложного типа, который является базовым типом определяемого сложного типа. [!NOTE]
> Этот атрибут неприменим в версии 1 для языка CSDL. В этой версии не поддерживается наследование для сложных типов.		
Абстрактный	Нет	Значение true , или false (значение по умолчанию) в зависимости от того, является ли сложный тип абстрактным типом. [!NOTE]
> Этот атрибут неприменим в версии 1 для языка CSDL. Сложные типы в этой версии не могут быть абстрактными типами.		

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ComplexType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано сложный тип, **адрес**, с помощью **EdmSimpleType** свойства **StreetAddress**, **Город**, **StateOrProvince**, **страны**, и **PostalCode**.

```
<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>
```

Чтобы определить сложный тип **адрес** (выше) как свойство типа сущности, необходимо объявить тип свойства в определении типа сущности. В следующем примере показан **адрес** свойство как сложный тип для типа сущности (**издателя**):

```
<EntityType Name="Publisher">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Type="Int32" Name="Id" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false" />
  <Property Type="BooksModel.Address" Name="Address" Nullable="false" />
  <NavigationProperty Name="Books" Relationship="BooksModel.PublishedBy"
    FromRole="Publisher" ToRole="Book" />
</EntityType>
```

Элемент DefiningExpression (CSDL)

DefiningExpression элемент на языке определения концептуальной схемы (CSDL) содержит выражение Entity SQL, которая определяет функцию в концептуальной модели.

NOTE

В целях проверки **DefiningExpression** элемент может содержать произвольное содержимое. Тем не менее, платформа Entity Framework приведет к возникновению исключения во время выполнения при **DefiningExpression** элемент не содержит допустимый язык Entity SQL.

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **DefiningExpression** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере используется **DefiningExpression** для определения функции, которая возвращает количество лет с момента публикации книги. Содержание **DefiningExpression** элемент создается на языке Entity SQL.

```
<Function Name="GetYearsInPrint" ReturnType="Edm.Int32" >
  <Parameter Name="book" Type="BooksModel.Book" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
  </DefiningExpression>
</Function>
```

Элемент Dependent (CSDL)

Зависимых элемента на языке определения концептуальной схемы (CSDL) является дочерним элементом для элемента **ReferentialConstraint** и определяет зависимый конец справочного ограничения. Объект **ReferentialConstraint** элемент определяет функциональные возможности, схожие с возможностями ограничения ссылочной целостности в реляционной базе данных. Свойство (или свойства) типа сущности могут ссылаться на ключ сущности в другом типе сущности также, как столбец (или столбцы) в таблице базы данных могут ссылаться на первичный ключ другой таблицы. Тип сущности, на который приведена ссылка называется **основного конца ограничения**. Тип сущности, который ссылается на основном конечном элементе называется **зависимого конца ограничения**. **PropertyRef** элементы, используемые для указания, какие разделы ссылаются на основной элемент.

Зависимых элемент может иметь следующие дочерние элементы (в указанном порядке):

- **PropertyRef** (один или несколько элементов)
- Элементы **annotation** (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **зависимых** элемент.

Имя атрибута	Необходимо	Значение
Роль	Да	Имя типа сущности в зависимом элементе ассоциации.

NOTE

Любое количество атрибутов **annotation** (настраиваемых атрибутов XML), которые могут быть применены к **зависимых** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ReferentialConstraint** элемента, используемого в качестве части определения **PublishedBy** ассоциации. **PublisherId** свойство **книги** делает тип сущности представляет зависимый элемент ссылочного ограничения.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Элемент Documentation (CSDL)

Документации элемента на языке определения концептуальной схемы (CSDL) можно использовать для предоставления сведений об объекте, который определен в родительском элементе. В файле EDMX когда **документации** элемент является дочерним для элемента, который отображается в виде объекта в области

конструктора в конструкторе EF (например, сущность, ассоциация или свойство), содержимое **документации** элемент будет отображаться в Visual Studio **свойства** для объекта.

Документации элемент может иметь следующие дочерние элементы (в указанном порядке):

- **Сводка**: краткое описание родительского элемента. (ноль или один элемент)
- **LongDescription**: подробное описание родительского элемента. (ноль или один элемент)
- Элементы annotation. (ноль или больше элементов)

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **документации** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **документации** элемент как дочерний элемент элемента EntityType. Если приведенный ниже фрагмент кода на языке CSDL содержитное EDMX, содержимое файла из **Сводка** и **LongDescription** появилось бы в Visual Studio **свойства** окно, при щелчке `Customer` типа сущности.

```
<EntityType Name="Customer">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Type="Int32" Name="CustomerId" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false" />
</EntityType>
```

Элемент End (CSDL)

Окончания элемент на языке определения концептуальной схемы (CSDL) может быть потомком элемента Association или элемент AssociationSet. В каждом случае роль **окончания** отличается элемент и применимые атрибуты отличаются.

Элемент End как дочерний по отношению к элементу Association

Окончания элемент (как дочерний **ассоциации** элемент) определяет тип сущности на одном конце ассоциации и число экземпляров типа сущности, которые могут существовать на одной конечной точке ассоциации. Элементы ассоциации определяются при определении ассоциации; ассоциация должна иметь два элемента. К экземплярам типов сущности на одном элементе ассоциации можно осуществлять доступ с помощью свойств навигации или внешних ключей при условии, что они были предоставлены в типах сущности.

Окончания элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- OnDelete (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **окончания** элементу, если он является дочерним элементом **ассоциации** элемент.

Имя атрибута	Необходимо	Значение
Type	Да	Имя типа сущности на одном элементе ассоциации.
Роль	Нет	Имя для элемента ассоциации. Если имя не было предоставлено, будет использовано имя типа сущности на элементе ассоциации.
Кратность	Да	<p>1, от 0 до 1, или * в зависимости от количества экземпляров типа сущности, которые могут быть в конце ассоциации.</p> <p>1 означает, что только один экземпляр типа сущности существует на элементе ассоциации.</p> <p>от 0 до 1 указывает, что на конечной точке ассоциации существует ноль или один экземпляров типа сущности.</p> <p>* Указывает, что на конечной точке ассоциации существует ноль, один или несколько экземпляров типа сущности.</p>

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **окончания** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, определяющий **CustomerOrders** ассоциации. **Кратность** значения для каждого **конечных** ассоциации указывают что многие **заказы** может быть связано с **клиента**, но только один **клиента** может быть связано с **порядок**. Кроме того **OnDelete** элемент указывает, что все **заказы**, относящихся к конкретному **клиента** и которые были загружены в контекст ObjectContext будут удаленные **клиента** удаляется.

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" />
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" /
    <OnDelete Action="Cascade" />
  </End>
</Association>
```

Элемент End как дочерний по отношению к элементу AssociationSet

Окончания элемент указывает один конец набора ассоциаций. **AssociationSet** элемент должен содержать два **окончания** элементов. Сведения, содержащиеся в **окончания** элемент используется в сопоставлении с источником данных набора ассоциаций.

Окончания элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

NOTE

Элементы Annotation должны следовать после всех остальных дочерних элементов. Элементы annotation допускаются только в том случае, в языке CSDL версии 2 и более поздних версий.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **окончания** элементу, если он является дочерним элементом **AssociationSet** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Набор EntitySet	Да	Имя EntitySet , определяющего один элемент родительского элемента AssociationSet элемент. EntitySet должен быть определен в том же контейнере сущностей, что и родительский элемент AssociationSet элемент.
Роль	Нет	Имя элемента набора ассоциаций. Если роли атрибут не используется, имя набора ассоциаций будет использоваться имя набора сущностей.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **окончания** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент с двумя **AssociationSet** элементов, каждый с двумя **окончания** элементов:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

Элемент EntityContainer (CSDL)

EntityContainer элемента на языке определения концептуальной схемы (CSDL) — это логический контейнер для наборов сущностей, наборов ассоциаций и функций импорта. Контейнер сущностей модели хранения посредством элемента EntityContainerMapping сопоставляет контейнер сущностей концептуальной модели. Контейнер сущностей режима хранения описывает структуру базы данных: наборы сущностей описывают таблицы, наборы ассоциаций описывают ограничения внешних ключей, функции импорта

описывают хранимые процедуры в базе данных.

EntityContainer элемент может иметь ноль или один элемент документации. Если **документации** элемент присутствует, он должен предшествовать **EntitySet**, **AssociationSet**, и **FunctionImport** элементов.

EntityContainer элемент может иметь ноль или более следующих дочерних элементов (в указанном порядке):

- EntitySet
- AssociationSet
- FunctionImport
- Элементы Annotation

Вы можете расширить **EntityContainer** элемент для включения содержимого другого **EntityContainer**, находящегося в одном пространстве имен. Чтобы включить содержимое другого **EntityContainer**, в ссылающейся **EntityContainer** элемент, установите для параметра **расширения** атрибут имени **EntityContainer** элемент, который требуется включить. Все дочерние элементы включенного **EntityContainer** элемент будет рассматриваться как дочерние элементы ссылающегося **EntityContainer** элемент.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **Using** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя контейнера сущностей.
Расширяет	Нет	Имя другого контейнера сущностей в том же пространстве имен.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EntityContainer** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент, определяющий три набора сущностей и два набора ассоциаций.

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

Элемент EntitySet (CSDL)

EntitySet элемент в языке определения концептуальной схемы — это логический контейнер для экземпляров типа сущности и экземпляров любого типа, который является производным от этого типа сущности. Связь между типом сущности и набором сущностей аналогична связи между строкой и таблицей в реляционной базе данных. Тип сущности, как и строка, определяет ряд взаимосвязанных данных, а набор сущностей, как и таблица, содержит экземпляры этого определения. Набор сущностей предоставляет конструкцию для группирования экземпляров типа сущности, чтобы их можно было сопоставлять со связанными структурами данных в источнике данных.

Можно определить больше одного набора сущностей для конкретного типа сущности.

NOTE

Конструктор EF не поддерживает концептуальные модели, которые содержат несколько наборов сущностей на тип.

EntitySet элемент может иметь следующие дочерние элементы (в указанном порядке):

- Элемент Documentation (ноль или один допустимое количество элементов)
- Элементы annotation (ноль или более допустимое количество элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntitySet** элементу.

Имя атрибута	Необходимо	Значение
Name	Да	Имя набора сущностей.
EntityType	Да	Полное имя типа сущности, для которого набор сущностей содержит экземпляры.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EntitySet** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент с тремя **EntitySet** элементов:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

Предусмотрена возможность определять несколько наборов сущностей на тип (модель MEST). В следующем примере определяется контейнер сущностей с двумя наборами сущностей для **книги** типа

сущности:

```
<EntityContainer Name="BooksContainer" >
  <EntityType Name="Books" EntityType="BooksModel.Book" />
  <EntityType Name="FictionBooks" EntityType="BooksModel.Book" />
  <EntityType Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntityType Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="BookAuthor" Association="BooksModel.BookAuthor">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

Элемент EntityType (CSDL)

EntityType элемент представляет структуру концепции верхнего уровня, например, клиента или заказа, в концептуальной модели. Тип сущности — это шаблон для экземпляров типов сущностей в приложении. Каждый шаблон содержит следующие сведения.

- Уникальное имя. (Обязательно).
- Ключ сущности, определяемый одним или несколькими свойствами. (Обязательно).
- Свойства содержащихся данных. (Необязательно.)
- Свойства навигации, позволяющие осуществлять переход от одного элемента ассоциации к другому. (Необязательно.)

В приложении экземпляр типа сущности представляет определенный объект (например, определенного клиента или заказ). Каждый экземпляр типа сущности в наборе сущностей должен иметь уникальный ключ сущности.

Два экземпляра типа сущности считаются равными, только если они являются экземплярами одного типа и значения их ключей сущности равны.

EntityType элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Ключ (ноль или один элемент)
- Свойство (ноль или более элементов)
- NavigationProperty (ноль или более элементов)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntityType** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя типа сущности.
BaseType	Нет	Имя другого типа сущности, являющегося базовым типом для определяемого типа сущности.

Имя атрибута	Необходимо	Значение
Абстрактный	Нет	Значение true, или False в зависимости от того, является ли тип сущности абстрактным типом.
OpenType	Нет	Значение true, или False в зависимости от того, является ли тип сущности является тип сущности открытым. [!NOTE]
> OpenType атрибут применяется только к типам сущностей, которые определены в концептуальной модели, которые используются со службами данных ADO.NET.		

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EntityType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityType** элемент с тремя **свойство** элементов и два **NavigationProperty** элементов:

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

Элемент EnumType (CSDL)

EnumType элемент представляет собой перечислимый тип.

EnumType элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Член (ноль или более элементов)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EnumType** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя типа сущности.
IsFlags	Нет	Значение true , или False , в зависимости от типа перечисления может использоваться как набор флагов. Значение по умолчанию — False .
UnderlyingType	Нет	Edm.Byte , Edm.Int16 , Edm.Int32 , Edm.Int64 или Edm.SByte Определение диапазона значений типа. Является базовым типом перечисления элементов по умолчанию Edm.Int32 .

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EnumType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EnumType** элемент с тремя **член** элементов:

```
<EnumType Name="Color" IsFlags="false" UnderlyingTyp="Edm.Byte">
  <Member Name="Red" />
  <Member Name="Green" />
  <Member Name="Blue" />
</EntityType>
```

Элемент Function (CSDL)

Функция элемент на языке определения концептуальной схемы (CSDL) используется для определения или объявления функций в концептуальной модели. Функция определяется с помощью элемент **DefiningExpression**.

Объект **функция** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Параметр (ноль или более элементов)
- **DefiningExpression** (ноль или один элемент)
- **ReturnType** (**Function**) (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Возвращает объект типа для функции должен быть указан с помощью **ReturnType** элемент (**Function**) или **ReturnType** атрибутов (см. ниже), но не оба. Возвращаемым типом может быть **EdmSimpleType**, тип сущности, сложный тип, строковый тип, ссылочный тип или коллекция, которая включает один из этих типов.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **функция** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя функции.
ReturnType	Нет	Тип, возвращаемый функцией.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **функция** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере используется **функция** элемент, чтобы определить функцию, которая возвращает количество лет, так как инструктор был принят на работу.

```
<Function Name="YearsSince" ReturnType="Edm.Int32">
  <Parameter Name="date" Type="Edm.DateTime" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(date)
  </DefiningExpression>
</Function>
```

Элемент FunctionImport (CSDL)

FunctionImport элемент на языке определения концептуальной схемы (CSDL) представляет функции, определенные в источнике данных, но доступна для объектов через концептуальную модель. Например элемента Function в модели хранения можно использовать для представления хранимой процедуры в базе данных. Объект **FunctionImport** элемента в концептуальной модели представляет соответствующую функцию в приложении Entity Framework и сопоставляется с функцией модели хранения с помощью элемента FunctionImportMapping. При вызове функции в приложении соответствующая хранимая процедура выполняется в базе данных.

FunctionImport элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один допустимое количество элементов)
- Параметр (ноль или более допустимое количество элементов)
- Элементы annotation (ноль или более допустимое количество элементов)
- ReturnType (FunctionImport) (ноль или более допустимое количество элементов)

Один **параметр** элемент должен быть определен для каждого параметра, принимаемого функцией.

Возвращает объект типа для функции должен быть указан с помощью **ReturnType** элемент (FunctionImport) или **ReturnType** атрибутов (см. ниже), но не оба. Типа возвращаемого значения должен быть коллекцией EdmSimpleType, тип EntityType или ComplexType.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **FunctionImport** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя импортируемой функции.

Имя атрибута	Необходимо	Значение
ReturnType	Нет	Тип, возвращаемый функцией. Не используйте этот атрибут, если функция не возвращает значение. В противном случае значение должно быть коллекция ComplexType, EntityType или EDMSSimpleType.
Набор EntitySet	Нет	Если функция возвращает коллекцию сущностей типов, значение EntitySet должен представлять собой набор сущностей, которому принадлежит коллекция. В противном случае EntitySet атрибут не должен использоваться.
IsComposable	Нет	Если присвоено значение true, функция — композицию (Table-valued Function) и может использоваться в запросе LINQ. Значение по умолчанию — false .

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **FunctionImport** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **FunctionImport** элемент, который принимает один параметр и возвращает коллекцию типов сущностей:

```
<FunctionImport Name="GetStudentGrades"
    EntitySet="StudentGrade"
    ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Элемент Key (CSDL)

Ключ элемент является дочерним элементом ElementType и определяет *ключ сущности* (свойство или набор свойств типа сущности, которые определяют идентификатор). Свойства, составляющие ключ сущности, выбираются во время разработки. Значения свойств ключа сущности должны уникально определять экземпляр типа сущности внутри набора сущностей во время выполнения. Свойства, составляющие ключ сущности, должны гарантировать уникальность экземпляра набора сущностей. **Ключ** элемент определяет ключ сущности, ссылаясь на одно или несколько свойств типа сущности.

Ключ элемент может иметь следующие дочерние элементы:

- PropertyRef (один или несколько элементов)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к

ключ элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В приведенном ниже примере определяется тип сущности с именем **книги**. Ключ сущности определяется ссылкой на **ISBN** свойство типа сущности.

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

ISBN свойство является хорошим выбором для ключа сущности, поскольку международный стандартный номер книги (ISBN) уникально определяет книгу.

В следующем примере показано типа сущности (**автор**), имеет ключ сущности, состоящий из двух свойств **имя** и **адрес**.

```
<EntityType Name="Author">
  <Key>
    <PropertyRef Name="Name" />
    <PropertyRef Name="Address" />
  </Key>
  <Property Type="String" Name="Name" Nullable="false" />
  <Property Type="String" Name="Address" Nullable="false" />
  <NavigationProperty Name="Books" Relationship="BooksModel.WrittenBy"
    FromRole="Author" ToRole="Book" />
</EntityType>
```

С помощью **имя** и **адрес** для сущности, ключ является разумным выбором, поскольку двух авторов с тем же именем, скорее всего, live по тому же адресу. Однако такой выбор ключа сущности не гарантирует уникальность ключей сущности в наборе сущностей. Добавление свойства, такие как **AuthorId**, который может использоваться для уникальной идентификации автора в этом случае рекомендуется.

Элемент Member (CSDL)

Член элемент является дочерним элементом EnumType и определяет член перечисляемого типа.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **FunctionImport** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя элемента.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Значение	Нет	Значение элемента. По умолчанию первый элемент имеет значение 0, и значение каждого последующего перечислителя увеличивается на 1. Может существовать несколько членов с одинаковыми значениями.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **FunctionImport** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EnumType** элемент с тремя **член** элементов:

```
<EnumType Name="Color">
  <Member Name="Red" Value="1"/>
  <Member Name="Green" Value="3" />
  <Member Name="Blue" Value="5"/>
</EntityType>
```

Элемент NavigationProperty (CSDL)

Объект **NavigationProperty** элемент определяет свойство навигации, который предоставляет ссылку на другом конце ассоциации. В отличие от свойства, определенные в элементе свойства навигации не определяют форму и характеристики данных. Они предоставляют возможность навигации по ассоциации между двумя типами сущностей.

Обратите внимание, что свойства навигации являются необязательными для обоих типов сущностей, расположенных в конечных элементах ассоциации. Если свойство навигации было определено для типа сущности на одном конечном элементе ассоциации, то определять его для типа сущности на другом конечном элементе необязательно.

Тип данных, возвращаемый свойством навигации, определяется кратностью в удаленном элементе ассоциации. Например, предположим, что свойство навигации, **OrdersNavProp**, существует на **клиента** типа сущности и устанавливает связь "один ко многим" между **клиента** и **Порядок**. Так как конечная точка удаленной ассоциации для свойства навигации имеет кратность многие (*), он имеет тип данных коллекции (из **порядок**). Аналогичным образом, если свойство навигации, **CustomerNavProp**, существует на **порядок** будет иметь тип сущности, тип данных **клиента** поскольку кратность удаленного конечного один (1).

Объект **NavigationProperty** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **NavigationProperty** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства навигации.

Имя атрибута	Необходимо	Значение
Связь	Да	Имя ассоциации, расположенной в пределах области модели.
ToRole	Да	Конечная точка ассоциации на которой заканчивается навигация. Значение ToRole атрибут должен быть таким же, как значение одного из роли атрибутами, определенными на одном из окончаний ассоциации (определенных в элементе AssociationEnd).
FromRole	Да	Конечная точка ассоциации с которой начинается навигация. Значение FromRole атрибут должен быть таким же, как значение одного из роли атрибутами, определенными на одном из окончаний ассоциации (определенных в элементе AssociationEnd).

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **NavigationProperty** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере определяется тип сущности (**книги**) с двумя свойствами навигации (**PublishedBy** и **WrittenBy**):

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

Элемент OnDelete (CSDL)

OnDelete элемент на языке определения концептуальной схемы (CSDL) определяет поведение, связанное с ассоциацией. Если **действие** атрибуту присваивается **Cascade** на одном конце ассоциации, связанные с типами сущностей на другой стороне ассоциации удаляются при удалении типа сущности в первом элементе. Если связь между двумя типами сущностей связь «первичный ключ к первичному ключа», то загруженный зависимый объект удаляется при удалении главного объекта, на другом конце ассоциации вне зависимости от **OnDelete** Спецификация.

NOTE

OnDelete элемент влияет только на поведение во время выполнения приложения; он не влияет на поведение в источнике данных. Поведение, определенное в источнике данных, должно совпадать с поведением, определенным для приложения.

OnDelete элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **OnDelete** элементу.

Имя атрибута	Необходимо	Значение
Действие	Да	CASCADE или None . Если Cascade , зависимые типы сущностей будут удалены при удалении основного типа сущности. Если None , зависимые типы сущностей не будут удалены при удалении основного типа сущности.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ассоциации** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, определяющий **CustomerOrders** ассоциации.

OnDelete элемент указывает, что все **заказы**, относящихся к конкретному **клиента** и были загружены в контекст ObjectContext будут удалены при **Клиент** удаляется.

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

Элемент Parameter (CSDL)

Параметр элемент на языке определения концептуальной схемы (CSDL) может быть дочерний элемент **FunctionImport**, либо функция.

Приложение, использующее элемент FunctionImport

Объект **параметр** элемент (как дочерний **FunctionImport** элемент) используется для определения входных и выходных параметров функций импорта, объявленных на языке CSDL.

Параметр элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один допустимое количество элементов)

- Элементы annotation (ноль или более допустимое количество элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **параметр** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя параметра.
Type	Да	Тип параметра. Значение должно быть EDMSimpleType или сложный тип, который находится в пределах области модели.
Режим	Нет	B , Out , или InOut в зависимости от того, является ли параметр входной, выходной или параметр ввода вывода.
MaxLength	Нет	Максимально допустимая длина параметра.
Точность	Нет	Точность параметра.
Масштаб	Нет	Масштаб параметра.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для параметров пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **параметр** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **FunctionImport** элемент с одним **параметр** дочерний элемент. Функция принимает один входной параметр и возвращает коллекцию типов сущностей.

```
<FunctionImport Name="GetStudentGrades"
    EntitySet="StudentGrade"
    ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Приложение, использующее элемент Function

Объект **параметр** элемент (как дочерний **функция** элемент) определяет параметры для функций, определяемых или объявляемых в концептуальной модели.

Параметр элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)

- CollectionType (ноль или один элемент)
- ReferenceType (ноль или один элемент)
- RowType (ноль или один элемент)

NOTE

Только один из **CollectionType**, **ReferenceType**, или **RowType** элементов может быть дочерним элементом элемента **свойство** элемент.

- Элементы annotation (ноль или более допустимое количество элементов)

NOTE

Элементы Annotation должны следовать после всех остальных дочерних элементов. Элементы annotation допускаются только в том случае, в языке CSDL версии 2 и более поздних версий.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **параметр** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя параметра.
Type	Нет	Тип параметра. Параметр может иметь любой из следующих типов (или быть коллекцией этих типов): EdmSimpleType тип сущности сложный тип тип строки ссылочный тип
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли свойство иметь null значение.
DefaultValue	Нет	Значение свойства по умолчанию.
MaxLength	Нет	Максимальная длина значения свойства.
FixedLength	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка фиксированной длины.
Точность	Нет	Точность значения свойства.
Масштаб	Нет	Масштаб значения свойства.

Имя атрибута	Необходимо	Значение
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .
Юникод	Нет	Значение true , или false в зависимости от того, будет ли значение свойства храниться как строка Юникода.
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **параметр** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **функция** элемент, который использует один **параметр** дочернего элемента для определения параметра функции.

```
<Function Name="GetYearsEmployed" ReturnType="Edm.Int32">
<Parameter Name="Instructor" Type="SchoolModel.Person" />
<DefiningExpression>
    Year(CurrentDateTime()) - Year(cast(Instructor.HireDate as DateTime))
</DefiningExpression>
</Function>
```

Элемент Principal (CSDL)

Участника элемента на языке определения концептуальной схемы (CSDL) является дочерним элементом для элемента **ReferentialConstraint**, который определяет основной конец справочного ограничения. Объект **ReferentialConstraint** элемент определяет функциональные возможности, схожие с возможностями ограничения ссылочной целостности в реляционной базе данных. Свойство (или свойства) типа сущности могут ссылаться на ключ сущности в другом типе сущности также, как столбец (или столбцы) в таблице базы данных могут ссылаться на первичный ключ другой таблицы. Тип сущности, на который приведена ссылка называется **основного конца ограничения**. Тип сущности, который ссылается на основном конечном элементе называется **зависимого конца ограничения**. **PropertyRef** элементы, используемые для указания, какие ключи ссылается зависимый конечный.

Участника элемент может иметь следующие дочерние элементы (в указанном порядке):

- **PropertyRef** (один или несколько элементов)
- Элементы **annotation** (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **участника** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Роль	Да	Имя типа сущности в основном конце ассоциации.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **участнику** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ReferentialConstraint** элемент, который является частью определения **PublishedBy** ассоциации. **Идентификатор** свойство **издателя** тип сущности составляет основной конец справочного ограничения.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" >
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Элемент Property (CSDL)

Свойство элемент на языке определения концептуальной схемы (CSDL) может быть дочерний элемент **EntityType**, **ComplexType**-элемент или элемент **RowType**.

Применение элементов **EntityType** и **ComplexType**

Свойство элементов (как дочерние элементы **EntityType** или **ComplexType** элементы) определяют форму и характеристики данных, который будет содержать экземпляр типа сущности или сложного типа . Свойства в концептуальной модели аналогичны свойствам, которые определены в классе. По такому же принципу, как свойства, относящиеся к классу, определяют форму класса и несут информацию об объектах, свойства в концептуальной модели определяют форму типа сущности и несут информацию об экземплярах типа сущности.

Свойство элемент может иметь следующие дочерние элементы (в указанном порядке):

- Элемент **Documentation** (ноль или один допустимое количество элементов)
- Элементы **annotation** (ноль или более допустимое количество элементов)

Следующие аспекты, которые могут применяться к **свойство** элемент: **Nullable**, **DefaultValue**, **MaxLength**, **FixedLength**, **точности**, **масштабирования**, **Юникода**, **параметры сортировки**, **ConcurrencyMode**.

Аспекты представляют собой атрибуты XML, которые предоставляют сведения о том, как значения свойств хранятся в хранилище данных.

NOTE

Аспекты может применяться только к свойствам этого типа **EDMSimpleType**.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **свойство** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства.
Type	Да	Тип значения свойства. Тип значения свойства должен быть EDMSimpleType или сложного типа (указано в полное доменное имя), который находится в пределах области модели.
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли свойство иметь значение null. [!NOTE]
> В языке CSDL v1 должен иметь свойство сложного типа <code>Nullable="False"</code> .		
DefaultValue	Нет	Значение свойства по умолчанию.
MaxLength	Нет	Максимальная длина значения свойства.
FixedLength	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка фиксированной длины.
Точность	Нет	Точность значения свойства.
Масштаб	Нет	Масштаб значения свойства.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .
Юникод	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка Юникода.

Имя атрибута	Необходимо	Значение
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.
ConcurrencyMode	Нет	Нет (значение по умолчанию) или Fixed . Если присвоено значение Fixed , значение свойства будет использоваться в проверках оптимистичного параллелизма.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **свойство** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityType** элемент с тремя **свойство** элементов:

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

В следующем примере показан **ComplexType** элемент с пятью **свойство** элементов:

```
<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>
```

Применение элемента RowType

Свойство элементы (будучи дочерними по отношению **RowType** элемент) определяют форму и характеристики данных, который может быть передан или возвращаемых определяемой моделью функции.

Свойство элемент может иметь только один из следующих дочерних элементов:

- CollectionType
- ReferenceType
- RowType

Свойство элемент может иметь любое число дочерних элементов заметки.

NOTE

Элементы annotation допускаются только в том случае, в языке CSDL версии 2 и более поздних версий.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **свойство** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства.
Type	Да	Тип значения свойства.
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли свойство иметь значение null. [!NOTE]
> В языке CSDL v1 должен иметь свойство сложного типа <code>Nullable="False"</code> .		
DefaultValue	Нет	Значение свойства по умолчанию.
MaxLength	Нет	Максимальная длина значения свойства.
FixedLength	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка фиксированной длины.
Точность	Нет	Точность значения свойства.
Масштаб	Нет	Масштаб значения свойства.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .
Юникод	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка Юникода.
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **свойство** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **свойство** элементы, используемые для определения вида тип возвращаемого значения определяемой моделью функции.

```
<Function Name="LastNamesAfter">
    <Parameter Name="someString" Type="Edm.String" />
    <ReturnType>
        <CollectionType>
            <RowType>
                <Property Name="FirstName" Type="Edm.String" Nullable="false" />
                <Property Name="LastName" Type="Edm.String" Nullable="false" />
            </RowType>
        </CollectionType>
    </ReturnType>
    <DefiningExpression>
        SELECT VALUE ROW(p.FirstName, p.LastName)
        FROM SchoolEntities.People AS p
        WHERE p.LastName &gt;= somestring
    </DefiningExpression>
</Function>
```

Элемент PropertyRef (CSDL)

PropertyRef элемент на языке определения концептуальной схемы (CSDL) ссылается на свойство типа сущности, чтобы указать, что свойство будет выступать в одной из следующих ролей:

- Часть ключа сущности (является свойством или набором свойств типа сущности, которые определяют идентификатор). Один или несколько **PropertyRef** элементы можно использовать для определения ключа сущности.
- Зависимый или основной конец справочного ограничения.

PropertyRef элемент может содержать только элементы annotation (ноль и более) как дочерние элементы.

NOTE

Элементы annotation допускаются только в том случае, в языке CSDL версии 2 и более поздних версий.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **PropertyRef** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя свойства, на которое дается ссылка.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **PropertyRef** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В приведенном ниже примере определяется тип сущности (**книги**). Ключ сущности определяется ссылкой на **ISBN** свойство типа сущности.

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

В следующем примере два **PropertyRef** элементов используются для указания, что два свойства (**идентификатор** и **PublisherId**) являются основным и зависимым окончанием ссылочного ограничение.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Элемент ReferenceType (CSDL)

ReferenceType элемент на языке определения концептуальной схемы (CSDL) задает ссылку на тип сущности. **ReferenceType** элемент может быть дочерним элементом следующих элементов:

- **ReturnType** (функция)
- Параметр
- **CollectionType**

ReferenceType элемент используется при задании параметра или возвращаемого типа для функции.

Объект **ReferenceType** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **ReferenceType** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Type	Да	Имя типа сущности, на который делается ссылка.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ReferenceType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ReferenceType** элемент, используемый в качестве дочернего элемента **параметр** элемент в определяемой моделью функции, которая принимает ссылку на **Person** сущности Тип:

```
<Function Name="GetYearsEmployed" ReturnType="Edm.Int32">
    <Parameter Name="instructor">
        <ReferenceType Type="SchoolModel.Person" />
    </Parameter>
    <DefiningExpression>
        Year(CurrentDateTime()) - Year(cast(instructor.HireDate as DateTime))
    </DefiningExpression>
</Function>
```

В следующем примере показан **ReferenceType** элемент, используемый в качестве дочернего элемента **ReturnType** элемент (Function) в определяемой моделью функции, которая возвращает ссылку на **Person** типа сущности:

```
<Function Name="GetPersonReference">
    <Parameter Name="p" Type="SchoolModel.Person" />
    <ReturnType>
        <ReferenceType Type="SchoolModel.Person" />
    </ReturnType>
    <DefiningExpression>
        REF(p)
    </DefiningExpression>
</Function>
```

Элемент ReferentialConstraint (CSDL)

Объект **ReferentialConstraint** элемент на языке определения концептуальной схемы (CSDL) определяет функциональные возможности, схожие с возможностями ограничения ссылочной целостности в реляционной базе данных. Свойство (или свойства) типа сущности могут ссылаться на ключ сущности в другом типе сущности также, как столбец (или столбцы) в таблице базы данных могут ссылаться на первичный ключ другой таблицы. Тип сущности, на который приведена ссылка называется **основного конца** ограничения. Тип сущности, который ссылается на основном конечном элементе называется **зависимого конца ограничения**.

Если внешний ключ, который предоставляется в одном типе сущности, ссылается на свойство другого типа сущности, **ReferentialConstraint** элемент определяет ассоциацию между двумя типами сущностей. Так как **ReferentialConstraint** элемент предоставляет связанные сведения о том, как два типа сущностей, соответствующий элемент AssociationSetMapping не является необходимым в языке определения сопоставлений (MSL). Связь между двумя типами сущностей, у которых нет представленных внешних

ключей должна иметь соответствующий **AssociationSetMapping** элемент для сопоставления сведений об ассоциации с источником данных.

Если внешний ключ недоступен для типа сущности **ReferentialConstraint** элемент можно определить только ключ к первичному ограничению первичного ключа между типом сущности и другого типа сущности.

Объект **ReferentialConstraint** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Участник (ровно один элемент)
- Зависимые (ровно один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

ReferentialConstraint элемент может иметь любое количество атрибутов annotation (настраиваемых атрибутов XML). Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ReferentialConstraint** элемента, используемого в качестве части определения **PublishedBy** ассоциации.

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Элемент ReturnType (Function) (язык CSDL)

ReturnType элемент (функция) в языке определения концептуальных схем (CSDL) определяет тип возвращаемого значения для функции, которая определяется в элементе функции. Тип возвращаемого значения также можно задать с помощью функции **ReturnType** атрибута.

Возвращаемым типом может быть любой **EdmSimpleType**, тип сущности, сложный тип, тип строки, ссылочный тип или коллекцию из одного из этих типов.

Тип возвращаемого значения функции, которые можно указать с помощью **тип** атрибут **ReturnType** элемент (Function), или с помощью одного из следующих дочерних элементов:

- CollectionType
- ReferenceType
- RowType

NOTE

Модель не пройдет проверку, если указан возвращаемый тип с обоими функции **тип** атрибут **ReturnType** (функция) и с помощью одного из дочерних элементов.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **ReturnType** элемент (Function).

Имя атрибута	Необходимо	Значение
ReturnType	Нет	Тип, возвращаемый функцией.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ReturnType** элемент (Function). Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере используется **функция** для определения функции, которая возвращает количество книги кондиционирования в годах печати. Обратите внимание, что тип возвращаемого значения определяется **тип** атрибут **ReturnType** элемент (Function).

```
<Function Name="GetYearsInPrint">
  <ReturnType Type=="Edm.Int32">
    <Parameter Name="book" Type="BooksModel.Book" />
    <DefiningExpression>
      Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
    </DefiningExpression>
  </ReturnType>
</Function>
```

Элемент ReturnType (FunctionImport) (язык CSDL)

ReturnType элемент (FunctionImport) на языке определения концептуальной схемы (CSDL) определяет тип возвращаемого значения для функции, которая определена в элемент FunctionImport. Тип возвращаемого значения также можно задать с помощью функции **ReturnType** атрибута.

Возвращаемым типом может быть любой коллекции типа сущности, сложный тип или **EdmSimpleType**.

Тип возвращаемого значения функции указан с **тип** атрибут **ReturnType** элемент (FunctionImport).

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **ReturnType** элемент (FunctionImport).

Имя атрибута	Необходимо	Значение
Type	Нет	Тип, возвращаемый функцией. Значение должно быть коллекция ComplexType, EntityType или EDMSimpleType.

Имя атрибута	Необходимо	Значение
Набор EntitySet	Нет	Если функция возвращает коллекцию сущностей типов, значение EntitySet должен представлять собой набор сущностей, которому принадлежит коллекция. В противном случае EntitySet атрибут не должен использоваться.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ReturnType** элемент (FunctionImport). Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере используется **FunctionImport**, возвращающий книги и издателей. Обратите внимание, что функция возвращает два результирующих набора и, следовательно, две **ReturnType** указаны элементы (FunctionImport).

```
<FunctionImport Name="GetBooksAndPublishers">
  <ReturnType Type=="Collection(BooksModel.Book )" EntitySet="Books">
  <ReturnType Type=="Collection(BooksModel.Publisher)" EntitySet="Publishers">
</FunctionImport>
```

Элемент RowType (язык CSDL)

Объект **RowType** элемента на языке определения концептуальной схемы (CSDL) определяет безымянную структуру в качестве параметра или тип возвращаемого значения для функции, определенной в концептуальной модели.

Объект **RowType** элемент может быть дочерним элементом следующих элементов:

- CollectionType
- Параметр
- ReturnType (функция)

Объект **RowType** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Свойство (один или несколько)
- Элементы annotation (ноль и более)

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **RowType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано определяемой моделью функции, который использует **CollectionType** элемента, чтобы указать, что функция возвращает коллекцию строк (как указано в **RowType** элемент).

```
<Function Name="LastNamesAfter">
  <Parameter Name="someString" Type="Edm.String" />
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="FirstName" Type="Edm.String" Nullable="false" />
        <Property Name="LastName" Type="Edm.String" Nullable="false" />
      </RowType>
    </CollectionType>
  </ReturnType>
  <DefiningExpression>
    SELECT VALUE ROW(p.FirstName, p.LastName)
    FROM SchoolEntities.People AS p
    WHERE p.LastName &gt;= somestring
  </DefiningExpression>
</Function>
```

Элемент Schema (CSDL)

Схемы элемент является корневым элементом определения концептуальной модели. Он содержит определения объектов, функций и контейнеров, из которых состоит концептуальная модель.

Схемы элемент может содержать ноль или более следующих дочерних элементов:

- Using
- EntityContainer
- EntityType
- EnumType
- Ассоциация
- ComplexType
- Функция

Объект **схемы** элемент может содержать ноль или один элемент заметки.

NOTE

Функция элементов и элементы annotation допускаются только в версии языка CSDL 2 и более поздних версий.

Схемы элемент использует **пространства имен** атрибут для определения пространства имен для типа сущности, сложный тип и объектов ассоциации в концептуальной модели. В пространстве имен не может быть двух объектов с одинаковым именем. Пространства имен может быть несколько **схемы** элементов и несколько файлов с расширением CSDL.

Пространство имен концептуальной модели отличается от пространства имен XML элемента **схемы** элемент. Пространство имен концептуальной модели (как определено **пространства имен** атрибут) — это логический контейнер для типов сущностей, сложные типы и типы ассоциаций. Пространство имен XML (обозначается **xmlns** атрибут) из **схемы** элемент является пространство имен по умолчанию для дочерних элементов и атрибутов из **схемы** элемент. Пространства имен XML вида <http://schemas.microsoft.com/ado/YYYY/MM/edm> (где YYYY и MM представляют год и месяц соответственно) зарезервированы для CSDL. Пользовательские элементы и атрибуты не могут присутствовать в пространствах имен такого вида.

Применимые атрибуты

В следующей таблице описываются атрибуты могут применяться к **схемы** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Пространство имен	Да	<p>Пространство имен концептуальной модели. Значение пространства имен атрибут используется, чтобы сформировать полное имя типа. Например если EntityType с именем <i>клиента</i> находится в пространстве имен Simple.Example.Model, а затем полностью квалифицированное имя EntityType является SimpleExampleModel.Customer. Следующие строки не может использоваться как значение для пространства имен атрибут: системы, временных, или Edm. Значение для пространства имен атрибут не может быть таким же, как значение для пространства имен атрибут в элементе SSDL-схеме.</p>
Alias	Нет	<p>Идентификатор, используемый в качестве имени пространства имен. Например если EntityType с именем <i>клиента</i> находится в пространстве имен Simple.Example.Model, а значение псевдоним атрибут модели, то в качестве полное доменное имя можно указать Model.Customer EntityType.</p>

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **схемы** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **схемы** элемент, содержащий **EntityContainer** элемент, двух **EntityType** элементов и один **ассоциации** элемент.

```

<Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"
    xmlns:cg="http://schemas.microsoft.com/ado/2009/11/codegeneration"
    xmlns:store="http://schemas.microsoft.com/ado/2009/11/edm/EntityStoreSchemaGenerator"
    Namespace="ExampleModel" Alias="Self">
    <EntityTypeContainer Name="ExampleModelContainer">
        <EntityTypeSet Name="Customers">
            EntityType="ExampleModel.Customer" />
        <EntityTypeSet Name="Orders" EntityType="ExampleModel.Order" />
        <AssociationSet
            Name="CustomerOrder"
            Association="ExampleModel.CustomerOrders">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Order" EntitySet="Orders" />
        </AssociationSet>
    </EntityTypeContainer>
    <EntityType Name="Customer">
        <Key>
            <PropertyRef Name="CustomerId" />
        </Key>
        <Property Type="Int32" Name="CustomerId" Nullable="false" />
        <Property Type="String" Name="Name" Nullable="false" />
        <NavigationProperty
            Name="Orders"
            Relationship="ExampleModel.CustomerOrders"
            FromRole="Customer" ToRole="Order" />
    </EntityType>
    <EntityType Name="Order">
        <Key>
            <PropertyRef Name="OrderId" />
        </Key>
        <Property Type="Int32" Name="OrderId" Nullable="false" />
        <Property Type="Int32" Name="ProductId" Nullable="false" />
        <Property Type="Int32" Name="Quantity" Nullable="false" />
        <NavigationProperty
            Name="Customer"
            Relationship="ExampleModel.CustomerOrders"
            FromRole="Order" ToRole="Customer" />
        <Property Type="Int32" Name="CustomerId" Nullable="false" />
    </EntityType>
    <Association Name="CustomerOrders">
        <End Type="ExampleModel.Customer"
            Role="Customer" Multiplicity="1" />
        <End Type="ExampleModel.Order"
            Role="Order" Multiplicity="*" />
        <ReferentialConstraint>
            <Principal Role="Customer">
                <PropertyRef Name="CustomerId" />
            </Principal>
            <Dependent Role="Order">
                <PropertyRef Name="CustomerId" />
            </Dependent>
        </ReferentialConstraint>
    </Association>
</Schema>

```

Элемент TypeRef (CSDL)

TypeRef элемент на языке определения концептуальной схемы (CSDL) предоставляет ссылку на существующий именованный тип. **TypeRef** элемент может быть дочерним элементом элемента `CollectionType`, который используется для указания, что функция имеет коллекцию в качестве параметра или возвращаемого типа.

Объект **TypeRef** элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **TypeRef** элемент. Обратите внимание, что **DefaultValue**, **MaxLength**, **FixedLength**, **точности**, **масштабирования**, **Юникод**, и **параметры сортировки** атрибуты применимы только к **EDMSimpleTypes**.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Type	Нет	Имя типа, на который дается ссылка.
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли свойство иметь значение null. [!NOTE]
> В языке CSDL v1 должен иметь свойство сложного типа <code>Nullable="False"</code> .		
DefaultValue	Нет	Значение свойства по умолчанию.
MaxLength	Нет	Максимальная длина значения свойства.
FixedLength	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка фиксированной длины.
Точность	Нет	Точность значения свойства.
Масштаб	Нет	Масштаб значения свойства.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .
Юникод	Нет	Значение true , или False в зависимости от того, будет ли значение свойства храниться как строка Юникода.
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **CollectionType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано определяемой моделью функции, который использует **TypeRef** элемент (как дочерний **CollectionType** элемент) для указания, что функция принимает коллекцию **Отдел** типов сущностей.

```
<Function Name="GetAvgBudget">
    <Parameter Name="Departments">
        <CollectionType>
            <TypeRef Type="SchoolModel.Department"/>
        </CollectionType>
    </Parameter>
    <ReturnType Type="Collection(Edm.Decimal)"/>
    <DefiningExpression>
        SELECT VALUE AVG(d.Budget) FROM Departments AS d
    </DefiningExpression>
</Function>
```

Элемент Using (CSDL)

Using элемента на языке определения концептуальной схемы (CSDL) импортирует содержимое концептуальной модели, который существует в другом пространстве имен. Установив значение **пространства имен** атрибут, можно ссылаться на типы сущностей, сложные типы и типы ассоциаций, которые определены в другой концептуальной модели. Более одного **Using** элемент может быть дочерним элементом **схемы** элемент.

NOTE

Using элемента на языке CSDL не работает абсолютно идентично **с помощью** инструкции на языке программирования. Путем импорта пространства имен **с помощью** инструкции на языке программирования, вы не влияют на объекты в исходном пространстве имен. В языке CSDL импортированное пространство имен может содержать тип сущности, производный от типа сущности в исходном пространстве имен. Это может влиять на наборы сущностей, объявленные в исходном пространстве имен.

Using элемент может иметь следующие дочерние элементы:

- Документация (ноль или один допустимое количество элементов)
- Элементы annotation (ноль или более допустимое количество элементов)

Применимые атрибуты

В следующей таблице описываются атрибуты могут применяться к **Using** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Пространство имен	Да	Имя импортируемого пространства имен.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Alias	Да	Идентификатор, используемый в качестве имени пространства имен. Хотя этот атрибут является обязательным, необязательно, чтобы он использовался вместо имени пространства имен при указании имен объектов.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **Using** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере демонстрируется **Using** элемента, используемого для импорта пространства имен, определенный в другом. Обратите внимание, что пространство имен для **схемы** является элемент, показанный `BooksModel`. `Address` Свойство `Publisher` **EntityType** — это сложный тип, который определен в `ExtendedBooksModel` пространства имен (импортированы с **Using** элемент).

```
<Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"
    xmlns:cg="http://schemas.microsoft.com/ado/2009/11/codegeneration"
    xmlns:store="http://schemas.microsoft.com/ado/2009/11/edm/EntityStoreSchemaGenerator"
    Namespace="BooksModel" Alias="Self">

    <Using Namespace="BooksModel.Extended" Alias="BMExt" />

    <EntityContainer Name="BooksContainer" >
        <EntityType Name="Publishers" EntityType="BooksModel.Publisher" />
    </EntityContainer>

    <EntityType Name="Publisher">
        <Key>
            <PropertyRef Name="Id" />
        </Key>
        <Property Type="Int32" Name="Id" Nullable="false" />
        <Property Type="String" Name="Name" Nullable="false" />
        <Property Type="BMExt.Address" Name="Address" Nullable="false" />
    </EntityType>

</Schema>
```

Атрибуты примечаний (CSDL)

Атрибуты annotation в языке CSDL представляют собой настраиваемые атрибуты XML в концептуальной модели. Атрибуты annotation должны не только иметь допустимую структуру XML, но и соответствовать следующим требованиям:

- Атрибуты annotation не должны находиться ни в одном пространстве имен XML, которое зарезервировано для языка CSDL.
- К данному конкретному элементу языка CSDL может применяться несколько атрибутов annotation.
- Полные имена любых двух атрибутов annotation не должны совпадать.

Атрибуты annotation могут использоваться для предоставления дополнительных метаданных об элементах в концептуальной модели. Метаданные, содержащиеся в элементах annotation может осуществляться во

время выполнения с помощью классов в пространстве имен System.Data.Metadata.Edm.

Пример

В следующем примере показан **EntityType** элемент с атрибутом annotation (**CustomAttribute**). В следующем примере показано применение элемента annotation к элементам типа сущности.

```
<Schema Namespace="SchoolModel" Alias="Self"
    xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
    <EntityContainer Name="SchoolEntities" annotation:LazyLoadingEnabled="true">
        <EntitySet Name="People" EntityType="SchoolModel.Person" />
    </EntityContainer>
    <EntityType Name="Person" xmlns:p="http://CustomNamespace.com"
        p:CustomAttribute="Data here.">
        <Key>
            <PropertyRef Name="PersonID" />
        </Key>
        <Property Name="PersonID" Type="Int32" Nullable="false"
            annotation:StoreGeneratedPattern="Identity" />
        <Property Name="LastName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="FirstName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="HireDate" Type="DateTime" />
        <Property Name="EnrollmentDate" Type="DateTime" />
        <p:CustomElement>
            Custom metadata.
        </p:CustomElement>
    </EntityType>
</Schema>
```

Следующий код извлекает метаданные из атрибута annotation и выводит их на консоль:

```
EdmItemCollection collection = new EdmItemCollection("School.csdl");
MetadataWorkspace workspace = new MetadataWorkspace();
workspace.RegisterItemCollection(collection);
EdmType contentType;
workspace.TryGetType("Person", "SchoolModel", DataSpace.CSpace, out contentType);
if (contentType.MetadataProperties.Contains("http://CustomNamespace.com:CustomAttribute"))
{
    MetadataProperty annotationProperty =
        contentType.MetadataProperties["http://CustomNamespace.com:CustomAttribute"];
    object annotationValue = annotationProperty.Value;
    Console.WriteLine(annotationValue.ToString());
}
```

Вышеуказанный код предполагает, что файл `School.csdl` находится в выходном каталоге проекта, а в проект добавлены инструкции `Imports` и `Using`:

```
using System.Data.Metadata.Edm;
```

Элементы примечаний (CSDL)

Элементы annotation в языке CSDL представляют собой настраиваемые элементы XML в концептуальной модели. Элементы annotation должны не только иметь допустимую структуру XML, но и соответствовать следующим требованиям:

- Элементы annotation не должны находиться в каком-либо пространстве имен XML, которое зарезервировано для языка CSDL.

- У данного элемента языка CSDL может быть больше одного дочернего элемента annotation.
- Полные имена любых двух элементов annotation не должны совпадать.
- Элементы annotation должны находиться после всех остальных дочерних элементов в данном элементе CSDL.

Элементы annotation могут использоваться для предоставления дополнительных метаданных об элементах в концептуальной модели. Начиная с .NET Framework версии 4, метаданные, содержащиеся в элементах annotation может осуществляться во время выполнения с помощью классов в пространстве имен System.Data.Metadata.Edm.

Пример

В следующем примере показан **EntityType** элемент с элементом annotation (**CustomElement**). В следующем примере показано применение атрибута annotation к элементам типа сущности.

```
<Schema Namespace="SchoolModel" Alias="Self"
    xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
    <EntityContainer Name="SchoolEntities" annotation:LazyLoadingEnabled="true">
        <EntityType Name="Person" EntityType="SchoolModel.Person" />
    </EntityContainer>
    <EntityType Name="Person" xmlns:p="http://CustomNamespace.com"
        p:CustomAttribute="Data here.">
        <Key>
            <PropertyRef Name="PersonID" />
        </Key>
        <Property Name="PersonID" Type="Int32" Nullable="false"
            annotation:StoreGeneratedPattern="Identity" />
        <Property Name="LastName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="FirstName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="HireDate" Type="DateTime" />
        <Property Name="EnrollmentDate" Type="DateTime" />
        <p:CustomElement
            Custom metadata.
        </p:CustomElement>
    </EntityType>
</Schema>
```

Следующий код извлекает метаданные из элемента annotation и выводит их на консоль.

```
EdmItemCollection collection = new EdmItemCollection("School.csdl");
MetadataWorkspace workspace = new MetadataWorkspace();
workspace.RegisterItemCollection(collection);
EdmType contentType;
workspace.TryGetType("Person", "SchoolModel", DataSpace.CSpace, out contentType);
if (contentType.MetadataProperties.Contains("http://CustomNamespace.com:CustomElement"))
{
    MetadataProperty annotationProperty =
        contentType.MetadataProperties["http://CustomNamespace.com:CustomElement"];
    object annotationValue = annotationProperty.Value;
    Console.WriteLine(annotationValue.ToString());
}
```

Вышеприведенный код предполагает, что файл School.csdl находится в выходном каталоге проекта, а в проект добавлены инструкции Imports И Using .

```
using System.Data.Metadata.Edm;
```

Типы концептуальной модели (CSDL)

Язык определения концептуальной схемы (CSDL) поддерживает набор абстрактных примитивных типов данных, называется **EDMSimpleTypes**, которые определяют свойства в концептуальной модели.

EDMSimpleTypes являются посредниками для примитивных типов данных, которые поддерживаются в среде хранения или размещения.

В приведенной ниже таблице представлены примитивные типы данных, поддерживаемые CSDL. В таблице также перечислены аспекты, которые могут применяться к каждому **EDMSimpleType**.

EDMSIMPLETYPE	ОПИСАНИЕ	ПРИМЕНИМЫЕ АСПЕКТЫ
Edm.Binary	Содержит двоичные данные.	MaxLength, FixedLength, Nullable, Default
Edm.Boolean	Содержит значение true или false .	Nullable, Default
Edm.Byte	Содержит 8-битное целое значение без знака.	Precision, Nullable, Default
Edm.DateTime	Представляет дату и время.	Precision, Nullable, Default
Edm.DateTimeOffset	Возвращает дату и время в виде смещения в минутах от времени GMT.	Precision, Nullable, Default
Edm.Decimal	Содержит точное числовое значение с заданной точностью и масштабом.	Precision, Nullable, Default
Edm.Double	Содержит число с плавающей запятой с точностью до 15 цифр	Precision, Nullable, Default
Edm.Float	Содержит число с плавающей запятой с точностью до 7 знаков.	Precision, Nullable, Default
Edm.Guid	Содержит уникальный 16-битный идентификатор.	Precision, Nullable, Default
Edm.Int16	Содержит 16-разрядное целое значение со знаком.	Precision, Nullable, Default
Edm.Int32	Содержит 32-разрядное целое значение со знаком.	Precision, Nullable, Default
Edm.Int64	Содержит 64-разрядное целое значение со знаком.	Precision, Nullable, Default
Edm.SByte	Содержит 8-разрядное целое значение со знаком.	Precision, Nullable, Default
Edm.String	Содержит символьные данные.	Unicode, FixedLength, MaxLength, Collation, Precision, Nullable, Default
Edm.Time	Содержит время дня.	Precision, Nullable, Default

EDMSIMPLETYPE	ОПИСАНИЕ	ПРИМЕНИМЫЕ АСПЕКТЫ
Edm.Geography		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyPoint		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyLineString		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyPolygon		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyMultiPoint		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyMultiLineString		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyMultiPolygon		Допускающие значение NULL, по умолчанию, SRID
Edm.GeographyCollection		Допускающие значение NULL, по умолчанию, SRID
Edm.Geometry		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryPoint		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryLineString		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryPolygon		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryMultiPoint		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryMultiLineString		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryMultiPolygon		Допускающие значение NULL, по умолчанию, SRID
Edm.GeometryCollection		Допускающие значение NULL, по умолчанию, SRID

Области (CSDL)

Аспекты в языке CSDL представляют ограничения для свойств типов сущности и сложных типов. Аспекты выглядят как XML-атрибуты в следующих элементах CSDL:

- Свойство.

- TypeRef
- Параметр

В следующей таблице описываются аспекты, поддерживаемые в CSDL. Все аспекты являются необязательными. Некоторые перечисленные ниже аспекты используются платформой Entity Framework, при создании базы данных из концептуальной модели.

NOTE

Сведения о типах данных в концептуальной модели см. в разделе типов концептуальной модели (CSDL).

АСПЕКТ	ОПИСАНИЕ	ПРИМЕНЕНИЕ	ИСПОЛЬЗУЕТСЯ ДЛЯ СОЗДАНИЯ БАЗЫ ДАННЫХ	ИСПОЛЬЗУЕТСЯ СРЕДОЙ ВЫПОЛНЕНИЯ
Параметры сортировки	Задает последовательность сортировки, которая будет использоваться при выполнении операций сравнения и упорядочивания для значений свойств.	Edm.String	Да	Нет
ConcurrencyMode	Указывает, что значение свойства должно использоваться в проверках оптимистического управления параллелизмом.	Все EDMSimpleType свойства	Нет	Да
Default	Задает значение по умолчанию для свойства в случае, если при создании экземпляра не было задано значение.	Все EDMSimpleType свойства	Да	Да
FixedLength	Указывает, может ли изменяться длина значения свойства.	Edm.Binary , Edm.String	Да	Нет
MaxLength	Указывает максимальную длину значения свойства.	Edm.Binary , Edm.String	Да	Нет
Допускающий значение NULL	Указывает, может ли свойство иметь null значение.	Все EDMSimpleType свойства	Да	Да

АСПЕКТ	ОПИСАНИЕ	ПРИМЕНЕНИЕ	ИСПОЛЬЗУЕТСЯ ДЛЯ СОЗДАНИЯ БАЗЫ ДАННЫХ	ИСПОЛЬЗУЕТСЯ СРЕДОЙ ВЫПОЛНЕНИЯ
Точность	Для свойств типа десятичное , количество десятичных разрядов, может иметь значение свойства. Для свойств типа время , DateTime , и DateTimeOffset , указывает количество цифр дробной части секунд значения свойства.	Edm.DateTime , Edm.DateTimeOffset , Edm.Decimal , Edm.Time	Да	Нет
Масштаб	Задает число цифр справа от десятичной запятой в значении свойства.	Edm.Decimal	Да	Нет
SRID	Указывает идентификатор системы пространственных ссылок системы. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .	Edm.Geography , Edm.GeographyPoint , Edm.GeographyLineString , Edm.GeographyPolygon , Edm.GeographyMultiPoint , Edm.GeographyMultiLineString , Edm.GeographyMultiPolygon , Edm.GeographyCollection , Edm.Geometry , Edm.GeometryPoint , Edm.GeometryLineString , Edm.GeometryPolygon , Edm.GeometryMultiPoint , Edm.GeometryMultiLineString , Edm.GeometryMultiPolygon , Edm.GeometryCollection	Нет	Да
Юникод	Указывает, будет ли значение свойства храниться в Юникоде.	Edm.String	Да	Да

NOTE

При создании базы данных из концептуальной модели, мастер создания базы данных будет распознавать значение **StoreGeneratedPattern** атрибут **свойство** элемент, если он находится в следующие пространство имен: <http://schemas.microsoft.com/ado/2009/02/edm/annotation>. Поддерживаемые значения для атрибута **удостоверений** и **вычисленные**. Значение **удостоверений** создаст столбец базы данных со значением идентификатора, который создается в базе данных. Значение **вычисленные** создаст столбец со значением, вычисляемым в базе данных.

Пример

В следующем примере показывается применение аспектов к свойствам типа сущности:

```
<EntityType Name="Product">
  <Key>
    <PropertyRef Name="ProductId" />
  </Key>
  <Property Type="Int32"
    Name="ProductId" Nullable="false"
    a:StoreGeneratedPattern="Identity"
    xmlns:a="http://schemas.microsoft.com/ado/2009/02/edm/annotation" />
  <Property Type="String"
    Name="ProductName"
    Nullable="false"
    MaxLength="50" />
  <Property Type="String"
    Name="Location"
    Nullable="true"
    MaxLength="25" />
</EntityType>
```

Спецификация MSL

02.10.2018 • 66 minutes to read • [Edit Online](#)

Язык определения сопоставлений (MSL) — это язык на основе XML, описывающий сопоставление между концептуальной моделью и моделью хранения приложения Entity Framework.

В приложении Entity Framework метаданные сопоставления загружаются из MSL-файла (написанного на языке MSL) во время сборки. Платформа Entity Framework метаданные сопоставления используются во время выполнения для преобразования запросов к концептуальной модели для конкретного хранилища команды.

Entity Framework Designer (конструктор EF) хранит сведения о сопоставлении в EDMX-файла во время разработки. Во время построения конструктор сущностей использует сведения в EDMX-файла, создаваемого MSL-файл, необходимый Entity Framework во время выполнения

Имена всех типов концептуальной модели или модели хранения, которые упоминаются в языке MSL, должны указываться вместе с именами соответствующих пространств имен. Сведения об имени пространства имен концептуальной модели, см. в разделе [спецификация языка CSDL](#). Сведения об имени пространства имен модели хранения, см. в разделе [спецификация SSDL](#).

Версии MSL различаются по пространствам имен XML.

ВЕРСИЯ MSL	ПРОСТРАНСТВО ИМЕН XML
Язык MSL v1	urn: schemas-microsoft-com:windows:storage:mapping:CS
Язык MSL v2	http://schemas.microsoft.com/ado/2008/09/mapping/cs
Язык MSL v3	http://schemas.microsoft.com/ado/2009/11/mapping/cs

Элемент Alias (язык MSL)

Псевдоним элемент в язык определения сопоставлений (MSL) — это дочерний элемент элемента сопоставления, который используется для определения псевдонимов для концептуальной модели и хранения пространства имен модели. Имена всех типов концептуальной модели или модели хранения, которые упоминаются в языке MSL, должны указываться вместе с именами соответствующих пространств имен. Сведения об имени пространства имен концептуальной модели см. в разделе элемент схемы (CSDL). Сведения об имени пространства имен модели хранения см. в разделе элемент схемы (SSDL).

Псевдоним элемент не может иметь дочерние элементы.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **псевдоним** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Key	Да	Псевдоним для пространства имен, который задается параметром значение атрибута.

Имя атрибута	Необходимо	Значение
Значение	Да	Пространство имен, для которого значение ключ элемент является псевдонимом.

Пример

В следующем примере показан **псевдоним** элемент, который определяет псевдоним, **с**, для типов, определенных в концептуальной модели.

```
<Mapping Space="C-S"
  xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
      <MappingFragment StoreEntitySet="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="Credits" ColumnName="Credits" />
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  <EntitySetMapping Name="Departments">
    <EntityTypeMapping TypeName="c.Department">
      <MappingFragment StoreEntitySet="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Budget" ColumnName="Budget" />
        <ScalarProperty Name="StartDate" ColumnName="StartDate" />
        <ScalarProperty Name="Administrator" ColumnName="Administrator" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
</EntityContainerMapping>
</Mapping>
```

Элемент AssociationEnd (MSL)

AssociationEnd элемент в язык определения сопоставлений (MSL) используется при сопоставлении функций изменения типа сущности в концептуальной модели с хранимыми процедурами в основной базе данных. Если хранимая процедура изменения принимает параметр, значение которого содержится в свойстве ассоциации, **AssociationEnd** сопоставляет значение свойства параметра. Дополнительные сведения см. в приведенных ниже примерах.

Дополнительные сведения о сопоставлении функций изменения типов сущностей с хранимыми процедурами см. в разделе элемент **ModificationFunctionMapping** (MSL) и пошаговое руководство: сопоставление сущности с хранимых процедур.

AssociationEnd элемент может иметь следующие дочерние элементы:

- **ScalarProperty**

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **AssociationEnd** элемент.

Имя атрибута	Необходимо	Значение
AssociationSet	Да	Имя сопоставляемой ассоциации.
From	Да	Значение FromRole атрибут свойства навигации, которое соответствует сопоставляемой ассоциации. Дополнительные сведения см. в разделе элемент NavigationProperty (CSDL).
Задача	Да	Значение ToRole атрибут свойства навигации, которое соответствует сопоставляемой ассоциации. Дополнительные сведения см. в разделе элемент NavigationProperty (CSDL).

Пример

Рассмотрим следующий тип сущности в концептуальной модели:

```
<EntityType Name="Course">
  <Key>
    <PropertyRef Name="CourseID" />
  </Key>
  <Property Type="Int32" Name="CourseID" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" MaxLength="100"
            FixedLength="false" Unicode="true" />
  <Property Type="Int32" Name="Credits" Nullable="false" />
  <NavigationProperty Name="Department"
    Relationship="SchoolModel.FK_Course_Department"
    FromRole="Course" ToRole="Department" />
</EntityType>
```

Предположим, имеется следующая хранимая процедура:

```
CREATE PROCEDURE [dbo].[UpdateCourse]
  @CourseID int,
  @Title nvarchar(50),
  @Credits int,
  @DepartmentID int
  AS
    UPDATE Course SET Title=@Title,
                     Credits=@Credits,
                     DepartmentID=@DepartmentID
    WHERE CourseID=@CourseID;
```

Чтобы сопоставить функцию обновления `Course` сущности в хранимую процедуру, необходимо указать значение для **DepartmentID** параметра. Значение для `DepartmentID` не соответствует свойству типа сущности. Оно содержится в независимом сопоставлении с сопоставлением, показанным ниже:

```

<AssociationSetMapping Name="FK_Course_Department"
    TypeName="SchoolModel.FK_Course_Department"
    StoreEntitySet="Course">
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <EndProperty Name="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </EndProperty>
</AssociationSetMapping>

```

В следующем коде показан **AssociationEnd** элемент, используемый для сопоставления **DepartmentID** свойство **FK_курс_отдел** связь с **UpdateCourse** хранимой процедуры (к которому функция обновления **курс** тип сущности сопоставлен):

```

<EntityTypeMapping TypeName="SchoolModel.Course">
    <MappingFragment StoreEntitySet="Course">
        <ScalarProperty Name="Credits" ColumnName="Credits" />
        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </MappingFragment>
</EntityTypeMapping>
<EntityTypeMapping TypeName="SchoolModel.Course">
    <ModificationFunctionMapping>
        <UpdateFunction FunctionName="SchoolModel.Store.UpdateCourse">
            <AssociationEnd AssociationSet="FK_Course_Department"
                From="Course" To="Department">
                <ScalarProperty Name="DepartmentID"
                    ParameterName="DepartmentID"
                    Version="Current" />
            </AssociationEnd>
            <ScalarProperty Name="Credits" ParameterName="Credits"
                Version="Current" />
            <ScalarProperty Name="Title" ParameterName="Title"
                Version="Current" />
            <ScalarProperty Name="CourseID" ParameterName="CourseID"
                Version="Current" />
        </UpdateFunction>
    </ModificationFunctionMapping>
</EntityTypeMapping>
</EntityTypeMapping>

```

Элемент AssociationSetMapping (MSL)

AssociationSetMapping элемент в язык определения сопоставлений (MSL) определяет сопоставление между ассоциацией в концептуальной модели и столбцами таблицы в основной базе данных.

Сопоставления в концептуальной модели — это типы, свойства которых представляют столбцы первичного и внешнего ключа в основной базе данных. **AssociationSetMapping** элемент использует два элемента **EndProperty**, чтобы определить сопоставления между свойствами типов сопоставлений и столбцами в базе данных. Можно определить условия сопоставления в элемент Condition. Сопоставление insert, update и delete функции для ассоциаций с хранимыми процедурами в базе данных с элементом ModificationFunctionMapping. Определите только для чтения сопоставление между ассоциациями и столбцами таблицы с помощью строк Entity SQL в QueryView-элемент.

NOTE

Если для ассоциации в концептуальной модели определено ссылочное ограничение, ассоциации необходимо сопоставить с **AssociationSetMapping** элемент. Если **AssociationSetMapping** элемент присутствует ассоциации со справочным ограничением, то сопоставление, определенное в **AssociationSetMapping** элемент будет игнорироваться. Дополнительные сведения см. в разделе элемент ReferentialConstraint (CSDL).

AssociationSetMapping элемент может иметь следующие дочерние элементы

- QueryView (ноль или один)
- EndProperty (ноль или два)
- Условие (ноль и более)
- ModificationFunctionMapping (ноль или один)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **AssociationSetMapping** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя набора ассоциаций концептуальной модели, с которым выполняется сопоставление.
Имя типа	Нет	Имя типа ассоциации концептуальной модели из полного пространства имен, с которым выполняется сопоставление.
StoreEntitySet	Нет	Имя таблицы, с которой производится сопоставление.

Пример

В следующем примере показан **AssociationSetMapping** элемента, в котором **FK_курс_отдел** набор ассоциаций в концептуальной модели сопоставляется с **Курс** таблицы в базе данных. Сопоставления между свойствами типов сопоставлений и столбцами таблицы указываются в дочерних **EndProperty** элементов.

```
<AssociationSetMapping Name="FK_Course_Department"
    TypeName="SchoolModel.FK_Course_Department"
    StoreEntitySet="Course">
    <EndProperty Name="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
</AssociationSetMapping>
```

Элемент ComplexProperty (MSL)

Объект **ComplexProperty** элемент в языке определения сопоставлений (MSL) определяет сопоставление между свойством сложного типа в концептуальной модели сущности типа и столбцами таблицы в основной базе данных. Сопоставления столбца свойств указываются в дочерних элементах **ScalarProperty**.

ComplexType свойство может иметь следующие дочерние элементы:

- ScalarProperty (ноль и более)
- **ComplexProperty** (ноль и более)
- ComplexTypeMapping (ноль и более)
- Условие (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **ComplexProperty** элементу:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя сложного свойства типа сущности в концептуальной модели, с которым выполняется сопоставление.
Имя типа	Нет	Полное имя пространства имен типа свойства концептуальной модели.

Пример

Следующий пример основан на модели School. К концептуальной модели добавлен следующий сложный тип.

```
<ComplexType Name="FullName">
  <Property Type="String" Name="LastName"
    Nullable="false" MaxLength="50"
    FixedLength="false" Unicode="true" />
  <Property Type="String" Name="FirstName"
    Nullable="false" MaxLength="50"
    FixedLength="false" Unicode="true" />
</ComplexType>
```

LastNames и **FirstNames** свойства **Person** типа сущности были заменены одним сложным свойством **Имя**:

```
<EntityType Name="Person">
  <Key>
    <PropertyRef Name="PersonID" />
  </Key>
  <Property Name="PersonID" Type="Int32" Nullable="false"
    annotation:StoreGeneratedPattern="Identity" />
  <Property Name="HireDate" Type="DateTime" />
  <Property Name="EnrollmentDate" Type="DateTime" />
  <Property Name="Name" Type="SchoolModel.FullName" Nullable="false" />
</EntityType>
```

Следующей спецификации MSL показан **ComplexProperty** элемент, используемый для сопоставления **Имя** свойства со столбцами в основной базе данных:

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate" ColumnName="EnrollmentDate" />
            <ComplexProperty Name="Name" TypeName="SchoolModel.FullName">
                <ScalarProperty Name="FirstName" ColumnName="FirstName" />
                <ScalarProperty Name="LastName" ColumnName="LastName" />
            </ComplexProperty>
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

Элемент ComplexTypeMapping (MSL)

ComplexTypeMapping элементом в язык определения сопоставлений (MSL) — это дочерний элемент ResultMapping и определяет сопоставление между импортом функции в концептуальной модели и хранимой процедурой в базовом База данных, при выполнении следующих условий:

- Импорт функции возвращает концептуальный сложный тип.
- Имена столбцов, возвращаемые хранимой процедурой, не соответствуют в точности именам свойств в сложном типе.

По умолчанию сопоставление между столбцами, возвращаемыми хранимой процедурой, и сложным типом основано на именах столбцов и свойств. Если имена столбцов не точно совпадают с именами свойств, необходимо использовать **ComplexTypeMapping** элемент для определения сопоставления. Пример сопоставления по умолчанию см. в разделе элемент FunctionImportMapping (MSL).

ComplexTypeMapping элемент может иметь следующие дочерние элементы:

- ScalarProperty (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **ComplexTypeMapping** элементу.

Имя атрибута	Необходимо	Значение
Имя типа	Да	Имя сопоставляемого сложного типа с указанием пространства имен.

Пример

Рассмотрим следующую хранимую процедуру:

```

CREATE PROCEDURE [dbo].[GetGrades]
    @student_Id int
    AS
    SELECT      EnrollmentID as enroll_id,
                Grade as grade,
                CourseID as course_id,
                StudentID as student_id
    FROM        dbo.StudentGrade
    WHERE       StudentID = @student_Id

```

Следует также обратить внимание на следующий сложный тип концептуальной модели:

```

<ComplexType Name="GradeInfo">
    <Property Type="Int32" Name="EnrollmentID" Nullable="false" />
    <Property Type="Decimal" Name="Grade" Nullable="true"
        Precision="3" Scale="2" />
    <Property Type="Int32" Name="CourseID" Nullable="false" />
    <Property Type="Int32" Name="StudentID" Nullable="false" />
</ComplexType>

```

Для создания импорта функции, возвращающей экземпляры предыдущего сложного типа, сопоставление между столбцами, возвращаемыми хранимой процедурой и тип сущности должен быть определен в **ComplexTypeMapping** элемент:

```

<FunctionImportMapping FunctionImportName="GetGrades"
    FunctionName="SchoolModel.Store.GetGrades" >
    <ResultMapping>
        <ComplexTypeMapping TypeName="SchoolModel.GradeInfo">
            <ScalarProperty Name="EnrollmentID" ColumnName="enroll_id"/>
            <ScalarProperty Name="CourseID" ColumnName="course_id"/>
            <ScalarProperty Name="StudentID" ColumnName="student_id"/>
            <ScalarProperty Name="Grade" ColumnName="grade"/>
        </ComplexTypeMapping>
    </ResultMapping>
</FunctionImportMapping>

```

Элемент Condition (MSL)

Условие элемент в языке определения сопоставлений (MSL) задает условия для сопоставления между концептуальной моделью и основной базе данных. Сопоставление, определенные в XML-узла является допустимым, если все условия, заданные в дочерних **условие** элементов, будут выполнены. В противном случае сопоставление недействительно. Например, если элемент MappingFragment содержит один или несколько **условие** дочерние элементы, то сопоставление, определенное в **MappingFragment** узел будет допустимым, если все условия дочернего элемента **условие** элементы будут выполнены.

Каждое условие может применяться либо **имя** (имя свойства сущности концептуальной модели, определяемое **имя** атрибут), или **ColumnName** (имя столбца в базе данных, определяемое **ColumnName** атрибут). Когда **имя** атрибут имеет значение, условие проверяется по значению свойства сущности. Когда **ColumnName** атрибут имеет значение, условие проверяется значение столбца. Только один из **имя** или **ColumnName** атрибут может быть указан в **условие** элемент.

NOTE

Когда **условие** элемент используется внутри элемента FunctionImportMapping только **имя** атрибут неприменим.

Условие элемент может быть дочерним элементом следующих элементов:

- AssociationSetMapping
- ComplexProperty
- EntitySetMapping
- MappingFragment
- EntityTypeMapping

Условие элемент может иметь дочерних элементов.

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **условие** элемент:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
ColumnName	Нет	Имя столбца таблицы, значение которого используется для оценки условия.
IsNull	Нет	Значение true, или False. Если значение равно True и значение столбца равно null , или если значение равно False и значение столбца не null , условие имеет значение true . В противном случае условие имеет значение false . IsNull и значение атрибуты нельзя использовать одновременно.
Значение	Нет	Значение, с которым сравнивается значение столбца. Если значения совпадают, условие верно. В противном случае условие имеет значение false . IsNull и значение атрибуты нельзя использовать одновременно.
Name	Нет	Имя свойства сущности концептуальной модели, значение которого используется для оценки условия. Этот атрибут неприменим Если условие элемент используется в элемент FunctionImportMapping.

Пример

В следующем примере показан **условие** элементы как дочерние элементы **MappingFragment** элементов. Когда **HireDate** не равно null и **EnrollmentDate** является null, то данные сопоставляются между **SchoolModel.Instructor** типа и **PersonID** и **HireDate** столбцы **Person** таблицы. Когда **EnrollmentDate** не равно null и **HireDate** является null, то данные сопоставляются между **SchoolModel.Student** типа и **PersonID** и **регистрации** столбцы **Person** таблицы.

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Person)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Instructor)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <Condition ColumnName="HireDate" IsNull="false" />
            <Condition ColumnName="EnrollmentDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Student)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="EnrollmentDate"
                ColumnName="EnrollmentDate" />
            <Condition ColumnName="EnrollmentDate" IsNull="false" />
            <Condition ColumnName="HireDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

Элемент DeleteFunction (MSL)

DeleteFunction в язык определения сопоставлений (MSL) сопоставляет функцию удаления типа сущности или ассоциации в концептуальной модели с хранимой процедурой в основной базе данных. Хранимые процедуры, с которыми сопоставляются функции изменения, должны объявляться в модели хранения. Дополнительные сведения см. в разделе элемента функция (SSDL).

NOTE

Если не выполнено сопоставление всех трех вставки, обновления или удаления типов сущностей с хранимыми процедурами, то несопоставленные операции завершится ошибкой, если во время выполнения и UpdateException возникает исключение.

Элемент DeleteFunction, применяемый к EntityTypeMapping

При применении к элементу EntityTypeMapping **DeleteFunction** сопоставляет функцию удаления типа сущности в концептуальной модели с хранимой процедурой.

DeleteFunction элемент может иметь следующие дочерние элементы при применении к **EntityTypeMapping** элемент:

- AssociationEnd (ноль и более)
- ComplexProperty (ноль и более)
- ScalarProperty (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **DeleteFunction** элемента при его применении к **EntityTypeMapping** элемент.

Имя атрибута	Необходимо	Значение
FunctionName	Да	Имя хранимой процедуры (с указанием пространства имен), с которой сопоставляется функция удаления. Хранимая процедура должна объявляться в модели хранения.
RowsAffectedParameter	Нет	Имя выходного параметра, возвращающего количество обработанных строк.

Пример

В следующем примере, основана на модели School и показывает **DeleteFunction** элемент сопоставления функции удаления **Person** тип сущности, **DeletePerson** Хранимая процедура. **DeletePerson** хранимая процедура объявлена в модели хранения.

```
<EntitySetMapping Name="People">
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="LastName" ColumnName="LastName" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="HireDate" ColumnName="HireDate" />
      <ScalarProperty Name="EnrollmentDate"
        ColumnName="EnrollmentDate" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <ModificationFunctionMapping>
      <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName" />
        <ScalarProperty Name="LastName" ParameterName="LastName" />
        <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
      </InsertFunction>
      <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate"
          Version="Current" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate"
          Version="Current" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName"
          Version="Current" />
        <ScalarProperty Name="LastName" ParameterName="LastName"
          Version="Current" />
        <ScalarProperty Name="PersonID" ParameterName="PersonID"
          Version="Current" />
      </UpdateFunction>
      <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
        <ScalarProperty Name="PersonID" ParameterName="PersonID" />
      </DeleteFunction>
    </ModificationFunctionMapping>
  </EntityTypeMapping>
</EntitySetMapping>
```

Элемент **DeleteFunction**, применяемый к **AssociationSetMapping**

При применении к элементу **AssociationSetMapping** **DeleteFunction** сопоставляет функцию удаления ассоциации в концептуальной модели с хранимой процедурой.

DeleteFunction элемент может иметь следующие дочерние элементы при применении к **AssociationSetMapping** элементу:

- EndProperty

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **DeleteFunction** элемента при его применении к **AssociationSetMapping** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
FunctionName	Да	Имя хранимой процедуры (с указанием пространства имен), с которой сопоставляется функция удаления. Хранимая процедура должна объявляться в модели хранения.
RowsAffectedParameter	Нет	Имя выходного параметра, возвращающего количество обработанных строк.

Пример

В следующем примере, основана на модели School и показывает **DeleteFunction** элемент, используемый для сопоставления функции удаления **CourseInstructor** взаимосвязи **DeleteCourseInstructor** хранимой процедуры. **DeleteCourseInstructor** хранимая процедура объявлена в модели хранения.

```
<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>
```

Элемент EndProperty (MSL)

EndProperty элемент в язык определения сопоставлений (MSL) определяет сопоставление между функцией изменения ассоциации концептуальной модели и основной базе данных или концом. Сопоставление

столбца свойств указано в дочерний элемент **ScalarProperty**.

Когда **EndProperty** элемент используется для определения сопоставления конца ассоциации концептуальной модели, он является дочерним элементом элемента **AssociationSetMapping**. Когда **EndProperty** элемент используется для определения сопоставления функции изменения ассоциации концептуальной модели, он является дочерним элементом **InsertFunction** или элемент **DeleteFunction**.

EndProperty элемент может иметь следующие дочерние элементы:

- **ScalarProperty** (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **EndProperty** элемент:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
name	Да	Имя сопоставляемого конца ассоциации.

Пример

В следующем примере показан **AssociationSetMapping** элемента, в котором **FK_курс_отдел** ассоциации в концептуальной модели сопоставляется с **Курс** таблицы в базе данных. Сопоставления между свойствами типов сопоставлений и столбцами таблицы указываются в дочерних **EndProperty** элементов.

```
<AssociationSetMapping Name="FK_Course_Department"
    TypeName="SchoolModel.FK_Course_Department"
    StoreEntitySet="Course">
    <EndProperty Name="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
</AssociationSetMapping>
```

Пример

В следующем примере показан **EndProperty** элемент сопоставления функции вставки и удаления ассоциации (**CourseInstructor**) с хранимыми процедурами в основной базе данных. Функции, с которыми выполняется сопоставление, объявляются в модели хранения.

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

Элемент EntityContainerMapping (MSL)

EntityContainerMapping в язык определения сопоставлений (MSL) сопоставляет контейнер сущностей в концептуальной модели контейнер сущностей в модели хранения. **EntityContainerMapping** элемент является дочерним элемента сопоставления.

EntityContainerMapping элемент может иметь следующие дочерние элементы (в указанном порядке):

- EntitySetMapping (ноль и более)
- AssociationSetMapping (ноль и более)
- FunctionImportMapping (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntityContainerMapping** элемент.

Имя атрибута	Необходимо	Значение
StorageModelContainer	Да	Имя контейнера сущностей модели хранения, с которым выполняется сопоставление.
CdmEntityContainer	Да	Имя контейнера сущностей концептуальной модели, с которым выполняется сопоставление.

Имя атрибута	Необходимо	Значение
Generateupdateviews как	Нет	Значение true, или False. Если False , представления обновлений не создаются. Этот атрибут должно быть присвоено False при наличии сопоставление только для чтения, которое может стать недействительным, так как данные могут быть не выполнить обратное преобразование успешно. Значение по умолчанию — True .

Пример

В следующем примере показан **EntityContainerMapping** сопоставляет **SchoolModelEntities** контейнера (контейнер сущностей концептуальной модели) для **SchoolModelStoreContainer** контейнера (контейнер сущностей модели хранения):

```
<EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
        <MappingFragment StoreEntitySet="Course">
            <ScalarProperty Name="CourseID" ColumnName="CourseID" />
            <ScalarProperty Name="Title" ColumnName="Title" />
            <ScalarProperty Name="Credits" ColumnName="Credits" />
            <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        </MappingFragment>
    </EntityTypeMapping>
</EntityTypeMapping>
<EntityTypeMapping TypeName="c.Department">
    <MappingFragment StoreEntitySet="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Budget" ColumnName="Budget" />
        <ScalarProperty Name="StartDate" ColumnName="StartDate" />
        <ScalarProperty Name="Administrator" ColumnName="Administrator" />
    </MappingFragment>
</EntityTypeMapping>
</EntityTypeMapping>
</EntityContainerMapping>
```

Элемент EntitySetMapping (MSL)

EntitySetMapping задает элемент в языке MSL сопоставляет все типы в сущности концептуальной модели с наборами сущностей в модели хранения. Набор сущностей в концептуальной модели — это логический контейнер для экземпляров сущностей одного типа (и производных типов). Набор сущностей в модели хранения представляет таблицу или представление в основной базе данных. Набор сущностей концептуальной модели задается значение **имя** атрибут **EntitySetMapping** элемент. Таблица или представление определяется **StoreEntitySet** атрибут в каждый дочерний элемент **MappingFragment** или на **EntitySetMapping** сам элемент.

EntitySetMapping элемент может иметь следующие дочерние элементы:

- **EntityTypeMapping** (ноль и более)
- **QueryView** (ноль или один)
- **MappingFragment** (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntitySetMapping** элементу.

Имя атрибута	Необходимо	Значение
Name	Да	Имя сопоставляемого набора сущностей концептуальной модели.
TypeName 1	Нет	Имя сопоставляемого типа сущности концептуальной модели.
StoreEntitySet 1	Нет	Имя сопоставляемого набора сущностей модели хранения.
MakeColumnsDistinct	Нет	Значение true, или False в зависимости от того, возвращаются только уникальные строки. Если этот атрибут имеет значение True , generateupdateviews как атрибут элемента EntityContainerMapping должно быть присвоено False .

1 **TypeName** и **StoreEntitySet** атрибуты можно использовать вместо дочерние элементы **EntityTypeMapping** и **MappingFragment** для сопоставления один тип сущности с одной таблицей.

Пример

В следующем примере показан **EntitySetMapping** элемент, сопоставляющий три типа (базового типа и два производных типа) в **курсы** набор сущностей в концептуальной модели с тремя разными таблицами основной базы данных. Таблицы заданы **StoreEntitySet** атрибута в каждом **MappingFragment** элемент.

```
<EntitySetMapping Name="Courses">
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.Course)">
    <MappingFragment StoreEntitySet="Course">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      <ScalarProperty Name="Credits" ColumnName="Credits" />
      <ScalarProperty Name="Title" ColumnName="Title" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.OnlineCourse)">
    <MappingFragment StoreEntitySet="OnlineCourse">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="URL" ColumnName="URL" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.OnsiteCourse)">
    <MappingFragment StoreEntitySet="OnsiteCourse">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="Time" ColumnName="Time" />
      <ScalarProperty Name="Days" ColumnName="Days" />
      <ScalarProperty Name="Location" ColumnName="Location" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

Элемент EntityTypeMapping (MSL)

EntityTypeMapping элемент в язык определения сопоставлений (MSL) определяет сопоставление между

типов сущности в концептуальной модели и таблицы или представления в основной базе данных. Сведения о типах сущностей концептуальной модели и базовых таблиц базы данных или представления см. в разделе элемент Entity Type (CSDL) и элемент Entity Set (SSDL). Определяется тип сущности концептуальной модели, с которым выполняется сопоставление **Type Name** атрибут Entity Type Mapping элемент. Таблицы или представления, с которым выполняется сопоставление определяются **Store Entity Set** атрибут Mapping Fragment дочернего элемента.

Modification Function Mapping дочерний элемент может использоваться для сопоставления вставки, обновления или удаления функций типов сущностей с хранимыми процедурами в базе данных.

Entity Type Mapping элемент может иметь следующие дочерние элементы:

- Mapping Fragment (ноль и более)
- Modification Function Mapping (ноль или один)
- Scalar Property
- Условие

NOTE

Mapping Fragment и **Modification Function Mapping** элементы не могут быть дочерними элементами элемента **Entity Type Mapping** элемент, в то же время.

NOTE

Scalar Property и **условие** элементов может быть только дочерние элементы **Entity Type Mapping** элемент, когда он используется внутри элемента Function Import Mapping.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **Entity Type Mapping** элемент.

Имя атрибута	Необходимо	Значение
Имя типа	Да	Имя с указанием пространства имен типа сущности концептуальной модели, с которым выполняется сопоставление. Если тип является абстрактным или производным типом, значение должно быть равно <code>IsOfType(Namespace-qualified_type_name)</code>

Пример

В следующем примере показано элемент Entity Set Mapping с двумя дочерними **Entity Type Mapping** элементов. В первом **Entity Type Mapping** элемент, **SchoolModel.Person** сопоставляется с типом сущности **Person** таблицы. Во втором **Entity Type Mapping** элемент, поэтому функциональные возможности обновления **SchoolModel.Person** тип сопоставлен с хранимой процедурой, **UpdatePerson**, в базе данных .

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate" ColumnName="EnrollmentDate" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <ModificationFunctionMapping>
            <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
                <ScalarProperty Name="EnrollmentDate" ParameterName="EnrollmentDate"
                               Version="Current" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate"
                               Version="Current" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName"
                               Version="Current" />
                <ScalarProperty Name="LastName" ParameterName="LastName"
                               Version="Current" />
                <ScalarProperty Name="PersonID" ParameterName="PersonID"
                               Version="Current" />
            </UpdateFunction>
        </ModificationFunctionMapping>
    </EntityTypeMapping>
</EntitySetMapping>

```

Пример

В следующем примере показано сопоставление иерархии типа, в которой корневой тип является абстрактным. Обратите внимание на использование `IsOfType` синтаксис **TypeName** атрибуты.

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Person)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Instructor)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <Condition ColumnName="HireDate" IsNull="false" />
            <Condition ColumnName="EnrollmentDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Student)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="EnrollmentDate"
                           ColumnName="EnrollmentDate" />
            <Condition ColumnName="EnrollmentDate" IsNull="false" />
            <Condition ColumnName="HireDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

Элемент FunctionImportMapping (MSL)

FunctionImportMapping элемент в языке определения сопоставлений (MSL) определяет сопоставление

между импортом функции в концептуальной модели и хранимой процедурой или функцией в основной базе данных. Импорт функций должен быть объявлен в концептуальной модели, а хранимые процедуры должны быть объявлены в модели хранения. Дополнительные сведения см. в разделе элемент FunctionImport (CSDL) и функция элемент (SSDL).

NOTE

По умолчанию импорт функции возвращает тип сущности концептуальной модели или сложный тип. В этом случае имена столбцов, которые вернула основная хранимая процедура, должны точно соответствовать именам свойств в типе концептуальной модели. Если имена столбцов не точно совпадают с именами свойств, сопоставление должно быть определено в элементе ResultMapping.

FunctionImportMapping элемент может иметь следующие дочерние элементы:

- ResultMapping (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **FunctionImportMapping** элементу:

Имя атрибута	Необходимо	Значение
FunctionImportName	Да	Имя импорта функции в концептуальной модели, с которым выполняется сопоставление.
FunctionName	Да	Имя импорта функции в модели хранения из полного пространства имен, с которым выполняется сопоставление.

Пример

Следующий пример основан на модели School. Рассмотрим следующую функцию в модели хранения:

```
<Function Name="GetStudentGrades" Aggregate="false"
          BuiltIn="false" NiladicFunction="false"
          IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
          Schema="dbo">
    <Parameter Name="StudentID" Type="int" Mode="In" />
</Function>
```

Также рассмотрим данный импорт функции в концептуальной модели:

```
<FunctionImport Name="GetStudentGrades" EntitySet="StudentGrades"
                  ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

В следующем примере показан **FunctionImportMapping** элемент, используемый для сопоставления функции и импорта функции друг с другом:

```
<FunctionImportMapping FunctionImportName="GetStudentGrades"
                      FunctionName="SchoolModel.Store.GetStudentGrades" />
```

Элемент InsertFunction (MSL)

InsertFunction в язык определения сопоставлений (MSL) сопоставляет функцию вставки типа сущности или ассоциации в концептуальной модели с хранимой процедурой в основной базе данных. Хранимые процедуры, с которыми сопоставляются функции изменения, должны объявляться в модели хранения. Дополнительные сведения см. в разделе элемента функция (SSDL).

NOTE

Если не выполнено сопоставление всех трех вставки, обновления или удаления типов сущностей с хранимыми процедурами, то несопоставленные операции завершится ошибкой, если во время выполнения и UpdateException возникает исключение.

InsertFunction элемент может быть дочерним элементом ModificationFunctionMapping и применить к EntityTypeMapping-элемент или элемент AssociationSetMapping.

Элемент InsertFunction, применяемый к EntityTypeMapping

При применении к элементу EntityTypeMapping **InsertFunction** сопоставляет функцию вставки типа сущности в концептуальной модели с хранимой процедурой.

InsertFunction элемент может иметь следующие дочерние элементы при применении к **EntityTypeMapping** элемент:

- AssociationEnd (ноль и более)
- ComplexProperty (ноль и более)
- ResultBinding (ноль или один)
- ScalarProperty (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **InsertFunction** элемент при применении к **EntityTypeMapping** элемент.

Имя атрибута	Необходимо	Значение
FunctionName	Да	Имя хранимой процедуры (с указанием пространства имен), с которой сопоставляется функция вставки. Хранимая процедура должна объявляться в модели хранения.
RowsAffectedParameter	Нет	Имя выходного параметра, возвращающего количество обработанных строк.

Пример

В следующем примере, основана на модели School и показывает **InsertFunction** элемент, используемый для сопоставления функции вставки типа сущности Person в **InsertPerson** хранимой процедуры. **InsertPerson** хранимая процедура объявлена в модели хранения.

```

<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>

```

Элемент **InsertFunction**, применяемый к **AssociationSetMapping**

При применении к элементу **AssociationSetMapping** **InsertFunction** сопоставляет функцию вставки ассоциации в концептуальной модели с хранимой процедурой.

InsertFunction элемент может иметь следующие дочерние элементы при применении к **AssociationSetMapping** элементу:

- **EndProperty**

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **InsertFunction** элемента при его применении к **AssociationSetMapping** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
FunctionName	Да	Имя хранимой процедуры (с указанием пространства имен), с которой сопоставляется функция вставки. Хранимая процедура должна объявляться в модели хранения.
RowsAffectedParameter	Нет	Имя выходного параметра, возвращающего количество обработанных строк.

Пример

В следующем примере, основана на модели School и показывает **InsertFunction** элемент, используемый для сопоставления функции вставки **CourseInstructor** взаимосвязи **InsertCourseInstructor** хранимой процедуры. **InsertCourseInstructor** хранимая процедура объявлена в модели хранения.

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

Элемент сопоставления (MSL)

Сопоставление элемент в язык определения сопоставлений (MSL) содержит сведения для сопоставления объектов, определенных в концептуальной модели в базу данных (как описано в модели хранения). Дополнительные сведения см. в разделе спецификации языка CSDL и SSDL спецификации.

Сопоставление элемент является корневым элементом спецификации сопоставления. Пространство имен XML для спецификаций сопоставлений <http://schemas.microsoft.com/ado/2009/11/mapping/cs>.

У элемента Mapping могут быть следующие дочерние элементы (в порядке перечисления):

- Псевдоним (ноль и более)
- EntityContainerMapping (ровно один элемент)

Имена типов концептуальной модели и модели хранения, которые упоминаются в языке MSL, должны указываться вместе с именами соответствующих пространств имен. Сведения об имени пространства имен концептуальной модели см. в разделе элемент схемы (CSDL). Сведения об имени пространства имен модели хранения см. в разделе элемент схемы (SSDL). Элемент Alias можно определить псевдонимы для пространств имен, используемых в языке MSL.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **сопоставление** элемент.

Имя атрибута	Необходимо	Значение
ПРОБЕЛ	Да	C-S. Это фиксированное значение, и его невозможно изменить.

Пример

В следующем примере показан **сопоставление** элемент, основанный на модели School. Дополнительные

сведения о модели School см. Краткое руководство (Entity Framework).

```
<Mapping Space="C-S"
      xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
      <MappingFragment StoreEntitySet="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="Credits" ColumnName="Credits" />
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  <EntitySetMapping Name="Departments">
    <EntityTypeMapping TypeName="c.Department">
      <MappingFragment StoreEntitySet="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Budget" ColumnName="Budget" />
        <ScalarProperty Name="StartDate" ColumnName="StartDate" />
        <ScalarProperty Name="Administrator" ColumnName="Administrator" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
</EntityContainerMapping>
</Mapping>
```

Элемент MappingFragment (MSL)

MappingFragment элемент в язык определения сопоставлений (MSL) определяет сопоставление между свойствами типа сущности концептуальной модели и таблицы или представления в базе данных. Сведения о типах сущностей концептуальной модели и базовых таблиц базы данных или представления см. в разделе элемент Entity-Type (CSDL) и элемент EntitySet (SSDL). **MappingFragment** может быть дочерним элементом элемента EntityTypeMapping-элемент или элемент EntitySetMapping.

MappingFragment элемент может иметь следующие дочерние элементы:

- ComplexType (ноль и более)
- ScalarProperty (ноль и более)
- Условие (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **MappingFragment** элемент.

Имя атрибута	Необходимо	Значение
StoreEntitySet	Да	Имя столбца таблицы или представления, с которым производится сопоставление.

Имя атрибута	Необходимо	Значение
MakeColumnsDistinct	Нет	Значение true, или False в зависимости от того, возвращаются только уникальные строки. Если этот атрибут имеет значение True , generateupdateviews как атрибут элемента EntityContainerMapping должно быть присвоено False .

Пример

В следующем примере показан **MappingFragment** элемент дочерним **EntityTypeMapping** элемент. В этом примере свойства **курс** типа в концептуальной модели сопоставляются со столбцами из **курс** таблицы в базе данных.

```
<EntitySetMapping Name="Courses">
  <EntityTypeMapping TypeName="SchoolModel.Course">
    <MappingFragment StoreEntitySet="Course">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="Title" ColumnName="Title" />
      <ScalarProperty Name="Credits" ColumnName="Credits" />
      <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

Пример

В следующем примере показан **MappingFragment** элемент дочерним **EntitySetMapping** элемент. Как показано в примере выше свойства **курс** типа в концептуальной модели сопоставляются со столбцами из **курс** таблицы в базе данных.

```
<EntitySetMapping Name="Courses" TypeName="SchoolModel.Course">
  <MappingFragment StoreEntitySet="Course">
    <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    <ScalarProperty Name="Title" ColumnName="Title" />
    <ScalarProperty Name="Credits" ColumnName="Credits" />
    <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
  </MappingFragment>
</EntitySetMapping>
```

Элемент ModificationFunctionMapping (MSL)

ModificationFunctionMapping в язык определения сопоставлений (MSL) сопоставляет вставки, обновления и удаления функций типа сущности концептуальной модели с хранимыми процедурами в основной базе данных. **ModificationFunctionMapping** элемент также можно сопоставить Вставка и удаление функций для многих ассоциаций в концептуальной модели с хранимыми процедурами в основной базе данных. Хранимые процедуры, с которыми сопоставляются функции изменения, должны объявляться в модели хранения. Дополнительные сведения см. в разделе элемента функция (SSDL).

NOTE

Если не выполнено сопоставление всех трех вставки, обновления или удаления типов сущностей с хранимыми процедурами, то несопоставленные операции завершится ошибкой, если во время выполнения и UpdateException возникает исключение.

NOTE

Если функции преобразования для одной сущности в иерархии наследования сопоставляются с хранимыми процедурами, то функции преобразования всех типов в иерархии должны быть сопоставлены с хранимыми процедурами.

ModificationFunctionMapping элемент может быть дочерний элемент EntityTypemapping или AssociationSetMapping-элемент.

ModificationFunctionMapping элемент может иметь следующие дочерние элементы:

- Элемент DeleteFunction (ноль или один)
- Элемент InsertFunction (ноль или один)
- UpdateFunction (ноль или один)

Нет атрибутов, применимых к **ModificationFunctionMapping** элемент.

Пример

В следующем примере показано сопоставление набора сущностей для **людей** набору сущностей в модели School. В дополнение к сопоставлению столбцов **Person** тип сущности, сопоставление вставки, обновления и удаления функций **Person** отображаются тип. Функции, с которыми выполняется сопоставление, объявляются в модели хранения.

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate"
                ColumnName="EnrollmentDate" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <ModificationFunctionMapping>
            <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
                <ScalarProperty Name="EnrollmentDate"
                    ParameterName="EnrollmentDate" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName" />
                <ScalarProperty Name="LastName" ParameterName="LastName" />
                <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
            </InsertFunction>
            <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
                <ScalarProperty Name="EnrollmentDate"
                    ParameterName="EnrollmentDate"
                    Version="Current" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate"
                    Version="Current" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName"
                    Version="Current" />
                <ScalarProperty Name="LastName" ParameterName="LastName"
                    Version="Current" />
                <ScalarProperty Name="PersonID" ParameterName="PersonID"
                    Version="Current" />
            </UpdateFunction>
            <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
                <ScalarProperty Name="PersonID" ParameterName="PersonID" />
            </DeleteFunction>
        </ModificationFunctionMapping>
    </EntityTypeMapping>
</EntitySetMapping>

```

Пример

В следующем примере показано сопоставление набора ассоциаций для **CourseInstructor** набор ассоциаций в модели School. В дополнение к сопоставлению столбцов **CourseInstructor** ассоциации, сопоставление функции insert и delete **CourseInstructor** показаны связи. Функции, с которыми выполняется сопоставление, объявляются в модели хранения.

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

Элемент QueryView (MSL)

QueryView элемент в языке определения сопоставлений (MSL) определяет сопоставление только для чтения между типом сущности или ассоциации в концептуальной модели и таблицей в основной базе данных. Сопоставление определяется с помощью запроса Entity SQL, которое вычисляется по модели хранения и express результирующий набор с точки зрения сущность или ассоциацию в концептуальной модели. Представления запросов доступны только для чтения, поэтому для обновления типов, которые определены представлениями запросов, не могут применяться стандартные команды обновления. Представления запросов доступны только для чтения, поэтому для обновления типов, которые определены представлениями запросов, не могут применяться стандартные команды обновления. Дополнительные сведения см. в разделе Практическое: функции изменения сопоставления хранимых процедур.

NOTE

В **QueryView** элемент, выражения Entity SQL, содержащие **GroupBy**, групповые статистические выражения и свойства навигации не поддерживаются.

QueryView элемент может быть дочерним элементом **EntityTypeMapping** или **AssociationSetMapping**-элемент. В первом случае представление запроса определяет сопоставление только для чтения для сущности в концептуальной модели. Во втором случае представление запроса определяет сопоставление только для чтения для связи в концептуальной модели.

NOTE

Если **AssociationSetMapping** элемент предназначен для связи со ссылочным ограничением, **AssociationSetMapping** элемент игнорируется. Дополнительные сведения см. в разделе элемент **ReferentialConstraint** (CSDL).

QueryView элемент не может иметь дочерние элементы.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **QueryView** элементу.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Имя типа	Нет	Имя типа концептуальной модели, сопоставляемой представлением запроса.

Пример

В следующем примере показан **QueryView** как дочерний элемент элемента **EntitySetMapping** элемент и определяет сопоставление представления запроса для **отдел** типа сущности в Модель School.

```
<EntitySetMapping Name="Departments" >
  <QueryView>
    SELECT VALUE SchoolModel.Department(d.DepartmentID,
                                         d.Name,
                                         d.Budget,
                                         d.StartDate)
    FROM SchoolModelStoreContainer.Department AS d
    WHERE d.Budget > 150000
  </QueryView>
</EntitySetMapping>
```

Поскольку запрос возвращает только подмножество членов **отдел** тип в модели хранения, **отдел** тип в модели School был изменен на базе этих сопоставлений следующим образом:

```
<EntityType Name="Department">
  <Key>
    <PropertyRef Name="DepartmentID" />
  </Key>
  <Property Type="Int32" Name="DepartmentID" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false"
            MaxLength="50" FixedLength="false" Unicode="true" />
  <Property Type="Decimal" Name="Budget" Nullable="false"
            Precision="19" Scale="4" />
  <Property Type="DateTime" Name="StartDate" Nullable="false" />
  <NavigationProperty Name="Courses"
    Relationship="SchoolModel.FK_Course_Department"
    FromRole="Department" ToRole="Course" />
</EntityType>
```

Пример

В следующем примере показан **QueryView** элемент дочерним **AssociationSetMapping** элемент и определяет сопоставление только для чтения для **FK_Course_Department** ассоциации в модели School.

```

<EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolEntities">
    <EntitySetMapping Name="Courses" >
        <QueryView>
            SELECT VALUE SchoolModel.Course(c.CourseID,
                c.Title,
                c.Credits)
            FROM SchoolModelStoreContainer.Course AS c
        </QueryView>
    </EntitySetMapping>
    <EntitySetMapping Name="Departments" >
        <QueryView>
            SELECT VALUE SchoolModel.Department(d.DepartmentID,
                d.Name,
                d.Budget,
                d.StartDate)
            FROM SchoolModelStoreContainer.Department AS d
            WHERE d.Budget > 150000
        </QueryView>
    </EntitySetMapping>
    <AssociationSetMapping Name="FK_Course_Department" >
        <QueryView>
            SELECT VALUE SchoolModel.FK_Course_Department(
                CREATEREF(SchoolEntities.Departments, row(c.DepartmentID), SchoolModel.Department),
                CREATEREF(SchoolEntities.Courses, row(c.CourseID)) )
            FROM SchoolModelStoreContainer.Course AS c
        </QueryView>
    </AssociationSetMapping>
</EntityContainerMapping>

```

Комментарии

Представления запросов можно определять для реализации следующих сценариев.

- Определение сущности в концептуальной модели, не включающей все свойства сущности в модели хранения. Сюда входят свойства, которые не имеют значения по умолчанию и не поддерживают **null** значения.
- Сопоставление вычисляемых столбцов в модели хранения со свойствами типов сущностей в концептуальной модели.
- Определение сопоставления, в котором условия, используемые для секционирования сущностей в концептуальной модели, не базируются на равенстве. При указании условного сопоставления с помощью **условие** элемент, указанное условие должно быть равно указанному значению. Дополнительные сведения см. в разделе элемент Condition (MSL).
- Сопоставление одного столбца в модели хранения с несколькими типами в концептуальной модели.
- Сопоставление нескольких типов с одной и той же таблицей.
- Определение ассоциаций в концептуальной модели, базирующихся на внешних ключах реляционной схемы.
- Применение пользовательской бизнес-логики для задания значений свойств в концептуальной модели. Например, можно сопоставить строковое значение «T» в источнике данных в значение **true**, логическое значение, в концептуальной модели.
- Определение условных фильтров для результатов запроса.
- Наложение меньшего числа ограничений на данные в концептуальной модели, нежели в модели хранения. Например, можно определить свойство в концептуальной модели допускает значения **null** даже если столбец, с которым сопоставляется не поддерживает **null** значения.

При определении представлений запросов для сущностей необходимо принимать во внимание следующие соображения.

- Представления запросов доступны только для чтения. Обновление сущностей можно осуществлять

только с помощью функций изменения.

- В случае определения типа сущности через представление запроса необходимо также определять все связанные сущности через представления запросов.
- При сопоставлении ассоциации многие ко многим сущности в модели хранения, которая представляет связанную таблицу в реляционной схеме, необходимо определить **QueryView** элемент в **AssociationSetMapping** элемент для данной таблицы связей.
- Представления запросов должны быть определены для всех типов в иерархии типов. Это можно сделать следующими способами:
 - С помощью одного **QueryView** элемент, который указывает один запрос Entity SQL, которая возвращает объединение всех типов сущностей в иерархии.
 - С помощью одного **QueryView** элемент, который указывает один запрос Entity SQL, использующий оператор CASE для возврата определенного типа сущности в иерархии на основе определенного условия.
 - С помощью дополнительного **QueryView** элемента для конкретного типа в иерархии. В этом случае следует использовать **TypeName** атрибут **QueryView** элемента, чтобы указать тип сущности для каждого представления.
- Когда представление запроса определено, нельзя указать **StorageSetName** атрибут **EntitySetMapping** элемент.
- Когда представление запроса определено, **EntitySetMapping** элемент не может также содержать **свойство** сопоставления.

Элемент ResultBinding (язык MSL)

ResultBinding в языке определения сопоставлений (MSL) сопоставляет значения столбцов, возвращаемых хранимыми процедурами свойствам сущности в концептуальной модели при сопоставлении функций изменения типа сущности хранимой процедурой в основной базе данных. Например, если возвращается значение столбца идентификаторов в инструкции insert хранимой процедуры, **ResultBinding** элемент возвращаемое значение сопоставляется со свойством типа сущности в концептуальной модели.

ResultBinding элемент может быть дочерним элементом элемента **InsertFunction** или элемента **UpdateFunction**.

ResultBinding элемент не может иметь дочерние элементы.

Применимые атрибуты

В следующей таблице описаны атрибуты, применимые к **ResultBinding** элемент:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства сущности в концептуальной модели, с которым выполняется сопоставление.
ColumnName	Да	Имя столбца, с которым выполняется сопоставление.

Пример

В следующем примере, основана на модели School и показывает **InsertFunction** элемент, используемый для сопоставления функции вставки **Person** тип сущности, **InsertPerson** Хранимая процедура. (**InsertPerson** хранимой процедуры ниже и объявлена в модели хранения.) Объект **ResultBinding** элемент используется для сопоставления значения столбца, который возвращается хранимой процедурой (**NewPersonID**) со свойством типа сущности (**PersonID**).

```

<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>

```

Следующий запрос Transact-SQL описывает **InsertPerson** хранимой процедуры:

```

CREATE PROCEDURE [dbo].[InsertPerson]
    @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @HireDate datetime,
    @EnrollmentDate datetime
AS
    INSERT INTO dbo.Person (LastName,
                           FirstName,
                           HireDate,
                           EnrollmentDate)
    VALUES (@LastName,
            @FirstName,
            @HireDate,
            @EnrollmentDate);
    SELECT SCOPE_IDENTITY() as NewPersonID;

```

Элемент ResultMapping (язык MSL)

ResultMapping элемент в язык определения сопоставлений (MSL) определяет сопоставление между импортом функции в концептуальной модели и хранимой процедурой в основной базе данных, при выполнении следующих условий:

- Импорт функции возвращает тип сущности концептуальной модели или сложный тип.
- Имена столбцов, возвращаемые хранимой процедурой, не совпадают в точности с именами свойств в сложном типе или типе сущности.

По умолчанию сопоставление между столбцами, возвращаемыми хранимой процедурой, и сложным типом или типом сущности выполняется по именам столбцов и свойств. Если имена столбцов не точно совпадают с именами свойств, необходимо использовать **ResultMapping** элемент для определения сопоставления.

Пример сопоставления по умолчанию см. в разделе элемент FunctionImportMapping (MSL).

ResultMapping элемент является дочерним элементом элемента FunctionImportMapping.

ResultMapping элемент может иметь следующие дочерние элементы:

- EntityTypeMapping (ноль и более)
- ComplexTypeMapping

Нет атрибутов, применимых к **ResultMapping** элементу.

Пример

Рассмотрим следующую хранимую процедуру:

```
CREATE PROCEDURE [dbo].[GetGrades]
    @student_Id int
    AS
        SELECT      EnrollmentID as enroll_id,
                    Grade as grade,
                    CourseID as course_id,
                    StudentID as student_id
        FROM dbo.StudentGrade
        WHERE StudentID = @student_Id
```

Рассмотрим следующий тип сущности концептуальной модели:

```
<EntityType Name="StudentGrade">
    <Key>
        <PropertyRef Name="EnrollmentID" />
    </Key>
    <Property Name="EnrollmentID" Type="Int32" Nullable="false"
              annotation:StoreGeneratedPattern="Identity" />
    <Property Name="CourseID" Type="Int32" Nullable="false" />
    <Property Name="StudentID" Type="Int32" Nullable="false" />
    <Property Name="Grade" Type="Decimal" Precision="3" Scale="2" />
</EntityType>
```

Для создания импорта функции, возвращающей экземпляры предыдущего типа сущности, сопоставление между столбцами, возвращаемыми хранимой процедурой и тип сущности должен быть определен в **ResultMapping** элемент:

```
<FunctionImportMapping FunctionImportName="GetGrades"
                           FunctionName="SchoolModel.Store.GetGrades" >
    <ResultMapping>
        <EntityTypeMapping TypeName="SchoolModel.StudentGrade">
            <ScalarProperty Name="EnrollmentID" ColumnName="enroll_id"/>
            <ScalarProperty Name="CourseID" ColumnName="course_id"/>
            <ScalarProperty Name="StudentID" ColumnName="student_id"/>
            <ScalarProperty Name="Grade" ColumnName="grade"/>
        </EntityTypeMapping>
    </ResultMapping>
</FunctionImportMapping>
```

Элемент ScalarProperty (MSL)

ScalarProperty в языке определения сопоставлений (MSL) сопоставляет свойство типа сущности концептуальной модели, сложного типа или сопоставления столбца или параметра хранимой процедуры в базе данных.

NOTE

Хранимые процедуры, с которыми сопоставляются функции изменения, должны объявляться в модели хранения. Дополнительные сведения см. в разделе элемента функция (SSDL).

ScalarProperty элемент может быть дочерним элементом следующих элементов:

- MappingFragment
- InsertFunction
- UpdateFunction
- DeleteFunction
- EndProperty
- ComplexProperty
- ResultMapping

Как дочерний **MappingFragment**, **ComplexProperty**, или **EndProperty** элемент, **ScalarProperty** сопоставляет свойство в концептуальной модели со столбцом в базе данных. Как дочерний **InsertFunction**, **UpdateFunction**, или **DeleteFunction** элемент, **ScalarProperty** сопоставляет свойство в концептуальной модели с параметром хранимой процедуры.

ScalarProperty элемент не может иметь дочерние элементы.

Применимые атрибуты

Атрибуты, которые применяются к **ScalarProperty** элементу различаться в зависимости от роли элемента.

В следующей таблице описаны атрибуты, которые применяются, когда **ScalarProperty** элемент используется для сопоставления свойства концептуальной модели со столбцом в базе данных:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства концептуальной модели, с которым выполняется сопоставление.
ColumnName	Да	Имя столбца таблицы, с которым производится сопоставление.

В следующей таблице описаны атрибуты, применимые к **ScalarProperty** элементу, если он используется для сопоставления свойства концептуальной модели с параметром хранимой процедуры:

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя свойства концептуальной модели, с которым выполняется сопоставление.
Имя_параметра	Да	Имя сопоставляемого параметра.
Version	Нет	Текущий или исходного в зависимости от того, является ли текущее значение или исходное значение свойства должен использоваться для проверки на параллелизм.

Пример

В следующем примере показан **ScalarProperty** элемента, используемого одним из двух способов:

- Для сопоставления свойства **Person** типа сущности со столбцами **Person** таблицы.
- Для сопоставления свойства **Person** типа сущности с параметрами **UpdatePerson** хранимой процедуры. Хранимые процедуры объявляются в модели хранения.

```
<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate"
                ColumnName="EnrollmentDate" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <ModificationFunctionMapping>
            <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
                <ScalarProperty Name="EnrollmentDate"
                    ParameterName="EnrollmentDate" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName" />
                <ScalarProperty Name="LastName" ParameterName="LastName" />
                <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
            </InsertFunction>
            <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
                <ScalarProperty Name="EnrollmentDate"
                    ParameterName="EnrollmentDate"
                    Version="Current" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate"
                    Version="Current" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName"
                    Version="Current" />
                <ScalarProperty Name="LastName" ParameterName="LastName"
                    Version="Current" />
                <ScalarProperty Name="PersonID" ParameterName="PersonID"
                    Version="Current" />
            </UpdateFunction>
            <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
                <ScalarProperty Name="PersonID" ParameterName="PersonID" />
            </DeleteFunction>
        </ModificationFunctionMapping>
    </EntityTypeMapping>
</EntitySetMapping>
```

Пример

В следующем примере показан **ScalarProperty** элемент, используемый для сопоставления вставки и удаления ассоциации концептуальной модели с хранимыми процедурами в базе данных функции. Хранимые процедуры объявляются в модели хранения.

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

Элемент UpdateFunction (MSL)

UpdateFunction в язык определения сопоставлений (MSL) сопоставляет функцию обновления типа сущности в концептуальной модели с хранимой процедурой в основной базе данных. Хранимые процедуры, с которыми сопоставляются функции изменения, должны объявляться в модели хранения. Дополнительные сведения см. в разделе элемента функция (SSDL).

NOTE

Если не выполнено сопоставление всех трех вставки, обновления или удаления типов сущностей с хранимыми процедурами, то несопоставленные операции завершится ошибкой, если во время выполнения и UpdateException возникает исключение.

UpdateFunction элемент может быть дочерним элементом **ModificationFunctionMapping** и применяется к элементу **EntityTypeMapping**.

UpdateFunction элемент может иметь следующие дочерние элементы:

- **AssociationEnd** (ноль и более)
- **ComplexProperty** (ноль и более)
- **ResultBinding** (ноль или один)
- **ScalarProperty** (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **UpdateFunction** элемент.

Имя атрибута	Необходимо	Значение
--------------	------------	----------

Имя атрибута	Необходимо	Значение
FunctionName	Да	Имя хранимой процедуры (с указанием пространства имен), с которой сопоставляется функция обновления. Хранимая процедура должна объявляться в модели хранения.
RowsAffectedParameter	Нет	Имя выходного параметра, возвращающего количество обработанных строк.

Пример

В следующем примере, основана на модели School и показывает **UpdateFunction** элемент, используемый для сопоставления функции обновления **Person** тип сущности, **UpdatePerson** Хранимая процедура. **UpdatePerson** хранимая процедура объявлена в модели хранения.

```
<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>
```

Спецификация SSDL

13.09.2018 • 55 minutes to read • [Edit Online](#)

SSDL представляет собой язык на основе XML, на котором описывается модель хранения в приложениях Entity Framework.

В приложении Entity Framework метаданные модели хранения, загружается из SSDL-файла (написанного на языке SSDL) в экземпляр System.Data.Metadata.StoreItemCollection и доступен с помощью методов в Класс System.Data.Metadata.Edm.MetadataWorkspace. Платформа Entity Framework метаданные модели хранения используются для преобразования запросов к концептуальной модели для конкретного хранилища команды.

Entity Framework Designer (конструктор EF) хранит сведения о модели хранения в EDMX-файла во время разработки. Во время построения конструктор сущностей использует сведения в EDMX-файла, создаваемого SSDL-файл, необходимый Entity Framework во время выполнения.

Версии языка SSDL различаются по пространствам имен XML.

ВЕРСИИ ЯЗЫКА SSDL	ПРОСТРАНСТВО ИМЕН XML
SSDL v1	http://schemas.microsoft.com/ado/2006/04/edm/ssdl
SSDL v2	http://schemas.microsoft.com/ado/2009/02/edm/ssdl
SSDL v3	http://schemas.microsoft.com/ado/2009/11/edm/ssdl

Элемент Association (SSDL)

Ассоциации в языке определения схемы хранения (SSDL) указывает столбцы таблицы, которые участвуют в ограничении внешнего ключа в основной базе данных. Два элемента обязательный дочерний указать таблицы в концах ассоциации и кратность в каждом элементе. Необязательный элемент ReferentialConstraint указывает основной и зависимый концы ассоциации, а также участвующих столбцов. Если не **ReferentialConstraint** элемент присутствует, элемент AssociationSetMapping должен использоваться для указания сопоставления столбцов для ассоциации.

Ассоциации элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- End (ровно два)
- ReferentialConstraint (ноль или один)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **ассоциации** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя соответствующего ограничения внешнего ключа в основной базе данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ассоциации** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, использующий **ReferentialConstraint** для указания столбцов, участвующих в элемент **FK_CustomerOrders** ограничение внешнего ключа:

```
<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Элемент AssociationSet (SSDL)

AssociationSet элемент языка определения схемы хранения (SSDL) представляет ограничение внешнего ключа между двумя таблицами в основной базе данных. Столбцы таблицы, которые участвуют в ограничении внешнего ключа задаются в элементе ассоциации. **Ассоциации** элемент, соответствующий заданной **AssociationSet** элемент указан в **ассоциации** атрибут **AssociationSet** элемент.

Наборы ассоциаций SSDL сопоставляются наборы ассоциаций CSDL AssociationSetMapping-элемент. Тем не менее, если ассоциация CSDL для сопоставления на основе заданного CSDL определяется с помощью элемента ReferentialConstraint, то соответствующий **AssociationSetMapping** необходим элемент. В этом случае если **AssociationSetMapping** элемент присутствует, то определяемые им сопоставления будут переопределены **ReferentialConstraint** элемент.

AssociationSet элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- End (ноль или два)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **AssociationSet** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя ограничения внешнего ключа, представленное набором ассоциаций.

Имя атрибута	Необходимо	Значение
Ассоциации	Да	Имя ассоциации, определяющее столбцы, которые участвуют в ограничении внешнего ключа.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **AssociationSet** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **AssociationSet**, представляющего `FK_CustomerOrders` ограничения внешнего ключа в основной базе данных:

```
<AssociationSet Name="FK_CustomerOrders"
    Association="ExampleModel.Store.FK_CustomerOrders">
    <End Role="Customers" EntitySet="Customers" />
    <End Role="Orders" EntitySet="Orders" />
</AssociationSet>
```

Элемент CollectionType (язык SSDL)

CollectionType элемент в языке определения схемы хранения (SSDL) указывает, что тип возвращаемого значения функции является коллекцией. **CollectionType** элемент является дочерним элемента **ReturnType**. Тип коллекции задается с помощью дочернего элемента **RowType**:

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **CollectionType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано функцию, которая использует **CollectionType** элемента, чтобы указать, что функция возвращает коллекцию строк.

```
<Function Name="GetProducts" IsComposable="true" Schema="dbo">
    <ReturnType>
        <CollectionType>
            <RowType>
                <Property Name="ProductID" Type="int" Nullable="false" />
                <Property Name="CategoryID" Type="bigint" Nullable="false" />
                <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
                <Property Name="UnitPrice" Type="money" />
                <Property Name="Discontinued" Type="bit" />
            </RowType>
        </CollectionType>
    </ReturnType>
</Function>
```

Элемент CommandText (SSDL)

CommandText элемент в языке определения схемы хранения (SSDL) — это дочерний элемент элемента функция, которая позволяет определить инструкцию SQL, выполняемую в базе данных. **CommandText** элемент позволяет добавить функциональные возможности, схожие с хранимой процедурой в базе данных, но определять **CommandText** элемент в модели хранения.

CommandText элемент не может иметь дочерние элементы. Тело **CommandText** элемент должен быть допустимой инструкцией SQL в основной базе данных.

Нет атрибутов, применимых к **CommandText** элемент.

Пример

В следующем примере показан **функция** элемент с дочерним **CommandText** элемент. Предоставлять **UpdateProductInOrder** функционировать как метод класса ObjectContext, импортируйте ее в концептуальной модели.

```
<Function Name="UpdateProductInOrder" IsComposable="false">
  <CommandText>
    UPDATE Orders
    SET ProductId = @productId
    WHERE OrderId = @orderId;
  </CommandText>
  <Parameter Name="productId"
    Mode="In"
    Type="int"/>
  <Parameter Name="orderId"
    Mode="In"
    Type="int"/>
</Function>
```

Элемент DefiningQuery (SSDL)

DefiningQuery элемент в языке определения схемы хранения (SSDL) позволяет выполнять инструкции SQL непосредственно в основной базе данных. **DefiningQuery** элемент обычно используется как представление базы данных, но оно определено в модели хранения вместо базы данных. Представление, определенное в **DefiningQuery** элемент можно сопоставить с типом сущности в концептуальной моделью с применением элемент EntitySetMapping. Такие сопоставления предназначены только для чтения.

В следующем синтаксисе SSDL показана объявление **EntitySet** следуют **DefiningQuery** элемент, содержащий запрос, используемый для получения представления.

```
<Schema>
  <EntitySet Name="Tables" EntityType="Self.STable">
    <DefiningQuery>
      SELECT TABLE_CATALOG,
        'test' as TABLE_SCHEMA,
        TABLE_NAME
      FROM INFORMATION_SCHEMA.TABLES
    </DefiningQuery>
  </EntitySet>
</Schema>
```

Можно использовать хранимые процедуры в Entity Framework для реализации сценариев чтения и записи применительно к представлениям. Представление источника данных или представление Entity SQL можно использовать в качестве базовой таблицы для выборки данных и обработки изменений с помощью хранимых процедур.

Можно использовать **DefiningQuery** элемента к целевому объекту Microsoft SQL Server Compact 3.5. Хотя SQL Server Compact 3.5 не поддерживает хранимые процедуры, вы можете реализовать аналогичные

функциональные возможности с **DefiningQuery** элемент. Этот элемент также может быть полезен при создании хранимых процедур для преодоления несоответствий между типами данных, используемых в языке программирования, и типами данных источника данных. Можно написать **DefiningQuery**, принимает определенный набор параметров, а затем вызывает хранимую процедуру с разными наборами параметров, например, хранимую процедуру, которая удаляет данные.

Элемент Dependent (SSDL)

Зависимых элемент в языке определения схемы хранения (SSDL) является дочерним элементом для элемента **ReferentialConstraint**, который определяет зависимый конец ограничения внешнего ключа (также называемого справочным ограничением). **Зависимых** элемент указывает столбец (или столбцы) в таблице, которые ссылаются на первичный ключевой столбец (или столбцы). **PropertyRef** элементы определяют, какие столбцы существуют ссылки. Основной элемент определяет столбцы первичного ключа, на которые ссылаются столбцы, которые указаны в **зависимых** элемент.

Зависимых элемент может иметь следующие дочерние элементы (в указанном порядке):

- **PropertyRef** (один или несколько)
- Элементы **annotation** (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **зависимых** элемент.

Имя атрибута	Необходимо	Значение
Роль	Да	То же значение роли атрибута (если используется) соответствующего элемента End ; в противном случае имя таблицы, содержащей ссылающийся столбец.

NOTE

Любое количество атрибутов **annotation** (настраиваемых атрибутов XML), которые могут быть применены к **зависимых** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано элемента ассоциации, использующего **ReferentialConstraint** для указания столбцов, участвующих в элемент **FK_CustomerOrders** внешнего ключа ограничение. **Зависимых** элемент указывает **CustomerId** столбец **порядок** таблицы как зависимый конец ограничения.

```

<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

Элемент Documentation (SSDL)

Документации можно использовать для предоставления сведений об объекте, который определен в родительском элементе элемента языка определения схемы хранения (SSDL).

Документации элемент может иметь следующие дочерние элементы (в указанном порядке):

- **Сводка:** краткое описание родительского элемента. (ноль или один элемент)
- **LongDescription:** подробное описание родительского элемента. (ноль или один элемент)

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **документации** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **документации** элемент как дочерний элемент ElementType.

```

<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>

```

Элемент End (SSDL)

Окончания в языке определения схемы хранения (SSDL) определяет таблицу и число строк на одном конце ограничения внешнего ключа в основной базе данных. **Окончания** элемент может быть потомком элемента Association или элемента AssociationSet. В каждом из этих случаев дочерние элементы и применимые атрибуты будут различными.

Элемент End как дочерний по отношению к элементу Association

Окончания элемент (как дочерний **ассоциации** элемент) задает таблицу и число строк в конце ограничения внешнего ключа в **типа** и **Кратность** атрибутами. Элементы ограничения внешнего ключа

определяются как часть ассоциации SSDL. Ассоциация SSDL должна содержать ровно два элемента.

Окончания элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- OnDelete (ноль или один элемент)
- Элементы annotation (ноль или более элементов)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **окончания** элементу, если он является дочерним элементом **ассоциации** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Type	Да	Полное имя набора сущностей языка SSDL, в конце ограничения внешнего ключа.
Роль	Нет	Значение роли атрибут в основном или зависимые от элемента соответствующий элемент ReferentialConstraint (если используется).
Кратность	Да	1, от 0 до 1 , или * в зависимости от количества строк, которые могут быть в конце ограничения внешнего ключа. 1 указывает на конце ограничения внешнего ключа существует ровно одна строка. от 0 до 1 указывает, что существует более одной строки в конце ограничения внешнего ключа. * Указывает, что на конечной точке ограничения внешнего ключа существует ноль, один или более строк.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **окончания** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, определяющий **FK_CustomerOrders** ограничение внешнего ключа. **Кратность** значения, указанные на каждом **окончания** указывают, что многие строки в **заказы** таблицы могут быть связаны со строкой в **клиентов** таблицы, но только одна строка в **клиентов** таблицы могут быть связаны со строкой в **заказы** таблицы. Кроме того **OnDelete** элемент указывает, что все строки в **заказы** таблицу, которая ссылается на отдельную строку в **клиентов** таблица будет удалена, если в строке **клиентов** таблица удаляется.

```

<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

Элемент End как дочерний по отношению к элементу AssociationSet

Окончания элемент (как дочерний **AssociationSet** элемент) задает таблицу на одном конце ограничения внешнего ключа в основной базе данных.

Окончания элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **окончания** элементу, если он является дочерним элементом **AssociationSet** элемент.

Имя атрибута	Необходимо	Значение
Набор EntitySet	Да	Имя набора сущностей языка SSDL, находится на конце ограничения внешнего ключа.
Роль	Нет	Значение одного из роли атрибутами, заданными на одном окончания для соответствующего элемента ассоциации.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **окончания** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент с **AssociationSet** элемент с двумя **окончания** элементов:

```

<EntityContainer Name="ExampleModelStoreContainer">
    <EntityType Name="Customers"
        EntityType="ExampleModel.Store.Customers"
        Schema="dbo" />
    <EntityType Name="Orders"
        EntityType="ExampleModel.Store.Orders"
        Schema="dbo" />
    <AssociationSet Name="FK_CustomerOrders"
        Association="ExampleModel.Store.FK_CustomerOrders">
        <End Role="Customers" EntitySet="Customers" />
        <End Role="Orders" EntitySet="Orders" />
    </AssociationSet>
</EntityContainer>

```

Элемент EntityContainer (SSDL)

EntityContainer языка определения схемы хранения (SSDL) описывает структуру базового источника данных в приложении Entity Framework: наборы сущностей SSDL (определенные в элементах `EntityType`) представляют таблицы в базе данных, типы сущностей SSDL (определенные в элементах `EntityType`) представляют строки в таблице, а наборы ассоциаций (определенные в элементах `AssociationSet`) представляют ограничения внешнего ключа в базе данных. Контейнер сущностей концептуальной модели посредством элемента `EntityContainerMapping` сопоставляет контейнер сущностей модели хранения.

EntityContainer элемент может иметь ноль или один элемент документации. Если **документации** элемент присутствует, он должен предшествовать всех остальных дочерних элементов.

EntityContainer элемент может иметь ноль или более следующих дочерних элементов (в указанном порядке):

- `EntityType`
- `AssociationSet`
- Элементы `Annotation`

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntityContainer** элемент.

Имя атрибута	Необходимо	Значение
<code>Name</code>	Да	Имя контейнера сущностей. Это имя не может содержать точек (.).

NOTE

Любое количество атрибутов `annotation` (настраиваемых атрибутов XML), которые могут быть применены к **EntityContainer** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент, определяющий два набора сущностей и один набор ассоциаций. Обратите внимание, что тип сущности и имена типов ассоциаций определяются именем пространства имен концептуальной модели.

```

<EntityContainer Name="ExampleModelStoreContainer">
    <EntityType Name="Customers"
        EntityType="ExampleModel.Store.Customers"
        Schema="dbo" />
    <EntityType Name="Orders"
        EntityType="ExampleModel.Store.Orders"
        Schema="dbo" />
    <AssociationSet Name="FK_CustomerOrders"
        Association="ExampleModel.Store.FK_CustomerOrders">
        <End Role="Customers" EntitySet="Customers" />
        <End Role="Orders" EntitySet="Orders" />
    </AssociationSet>
</EntityContainer>

```

Элемент EntitySet (SSDL)

EntitySet элемент языка определения схемы хранения (SSDL) представляет таблицу или представление в основной базе данных. Элемент EntityType языка SSDL представляет строку в таблице или представлении.

EntityType атрибут **EntitySet** элемента указывает определенный тип сущности языка SSDL, представляющий строки в наборе сущностей языка SSDL. Сопоставление набора сущностей языка CSDL и SSDL набор сущностей, указывается в элементе EntitySetMapping.

EntitySet элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- DefiningQuery (ноль или один элемент)
- Элементы Annotation

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntitySet** элементу.

NOTE

Некоторые атрибуты (не указанные здесь) могут быть дополнены **хранилия** псевдонимом. Эти атрибуты используются мастером обновления модели при обновлении модели.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя набора сущностей.
EntityType	Да	Полное имя типа сущности, для которого набор сущностей содержит экземпляры.
Схема	Нет	Схема базы данных.
Таблица	Нет	Таблица базы данных.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EntitySet** элементу. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityContainer** элемент, имеющий два **EntitySet** элементов и один **AssociationSet** элемент:

```
<EntityContainer Name="ExampleModelStoreContainer">
  <EntitySet Name="Customers"
    EntityType="ExampleModel.Store.Customers"
    Schema="dbo" />
  <EntitySet Name="Orders"
    EntityType="ExampleModel.Store.Orders"
    Schema="dbo" />
  <AssociationSet Name="FK_CustomerOrders"
    Association="ExampleModel.Store.FK_CustomerOrders">
    <End Role="Customers" EntitySet="Customers" />
    <End Role="Orders" EntitySet="Orders" />
  </AssociationSet>
</EntityContainer>
```

Элемент EntityType (SSDL)

EntityType в языке определения схемы хранения (SSDL) представляет строку в таблицу или представление базы данных. Элемент EntitySet на языке SSDL представляет таблицу или представление, в котором появляются строки. **EntityType** атрибут **EntitySet** элемента указывает определенный тип сущности языка SSDL, представляющий строки в наборе сущностей языка SSDL. Сопоставление между типом сущности SSDL и типом сущности CSDL указывается в элементе EntityTypeMapping.

EntityType элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один элемент)
- Ключ (ноль или один элемент)
- Элементы Annotation

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **EntityType** элементу.

Имя атрибута	Необходимо	Значение
Name	Да	Имя типа сущности. Данное значение обычно совпадает с именем таблицы, в которой тип сущности представляется строку. Это значение не может содержать точек (.) .

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **EntityType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityType** элемент с двумя свойствами:

```

<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>

```

Элемент Function (SSDL)

Функция элемент в языке определения схемы хранения (SSDL) указывает хранимую процедуру, которая существует в базе данных.

Функция элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- Параметр (ноль и более)
- CommandText (ноль или один)
- ReturnType (ноль и более)
- Элементы annotation (ноль и более)

Возвращает объект типа для функции должен быть указан с помощью **ReturnType** элемент или **ReturnType** атрибутов (см. ниже), но не оба.

Хранимая процедура, указанная в модели хранения, может быть импортирована в концептуальную модель приложения. Дополнительные сведения см. в разделе [выполнение запросов с помощью хранимых процедур](#). **Функция** элемент также может использоваться для определения пользовательских функций в модели хранения.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **функция** элемент.

NOTE

Некоторые атрибуты (не указанные здесь) могут быть дополнены **хранилища** псевдонимом. Эти атрибуты используются мастером обновления модели при обновлении модели.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Name	Да	Имя хранимой процедуры.
ReturnType	Нет	Возвращаемый тип хранимой процедуры.
Aggregate	Нет	Значение true , Если хранимая процедура возвращает статистическое значение; в противном случае False .

Имя атрибута	Необходимо	Значение
Встроенные	Нет	Значение true , Если функция встроена в ¹ функции; в противном случае False .
StoreFunctionName	Нет	Имя хранимой процедуры.
NiladicFunction	Нет	Значение true , Если функция является функцией без параметров ² функции; False в противном случае.
IsComposable	Нет	Значение true , Если функция составляема ³ функции; False в противном случае.
ParameterTypeSemantics	Нет	Перечисление, определяющее семантику типа, которая используется для разрешения перегрузок функций. Перечисление определено в манифесте поставщика для определения функции. Значение по умолчанию — AllowImplicitConversion .
Схема	Нет	Имя схемы, в которой определена хранимая процедура.

¹ встроенная функция — это функция, которая определена в базе данных. Сведения о функциях, которые определены в модели хранения см. в разделе элемент CommandText (SSDL).

² функция без параметров — функция, которая не принимает параметры и, при вызове скобки не требуются.

³ две функции допускают композицию, если выходные данные одной функции может быть входных данных другой функции.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **функция** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **функция** элемент, соответствующий **UpdateOrderQuantity** хранимой процедуры. Хранимая процедура принимает два параметра и не возвращает значение.

```
<Function Name="UpdateOrderQuantity"
    Aggregate="false"
    BuiltIn="false"
    NiladicFunction="false"
    IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
```

Элемент Key (SSDL)

Ключ элемент языка определения схемы хранения (SSDL) представляет первичный ключ таблицы в основной базе данных. **Ключ** является дочерним элементом элемента EntityType, представляющий строку в таблице. Первичный ключ определен в **ключ** элемент, ссылаясь на один или несколько свойств элементов, которые определены на EntityType элемент.

Ключ элемент может иметь следующие дочерние элементы (в указанном порядке):

- PropertyRef (один или несколько)
- Элементы Annotation

Нет атрибутов, применимых к **ключ** элемент.

Пример

В следующем примере показан EntityType элемент с ключом, который ссылается на одно свойство:

```
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

Элемент OnDelete (SSDL)

OnDelete элемент в языке определения схемы хранения (SSDL) отражает поведение базы данных, при удалении строки, которая участвует в ограничении внешнего ключа. Если задано действие **Cascade**, то также будут удалены строки, на которые ссылается на строку, которая удаляется. Если задано действие **None**, то также не удаляются строки, на которые ссылается на строку, которая удаляется. **OnDelete** элемент является дочерним элементом элемента End.

OnDelete элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **OnDelete** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Действие	Да	CASCADE или None . (Значение Restricted является допустимым, но действует так же, как None .)

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **OnDelete** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, определяющий **FK_CustomerOrders** ограничение внешнего ключа. **OnDelete** элемент указывает, что все строки в **заказы** таблицу, которая ссылается на отдельную строку в **клиентов** таблица будет удалена, если в строке **Клиентов** таблица удаляется.

```
<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
<ReferentialConstraint>
  <Principal Role="Customers">
    <PropertyRef Name="CustomerId" />
  </Principal>
  <Dependent Role="Orders">
    <PropertyRef Name="CustomerId" />
  </Dependent>
</ReferentialConstraint>
</Association>
```

Элемент Parameter (SSDL)

Параметр элемент в языке определения схемы хранения (SSDL) является дочерним элемента Function, задающая параметры для хранимой процедуры в базе данных.

Параметр элемент может иметь следующие дочерние элементы (в указанном порядке):

- Документация (ноль или один)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **параметр** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя параметра.
Type	Да	Тип параметра.
Режим	Нет	B, Out, или InOut в зависимости от того, является ли параметр входной, выходной или параметр ввода вывода.
MaxLength	Нет	Максимальная длина параметра.
Точность	Нет	Точность параметра.
Масштаб	Нет	Масштаб параметра.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для параметров пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **параметр** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **функция** элемент, имеющий два **параметр** элементы, которые определяют входные параметры:

```
<Function Name="UpdateOrderQuantity"
    Aggregate="false"
    BuiltIn="false"
    NiladicFunction="false"
    IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
```

Элемент Principal (SSDL)

Участника элемент в языке определения схемы хранения (SSDL) является дочерним элементом для элемента ReferentialConstraint, который определяет основной конец ограничения внешнего ключа (также называемого справочным ограничением). **Участника** элемент определяет столбец первичного ключа (или столбцы) в таблице, на который ссылается другой столбец (или столбцы). **PropertyRef** элементы определяют, какие столбцы существуют ссылки. Элемент Dependent указывает столбцы, которые ссылаются на столбцы первичного ключа, которые указаны в **участника** элемент.

Участника элемент может иметь следующие дочерние элементы (в указанном порядке):

- PropertyRef (один или несколько)
- Элементы annotation (ноль и более)

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **участника** элемент.

ИМЯ АТРИБУТА	НЕОБХОДИМО	ЗНАЧЕНИЕ
Роль	Да	То же значение роли атрибута (если используется) соответствующего элемента End; в противном случае имя таблицы, содержащей ссылочный столбец.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **участнику** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показано элемента ассоциации, использующего **ReferentialConstraint** для указания столбцов, участвующих в элемент **FK_CustomerOrders** внешнего ключа ограничение. **Участника** элемент указывает **CustomerId** столбец **клиента** таблицу в качестве основной конец ограничения.

```
<Association Name="FK_CustomerOrders">
    <End Role="Customers"
        Type="ExampleModel.Store.Customers" Multiplicity="1">
        <OnDelete Action="Cascade" />
    </End>
    <End Role="Orders"
        Type="ExampleModel.Store.Orders" Multiplicity="*" />
    <ReferentialConstraint>
        <Principal Role="Customers">
            <PropertyRef Name="CustomerId" />
        </Principal>
        <Dependent Role="Orders">
            <PropertyRef Name="CustomerId" />
        </Dependent>
    </ReferentialConstraint>
</Association>
```

Элемент Property (SSDL)

Свойство элемент языка определения схемы хранения (SSDL) представляет столбец в таблице в базе данных. **Свойство** элементы являются дочерними для элементов **EntityType**, представляющих строки в таблице. Каждый **свойство** элемент, определенный на **EntityType** элемент представляет столбец.

Объект **свойство** элемент не может иметь дочерние элементы.

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **свойство** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя соответствующего столбца.
Type	Да	Тип соответствующего столбца.
Допускающий значение NULL	Нет	Значение true , (значение по умолчанию) или False в зависимости от того, может ли соответствующий столбец иметь значение null.
DefaultValue	Нет	Значение по умолчанию соответствующего столбца.
MaxLength	Нет	Максимальная длина соответствующего столбца.

Имя атрибута	Необходимо	Значение
FixedLength	Нет	Значение true, или False в зависимости от того, будет ли значение соответствующего столбца храниться как строка фиксированной длины.
Точность	Нет	Точность соответствующего столбца.
Масштаб	Нет	Масштаб соответствующего столбца.
Юникод	Нет	Значение true, или False в зависимости от того, будет ли значение соответствующего столбца храниться как строка Юникода.
Параметры сортировки	Нет	Строка, указывающая последовательность сортировки, которая должна использоваться в источнике данных.
SRID	Нет	Идентификатор ссылки Пространственные системы. Допустимо только для свойства пространственных типов. Дополнительные сведения см. в разделе SRID и SRID (SQL Server) .
StoreGeneratedPattern	Нет	Нет, удостоверений (если соответствующее значение столбца является удостоверением, создается в базе данных), или вычисленные (если соответствующее значение столбца вычисляется в базе данных). Не допустимо для RowType свойства.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **свойство** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **EntityType** элемент с двумя дочерними **свойство** элементов:

```
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

Элемент PropertyRef (SSDL)

PropertyRef элемент в языке определения схемы хранения (SSDL) ссылается на свойство, определенное в элементе **EntityType**, чтобы указать, что свойство будет выступать в одной из следующих ролей:

- Быть частью первичного ключа таблицы, **EntityType** представляет. Один или несколько **PropertyRef** элементы можно использовать для определения первичного ключа. Дополнительные сведения см. в разделе **Ключ элемента**.
- Будет зависимым или основным элементом ссылочного ограничения. Дополнительные сведения см. в разделе элемент **ReferentialConstraint**.

PropertyRef элемент может иметь только следующие дочерние элементы:

- Документация (ноль или один)
- Элементы Annotation

Применимые атрибуты

В следующей таблице описаны атрибуты, которые могут применяться к **PropertyRef** элемент.

Имя атрибута	Необходимо	Значение
Name	Да	Имя свойства, на которое дается ссылка.

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **PropertyRef** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для CSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **PropertyRef** элемент, используемый для определения первичного ключа путем ссылки на свойства, которое определено на **EntityType** элемент.

```
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

Элемент ReferentialConstraint (SSDL)

ReferentialConstraint элемент языка определения схемы хранения (SSDL) представляет ограничение внешнего ключа (также называемый ограничением ссылочной целостности) в основной базе данных. Основной и зависимый концы ограничения определяются участника и зависимые от дочерних элементов, соответственно. Столбцы, участвующие в основной и зависимой концах находятся ссылки на элементы **PropertyRef**.

ReferentialConstraint элемент — необязательный дочерний элемент элемента **Association**. Если

ReferentialConstraint элемент не используется для сопоставления ограничения внешнего ключа, который указан в **ассоциации** элемент AssociationSetMapping, элемент должен использоваться для этого.

ReferentialConstraint элемент может иметь следующие дочерние элементы:

- Документация (ноль или один)
- Участник (ровно один элемент)
- Зависимые (ровно один элемент)
- Элементы annotation (ноль и более)

Применимые атрибуты

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ReferentialConstraint** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере показан **ассоциации** элемент, использующий **ReferentialConstraint** для указания столбцов, участвующих в элемент **FK_CustomerOrders** ограничение внешнего ключа:

```
<Association Name="FK_CustomerOrders">
    <End Role="Customers"
        Type="ExampleModel.Store.Customers" Multiplicity="1">
            <OnDelete Action="Cascade" />
    </End>
    <End Role="Orders"
        Type="ExampleModel.Store.Orders" Multiplicity="*" />
    <ReferentialConstraint>
        <Principal Role="Customers">
            <PropertyRef Name="CustomerId" />
        </Principal>
        <Dependent Role="Orders">
            <PropertyRef Name="CustomerId" />
        </Dependent>
    </ReferentialConstraint>
</Association>
```

Элемент ReturnType (SSDL)

ReturnType элемент в языке определения схемы хранения (SSDL) указывает тип возвращаемого значения для функции, которая определена в **функция** элемент. Тип возвращаемого значения также можно задать с помощью функции **ReturnType** атрибута.

Тип возвращаемого значения функции указан с **тип** атрибут или **ReturnType** элемент.

ReturnType элемент может иметь следующие дочерние элементы:

- CollectionType (один)

NOTE

Любое количество атрибутов annotation (настраиваемых атрибутов XML), которые могут быть применены к **ReturnType** элемент. Однако настраиваемые атрибуты не могут принадлежать к любому пространству имен XML, зарезервированному для SSDL. Полные имена любых двух настраиваемых атрибутов не могут совпадать.

Пример

В следующем примере используется **функция**, возвращающий коллекцию строк.

```

<Function Name="GetProducts" IsComposable="true" Schema="dbo">
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="ProductID" Type="int" Nullable="false" />
        <Property Name="CategoryID" Type="bigint" Nullable="false" />
        <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
        <Property Name="UnitPrice" Type="money" />
        <Property Name="Discontinued" Type="bit" />
      </RowType>
    </CollectionType>
  </ReturnType>
</Function>

```

Элемент RowType (язык SSDL)

Объект **RowType** элемент в языке определения схемы хранения (SSDL) определяет безымянную структуру как возврат типа для функции, определенной в хранилище.

Объект **RowType** элемент является дочерним элементом элемента **CollectionType** элемент:

Объект **RowType** элемент может иметь следующие дочерние элементы:

- Свойство (один или несколько)

Пример

В следующем примере показано функции хранилища, которая использует **CollectionType** элемента, чтобы указать, что функция возвращает коллекцию строк (как указано в **RowType** элемент).

```

<Function Name="GetProducts" IsComposable="true" Schema="dbo">
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="ProductID" Type="int" Nullable="false" />
        <Property Name="CategoryID" Type="bigint" Nullable="false" />
        <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
        <Property Name="UnitPrice" Type="money" />
        <Property Name="Discontinued" Type="bit" />
      </RowType>
    </CollectionType>
  </ReturnType>
</Function>

```

Элемент Schema (SSDL)

Схемы элемент в языке определения схемы хранения (SSDL) является корневым элементом определения модели хранения. Он содержит определения объектов, функций и контейнеров, из которых состоит модель хранения.

Схемы элемент может содержать ноль или более следующих дочерних элементов:

- Ассоциация
- EntityType
- EntityContainer
- Функция

Схемы элемент использует **пространства имен** атрибут определяет пространство имен для объектов сущности типа и ассоциации в модели хранения. В пространстве имен не может быть двух объектов с

одинаковым именем.

Пространство имен модели хранения отличается от пространства имен XML элемента **схемы** элемент. Пространство имен модели хранения (как определено **пространства имен** атрибут) — это логический контейнер для типов сущностей и типов ассоциаций. Пространство имен XML (обозначается **xmlns** атрибут) из **схемы** элемент является пространство имен по умолчанию для дочерних элементов и атрибутов из **схемы** элемент. Пространства имен XML вида <http://schemas.microsoft.com/ado/YYYY/MM/edm/ssdl> (где YYYY и MM представляют год и месяц соответственно) зарезервированы для SSDL. Пользовательские элементы и атрибуты не могут присутствовать в пространствах имен такого вида.

Применимые атрибуты

В следующей таблице описываются атрибуты могут применяться к **схемы** элемент.

Имя атрибута	Необходимо	Значение
Пространство имен	Да	Пространство имен модели хранения. Значение пространства имен атрибут используется, чтобы сформировать полное имя типа. Например если EntityType с именем <i>клиента</i> находится в пространстве имен ExampleModel.Store, то полное доменное имя EntityType является ExampleModel.Store.Customer. Следующие строки не может использоваться как значение для пространства имен атрибут: системы, временных, или Edm . Значение для пространства имен атрибут не может быть таким же, как значение для пространства имен атрибут в элементе CSDL-схеме.
Alias	Нет	Идентификатор, используемый в качестве имени пространства имен. Например если EntityType с именем <i>клиента</i> находится в пространстве имен ExampleModel.Store, а значение псевдоним атрибут <i>StorageModel</i> , то в качестве полное доменное имя можно указать StorageModel.Customer EntityType .
Поставщик	Да	Поставщик данных.
ProviderManifestToken	Да	Маркер, который указывает поставщику, какой манифест должен быть возвращен. Формат маркера не определен. Значения для маркера определяются поставщиком. Сведения о маркерах манифеста поставщика SQL Server см. в разделе SqlClient для Entity Framework .

Пример

В следующем примере показан **схемы** элемент, содержащий **EntityContainer** элемент, двух **EntityType** элементов и один **ассоциации** элемент.

```
<Schema Namespace="ExampleModel.Store"
```

```

    Alias="Self" Provider="System.Data.SqlClient"
    ProviderManifestToken="2008"
    xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">
<EntityContainer Name="ExampleModelStoreContainer">
    <EntityType Name="Customers">
        EntityType="ExampleModel.Store.Customers"
        Schema="dbo" />
    <EntityType Name="Orders">
        EntityType="ExampleModel.Store.Orders"
        Schema="dbo" />
    <AssociationSet Name="FK_CustomerOrders">
        Association="ExampleModel.Store.FK_CustomerOrders">
            <End Role="Customers" EntitySet="Customers" />
            <End Role="Orders" EntitySet="Orders" />
        </AssociationSet>
    </EntityType>
</EntityContainer>
<EntityType Name="Customers">
    <Documentation>
        <Summary>Summary here.</Summary>
        <LongDescription>Long description here.</LongDescription>
    </Documentation>
    <Key>
        <PropertyRef Name="CustomerId" />
    </Key>
    <Property Name="CustomerId" Type="int" Nullable="false" />
    <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
    <Key>
        <PropertyRef Name="OrderId" />
    </Key>
    <Property Name="OrderId" Type="int" Nullable="false"
        c:CustomAttribute="someValue"/>
    <Property Name="ProductId" Type="int" Nullable="false" />
    <Property Name="Quantity" Type="int" Nullable="false" />
    <Property Name="CustomerId" Type="int" Nullable="false" />
    <c:CustomElement>
        Custom data here.
    </c:CustomElement>
</EntityType>
<Association Name="FK_CustomerOrders">
    <End Role="Customers"
        Type="ExampleModel.Store.Customers" Multiplicity="1">
        <OnDelete Action="Cascade" />
    </End>
    <End Role="Orders"
        Type="ExampleModel.Store.Orders" Multiplicity="*" />
<ReferentialConstraint>
    <Principal Role="Customers">
        <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
        <PropertyRef Name="CustomerId" />
    </Dependent>
</ReferentialConstraint>
</Association>
<Function Name="UpdateOrderQuantity">
    Aggregate="false"
    BuiltIn="false"
    NiladicFunction="false"
    IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
<Function Name="UpdateProductInOrder" IsComposable="false">
    <CommandText>
        UPDATE Orders

```

```

    SET ProductId = @productId
    WHERE OrderId = @orderId;
</CommandText>
<Parameter Name="productId"
    Mode="In"
    Type="int"/>
<Parameter Name="orderId"
    Mode="In"
    Type="int"/>
</Function>
</Schema>

```

Атрибуты примечаний

Атрибуты annotation в языке SSDL представляют собой настраиваемые атрибуты XML в модели хранения, которые содержат дополнительные метаданные об элементах в модели хранения. Атрибуты annotation должны не только иметь допустимую структуру XML, но и соответствовать следующим ограничениям.

- Атрибуты annotation не должны находиться ни в одном пространстве имен XML, которое зарезервировано для языка SSDL.
- Полные имена любых двух атрибутов annotation не должны совпадать.

К данному конкретному элементу языка SSDL может применяться несколько атрибутов annotation.

Метаданные, содержащиеся в элементах annotation может осуществляться во время выполнения с помощью классов в пространстве имен System.Data.Metadata.Edm.

Пример

В следующем примере показано EntityType элемент, имеющий атрибут annotation применяется к **OrderId** свойство. В примере также показывается элемент заметки, добавляемый **EntityType** элемент.

```

<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
    <Key>
        <PropertyRef Name="OrderId" />
    </Key>
    <Property Name="OrderId" Type="int" Nullable="false"
        c:CustomAttribute="someValue"/>
    <Property Name="ProductId" Type="int" Nullable="false" />
    <Property Name="Quantity" Type="int" Nullable="false" />
    <Property Name="CustomerId" Type="int" Nullable="false" />
    <c:CustomElement>
        Custom data here.
    </c:CustomElement>
</EntityType>

```

Элементы Annotation (SSDL)

Элементы annotation в языке SSDL представляют собой настраиваемые элементы XML в модели хранения, которые содержат дополнительные метаданные о модели хранения. Элементы annotation должны не только иметь верную структуру XML, но и соответствовать следующим ограничениям.

- Элементы annotation не должны находиться в каком-либо пространстве имен XML, которое зарезервировано для языка SSDL.
- Полные имена любых двух элементов annotation не должны совпадать.
- Элементы annotation должны находиться после всех остальных дочерних элементов в данном элементе SSDL.

У данного элемента языка SSDL может быть больше одного дочернего элемента annotation. Начиная с .NET Framework версии 4, метаданные, содержащиеся в элементах annotation может осуществляться во время

выполнения с помощью классов в пространстве имен System.Data.Metadata.Edm.

Пример

В следующем примере показано элемент EntityType с элементом annotation (**CustomElement**). В примере также показано атрибут annotation применяется к **OrderId** свойство.

```
<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
  <Key>
    <PropertyRef Name="OrderId" />
  </Key>
  <Property Name="OrderId" Type="int" Nullable="false"
    c:CustomAttribute="someValue"/>
  <Property Name="ProductId" Type="int" Nullable="false" />
  <Property Name="Quantity" Type="int" Nullable="false" />
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <c:CustomElement>
    Custom data here.
  </c:CustomElement>
</EntityType>
```

Аспекты (модель SSDL)

Аспекты в языке определения схемы хранения (SSDL) представляют ограничения для типов столбцов, которые указаны в свойствах элементов. Аспекты реализуются как атрибуты XML на **свойство** элементов.

В следующей таблице описываются аспекты, поддерживаемые в SSDL.

АСПЕКТ	ОПИСАНИЕ
Параметры сортировки	Задает последовательность сортировки, которая будет использоваться при выполнении операций сравнения и упорядочивания для значений свойств.
FixedLength	Указывает, может ли изменяться длина значения столбца.
MaxLength	Указывает максимальную длину значения столбца.
Точность	Для свойств типа десятичное , количество десятичных разрядов, может иметь значение свойства. Для свойств типа время , DateTime , и DateTimeOffset , указывает количество цифр дробной части секунд значения столбца.
Масштаб	Задает число цифр справа от десятичной запятой в значении столбца.
Юникод	Указывает, будет ли значение столбца храниться в Юникоде.

Определяющий запрос - конструктор EF

13.09.2018 • 9 minutes to read • [Edit Online](#)

В этом пошаговом руководстве показан способ добавления определяющего запроса и соответствующей сущности, тип модели, в конструкторе EF. Определяющий запрос обычно используется для предоставления функций, которые аналогичны предоставляемым представления базы данных, но оно определено в модели, а не в базе данных. Определяющий запрос позволяет выполнить инструкцию SQL, который указан в **DefiningQuery** элемент EDMX-файла. Дополнительные сведения см. в разделе **DefiningQuery** в спецификации SSDL.

При использовании определение запросов, необходимо также определить тип сущности в модели. Тип сущности используется для отображения данных с помощью определения запроса. Обратите внимание, что данные, отображаемые через этот тип сущности только для чтения.

Параметризованные запросы не могут выполняться как определяющие запросы. Однако их обновление возможно через сопоставление функций вставки, обновления и удаления типа сущности, который отображает данные в хранимых процедурах. Дополнительные сведения см. в разделе [вставки, обновления и удаления с помощью хранимых процедур](#).

В этом разделе показано, как выполнять следующие задачи.

- Добавление определяющего запроса
- Добавление типа сущности в модель
- Тип сущности сопоставляются с определяющим запросом

Предварительные требования

Для выполнения данного пошагового руководства требуется:

- Последнюю версию Visual Studio.
- [Образца базы данных School](#).

Настройка проекта

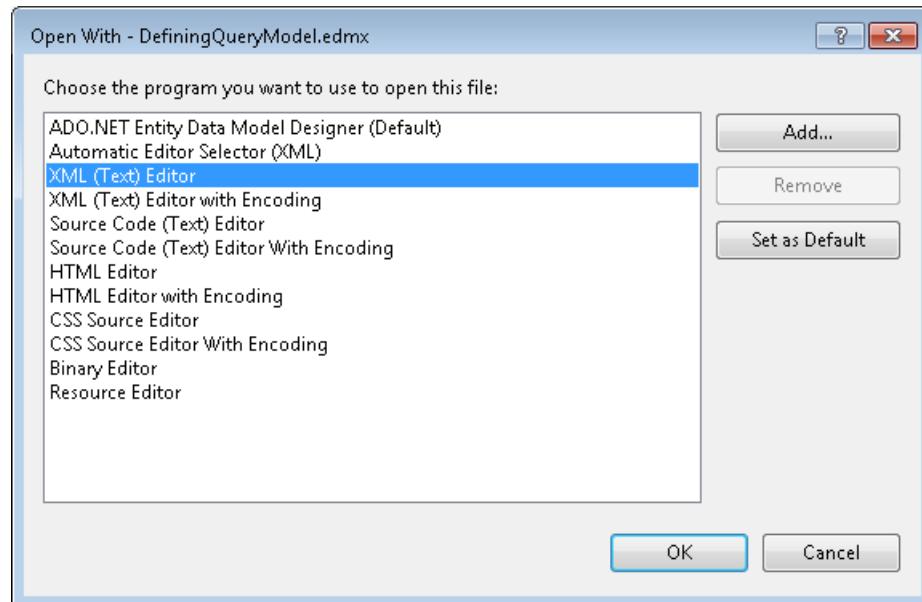
В этом пошаговом руководстве используется Visual Studio 2012 или более поздней версии.

- Запустите Visual Studio.
- В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**.
- В левой области щелкните **Visual C#**, а затем выберите **консольное приложение** шаблона.
- Введите **DefiningQuerySample** как имя проекта и нажмите кнопку **OK**.

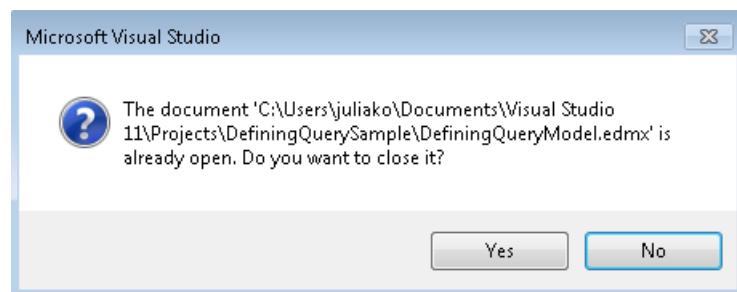
Создать модель, основанную на базе данных School

- Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**.
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в области шаблонов.
- Введите **DefiningQueryModel.edmx** имя файла, а затем нажмите кнопку **добавить**.
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**.

- Щелкните новое подключение. В диалоговом окне Свойства соединения, введите имя сервера (например, **(localdb)\mssqllocaldb**) выберите метод проверки подлинности, тип **School** для имени базы данных, а затем Нажмите кнопку **OK**. В диалоговом окне Выбор подключения к базе данных обновляется параметр подключения базы данных.
- В диалоговом окне Выбор объектов базы данных проверьте **таблицы** узла. Это добавит все таблицы, которые **School** модели.
- Нажмите кнопку **Готово**.
- В обозревателе решений щелкните правой кнопкой мыши **DefiningQueryModel.edmx** файл и выберите **открыть с помощью...**.
- Выберите **редактор (текстовый) XML**.



- Нажмите кнопку **Да** при появлении соответствующего запроса со следующим сообщением:



Добавление определяющего запроса

На этом шаге мы воспользуемся редактором XML для добавления определяющего запроса и тип сущности в разделе SSDL EDMX-файла.

- Добавить **EntitySet** элемент в раздел SSDL EDMX-файла (строка 5 по 13). Укажите следующие сведения:
 - Только **имя** и **EntityType** атрибуты **EntitySet** указанных элементов.
 - Полное имя типа сущности используется в **EntityType** атрибута.
 - Инструкция SQL для выполнения указана в **DefiningQuery** элемент.

```

<!-- SSDL content -->
<edmx:StorageModels>
    <Schema Namespace="SchoolModel.Store" Alias="Self" Provider="System.Data.SqlClient"
ProviderManifestToken="2008"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">
        <EntityContainer Name="SchoolModelStoreContainer">
            <EntityType Name="GradeReport" EntityType="SchoolModel.Store.GradeReport">
                <DefiningQuery>
                    SELECT CourseID, Grade, FirstName, LastName
                    FROM StudentGrade
                    JOIN
                    (SELECT * FROM Person WHERE EnrollmentDate IS NOT NULL) AS p
                    ON StudentID = p.PersonID
                </DefiningQuery>
            </EntityType>
            <EntityType Name="Course" EntityType="SchoolModel.Store.Course" store:Type="Tables" Schema="dbo" />

```

- Добавить **EntityType** элемент в раздел SSDL EDMX-файла. файл, как показано ниже. Обратите внимание на следующее условия:
 - Значение **имя** атрибут соответствует значению **EntityType** атрибут в **EntityType** элемент выше, несмотря на то что полное доменное имя Тип сущности используется в **EntityType** атрибута.
 - Имена свойств соответствуют именам столбцов, возвращаемых инструкцией SQL в **DefiningQuery** элемент (см. выше).
 - В данном примере ключ сущности состоит из трех свойств, что обеспечивает уникальность значения ключа.

```

<EntityType Name="GradeReport">
    <Key>
        <PropertyRef Name="CourseID" />
        <PropertyRef Name="FirstName" />
        <PropertyRef Name="LastName" />
    </Key>
    <Property Name="CourseID"
        Type="int"
        Nullable="false" />
    <Property Name="Grade"
        Type="decimal"
        Precision="3"
        Scale="2" />
    <Property Name="FirstName"
        Type="nvarchar"
        Nullable="false"
        MaxLength="50" />
    <Property Name="LastName"
        Type="nvarchar"
        Nullable="false"
        MaxLength="50" />
</EntityType>

```

NOTE

Если вы запустите **мастер обновления моделей** диалоговое окно, любые изменения, внесенные в модель хранения, включая определение запросов, будут перезаписаны.

Добавление типа сущности в модель

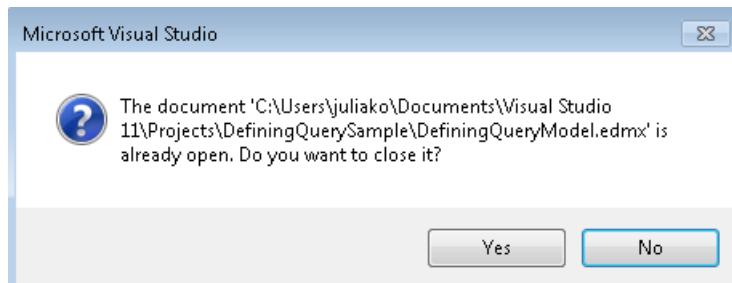
На этом этапе мы добавим тип сущности в концептуальную модель, в конструкторе EF. Обратите внимание

на следующее условия:

- **Имя** сущности соответствует значению **EntityType** атрибут в **EntitySet** элемент выше.
- Имена свойств соответствуют именам столбцов, возвращаемых инструкцией SQL в **DefiningQuery** элемент выше.
- В данном примере ключ сущности состоит из трех свойств, что обеспечивает уникальность значения ключа.

Откройте модель в конструкторе EF.

- Дважды щелкните DefiningQueryModel.edmx.
- Скажем **Да** следующее сообщение:



Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается.

- Щелкните правой кнопкой мыши в конструкторе и выберите пункт **Add New->сущности...**.
- Укажите **GradeReport** имя сущности и **CourseID** для **свойство Key**.
- Щелкните правой кнопкой мыши **GradeReport** сущности и выберите **Add New - > скалярное свойство**.
- Изменить имя по умолчанию свойства **FirstName**.
- Добавить еще одно скалярное свойство и указать **LastName** для имени.
- Добавить еще одно скалярное свойство и указать **корпоративного класса** для имени.
- В **свойства** измените **корпоративного класса тип** свойства **десятичное**.
- Выберите **FirstName** и **LastName** свойства.
- В **свойства** измените **EntityKey** значение свойства **True**.

Таким образом, были добавлены следующие элементы **CSDL** раздел EDMX-файла.

```
<EntitySet Name="GradeReport" EntityType="SchoolModel.GradeReport" />

<EntityType Name="GradeReport">
  ...
</EntityType>
```

Тип сущности сопоставляются с определяющего запроса

На этом шаге мы будем использовать окно **Mapping Details** для сопоставления концептуальной и типа сущности хранилища.

- Щелкните правой кнопкой мыши **GradeReport** сущности в область конструктора и выберите **сопоставление таблицы**.
Сведения о сопоставлении откроется диалоговое окно.
- Выберите **GradeReport** из **<добавить таблицу или представление>** раскрывающегося списка (расположенный в **таблицы**).

По умолчанию сопоставления между концептуальной и хранения **GradeReport** отображаются тип сущности.

Mapping Details - Person					
Parameter / Column	Operator	Property	Use Original...	Rows Affected	Parameter
Functions					
Insert Using InsertPerson					
Parameters					
@ LastName : nvarchar	←	LastName : String	<input type="checkbox"/>		
@ FirstName : nvarchar	←	FirstName : String	<input type="checkbox"/>		
@ HireDate : datetime	←	HireDate : DateTime	<input type="checkbox"/>		
@ EnrollmentDate : datetime	←	EnrollmentDate : DateTime	<input type="checkbox"/>		
@ Discriminator : nvarchar	←	Discriminator : String	<input type="checkbox"/>		
Result Column Bindings					
NewPersonID	→	PersonID : Int32	<input checked="" type="checkbox"/>		

В результате **EntitySetMapping** элемент добавляется в раздел сопоставления EDMX-файла.

```
<EntitySetMapping Name="GradeReports">
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.GradeReport)">
        <MappingFragment StoreEntitySet="GradeReport">
            <ScalarProperty Name="LastName" ColumnName="LastName" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="Grade" ColumnName="Grade" />
            <ScalarProperty Name="CourseID" ColumnName="CourseID" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>
```

- Скомпилируйте приложение.

Вызывать определяющего запроса в коде

Теперь, определяющий запрос можно выполнить с помощью **GradeReport** типа сущности.

```
using (var context = new SchoolEntities())
{
    var report = context.GradeReports.FirstOrDefault();
    Console.WriteLine("{0} {1} got {2}",
        report.FirstName, report.LastName, report.Grade);
}
```

Хранимые процедуры с несколькими результирующими наборами

13.09.2018 • 9 minutes to read • [Edit Online](#)

Иногда при использовании хранимых процедур, которые требуется вернуть несколько результатов значение. Этот сценарий обычно используется для сокращения объема базы данных циклами, необходимые для составления на одном экране. До EF5 Entity Framework позволяет вызывать хранимую процедуру, но возвратит только первый результирующий набор в вызывающий код.

В этой статье мы покажем два способа, которые можно использовать для доступа к более чем одного результирующего набора из хранимой процедуры в Entity Framework. Используется только код и работает с и код сначала и конструкторе EF и тот, который работает только с помощью конструктора EF. Средства и API поддерживает это должно улучшить в будущих версиях Entity Framework.

Модель

В примерах в этой статье используется базовый блог и сообщения модель, где блог имеет записях и блога принадлежит к одной записи. Мы будем использовать хранимую процедуру в базе данных, который возвращает все блогов и записей, примерно так:

```
CREATE PROCEDURE [dbo].[GetAllBlogsAndPosts]
AS
    SELECT * FROM dbo.Blogs
    SELECT * FROM dbo.Posts
```

Доступ к несколько результирующих наборов с помощью кода

Позволяет выполнить использовать код для выдачи необработанные команду SQL для выполнения хранимой процедуры. Этот подход удобен для работы с и код сначала и конструкторе EF.

Чтобы получить несколько результирующих задает рабочего, нам нужно уменьшится до API ObjectContext с помощью интерфейса IObjectContextAdapter.

Когда у нас ObjectContext мы можем использовать метод Translate преобразовать результаты наших хранимой процедуры, в сущности, которые могут отслеживаться и использоваться в EF в обычном режиме. В следующем образце кода демонстрируется это в действии.

```

using (var db = new BloggingContext())
{
    // If using Code First we need to make sure the model is built before we open the connection
    // This isn't required for models created with the EF Designer
    db.Database.Initialize(force: false);

    // Create a SQL command to execute the sproc
    var cmd = db.Database.Connection.CreateCommand();
    cmd.CommandText = "[dbo].[GetAllBlogsAndPosts]";

    try
    {

        db.Database.Connection.Open();
        // Run the sproc
        var reader = cmd.ExecuteReader();

        // Read Blogs from the first result set
        var blogs = ((IObjectContextAdapter)db)
            .ObjectContext
            .Translate<Blog>(reader, "Blogs", MergeOption.AppendOnly);

        foreach (var item in blogs)
        {
            Console.WriteLine(item.Name);
        }

        // Move to second result set and read Posts
        reader.NextResult();
        var posts = ((IObjectContextAdapter)db)
            .ObjectContext
            .Translate<Post>(reader, "Posts", MergeOption.AppendOnly);

        foreach (var item in posts)
        {
            Console.WriteLine(item.Title);
        }
    }
    finally
    {
        db.Database.Connection.Close();
    }
}

```

Метод Translate принимает средство чтения, мы получили, когда выполнена процедура, является именем набора сущностей и MergeOption. Имя EntitySet будет таким же, как свойства DbSet в вашем производном контексте. Перечисление MergeOption управляет, как обрабатываются результаты, если та же сущность уже существует в памяти.

Здесь мы итерации по коллекции блогов, прежде чем мы называем NextResult, это очень важно, учитывая приведенный выше код так, как первый результирующий набор, которые должны использоваться перед переходом к следующему результирующему набору.

После перевода двух методы вызываются, то сущности Blog и Post отслеживаются EF так же, как любой другой сущности и поэтому можно изменить или удалить и сохранен в обычном режиме.

NOTE

При создании сущности с помощью метода Translate EF не принимает любое сопоставление в учетной записи. Он просто будет соответствовать имена столбцов в результирующем наборе с именами свойств в классах.

NOTE

Что если имеется отложенная загрузка включена, при обращении к свойству сообщения на одной сущности блог затем EF будет соединиться с базой данных для пассивной загрузки все сообщения, несмотря на то, что мы уже загружен их все. Это обусловлено EF не знает ли вы загрузили все сообщения, или если есть и другие базы данных. Если вы хотите избежать этого, необходимо отключить отложенную загрузку.

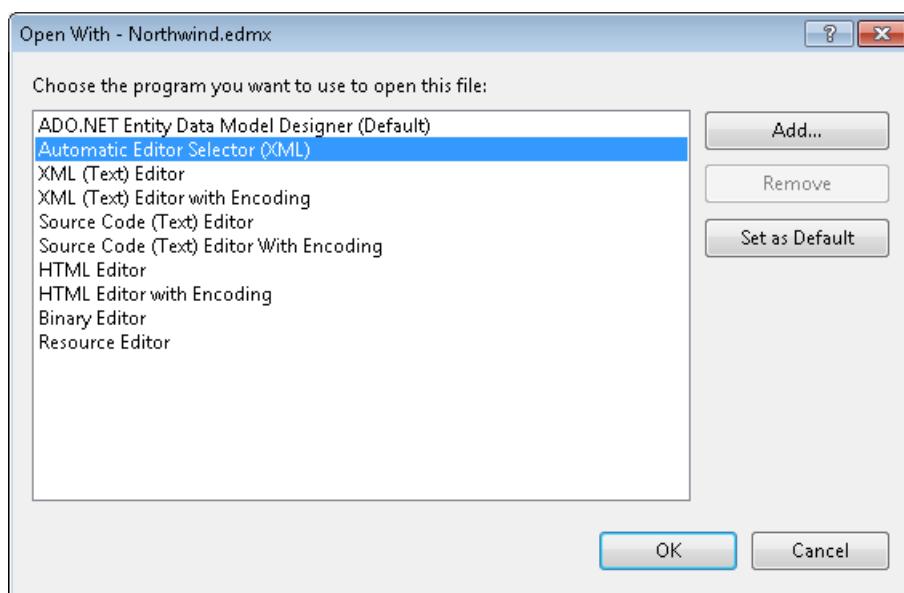
Несколько результирующих наборов с помощью настроенного в EDMX

NOTE

Необходимо ориентироваться .NET Framework 4.5, чтобы иметь возможность настроить несколько результирующих наборов в EDMX. Если вы ориентируетесь на .NET 4.0, можно использовать метод на основе кода, показанный в предыдущем разделе.

Если вы используете конструктор EF, знает о различных результирующих наборах, которые будут возвращены, также можно изменить модель. Необходимо знать, прежде чем вручную в том, что инструментарий не несколько результирующих значение помнить, поэтому необходимо будет вручную изменить EDMX-файла. Изменения в EDMX-файл, как это будет работать, но он также отключению проверки модели в Visual STUDIO. Поэтому если проверка модели вы всегда получите ошибки.

- Для этого необходимо добавить хранимую процедуру в модель, как это делается для одного результирующего набора запроса.
- После этого необходимо щелкнуть правой кнопкой мыши модель и выберите **открыть с помощью...** затем **Xml**



Создав модель открывается как XML, то необходимо выполнить следующие действия.

- Найти сложный тип и функцию импорта в модель:

```

<!-- CSDL content -->
<edmx:ConceptualModels>

...

<FunctionImport Name="GetAllBlogsAndPosts" ReturnType="Collection(BlogModel.GetAllBlogsAndPosts_Result)"
/>

...

<ComplexType Name="GetAllBlogsAndPosts_Result">
<Property Type="Int32" Name="BlogId" Nullable="false" />
<Property Type="String" Name="Name" Nullable="false" MaxLength="255" />
<Property Type="String" Name="Description" Nullable="true" />
</ComplexType>

...

</edmx:ConceptualModels>

```

- Удаление сложного типа
- Обновления, импорта функции, таким образом, чтобы он был сопоставлен сущностей, в нашем случае он будет выглядеть следующим образом:

```

<FunctionImport Name="GetAllBlogsAndPosts">
<ReturnType EntitySet="Blogs" Type="Collection(BlogModel.Blog)" />
<ReturnType EntitySet="Posts" Type="Collection(BlogModel.Post)" />
</FunctionImport>

```

Это сообщает модели, что хранимая процедура вернет двух коллекций, одну из записей в блогах и из операции post.

- Найдите элемент сопоставления функции:

```

<!-- C-S mapping content -->
<edmx:Mappings>

...

<FunctionImportMapping FunctionImportName="GetAllBlogsAndPosts"
FunctionName="BlogModel.Store.GetAllBlogsAndPosts">
<ResultMapping>
<ComplexTypeMapping TypeName="BlogModel.GetAllBlogsAndPosts_Result">
<ScalarProperty Name="BlogId" ColumnName="BlogId" />
<ScalarProperty Name="Name" ColumnName="Name" />
<ScalarProperty Name="Description" ColumnName="Description" />
</ComplexTypeMapping>
</ResultMapping>
</FunctionImportMapping>

...

</edmx:Mappings>

```

- Замените сопоставление результат одной для каждой сущности, возвращаемых, например следующие:

```

<ResultMapping>
  <EntityTypeMapping TypeName = "BlogModel.Blog">
    <ScalarProperty Name="BlogId" ColumnName="BlogId" />
    <ScalarProperty Name="Name" ColumnName="Name" />
    <ScalarProperty Name="Description" ColumnName="Description" />
  </EntityTypeMapping>
</ResultMapping>
<ResultMapping>
  <EntityTypeMapping TypeName="BlogModel.Post">
    <ScalarProperty Name="BlogId" ColumnName="BlogId" />
    <ScalarProperty Name="PostId" ColumnName="PostId"/>
    <ScalarProperty Name="Title" ColumnName="Title" />
    <ScalarProperty Name="Text" ColumnName="Text" />
  </EntityTypeMapping>
</ResultMapping>

```

Это также можно сопоставить с результирующими наборами сложные типы, например, созданный по умолчанию. Для этого создайте новый сложный тип, а отключите их и использовать сложные типы везде, вы использовали имена сущностей в приведенных выше примерах.

После изменились эти сопоставления можно сохранить модель и выполните следующий код, чтобы хранимая процедура:

```

using (var db = new BlogEntities())
{
  var results = db.GetAllBlogsAndPosts();

  foreach (var result in results)
  {
    Console.WriteLine("Blog: " + result.Name);
  }

  var posts = results.GetNextResult<Post>();

  foreach (var result in posts)
  {
    Console.WriteLine("Post: " + result.Title);
  }

  Console.ReadLine();
}

```

NOTE

Если вы вручную вносили в EDMX-файл для модели он будет перезаписан, если вы когда-либо повторно создать модель из базы данных.

Сводка

Здесь мы показали, два метода доступа к несколько результирующих наборов с помощью Entity Framework. Оба значения являются одинаково допустимыми зависимостями от конкретной ситуации и предпочтения и вам следует выбрать тот, который кажется наиболее обстоятельствах. Планируется, что поддержка нескольких результирующих наборов, будет улучшена в будущем версиях Entity Framework и, выполнив действия в этом документе будет больше не нужен.

Возвращающие табличные значения функции (TVF)

27.09.2018 • 6 minutes to read • [Edit Online](#)

NOTE

EF5 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в Entity Framework 5. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

Видео и пошаговые руководства показано, как сопоставить возвращающих табличные значения функции (TVF), с помощью Entity Framework Designer. Он также демонстрирует вызов функции с табличным значением из запроса LINQ.

Возвращающие табличное значение функции в настоящий момент поддерживается только в базе данных первого рабочего процесса.

Возвращающая табличное значение Функция поддержки появилась в версии 5 платформы Entity Framework. Обратите внимание, что для использования новых возможностей, таких как функции, возвращающие табличные значения, перечислимые типы и Пространственные типы .NET Framework 4.5 необходимо ориентироваться. Visual Studio 2012 предназначенного для .NET 4.5 по умолчанию.

Возвращающие табличное значение функции очень похожи на хранимые процедуры с одним основным отличием: составляется результатом функции с табличным значением. Это означает, что результаты из функции с табличным значением может использоваться в запросе LINQ, хотя результаты хранимой процедуры нельзя.

Просмотреть видео

Представленный: Юлия Корнич

[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

Предварительные требования

Для выполнения этого пошагового руководства, необходимо:

- Установка [базы данных School](#).
- У последняя версия Visual Studio

Настройка проекта

1. Открытие Visual Studio
2. На **файл** последовательно выберите пункты **New**, а затем нажмите кнопку **проекта**
3. В левой области щелкните **Visual C#**, а затем выберите **консоли** шаблона
4. Введите **возвращающей табличное значение функции** как имя проекта и нажмите кнопку **OK**

Добавление функции с табличным значением в базу данных

- Выберите **представление** —> **обозреватель объектов SQL Server**

- Если LocalDB отсутствует в списке серверов: щелкните правой кнопкой мыши **SQL Server** и выберите **добавить SQL Server** используйте значение по умолчанию **проверки подлинности Windows** для подключения к серверу LocalDB
- Разверните узел LocalDB
- В узле базы данных, щелкните правой кнопкой мыши узел базы данных School и выберите **новый запрос...**
- В редакторе T-SQL, вставьте следующее определение функции с табличным значением

```

CREATE FUNCTION [dbo].[GetStudentGradesForCourse]
(
@CourseID INT
)
RETURNS TABLE
AS
RETURN
SELECT [EnrollmentID],
       [CourseID],
       [StudentID],
       [Grade]
FROM   [dbo].[StudentGrade]
WHERE CourseID = @CourseID

```

- Щелкните правой кнопкой мыши в редакторе T-SQL и выберите **Execute**
- Функция GetStudentGradesForCourse будет добавлена в базу данных School

Создание модели

- Щелкните правой кнопкой мыши имя проекта в обозревателе решений, выберите пункт **добавить**, а затем нажмите кнопку **новый элемент**
- Выберите **данных** меню слева, а затем выберите **ADO.NET Entity Data Model** в **шаблоны** области
- Введите **TVFModel.edmx** имя файла, а затем нажмите кнопку **добавить**
- В диалоговом окне Выбор содержимого модели выберите **создать из базы данных**, а затем нажмите кнопку **Далее**
- Нажмите кнопку **новое подключение** ввод **(localdb)\mssqllocaldb** в текст имени сервера поле Ввод **School** для базы данных имя щелкните **OK**
- В поле выберите ваши объекты базы данных диалогового списка **таблиц** выберите **Person, StudentGrade, и курс** таблиц
- Выберите **GetStudentGradesForCourse** функции, расположенный в **хранимые процедуры и функции** узла Обратите внимание, что, начиная с Visual Studio 2012 г. конструктор сущностей позволяет пакетный Импорт Хранимые процедуры и функции
- Нажмите кнопку **Готово**
- Конструктор сущностей, который предоставляет область конструктора для изменения модели, отображается. Все объекты, которые выбраны в **Choose Your Database Objects** диалоговое окно добавляются в модель.
- По умолчанию результирующая форма каждый импортированный хранимой процедуры или функции автоматически становится новый сложный тип в модели сущности. Но нам нужно сопоставить с сущностью StudentGrade результаты функции GetStudentGradesForCourse: щелкните правой кнопкой мыши область конструктора и выберите **браузер моделей** в браузере моделей выберите **импортируемые функции**, а затем дважды щелкните **GetStudentGradesForCourse** функции в изменение импорта функции установите флажок **сущностей** и выберите **StudentGrade**

Сохранения и извлечения данных

Откройте файл, в котором определен метод Main. Добавьте следующий код в функцию Main.

Следующий код демонстрирует построение запроса, использующего Table-valued Function. Результаты запроса проецируются в анонимный тип, содержащий связанные название курса и связанных слушателям оценку больше или равно 3.5.

```
using (var context = new SchoolEntities())
{
    var CourseID = 4022;
    var Grade = 3.5M;

    // Return all the best students in the Microeconomics class.
    var students = from s in context.GetStudentGradesForCourse(CourseID)
                  where s.Grade >= Grade
                  select new
                  {
                      s.Person,
                      s.Course.Title
                  };

    foreach (var result in students)
    {
        Console.WriteLine(
            "Couse: {0}, Student: {1} {2}",
            result.Title,
            result.Person.FirstName,
            result.Person.LastName);
    }
}
```

Скомпилируйте и запустите приложение. Программа выдает следующие результаты.

```
Couse: Microeconomics, Student: Arturo Anand
Couse: Microeconomics, Student: Carson Bryant
```

Сводка

В этом пошаговом руководстве мы рассмотрели способ сопоставления возвращающих табличные значения функции (TVF), с помощью Entity Framework Designer. Кроме того, вы узнали, как для вызова функции с табличным значением из запроса LINQ.

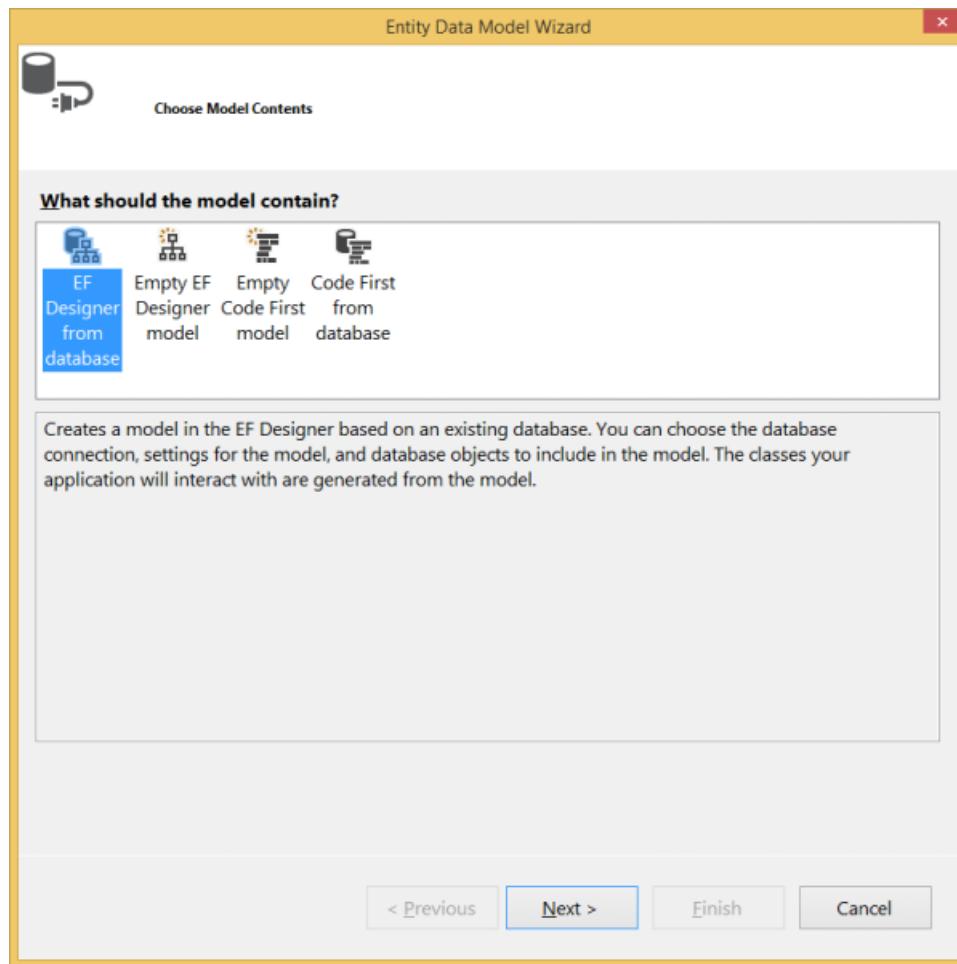
Entity Framework сочетания клавиш в конструкторе

13.09.2018 • 12 minutes to read • [Edit Online](#)

Эта страница предоставляет список сочетания клавиатуры, которые доступны в различных экранов инструменты Entity Framework для Visual Studio.

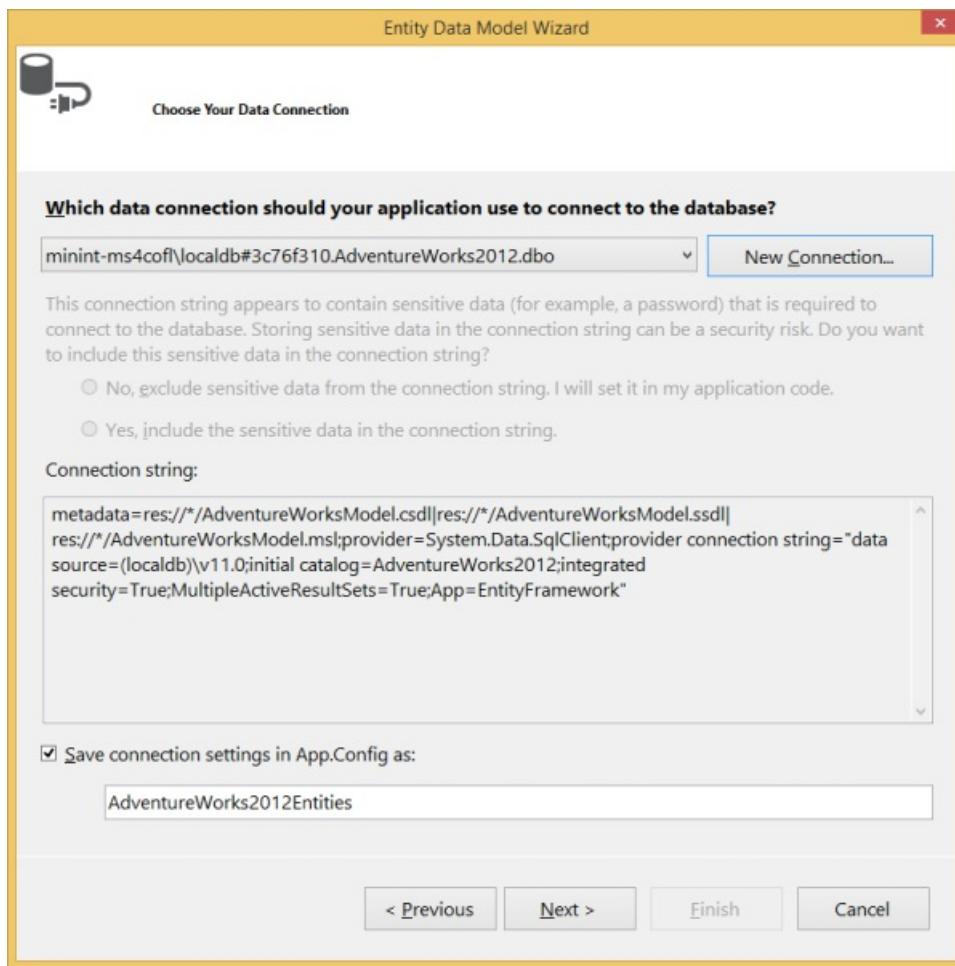
Мастер моделей EDM ADO.NET

Шаг один: Выбор содержимого модели



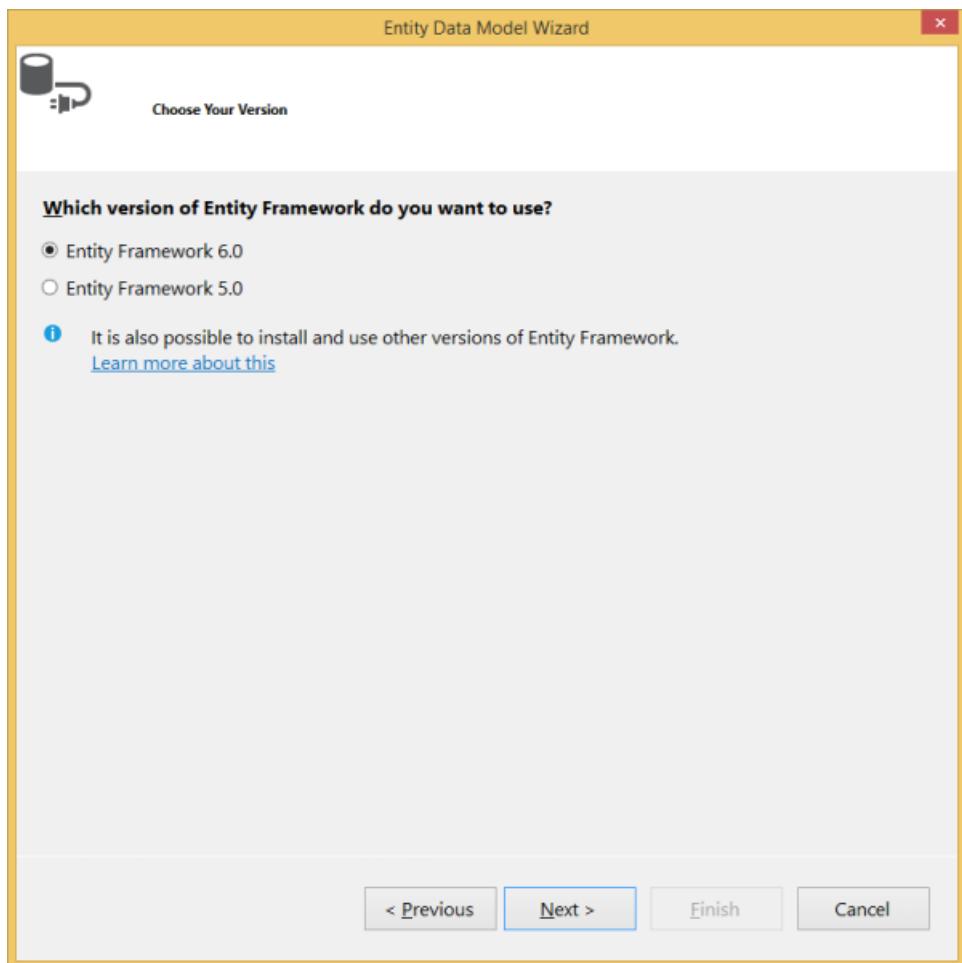
СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
ALT + n	Перемещение к следующему экрану	Не доступно для всех выбранных элементов для модели содержимого.
ALT + f	Завершение работы мастера	Не доступно для всех выбранных элементов для модели содержимого.
ALT + w	Переключить фокус на «что должна содержать модель?» область.	

Шаг 2: Выбор подключения



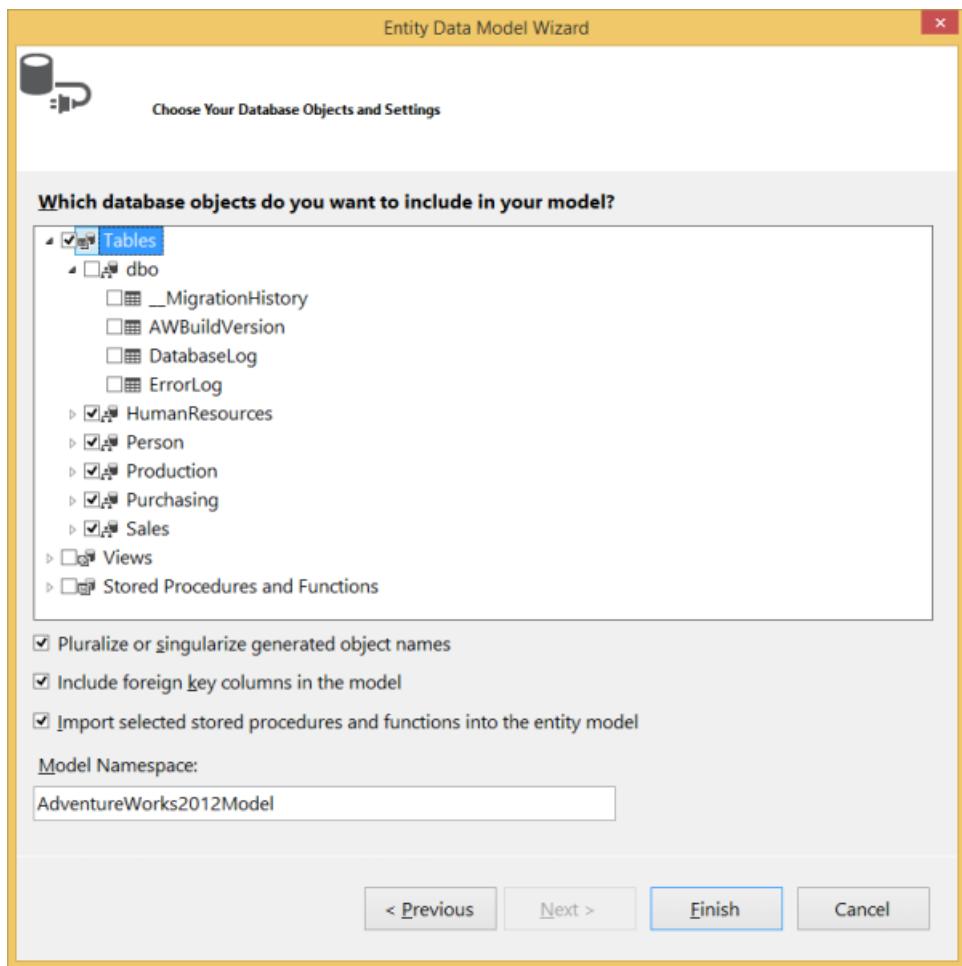
СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
ALT + n	Перемещение к следующему экрану	
ALT + p	Перемещение к предыдущему экрану	
ALT + w	Переключить фокус на «что должна содержать модель?» область.	
ALT + c	Откройте окно «Свойства подключения»	Позволяет определять новые соединения с базой данных.
ALT + e	Исключить конфиденциальные данные из строки подключения	
ALT + i	Включить конфиденциальные данные в строке подключения	
ALT + s	Укажите для параметра «Сохранить параметры подключения в файле App.Config»	

Шаг 3: Выбор версии



СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
ALT + n	Перемещение к следующему экрану	
ALT + p	Перемещение к предыдущему экрану	
ALT + w	Переключить фокус на выбор версии Entity Framework	Позволяет указать другую платформы Entity Framework для использования в проекте.

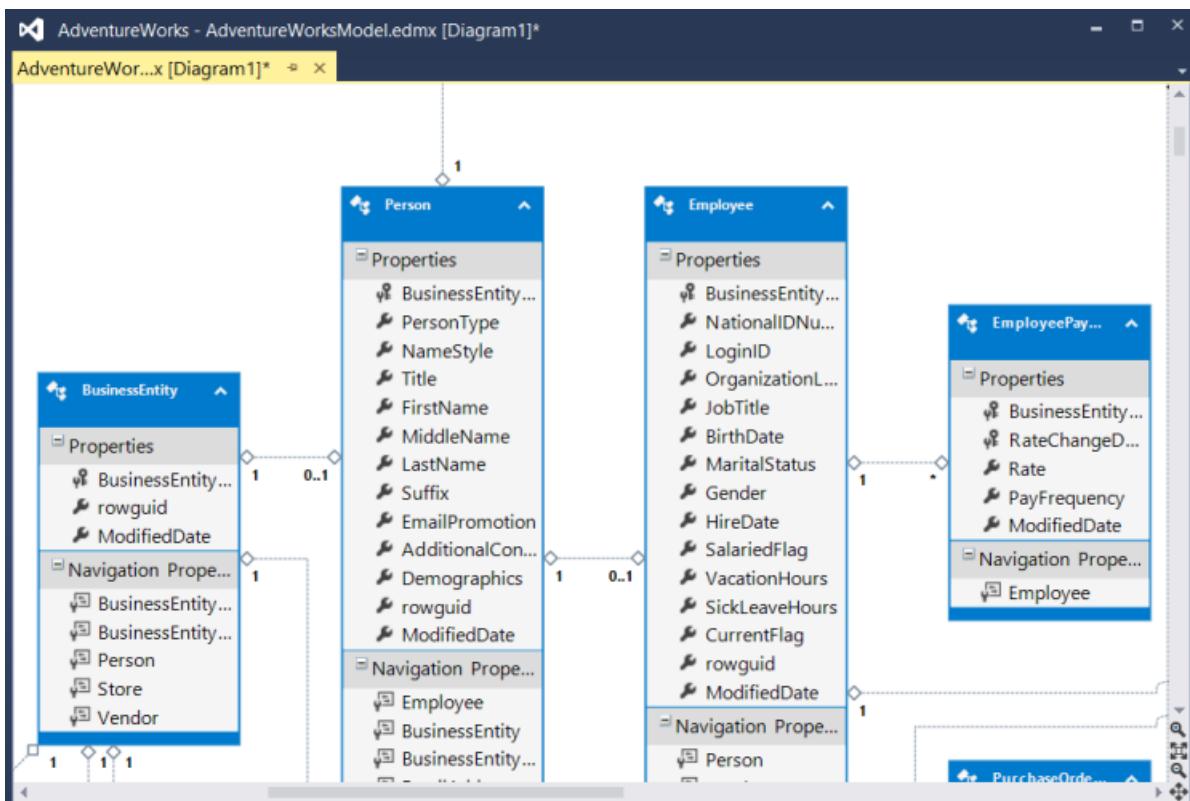
Шаг 4: Выбор объектов базы данных и параметры



СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
ALT + f	Завершение работы мастера	
ALT + p	Перемещение к предыдущему экрану	
ALT + w	Переключить фокус на панели выбора объектов базы данных	Позволяет создавать для указания объектов базы данных для обратного разбору.
ALT + s	Переключатель «множественном или единственном числе имена создаваемых объектов» параметр	
ALT + k	Укажите для параметра «Включить столбцы внешнего ключа в модели»	Не доступно для всех выбранных элементов для модели содержимого.
ALT + i	Переключить режим «Импортировать выбранные хранимые процедуры и функции в модели»	Не доступно для всех выбранных элементов для модели содержимого.
ALT + m	Перенос фокуса ввода в текстовое поле «Пространство имен модели»	Не доступно для всех выбранных элементов для модели содержимого.
ПРОБЕЛ	Переключить выделение в элементе	Если элемент имеет дочерние элементы, все дочерние элементы будут включаться также
Слева	Свернуть дочернего дерева	

СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
Справа	Разверните дерево дочерних	
Вверх	Перейдите на предыдущий элемент в дереве	
Вниз	Перейдите к следующему элементу в дереве	

Область конструктора EF



СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
Или введите пространства	Переключить выделение	Переключает выбор в объект с фокусом.
ESC	Отменить выделение	Отмена текущего выделения.
CTRL + A	Выделить все	Выбирает все фигуры на поверхности разработки.
Стрелка вверх	Переместить вверх	Перемещение выбранных сущностей один шаг сетки вверх. Если в списке, перемещается к предыдущему вложенному.
Стрелка вниз	Переместить вниз	Перемещение выбранных сущностей вниз один шаг сетки. Если в списке, перемещает Далее вложенному.

СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
СТРЕЛКА ВЛЕВО	Переместить влево	Перемещение выбранных сущностей один шаг сетки влево. Если в списке, перемещается к предыдущему вложенному.
СТРЕЛКА ВПРАВО	Переместить вправо	Перемещение выделенных шаг сетки вправо на одну сущность. Если в списке, перемещает Далее вложенному.
Shift + Стрелка влево	Размер фигуры влево	Уменьшение ширины элемента выбранной сущности на один шаг сетки.
Shift + Стрелка вправо	Размер фигуры вправо	Увеличение ширины элемента выбранной сущности на один шаг сетки.
Корневая папка	Первый узел	Перемещение фокуса и выделение на первый объект в области конструктора на том же уровне.
END	Последний однорангового узла	Перемещение фокуса и выделения до последнего объекта в области конструктора на том же уровне.
CTRL + Home	Первый узел (фокус)	Совпадает со значением первого узла, но перемещение фокуса без перемещения фокуса и выделения.
Ctrl + End	Последний однорангового узла (фокус)	Так же, как Дата последнего узла, но можно перемещать фокус без перемещения фокуса и выделения.
TAB	Далее однорангового узла	Перемещение фокуса и выделение на следующий объект в области конструктора на том же уровне.
SHIFT+TAB	Предыдущий однорангового узла	Перемещение фокуса и выделение предыдущего объекта в области конструктора на том же уровне.
Alt + Ctrl + Tab	Далее однорангового узла (фокус)	Так же, как следующего узла, но можно перемещать фокус без перемещения фокуса и выделения.
Alt + Ctrl + Shift + Tab	Предыдущий узел (фокус)	Совпадает со значением предыдущего однорангового узла, но перемещение фокуса без перемещения фокуса и выделения.

СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
<	Перемещения вверх	Переход к следующему объекту на проектирование поверхности один уровень выше в иерархии. Если нет ни одной фигуры, расположенные уровнем выше в иерархии (то есть объект помещается непосредственно в рабочей области конструирования), выбирается схема.
>	Опускаются	Переходит к следующей содержащейся объект на поверхности разработки один уровень, ниже в иерархии. Если не содержащегося объекта, это холостой.
CTRL + <	Ascend (фокус)	Так же, как ascend команды, однако перемещение фокуса без выделения.
CTRL + >	Опускаются (фокус)	Так же, как опускаются команды, однако перемещение фокуса без выделения.
Shift + End	Выполните подключенный	Из сущности перемещает сущность, которая подключена к этой сущности.
DEL	Удаление	Удалите из схемы объекта или соединитель.
Модули	Insert	Добавляет новое свойство сущности, при выборе заголовка секции «Скалярные свойства» или само свойство.
Стр.	Схема прокрутки вверх	Прокручивает область конструктора, с шагом, равным 75% от высоты в области конструктора, видимой в данный момент.
Стр.	Схема прокрутки вниз	Прокручивает область конструктора вниз.
SHIFT + стр.	Схема прокрутки вправо	Прокручивает область конструктора, чтобы справа.
SHIFT + стр.	Прокрутите схему влево	Прокручивает область конструктора, чтобы слева.
F2	Режим редактирования	Стандартные сочетания клавиш для вход в режим правки для текстового элемента управления.
SHIFT + F10	Отображение контекстного меню	Стандартные сочетания клавиш для отображения контекстного меню выбранного элемента.

СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
Control + Shift + мыши щелчок левой кнопкой Control + Shift + колесико мыши вперед	Семантический масштаб в	Увеличение масштаба области под курсором мыши представления "Схема".
Control + Shift + мыши щелкните правой кнопкой мыши Control + Shift + колесико мыши назад	Семантический масштаб Out	Уменьшает масштаб в области представления "Схема" под курсором мыши. Повторно диаграмме центров, при уменьшении слишком далеко для текущего центра схеме.
Удерживая нажатой клавишу CTRL Shift + «+» Control + колесико мыши вперед	Увеличить	Увеличение в центре представления "Схема".
Удерживая нажатой клавишу CTRL Shift + “-” Control + колесико мыши назад	Уменьшить	Уменьшает масштаб из выбранной области представления "Схема". Повторно диаграмме центров, при уменьшении слишком далеко для текущего центра схеме.
Управлять + Shift + нарисовать прямоугольник с левой кнопки мыши	Области увеличения	Увеличение по центру области, которые вы выбрали. При удержании нажатой клавиши Shift + элемента управления, вы увидите сообщение о том, что курсор изменяется в лупу, можно задать область для увеличения масштаба.
Клавиша контекстного меню + меня "	Откройте окно сведений о сопоставлении	Откроется окно сведения о сопоставлении для изменения сопоставления для выбранной сущности

Окно «Сведения о сопоставлении»

The screenshot shows the 'Mapping Details - Person' window with the following details:

- Tables:** Shows 'Maps to Person' and 'Column Mappings' sections.
- Column Mappings:**

Column	Operat...	Value / Property
BusinessEntityID : int	→	BusinessEntityID : Int32
PersonType : nchar	→	PersonType : String
NameStyle : bit	→	NameStyle : Boolean
Title : nvarchar	→	Title : String
FirstName : nvarchar	→	FirstName : String
MiddleName : nvarchar	→	MiddleName : String
LastName : nvarchar	→	LastName : String
Suffix : nvarchar	→	Suffix : String
EmailPromotion : int	→	EmailPromotion : Int32
AdditionalContactInfo : xml	→	AdditionalContactInfo : String
Demographics : xml	→	Demographics : String
rowguid : uniqueidentifier	→	rowguid : Guid
ModifiedDate : datetime	→	ModifiedDate : DateTime
- Additions:** Includes '<Add a Condition>' and '<Add a Table or View>' options.

СОЧЕТАНИЕ КЛАВИШ	ДЕЙСТВИЕ	ПРИМЕЧАНИЯ
TAB	Переключение контекста	Переключение между основную область окна и панели инструментов слева
Клавиши со стрелками	Навигация	Перемещение вверх и вниз строки, или вправо и влево по столбцам в основную область окна. Перемещение между кнопками на панели инструментов в левой части.
ВВОД ПРОБЕЛ	Выбрать	Выбор кнопки на панели инструментов в левой части.
Alt + Стрелка вниз	Открытие списка	Раскрывающийся список, если ячейка выбрана, имеющий раскрывающегося списка.
ВВОД	Выберите	Выбирает элемент в раскрывающемся списке.
ESC	Закройте список	Закрытие раскрывающегося списка.

Visual Studio навигации

Платформа Entity Framework также предоставляет ряд определенных действий, которые могут иметь сопоставленные пользовательских сочетаний клавиш (по умолчанию сопоставлены никакие ярлыки). Чтобы создать эти настраиваемые сочетания клавиш, щелкните меню "Сервис", а затем параметры. В среде выберите клавиатура. Назначьте прокрутите вниз список в середине пока можно выбрать нужную команду, введите в текстовом поле «Ведите сочетание клавиш» и нажмите кнопку. Ниже приведены возможные сочетания клавиш.

СОЧЕТАНИЕ КЛАВИШ
OtherContextMenus.MicrosoftDataEntityDesignContext.Add.ComplexProperty.ComplexTypes
OtherContextMenus.MicrosoftDataEntityDesignContext.AddCodeGenerationItem
OtherContextMenus.MicrosoftDataEntityDesignContext.AddFunctionImport
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.AddEnumType
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.Association
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.ComplexProperty
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.ComplexType
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.Entity
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.FunctionImport
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.Inheritance

СОЧЕТАНИЕ КЛАВИШ

OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.NavigationProperty

OtherContextMenus.MicrosoftDataEntityDesignContext.AddNew.ScalarProperty

OtherContextMenus.MicrosoftDataEntityDesignContext.AddNewDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.AddtoDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.Close

OtherContextMenus.MicrosoftDataEntityDesignContext.Collapse

OtherContextMenus.MicrosoftDataEntityDesignContext.ConverttoEnum

OtherContextMenus.MicrosoftDataEntityDesignContext.Diagram.CollapseAll

OtherContextMenus.MicrosoftDataEntityDesignContext.Diagram.ExpandAll

OtherContextMenus.MicrosoftDataEntityDesignContext.Diagram.ExportasImage

OtherContextMenus.MicrosoftDataEntityDesignContext.Diagram.LayoutDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.Edit

OtherContextMenus.MicrosoftDataEntityDesignContext.EntityKey

OtherContextMenus.MicrosoftDataEntityDesignContext.Expand

OtherContextMenus.MicrosoftDataEntityDesignContext.FunctionImportMapping

OtherContextMenus.MicrosoftDataEntityDesignContext.GenerateDatabasefromModel

OtherContextMenus.MicrosoftDataEntityDesignContext.GoToDefinition

OtherContextMenus.MicrosoftDataEntityDesignContext.Grid>ShowGrid

OtherContextMenus.MicrosoftDataEntityDesignContext.Grid.SnaptoGrid

OtherContextMenus.MicrosoftDataEntityDesignContext.IncludeRelated

OtherContextMenus.MicrosoftDataEntityDesignContext.Mapping Details

OtherContextMenus.MicrosoftDataEntityDesignContext.ModelBrowser

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveDiagramstoSeparateFile

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Down

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Down5

СОЧЕТАНИЕ КЛАВИШ

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.ToBottom

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.ToTop

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Up

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Up5

OtherContextMenus.MicrosoftDataEntityDesignContext.MovetonewDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.Open

OtherContextMenus.MicrosoftDataEntityDesignContext.Refactor.MoveToNewComplexType

OtherContextMenus.MicrosoftDataEntityDesignContext.Refactor.Rename

OtherContextMenus.MicrosoftDataEntityDesignContext.RemovefromDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.Rename

OtherContextMenus.MicrosoftDataEntityDesignContext.ScalarPropertyFormat.DisplayName

OtherContextMenus.MicrosoftDataEntityDesignContext.ScalarPropertyFormat.DisplayNameandType

OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Base Type

OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Entity

OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Property

OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Subtype

OtherContextMenus.MicrosoftDataEntityDesignContext.SelectAll

OtherContextMenus.MicrosoftDataEntityDesignContext.SelectAssociation

OtherContextMenus.MicrosoftDataEntityDesignContext.ShowinDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.ShowinModelBrowser

OtherContextMenus.MicrosoftDataEntityDesignContext.StoredProcedureMapping

OtherContextMenus.MicrosoftDataEntityDesignContext.TableMapping

OtherContextMenus.MicrosoftDataEntityDesignContext.UpdateModelfromDatabase

OtherContextMenus.MicrosoftDataEntityDesignContext.Validate

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.10

СОЧЕТАНИЕ КЛАВИШ

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.100

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.125

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.150

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.200

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.25

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.300

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.33

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.400

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.50

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.66

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.75

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.Custom

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.ZoomIn

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.ZoomOut

OtherContextMenus.MicrosoftDataEntityDesignContext.Zoom.ZoomtoFit

View.EntityDataModelBrowser

View.EntityDataModelMappingDetails

Запросы и поиск сущностей

13.09.2018 • 5 minutes to read • [Edit Online](#)

В этом разделе рассматриваются различные способы запроса данных с помощью Entity Framework, включая LINQ и метод Find. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Поиск сущностей с помощью запроса

DbSet и IDbSet реализуют IQueryable, поэтому их можно использовать как начальную точку для написания запроса LINQ к базе данных. Здесь мы не будем подробно рассматривать LINQ, но приведем несколько простых примеров:

```
using (var context = new BloggingContext())
{
    // Query for all blogs with names starting with B
    var blogs = from b in context.Blogs
                where b.Name.StartsWith("B")
                select b;

    // Query for the Blog named ADO.NET Blog
    var blog = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .FirstOrDefault();
}
```

Обратите внимание, что DbSet и IDbSet всегда создают запросы к базе данных и всегда используют круговой путь к базе данных, даже если возвращаемые сущности уже присутствуют в контексте. Запрос выполняется в базе данных, если:

- Он перечисляется с помощью инструкции **foreach** (C#) или **For Each** (Visual Basic).
- Он перечисляется операцией коллекции, например **ToArray**, **ToDictionary** или **ToList**.
- Во внешней части запроса заданы операторы LINQ, например **First** или **Any**.
- Вызываются следующие методы: метод расширения **Load** в DbSet, **DbEntityEntry.Reload** и **Database.ExecuteSqlCommand**.

Когда из базы данных возвращаются результаты, объекты, отсутствующие в контексте, присоединяются к контексту. Если объект уже есть в контексте, возвращается существующий объект (текущие и исходные значения свойств объекта в записи **не** переписываются значениями из базы данных).

Когда вы выполняете запрос, сущности, которые были добавлены в контекст, но еще не были сохранены в базе данных, не возвращаются в составе результирующего набора. Чтобы получить данные из контекста, см. раздел [Локальные данные](#).

Если запрос не возвращает строки из базы данных, результатом будет пустая коллекция, а не **NULL**.

Поиск сущностей с помощью первичных ключей

Метод Find в классе DbSet использует значение первичного ключа, чтобы найти сущность, отслеживаемую контекстом. Если сущность не найдена в контексте, запрос отправляется в базу данных для поиска сущности там. Если сущность не найдена в контексте или в базе данных, возвращается значение NULL.

Метод Find имеет два важных отличия от запроса:

- Круговой путь к базе данных будет использоваться только в том случае, если сущность с указанным ключом не найдена в контексте.
- Метод Find возвращает сущности в состоянии "Добавлено". Это значит, что метод Find возвращает сущности, которые были добавлены в контекст, но еще не были сохранены в базе данных.

Поиск сущности по первичному ключу

В коде ниже приведено несколько примеров использования метода Find.

```
using (var context = new BloggingContext())
{
    // Will hit the database
    var blog = context.Blogs.Find(3);

    // Will return the same instance without hitting the database
    var blogAgain = context.Blogs.Find(3);

    context.Blogs.Add(new Blog { Id = -1 });

    // Will find the new blog even though it does not exist in the database
    var newBlog = context.Blogs.Find(-1);

    // Will find a User which has a string primary key
    var user = context.Users.Find("johndoe1987");
}
```

Поиск сущности по составному первичному ключу

Платформа Entity Framework позволяет сущностям иметь составные ключи, то есть ключи, состоящие из нескольких свойств. Например, вы можете создать сущность BlogSettings, которая представляет собой параметры пользователей для конкретного блога. Так как пользователю необходима только одна сущность BlogSettings для каждого блога, первичный ключ для BlogSettings может состоять из комбинации идентификатора блога и имени пользователя. Следующий код пытается найти BlogSettings по идентификатору = 3 и имени пользователя = johndoe1987:

```
using (var context = new BloggingContext())
{
    var settings = context.BlogSettings.Find(3, "johndoe1987");
}
```

Если у вас есть составные ключи, вам нужно использовать ColumnAttribute или текущий API, чтобы указать порядок свойств составного ключа. В вызове метода Find эти значения ключа должны указываться в том же порядке.

Метод Load

13.09.2018 • 2 minutes to read • [Edit Online](#)

Существует несколько сценариев, где может потребоваться загрузка сущностей из базы данных в контекст, не выполняя никаких действий с этими сущностями немедленно. Хорошим примером этого загрузка сущностей для привязки данных, как описано в разделе [локальных данных](#). Один из способов сделать это — написать запрос LINQ, а затем вызывать `ToList` в его, только для немедленного удаления созданный список. Метода расширения `Load` работает так же, как `ToList`, за исключением того, что позволяет избежать создания списка полностью.

Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Ниже приведены два примера использования нагрузки. Первый берется из приложения привязки данных Windows Forms, где нагрузки используется для запроса сущностей перед привязкой к локальной коллекции, как описано в разделе [локальных данных](#):

```
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);

    _context = new ProductContext();

    _context.Categories.Load();
    categoryBindingSource.DataSource = _context.Categories.Local.ToBindingList();
}
```

Второй пример показывает использование загрузки для загрузки фильтрованную коллекцию связанных сущностей, как описано в [загрузка связанных сущностей](#):

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog
    context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();
}
```

Локальные данные

13.09.2018 • 16 minutes to read • [Edit Online](#)

Выполняется запрос LINQ непосредственно к DbSet, всегда отправляет запрос к базе данных, но можно получить доступ к данным, который в данный момент в памяти с помощью свойства DbSet.Local. Можно также открыть нужные дополнительные данные отслеживания EF о сущности с помощью методов DbContext.Entry и DbContext.ChangeTracker.Entries. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Использовать локальную для просмотра локальных данных

Локальное свойство DbSet предоставляет простой доступ к сущностям из набора, которые отслеживаются контекстом и не были помечены как удаленные. Доступ к локальному свойству никогда не вызывает запрос для отправки в базу данных. Это означает, что оно обычно используется после запроса уже была выполнена. Метода расширения Load может использоваться для выполнения запроса, чтобы контекст отслеживает результаты. Пример:

```
using (var context = new BloggingContext())
{
    // Load all blogs from the database into the context
    context.Blogs.Load();

    // Add a new blog to the context
    context.Blogs.Add(new Blog { Name = "My New Blog" });

    // Mark one of the existing blogs as Deleted
    context.Blogs.Remove(context.Blogs.Find(1));

    // Loop over the blogs in the context.
    Console.WriteLine("In Local: ");
    foreach (var blog in context.Blogs.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            blog.BlogId,
            blog.Name,
            context.Entry(blog).State);
    }

    // Perform a query against the database.
    Console.WriteLine("\nIn DbSet query: ");
    foreach (var blog in context.Blogs)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            blog.BlogId,
            blog.Name,
            context.Entry(blog).State);
    }
}
```

Если у нас два блоги в базе данных - «ADO.NET блог» с BlogId 1 - и «Блоге по Visual Studio» с BlogId 2 мы могла бы ожидать следующие выходные данные:

```
In Local:
```

```
Found 0: My New Blog with state Added  
Found 2: The Visual Studio Blog with state Unchanged
```

```
In DbSet query:
```

```
Found 1: ADO.NET Blog with state Deleted  
Found 2: The Visual Studio Blog with state Unchanged
```

Это иллюстрирует три точки:

- Новый блог «Мой новый блог» входит в локальную коллекцию, несмотря на то, что он еще не были сохранены в базе данных. Этот блог имеет первичный ключ нулевой, так как в базе данных еще не создал фактическую клавишу для сущности.
- В «записи ADO.NET» отсутствует в локальной коллекции, несмотря на то, что он по-прежнему отслеживается контекстом. Это так, как мы удалили из DbSet, тем самым пометив его как удаленные.
- Когда DbSet, используется для выполнения запроса блога, помечен для удаления (блог ADO.NET) включается в результаты и нового блога (Мой новый блог), который еще не были сохранены в базе данных не включается в результаты. Это обусловлено DbSet выполняет запрос к базе данных и результатов, возвращаемых всегда оно совпадало с указанным в базе данных.

Использовать локальную для добавления и удаления сущностей из контекста

Возвращает свойства локально DbSet [ObservableCollection](#) с событиями, привязываем ее таким образом, что он остается синхронизированной с содержимое контекста. Это означает, что сущности можно добавлять или удалять из локальной коллекции или DbSet. Это также означает, что запросы, которые привносят новые сущности в контекст, приведет к локальной коллекции, обновляемой с этими сущностями.

Пример:

```

using (var context = new BloggingContext())
{
    // Load some posts from the database into the context
    context.Posts.Where(p => p.Tags.Contains("entity-framework")).Load();

    // Get the local collection and make some changes to it
    var localPosts = context.Posts.Local;
    localPosts.Add(new Post { Name = "What's New in EF" });
    localPosts.Remove(context.Posts.Find(1));

    // Loop over the posts in the context.
    Console.WriteLine("In Local after entity-framework query: ");
    foreach (var post in context.Posts.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            post.Id,
            post.Title,
            context.Entry(post).State);
    }

    var post1 = context.Posts.Find(1);
    Console.WriteLine(
        "State of post 1: {0} is {1}",
        post1.Name,
        context.Entry(post1).State);

    // Query some more posts from the database
    context.Posts.Where(p => p.Tags.Contains("asp.net")).Load();

    // Loop over the posts in the context again.
    Console.WriteLine("\nIn Local after asp.net query: ");
    foreach (var post in context.Posts.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            post.Id,
            post.Title,
            context.Entry(post).State);
    }
}

```

При условии, что у нас было несколько помеченные с «entity framework» и «asp.net» выходные данные записи может выглядеть следующим образом:

```

In Local after entity-framework query:
Found 3: EF Designer Basics with state Unchanged
Found 5: EF Code First Basics with state Unchanged
Found 0: What's New in EF with state Added
State of post 1: EF Beginners Guide is Deleted

In Local after asp.net query:
Found 3: EF Designer Basics with state Unchanged
Found 5: EF Code First Basics with state Unchanged
Found 0: What's New in EF with state Added
Found 4: ASP.NET Beginners Guide with state Unchanged

```

Это иллюстрирует три точки:

- «Новые возможности в EF» новую запись, добавленный к локальной коллекции становится отслеживаемая в контексте в добавленном состоянии. Она будет таким образом вставлена в базу данных при вызове метода SaveChanges.
- Запрос post, который был удален из локальной коллекции (руководство по профилированию EF) теперь

помечены как удаленные в контексте. Его таким образом удаляются из базы данных при вызове метода SaveChanges.

- Дополнительные post (руководство по профилированию ASP.NET), загруженной в контекст, с помощью второй запрос автоматически добавляется к локальной коллекции.

Заключительную следует помнить о локальной обусловлено тем, что ObservableCollection производительности не очень удобно для большого количества сущностей. Поэтому если вы имеете дело с тысячами сущностей в контексте может оказаться рекомендуется использовать локальный.

Использовать локальную для привязки данных WPF

Свойства DbSet локально можно использовать непосредственно для привязки данных в приложении WPF, так как он является экземпляром ObservableCollection. Как описано в предыдущих разделах, это означает, что он будет автоматически синхронизироваться с содержимое контекста и содержимое контекста будут автоматически синхронизированы с ним. Обратите внимание на то, что вам нужно предварительно заполнить локальную коллекцию с данными не исключена действий, которые необходимо привязать к, так как локальный никогда не вызывает запрос к базе данных.

Это не является подходящим местом для полный пример привязки данных WPF, но являются ключевые элементы:

- Настройка источника привязки
- Привяжите его к локальному свойству набора параметров
- Заполнение локальной, с помощью запроса к базе данных.

Привязки WPF для свойств навигации

Если вы выполняете данных «основной/подробности», то может потребоваться привязать к свойству навигации, одной из сущностей представлении «Подробности». Простой способ выполнения этой работы является использование коллекции ObservableCollection для свойства навигации. Пример:

```
public class Blog
{
    private readonly ObservableCollection<Post> _posts =
        new ObservableCollection<Post>();

    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual ObservableCollection<Post> Posts
    {
        get { return _posts; }
    }
}
```

Использовать локальную для очистки сущностей в SaveChanges

В большинстве случаев сущности удалены из свойства навигации не помечается, автоматически как удаленный в контексте. Например при удалении объекта Post из коллекции Blog.Posts, затем выполните, не удаляются автоматически при вызове метода SaveChanges. Если это необходимо для удаления затем может потребоваться найти эти несвязанные сущности и пометить их как удаленные, перед вызовом метода SaveChanges, или как часть переопределенного метода SaveChanges. Пример:

```
public override int SaveChanges()
{
    foreach (var post in this.Posts.Local.ToList())
    {
        if (post.Blog == null)
        {
            this.Posts.Remove(post);
        }
    }

    return base.SaveChanges();
}
```

Приведенный выше код использует локальную коллекцию, чтобы найти все сообщения и метки, которые не имеют ссылку на блог как удаленные. Вызов `ToList` является обязательным, поскольку в противном случае коллекции будет изменена `Remove` вызвать во время его перечисления. В большинстве случаев вы можете запрашивать непосредственно с локального свойства без использования `ToList` сначала.

Использование привязки данных для локальных и `ToBindingList` for Windows Forms

Windows Forms не поддерживает привязку данных полноценной непосредственно с помощью `ObservableCollection`. Тем не менее чтобы получить все преимущества, описанные в предыдущих разделах по-прежнему можно использовать `DbSet` локального свойства для привязки данных. Это достигается с помощью метода расширения `ToBindingList`, который создает `IBindingList` реализации поддерживающий локальной коллекции `ObservableCollection`.

Это не является подходящим местом для полный пример привязки данных Windows Forms, но являются ключевые элементы:

- Настройка объекта источника привязки
- Привяжите его к локальному свойству с помощью `Local.ToBindingList()` набора
- Заполнение локальной, с помощью запроса к базе данных

Получение подробных сведений о отслеживаемые сущности

Во многих примерах в этой серии запись метод позволяет вернуть экземпляр `dbentityentry`, который должен для сущности. Затем этот объект записи выступает в качестве отправной точки для сбора сведений о сущности, например текущего состояния, а также для выполнения операций с сущностями, например явная загрузка связанных сущностей.

Методы операции возвращают объекты `dbentityentry`, который должен для нескольких или всех сущностей, отслеживается контекстом. Это позволяет собирать информацию или выполнить операции на множество сущностей, а не только одна запись. Пример:

```

using (var context = new BloggingContext())
{
    // Load some entities into the context
    context.Blogs.Load();
    context.Authors.Load();
    context.Readers.Load();

    // Make some changes
    context.Blogs.Find(1).Title = "The New ADO.NET Blog";
    context.Blogs.Remove(context.Blogs.Find(2));
    context.Authors.Add(new Author { Name = "Jane Doe" });
    context.Readers.Find(1).Username = "johndoe1987";

    // Look at the state of all entities in the context
    Console.WriteLine("All tracked entities: ");
    foreach (var entry in context.ChangeTracker.Entries())
    {
        Console.WriteLine(
            "Found entity of type {0} with state {1}",
            ObjectContext.GetObjectType(entry.Entity.GetType()).Name,
            entry.State);
    }

    // Find modified entities of any type
    Console.WriteLine("\nAll modified entities: ");
    foreach (var entry in context.ChangeTracker.Entries()
        .Where(e => e.State == EntityState.Modified))
    {
        Console.WriteLine(
            "Found entity of type {0} with state {1}",
            ObjectContext.GetObjectType(entry.Entity.GetType()).Name,
            entry.State);
    }

    // Get some information about just the tracked blogs
    Console.WriteLine("\nTracked blogs: ");
    foreach (var entry in context.ChangeTracker.Entries<Blog>())
    {
        Console.WriteLine(
            "Found Blog {0}: {1} with original Name {2}",
            entry.Entity.BlogId,
            entry.Entity.Name,
            entry.Property(p => p.Name).OriginalValue);
    }

    // Find all people (author or reader)
    Console.WriteLine("\nPeople: ");
    foreach (var entry in context.ChangeTracker.Entries<IPerson>())
    {
        Console.WriteLine("Found Person {0}", entry.Entity.Name);
    }
}

```

Можно заметить, мы представляем класс автора и чтения в пример — оба эти класса реализуют интерфейс IPerson.

```

public class Author : IPerson
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public string Biography { get; set; }
}

public class Reader : IPerson
{
    public int ReaderId { get; set; }
    public string Name { get; set; }
    public string Username { get; set; }
}

public interface IPerson
{
    string Name { get; }
}

```

Предположим, что у нас есть следующие данные в базе данных:

Блог с BlogId = 1 "и" имя = «Блог ADO.NET»
 Блог с BlogId = 2, а также имя = «Блог Visual Studio»
 Блог с BlogId = 3 "и" имя = «Блог по .NET Framework»
 Разработка с помощью AuthorId = 1 "и" имя = «Joe Bloggs»
 Средства чтения с ReaderId = 1 "и" имя = «John Doe»

В результате выполнения кода будет следующим:

```

All tracked entities:
Found entity of type Blog with state Modified
Found entity of type Blog with state Deleted
Found entity of type Blog with state Unchanged
Found entity of type Author with state Unchanged
Found entity of type Author with state Added
Found entity of type Reader with state Modified

All modified entities:
Found entity of type Blog with state Modified
Found entity of type Reader with state Modified

Tracked blogs:
Found Blog 1: The New ADO.NET Blog with original Name ADO.NET Blog
Found Blog 2: The Visual Studio Blog with original Name The Visual Studio Blog
Found Blog 3: .NET Framework Blog with original Name .NET Framework Blog

People:
Found Person John Doe
Found Person Joe Bloggs
Found Person Jane Doe

```

В следующих примерах показаны несколько точек:

- Методы записи, возвращают записи для сущности во всех состояниях, включая удаленные. Сравните это для локального параметра, который исключает удаление сущностей.
- При использовании метода нестандартную записей, возвращаются записи для всех типов сущностей. При использовании метода универсального записи только возвращаются записи для сущностей, которые являются экземплярами универсального типа. Использовался выше для получения записи для всех блогах. Он также использовался для получения записи для всех сущностей, которые реализуют IPerson. Это показывает, что универсальный тип имеет тип фактического объекта.
- LINQ к объектам можно использовать для фильтрации возвращаемых результатов. Использовался выше

для поиска объектов любого типа, до тех пор, пока они будут изменены.

Обратите внимание, что экземпляры dbentityentry, который должен всегда содержать сущности отличные от null. Записей отношений и записи заглушки не представлены в виде экземпляров dbentityentry, который должен, нет необходимости для фильтрации для них.

и без отслеживания

13.09.2018 • 2 minutes to read • [Edit Online](#)

Иногда может потребоваться вернуться к сущности на основе запроса, но нет элементов, отслеживаемые по контексту. Это может привести к повышению производительности, при запросе большого количества сущностей в сценариях только для чтения. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Новый метод расширения `AsNoTracking` позволяет любому запросу, чтобы выполняться таким образом.

Пример:

```
using (var context = new BloggingContext())
{
    // Query for all blogs without tracking them
    var blogs1 = context.Blogs.AsNoTracking();

    // Query for some blogs without tracking them
    var blogs2 = context.Blogs
        .Where(b => b.Name.Contains(".NET"))
        .AsNoTracking()
        .ToList();
}
```

Необработанные SQL-запросы

27.09.2018 • 4 minutes to read • [Edit Online](#)

Платформа Entity Framework позволяет выполнять запросы с помощью LINQ с помощью классов сущностей. Тем не менее могут возникнуть ситуации, которые вы хотите выполнять запросы с помощью необработанных SQL непосредственно в базе данных. Это включает в себя вызов хранимых процедур, которые могут оказаться полезными для Code First моделей, которые в настоящее время не поддерживает сопоставление с хранимыми процедурами. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Написание SQL-запросов для сущностей

Метод SQL-запрос на DbSet позволяет необработанный SQL-запрос для записи, который возвращает экземпляры сущностей. Возвращаемые объекты будут отслеживаемые по контексту, так же, как было бы, если они были возвращены запросом LINQ. Пример:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();
}
```

Обратите внимание, что, так же как и запросы LINQ, запрос не выполняется до перечисляются результаты — в приведенном выше примере это делается с помощью вызова `ToList`.

Следует соблюдать осторожность, каждый раз, когда необработанные SQL-запросы записываются по двум причинам. Во-первых запрос должны быть написаны так, чтобы убедиться, что он только возвращает сущности, которые действительно запрошенного типа. Например при использовании механизмы вроде наследования можно легко написать запрос, который будет создание сущностей, которые имеют неправильный тип среди CLR.

Во-вторых некоторые виды необработанный SQL-запрос предоставляет потенциальные угрозы безопасности, особенно в отношении атаки путем внедрения кода SQL. Убедитесь в том, с использованием параметров в запросе в правильный способ защититься от таких атак.

Загрузка сущностей из хранимых процедур

`DbSet.SqlQuery` можно использовать для загрузки сущностей из результатов хранимой процедуры.

Например следующий код вызывает `dbo`. Процедура `GetBlogs` в базе данных.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("dbo.GetBlogs").ToList();
}
```

Вы также можете передавать параметры в хранимую процедуру, используя следующий синтаксис:

```
using (var context = new BloggingContext())
{
    var blogId = 1;

    var blogs = context.Blogs.SqlQuery("dbo.GetBlogById @p0", blogId).Single();
}
```

Написание SQL-запросов для несущностным типам

SQL-запроса возвращаются экземпляры любого типа, включая типы-примитивы, могут создаваться с помощью метода SQL-запрос к классу базы данных. Пример:

```
using (var context = new BloggingContext())
{
    var blogNames = context.Database.SqlQuery<string>(
        "SELECT Name FROM dbo.Blogs").ToList();
}
```

Результаты, возвращенные SQL-запрос в базе данных никогда не быть отслеживаемые по контексту, даже если объекты являются экземплярами типа сущности.

Отправка необработанных команды в базу данных

Команды запросов не могут отправляться в базу данных, используя метод ExecuteSqlCommand в базе данных. Пример:

```
using (var context = new BloggingContext())
{
    context.Database.ExecuteSqlCommand(
        "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
}
```

Обратите внимание, что любые изменения, внесенные в данные в базе данных с помощью ExecuteSqlCommand непрозрачности в контекст пока сущностей не будут загружены или перезагружена из базы данных.

Выходные параметры

Если используются выходные параметры, их значения не будут доступны, пока не полностью считать результаты. Это происходит из-за особенностей DbDataReader, см. в разделе [получение данных с помощью объекта DataReader](#) для получения дополнительных сведений.

Загрузка связанных сущностей

29.09.2018 • 9 minutes to read • [Edit Online](#)

Платформа Entity Framework поддерживает три способа загрузки связанных данных — Безотложная загрузка, отложенная загрузка и явная загрузка. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Загрузка заранее

Безотложная загрузка — это процесс, при котором запрос для одного типа сущности также загружает связанные сущности как часть запроса. Безотложная загрузка достигается за счет применения метода `Include`. Например приведенных ниже запросах загрузит блогов и записей, связанных с каждого блога.

```
using (var context = new BloggingContext())
{
    // Load all blogs and related posts
    var blogs1 = context.Blogs
        .Include(b => b.Posts)
        .ToList();

    // Load one blogs and its related posts
    var blog1 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include(b => b.Posts)
        .FirstOrDefault();

    // Load all blogs and related posts
    // using a string to specify the relationship
    var blogs2 = context.Blogs
        .Include("Posts")
        .ToList();

    // Load one blog and its related posts
    // using a string to specify the relationship
    var blog2 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include("Posts")
        .FirstOrDefault();
}
```

Обратите внимание, что включить метод расширения в пространстве имен `System.Data.Entity` поэтому убедитесь, что при использовании этого пространства имен.

Заранее загрузки несколько уровней

Можно также заранее загрузить несколько уровней связанных сущностей. Приведенных ниже запросах приведены примеры того, как сделать это для коллекции и свойства навигации по ссылке.

```

using (var context = new BloggingContext())
{
    // Load all blogs, all related posts, and all related comments
    var blogs1 = context.Blogs
        .Include(b => b.Posts.Select(p => p.Comments))
        .ToList();

    // Load all users, their related profiles, and related avatar
    var users1 = context.Users
        .Include(u => u.Profile.Avatar)
        .ToList();

    // Load all blogs, all related posts, and all related comments
    // using a string to specify the relationships
    var blogs2 = context.Blogs
        .Include("Posts.Comments")
        .ToList();

    // Load all users, their related profiles, and related avatar
    // using a string to specify the relationships
    var users2 = context.Users
        .Include("Profile.Avatar")
        .ToList();
}

```

Обратите внимание на то, что он не является в настоящее время можно фильтровать, какие связанные сущности загружаются. Будет всегда использовать в все связанные сущности.

Отложенная загрузка

Отложенная загрузка — это процесс, при котором сущность или коллекцию сущностей автоматически загружается из базы данных при первом обращении к свойству, ссылки на сущности или сущностей. При использовании типов сущностей РОСО, отложенная загрузка достигается путем создания экземпляров типов производного прокси-сервера и затем переопределение виртуальных свойств для добавления обработчика загрузки. Например при использовании класса сущности блога, определенная ниже, связанных сообщений будут загружены при первом обращении к свойству навигации сообщения:

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}

```

Включить отложенная загрузка для сериализации

Отложенная загрузка и сериализация не следует смешивать хорошо, и если не соблюдать осторожность можно в итоге запросов для всей базы данных, только потому, что отложенная загрузка включена.

Большинство сериализаторов работать с помощью каждого свойства в экземпляре типа. Доступ к свойству активирует отложенной загрузки, поэтому сериализуются больше сущностей. В этих сущностях доступа к свойствам и даже дополнительные сущности загружаются. Рекомендуется включить отложенной загрузки из системы, прежде чем сериализации сущности. Ниже показано, как это сделать.

Отключение отложенная загрузка для свойств конкретного навигации

Можно отключить отложенную загрузку набор Posts, сделав невиртуальный свойства сообщения:

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

Загрузка в записях коллекции можно по-прежнему добиться, используя безотложную загрузку (см. в разделе [заранее Загрузка](#) выше) или метода Load (см. в разделе [явная загрузка](#) ниже).

Отключить отложенную загрузку для всех сущностей

Отложенная загрузка можно отключить для всех сущностей в контексте, установив флаг в свойстве конфигурации. Пример:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Загрузкой связанных сущностей по-прежнему может осуществляться с помощью Безотложная загрузка (см. в разделе [заранее Загрузка](#) выше) или метода Load (см. в разделе [явная загрузка](#) ниже).

Явная загрузка

Даже с отложенная загрузка отключена возможность медленно загрузить связанные сущности, но это должно быть сделано с помощью явного вызова. Для этого использовать метод Load в связанной сущности операции. Пример:

```
using (var context = new BloggingContext())
{
    var post = context.Posts.Find(2);

    // Load the blog related to a given post
    context.Entry(post).Reference(p => p.Blog).Load();

    // Load the blog related to a given post using a string
    context.Entry(post).Reference("Blog").Load();

    var blog = context.Blogs.Find(1);

    // Load the posts related to a given blog
    context.Entry(blog).Collection(p => p.Posts).Load();

    // Load the posts related to a given blog
    // using a string to specify the relationship
    context.Entry(blog).Collection("Posts").Load();
}
```

Обратите внимание на то, что метода ссылки следует использовать, если сущность имеет свойство навигации, чтобы другой одной сущности. С другой стороны метод сбора следует использовать, если сущность имеет свойство навигации, чтобы коллекция других сущностей.

Применение фильтров, если явная загрузка связанных сущностей

Метод запроса предоставляет доступ к базовый запрос, который будет использовать Entity Framework, при загрузке связанных сущностей. Затем можно использовать LINQ для фильтрации запроса перед его выполнением с помощью вызова в метод расширения LINQ, вроде `ToList`, нагрузки и т. д. Метод запроса может использоваться со свойствами навигации как ссылки, так и коллекцию, но чрезвычайно полезна для коллекций, где его можно использовать для загрузки только частью коллекции. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog
    context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();

    // Load the posts with the 'entity-framework' tag related to a given blog
    // using a string to specify the relationship
    context.Entry(blog)
        .Collection("Posts")
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();
}
```

При использовании метода `Query` обычно лучше отключить отложенную загрузку для свойства навигации. Это обусловлено тем, в противном случае вся коллекция может получить автоматически загружать механизм отложенной загрузки до или после выполнения отфильтрованного запроса.

Обратите внимание, что хотя связи должно лежать в виде строки вместо лямбда-выражение, возвращаемое `IQueryable` не универсального когда строка используется в том случае, и поэтому метод приведения обычно требуется, чтобы сделать ничего полезного с ним.

С помощью запроса для подсчета без их загрузки связанных сущностей

Иногда бывает полезно знать, сколько сущности связаны с другой сущностью в базе данных без фактически затрат на загрузку этих сущностей. Метод запроса с помощью метода LINQ счетчик может использоваться для этого. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Count how many posts the blog has
    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

Сохранение данных в Entity Framework 6

14.09.2018 • 2 minutes to read • [Edit Online](#)

В этом разделе вы найдете сведения о возможностях отслеживания изменений платформы EF и о том, что происходит при вызове метода `SaveChanges` для сохранения всех изменений объектов в базе данных.

Автоматическое обнаружение изменений

13.09.2018 • 2 minutes to read • [Edit Online](#)

При использовании большинство сущностей POCO определение как была изменена сущность (и поэтому какие обновления должны отправляться в базу данных) обрабатывается алгоритмом обнаружение изменений. Обнаружение изменений работает путем выявления различий между текущие значения свойств сущности и исходные значения свойств, которые хранятся в моментальном снимке, когда сущность была запросе или присоединении. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

По умолчанию Entity Framework автоматически выполняет обнаружение изменений, при вызове следующих методов:

- `DbSet.Find`
- `DbSet.Local`
- `DbSet.Add`
- `DbSet.AddRange`
- `DbSet.Remove`
- `DbSet.RemoveRange`
- `DbSet.Attach`
- `DbContext.SaveChanges`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`

Отключить автоматическое обнаружение изменений

Если вы наблюдаете массу сущностей в контексте, вызовите один из этих методов много раз в цикле может добиться заметного повышения производительности, отключив обнаружение изменений в течение всего цикла. Пример:

```
using (var context = new BloggingContext())
{
    try
    {
        context.Configuration.AutoDetectChangesEnabled = false;

        // Make many calls in a loop
        foreach (var blog in aLotOfBlogs)
        {
            context.Blogs.Add(blog);
        }
    }
    finally
    {
        context.Configuration.AutoDetectChangesEnabled = true;
    }
}
```

Не забудьте снова включить обнаружение изменений после цикла, мы использовали `try/finally`, чтобы он всегда повторно включен даже если код в цикле вызывает исключение.

Альтернативы для отключения и повторного включения это поле остается автоматическое обнаружение изменений, отключить все значения времени и либо контекста вызова. ChangeTracker.DetectChanges явным образом или используйте посредников для отслеживания изменений, тщательно. Обе функции являются расширенными и может привести к тонким ошибкам в приложение таким образом их использовать с осторожностью.

Если вам нужно добавить или удалить множество объектов из контекста, рассмотрите возможность использования DbSet.AddRange и DbSet.RemoveRange. Этот метод автоматически обнаруживает изменения только один раз после завершения операций добавления или удаления.

Работа с состояния сущностей

13.09.2018 • 9 minutes to read • [Edit Online](#)

В этом разделе будут рассмотрены способы добавления и присоединить сущности к контексту и как платформа Entity Framework преобразует их во время SaveChanges. Платформа Entity Framework берет на себя отслеживания состояния сущностей, пока они подключены к контексту, но в отключенном или N-уровневых сценариях можно позволить платформе EF, какое состояние сущности должны находиться в. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Состояния сущностей и SaveChanges

Сущность может быть в одном из пяти состояний, как определено в перечислении EntityState. Возможны следующие состояния:

- Добавлена: сущность отслеживается контекстом, но еще не существует в базе данных
- Без изменений: отслеживается контекстом и сущности в базе данных и значения его свойств, не отличаются от значений в базе данных
- Изменен: сущность отслеживается контекстом и существует в базе данных, а также были изменены некоторые или все значения его свойств
- Удаленные: сущности отслеживается контекстом в базе данных, но теперь помечено для удаления из базы данных в следующий раз в вызове метода SaveChanges
- Отсоединенные: сущность не отслеживается контекстом

SaveChanges выполняет различные действия для сущностей в различных состояниях.

- Без изменений сущности, не менялись с SaveChanges. Обновления не отправляются в базу данных для сущностей в неизмененном состоянии.
- Добавлены сущности вставляются в базу данных и затем становятся без изменений при возвращают SaveChanges.
- Измененных сущностей обновляются в базе данных и затем становятся не изменяются при возвращает SaveChanges.
- Удаленные сущности будут удалены из базы данных и затем отсоединяются от контекста.

Ниже приведены примеры способов, в котором можно изменить состояние объекта или графа объекта.

Добавление новой сущности в контекст

Можно добавить новую сущность в контекст путем вызова метода Add для DbSet. Этот запрос помещает сущность в добавленном состоянии, это означает, что она будет вставлена в базу данных во время следующего вызова метода SaveChanges. Пример:

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Чтобы изменить его состояние на добавленное является другим способом добавления новой сущности в

контекст. Пример:

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Entry(blog).State = EntityState.Added;
    context.SaveChanges();
}
```

Наконец можно добавить новую сущность в контекст путем ее прикрепления к другой сущности, которая уже отслеживается. Это может быть путем добавления новой сущности к другой сущности свойство навигации по коллекции, или установив свойства навигации ссылки другой сущности, чтобы он указывал на новую сущность. Пример:

```
using (var context = new BloggingContext())
{
    // Add a new User by setting a reference from a tracked Blog
    var blog = context.Blogs.Find(1);
    blog.Owner = new User { UserName = "johndoe1987" };

    // Add a new Post by adding to the collection of a tracked Blog
    var blog = context.Blogs.Find(2);
    blog.Posts.Add(new Post { Name = "How to Add Entities" });

    context.SaveChanges();
}
```

Обратите внимание, что для всех этих примерах, если добавляемый сущность имеет ссылки на другие сущности, которые не отслеживаются затем еще эти новые сущности также будут добавляться к контексту и будет вставлен в базе данных при последующем вызове метода SaveChanges.

Присоединение существующей сущности к контексту

При наличии сущности, вы уже знаете, существует в базе данных, но которой не сейчас отслеживается контекстом можно указать контекст для отслеживания использует метод присоединения на DbSet сущности. Сущность будет в неизмененном состоянии, в контексте. Пример:

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);

    // Do some more work...

    context.SaveChanges();
}
```

Обратите внимание на то, что изменения не будут внесены в базу данных при вызове метода SaveChanges, не выполняя другие операции вложенные сущности. Это обусловлено сущность находится в неизмененном состоянии.

Другой способ Присоединение существующей сущности к контексту — изменяет свое состояние на неизмененное. Пример:

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

Обратите внимание, что для обоих этих примерах Если присоединяемый объект имеет ссылки на другие сущности, которые не отслеживаются затем эти новые сущности будет также присоединен к контексту в неизмененном состоянии.

Присоединение существующего но измененная сущность к контексту

При наличии сущности, которые известны как уже существует в базе данных, но какие изменения внесены можно указать контекст, чтобы вложить сущность и задайте для его состояние Modified. Пример:

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Modified;

    // Do some more work...

    context.SaveChanges();
}
```

При изменении состояния для изменения все свойства сущности будут помечены как измененные и все значения свойств будут отправляться в базу данных при вызове метода SaveChanges.

Обратите внимание, что если присоединяемый объект имеет ссылки на другие сущности, которые не отслеживаются, затем эти новые сущности будет присоединен к контексту в неизмененном состоянии, они не автоматически будут внесены изменения. При наличии нескольких сущностей, которые должны быть помечены Modified необходимо задать состояние для каждого из этих сущностей по отдельности.

Изменение состояния для отслеживаемой сущности

Можно изменить состояние сущности, которая уже отслеживается, задав свойство состояния на соответствующую запись. Пример:

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

Обратите внимание на то, что обращаясь добавить или присоединить сущность, которая уже отслеживается

может также использоваться для изменения состояния сущности. Например вызов присоединения для сущности, которая в настоящее время находится в добавленном состоянии изменит свое состояние на неизмененное.

Вставить или обновить шаблон

Распространенный подход для некоторых приложений является либо добавить сущность в качестве нового (в результате чего вставки баз данных), или вложить сущность как существующих и пометить его как измененный (что обновления базы данных) в зависимости от значения первичного ключа. Например при использовании сформированный базой данных целочисленных первичных ключей довольно часто рассматривать сущность с нулям ключ как новый и сущность с ключом ненулевое значение, как существующий. Этот шаблон можно сделать путем установки состояние сущности на основе проверки значения первичного ключа. Пример:

```
public void InsertOrUpdate(Blog blog)
{
    using (var context = new BloggingContext())
    {
        context.Entry(blog).State = blog.BlogId == 0 ?
            EntityState.Added :
            EntityState.Modified;

        context.SaveChanges();
    }
}
```

Обратите внимание, что при изменении состояния для изменения все свойства сущности будут помечены как измененные, и все значения свойств будут отправляться в базу данных при вызове метода SaveChanges.

Работа со значениями свойств

13.09.2018 • 15 minutes to read • [Edit Online](#)

В большинстве случаев платформа Entity Framework позаботится о отслеживания состояния, исходные значения и текущие значения свойств экземпляров сущности. Тем не менее возможны некоторые ситуации — например без подключения к сети —, где вы хотите просмотреть или изменить сведения, которые EF о свойствах. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Платформа Entity Framework хранит информацию о двух значений для каждого свойства отслеживаемой сущности. Текущее значение является, как видно из названия, текущее значение свойства в сущности. Исходное значение является значением, которое свойство имело при запросе из базы данных или присоединен к контексту сущности.

Существует два механизма, общие для работы со значениями свойств:

- Значение одного свойства можно получить в строго типизированным способом с помощью метода `GetValue`.
- Значения для всех свойств сущности можно считать в объект `DbPropertyValues`. `DbPropertyValues` действует как объект подобная словарю, чтобы разрешить читать и устанавливать значения свойств. Можно задать значения в объекте `DbPropertyValues`, на основе значений в другой объект `DbPropertyValues` или на основе значений в другой объект, например другую копию сущности или объект передачи данных (DTO).

В следующих разделах приведены примеры использования обоих указанных выше механизмов.

Получение и задание текущее или исходное значение отдельного свойства

В приведенном ниже примере показано, как текущее значение свойства можно считывать и задайте новое значение:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(3);

    // Read the current value of the Name property
    string currentName1 = context.Entry(blog).Property(u => u.Name).CurrentValue;

    // Set the Name property to a new value
    context.Entry(name).Property(u => u.Name).CurrentValue = "My Fancy Blog";

    // Read the current value of the Name property using a string for the property name
    object currentName2 = context.Entry(blog).Property("Name").CurrentValue;

    // Set the Name property to a new value using a string for the property name
    context.Entry(blog).Property("Name").CurrentValue = "My Boring Blog";
}
```

Свойство `OriginalValue` вместо свойства `CurrentValue` прочитать или задать исходное значение.

Обратите внимание, что возвращаемое значение типизируется как «объект», когда строка используется для указания имени свойства. С другой стороны возвращаемое значение является строго типизированным, если

используется лямбда-выражение.

Установка значения свойства следующим образом только отмечает свойство, как изменить значение, если новое значение отличается от предыдущего значения.

Если значение свойства имеет значение таким образом изменение обнаруживается автоматически даже если AutoDetectChanges находится в отключенном состоянии.

Получение и задание текущее значение несопоставленного свойства

Текущее значение свойства, которое не сопоставлен с базы данных также могут считываться. Пример несопоставленного свойства может быть свойством RssLink в блоге. Это значение может быть вычислены на основании BlogId и поэтому не должны храниться в базе данных. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    // Read the current value of an unmapped property
    var rssLink = context.Entry(blog).Property(p => p.RssLink).CurrentValue;

    // Use a string to specify the property name
    var rssLinkAgain = context.Entry(blog).Property("RssLink").CurrentValue;
}
```

Текущее значение можно задать также в том случае, если свойство предоставляет метод задания.

Чтение значений параметров несопоставленные свойства полезно в том случае, при проверке платформы Entity Framework несопоставленные свойства. По этой же причине текущие значения можно читать и задавать для свойства сущностей, которые не отслеживаются контекстом. Пример:

```
using (var context = new BloggingContext())
{
    // Create an entity that is not being tracked
    var blog = new Blog { Name = "ADO.NET Blog" };

    // Read and set the current value of Name as before
    var currentName1 = context.Entry(blog).Property(u => u.Name).CurrentValue;
    context.Entry(blog).Property(u => u.Name).CurrentValue = "My Fancy Blog";
    var currentName2 = context.Entry(blog).Property("Name").CurrentValue;
    context.Entry(blog).Property("Name").CurrentValue = "My Boring Blog";
}
```

Обратите внимание на то, что исходные значения недоступны для несопоставленные свойства или свойства сущностей, которые не отслеживаются контекстом.

Проверка, помечен ли свойство как измененное

В приведенном ниже примере показано, как проверить ли отдельное свойство отмечается как измененное:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var nameIsModified1 = context.Entry(blog).Property(u => u.Name).IsModified;

    // Use a string for the property name
    var nameIsModified2 = context.Entry(blog).Property("Name").IsModified;
}
```

Значения измененных свойств, отправляются в виде обновлений в базу данных при вызове метода `SaveChanges`.

Пометить как измененное свойство

В приведенном ниже примере показано, как заставить отдельное свойство помечается как измененный:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    context.Entry(blog).Property(u => u.Name).IsModified = true;

    // Use a string for the property name
    context.Entry(blog).Property("Name").IsModified = true;
}
```

Пометка измененный принудительное обновление быть отправки в базу данных для свойства при вызове метода `SaveChanges`, даже если текущее значение свойства является таким же, как исходное значение свойства.

Не сейчас невозможно сбросить отдельного свойства, которые не будут изменяться после он был помечен как измененный. Это то, что мы планируем реализовать поддержку в будущем выпуске.

Считывание текущего, исходные и значения базы данных для всех свойств сущности

В приведенном ниже примере показано, как считывать значения фактически в базе данных для всех сопоставленных свойств сущности, исходные значения и текущие значения.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Make a modification to Name in the tracked entity
    blog.Name = "My Cool Blog";

    // Make a modification to Name in the database
    context.Database.SqlCommand("update dbo.Blogs set Name = 'My Boring Blog' where Id = 1");

    // Print out current, original, and database values
    Console.WriteLine("Current values:");
    PrintValues(context.Entry(blog).CurrentValues);

    Console.WriteLine("\nOriginal values:");
    PrintValues(context.Entry(blog).OriginalValues);

    Console.WriteLine("\nDatabase values:");
    PrintValues(context.Entry(blog).GetDatabaseValues());
}

public static void PrintValues(DbPropertyValues values)
{
    foreach (var propertyName in values.PropertyNames)
    {
        Console.WriteLine("Property {0} has value {1}",
                          propertyName, values[propertyName]);
    }
}
```

Текущие значения — это значения, в настоящий момент содержат свойства сущности. Исходные значения — это значения, считанных из базы данных, на которые запросила сущности. Значения базы данных, значения в настоящее время хранятся в базе данных. Получение значений базы данных полезно в том случае, когда значения в базе данных мог быть изменен, так как сущность запросила, такие как стала при одновременное изменение в базе данных другим пользователем.

Текущее и исходное значения из другого объекта

Текущее и исходное значения отслеживаемой сущности можно обновить путем копирования значения из другого объекта. Пример:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    var coolBlog = new Blog { Id = 1, Name = "My Cool Blog" };
    var boringBlog = new BlogDto { Id = 1, Name = "My Boring Blog" };

    // Change the current and original values by copying the values from other objects
    var entry = context.Entry(blog);
    entry.CurrentValues.SetValues(coolBlog);
    entry.OriginalValues.SetValues(boringBlog);

    // Print out current and original values
    Console.WriteLine("Current values:");
    PrintValues(entry.CurrentValues);

    Console.WriteLine("\nOriginal values:");
    PrintValues(entry.OriginalValues);
}

public class BlogDto
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

Под управлением приведенный выше код выводит на экран:

```

Current values:
Property Id has value 1
Property Name has value My Cool Blog

Original values:
Property Id has value 1
Property Name has value My Boring Blog

```

Этот способ иногда используется при обновлении сущности с помощью значений, полученных из вызова службы или клиента в п уровневого приложения. Обратите внимание, что объект используется не быть того же типа как сущность, до тех пор, пока он имеет свойства, имена которых совпадают с сущности. В приведенном выше примере экземпляр BlogDTO используется для обновления исходных значений.

Обратите внимание на то, что только те свойства, которые заданы разные значения, при копировании с другой объект будет помечен как измененный.

Параметр текущее и исходное значения из словаря

Текущее и исходное значения отслеживаемой сущности можно обновить путем копирования значения из словаря или некоторые другие структуры данных. Пример:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var newValues = new Dictionary<string, object>
    {
        { "Name", "The New ADO.NET Blog" },
        { "Url", "blogs.msdn.com/adonet" },
    };

    var currentValues = context.Entry(blog).CurrentValues;

    foreach (var propertyName in newValues.Keys)
    {
        currentValues[propertyName] = newValues[propertyName];
    }

    PrintValues(currentValues);
}

```

Используйте свойство `OriginalValues` вместо свойства `CurrentValues` для установки исходных значений.

Параметр текущее и исходное значения из словаря с помощью свойства

Использование метода свойство для задания значения каждого свойства является альтернативой использованию `CurrentValues` или `OriginalValues`, как показано выше. Это может быть более предпочтительным, чем при необходимости задайте значения свойств сложных свойств. Пример:

```

using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    var newValues = new Dictionary<string, object>
    {
        { "Name", "John Doe" },
        { "Location.City", "Redmond" },
        { "Location.State.Name", "Washington" },
        { "Location.State.Code", "WA" },
    };

    var entry = context.Entry(user);

    foreach (var propertyName in newValues.Keys)
    {
        entry.Property(propertyName).CurrentValue = newValues[propertyName];
    }
}

```

В примере выше сложных свойств осуществляется с помощью точечно имени. Другие способы доступа к сложным свойствам см. в двух разделах далее в этом разделе, в частности о сложных свойствах.

Создание клонированный объект, содержащий текущий, исходные или значения базы данных

`DbPropertyValues` объект, возвращенный `CurrentValues` `OriginalValues`, или `GetDatabaseValues` может использоваться для создания клона сущности. Этот клон будет содержать значения свойств из объекта `DbPropertyValues`, использованного для его создания. Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var clonedBlog = context.Entry(blog).GetDatabaseValues().ToObject();
}
```

Обратите внимание, что возвращенный объект не является ни сущностью и не отслеживается контекстом. Возвращаемый объект также имеет все связи, задайте для других объектов.

Клонированный объект может быть полезно при решении проблем, связанных с одновременных обновлений в базу данных, особенно когда используется пользовательский Интерфейс, который включает в себя привязку данных к объектам определенного типа.

Получение и установка значения текущей или исходной сложных свойств

Чтение и задается с помощью метода свойства так же, как это может быть простое свойство, допускается значение всего сложного объекта. Кроме того можно выполнить детализацию в сложного объекта и чтения или заданы свойства этого объекта, или даже вложенного объекта. Далее приводятся некоторые примеры.

```

using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    // Get the Location complex object
    var location = context.Entry(user)
        .Property(u => u.Location)
        .CurrentValue;

    // Get the nested State complex object using chained calls
    var state1 = context.Entry(user)
        .ComplexProperty(u => u.Location)
        .Property(l => l.State)
        .CurrentValue;

    // Get the nested State complex object using a single lambda expression
    var state2 = context.Entry(user)
        .Property(u => u.Location.State)
        .CurrentValue;

    // Get the nested State complex object using a dotted string
    var state3 = context.Entry(user)
        .Property("Location.State")
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using chained calls
    var name1 = context.Entry(user)
        .ComplexProperty(u => u.Location)
        .ComplexProperty(l => l.State)
        .Property(s => s.Name)
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using a single lambda expression
    var name2 = context.Entry(user)
        .Property(u => u.Location.State.Name)
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using a dotted string
    var name3 = context.Entry(user)
        .Property("Location.State.Name")
        .CurrentValue;
}

```

Свойство `OriginalValue` вместо свойства `CurrentValue` требуется получить или задать исходное значение.

Обратите внимание на то, что свойство или метод `ComplexProperty` может использоваться для доступа к сложного свойства. Тем не менее метод `ComplexProperty` должен использоваться, если вы хотите углубиться в сложного объекта с дополнительным свойством или вызывает `ComplexProperty`.

Использование `DbPropertyValues` для доступа к сложным свойствам

При использовании `CurrentValues`, `OriginalValues` или `GetDatabaseValues` для получения исходного все текущие, или значения базы данных для сущности, как вложенные объекты `DbPropertyValues` возвращаются значения любого из сложных свойств. Эти вложенные объекты могут затем использоваться для получения значения сложного объекта. Например следующий метод напечатает значения всех свойств, включая значения любого сложных свойств и вложенных составных свойств.

```
public static void WritePropertyValues(string parentPropertyName, DbPropertyValues propertyValues)
{
    foreach (var propertyName in propertyValues.PropertyNames)
    {
        var nestedValues = propertyValues[propertyName] as DbPropertyValues;
        if (nestedValues != null)
        {
            WritePropertyValues(parentPropertyName + propertyName + ".", nestedValues);
        }
        else
        {
            Console.WriteLine("Property {0}{1} has value {2}",
                parentPropertyName, propertyName,
                propertyValues[propertyName]);
        }
    }
}
```

Чтобы распечатать все текущие значения свойств, метод будет вызываться следующим образом:

```
using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    WritePropertyValues("", context.Entry(user).CurrentValue);
}
```

Обработка конфликтов параллелизма

13.09.2018 • 8 minutes to read • [Edit Online](#)

Оптимистичный параллелизм предусматривает оптимистическую попытку сохранить сущность в базу данных в надежде, что данные оттуда не изменились после сущность была загружена. Если оказывается, что данные изменились, то возникает исключение, и необходимо разрешить конфликт, прежде чем сохранить еще раз. В этом разделе описывается, как для обработки таких исключений в Entity Framework. Методы, представленные в этом разделе, также применимы к моделям, созданным с помощью Code First и конструктора EF.

Это сообщение не подходит местом для полное описание оптимистичного параллелизма. В следующих разделах, предполагают, что некоторые разрешения параллелизма и отобразите шаблоны для распространенных задач.

Многие из этих шаблонов использования темам, обсуждавшимся в [работа со значениями свойств](#).

Устранение проблем параллелизма, при использовании независимых сопоставлений (которой внешний ключ не сопоставлен со свойством сущности) гораздо сложнее, чем при использовании внешнего ключа ассоциации. Таким образом Если необходимо выполнить разрешение параллелизма в приложении рекомендуется всегда сопоставляются внешних ключей в сущности. Всех приведенных ниже примерах предполагается, что вы используете ассоциации внешних ключей.

`DbUpdateConcurrencyException` генерируется `SaveChanges` при обнаружении исключения оптимистичного параллелизма при попытке сохранить сущность, которая использует связи внешнего ключа.

Разрешения исключений оптимистичного параллелизма с перезагрузить базы данных wins)

Метод перезагрузки может использоваться для перезаписи текущих значений сущности со значениями, теперь в базе данных. Сущность обычно предоставляется обратно пользователю в определенной форме, и они пытаются снова внести свои изменения и повторно сохраните. Пример:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;

        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Update the values of the entity that failed to save from the store
            ex.Entries.Single().Reload();
        }
    } while (saveFailed);
}

```

Чтобы установить точку останова в вызове метода `SaveChanges`, а затем измените сущности, который сохраняется в базе данных с помощью другого средства, например SQL Management Studio является хорошим способом имитации исключения параллельности. Вы также может вставить строку перед `SaveChanges` для обновления базы данных, непосредственно с помощью `SqlCommand`. Пример:

```

context.Database.SqlCommand(
    "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");

```

Метод записи на `DbUpdateConcurrencyException` возвращает `dbentityentry`, который должен экземпляры сущности, которые не удалось обновить. (Это свойство в настоящее время всегда возвращает одно значение для проблем параллелизма. Он может возвращать несколько значений для обновления общего исключения.) Альтернативы в некоторых случаях может быть получение записей для всех сущностей, которые может потребоваться перезагрузить из базы данных и перезагрузить вызова для каждого из них.

Разрешения исключений оптимистичного параллелизма, как приоритет клиента

Приведенном выше примере, использующий перезагрузить иногда называют базы данных wins или победы хранилища, так как значения в сущности, перезаписываются значениями из базы данных. Иногда может потребоваться сделать обратное и перезаписи значений базы данных с текущими значениями в сущности. Это иногда называется приоритет клиента и можно сделать, получив текущие значения базы данных и их настройке в качестве исходных значений для сущности. (См. в разделе [работа со значениями свойств](#) сведения на текущие и исходные значения.) Пример:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Update original values from the database
            var entry = ex.Entries.Single();
            entry.OriginalValues.SetValues(entry.GetDatabaseValues());
        }
    } while (saveFailed);
}
```

Пользовательское разрешение исключений оптимистичного параллелизма

Иногда может потребоваться объединить значения, в настоящее время в базе данных с текущими значениями в сущности. Это обычно требует некоторых пользовательских взаимодействие логики или пользователя. Например формы может представлять пользователю, содержащий текущие значения, значения в базе данных, и значение по умолчанию набора разрешенных значений. Пользователь может затем измените разрешенные значения при необходимости и было бы эти разрешенного значения, которые сохраняются в базу данных. Это можно сделать с помощью `DbPropertyValues` объекты, возвращаемые `CurrentValue`s и `GetDatabaseValues` на запись сущности. Пример:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Get the current entity values and the values in the database
            var entry = ex.Entries.Single();
            var currentValue = entry.CurrentValues;
            var databaseValues = entry.GetDatabaseValues();

            // Choose an initial set of resolved values. In this case we
            // make the default be the values currently in the database.
            var resolvedValues = databaseValues.Clone();

            // Have the user choose what the resolved values should be
            HaveUserResolveConcurrency(currentValue, databaseValues, resolvedValues);

            // Update the original values with the database values and
            // the current values with whatever the user choose.
            entry.OriginalValues.SetValues(databaseValues);
            entry.CurrentValues.SetValues(resolvedValues);
        }
    } while (saveFailed);
}

public void HaveUserResolveConcurrency(DbPropertyValues currentValue,
                                       DbPropertyValues databaseValues,
                                       DbPropertyValues resolvedValues)
{
    // Show the current, database, and resolved values to the user and have
    // them edit the resolved values to get the correct resolution.
}

```

Пользовательское разрешение исключений оптимистичного параллелизма, с помощью объектов

Приведенный выше код использует `DbPropertyValues` экземпляры для передачи текущей базы данных и разрешенные значения. Иногда может быть проще использовать экземпляры типа сущности. Это можно сделать с помощью методов `ToObject` и `SetValues` `DbPropertyValues`. Пример:

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Get the current entity values and the values in the database
            // as instances of the entity type
            var entry = ex.Entries.Single();
            var databaseValues = entry.GetDatabaseValues();
            var databaseValuesAsBlog = (Blog)databaseValues.ToObject();

            // Choose an initial set of resolved values. In this case we
            // make the default be the values currently in the database.
            var resolvedValuesAsBlog = (Blog)databaseValues.ToObject();

            // Have the user choose what the resolved values should be
            HaveUserResolveConcurrency((Blog)entry.Entity,
                databaseValuesAsBlog,
                resolvedValuesAsBlog);

            // Update the original values with the database values and
            // the current values with whatever the user choose.
            entry.OriginalValues.SetValues(databaseValues);
            entry.CurrentValues.SetValues(resolvedValuesAsBlog);
        }
    } while (saveFailed);
}

public void HaveUserResolveConcurrency(Blog entity,
    Blog databaseValues,
    Blog resolvedValues)
{
    // Show the current, database, and resolved values to the user and have
    // them update the resolved values to get the correct resolution.
}

```

Работа с транзакциями

13.09.2018 • 14 minutes to read • [Edit Online](#)

NOTE

Только в EF6 и более поздних версиях. Функции, API и другие возможности, описанные на этой странице, появились в Entity Framework 6. При использовании более ранней версии могут быть неприменимы некоторые или все сведения.

В этом документе описано использование транзакций в EF6, включая усовершенствования, реализованной с момента EF5 для облегчения работы с транзакциями.

По умолчанию не EF

Во всех версиях Entity Framework, при каждом выполнении **SaveChanges()** вставить, обновить или удалить на базе платформы будет служить оболочкой этой операции в транзакции. Данная транзакция продолжается только достаточно долго для того, для выполнения операции, а затем завершает работу. При выполнении другой операции запускается новая транзакция.

Начиная с EF6 **Database.ExecuteSqlCommand()** по умолчанию будет служить оболочкой команды в транзакции, если один еще не. Существуют перегрузки этого метода, которые дают возможность переопределить это поведение, при необходимости. Также в EF6 выполнение хранимых процедур, включенных в модель через API-интерфейсы, такие как **ObjectContext.ExecuteFunction()** делает то же самое (за исключением того, что поведение по умолчанию не может в данный момент переопределяться).

В любом случае уровень изоляции транзакции — независимо от уровня изоляции считает, что поставщик базы данных по умолчанию этого параметра. По умолчанию например, на сервере SQL Server это READ COMMITTED.

Платформа Entity Framework не перенос запросов в транзакции.

Эта функция по умолчанию подходит для многих пользователей, и если так что нет необходимости делать какие-либо изменения в EF6; просто напишите код, как это делалось всегда.

Однако некоторым пользователям требуется больший контроль над их транзакций — это рассматривается в следующих разделах.

Работе с API

Прежде чем платформа Entity Framework EF6 настаивали на том, на открытие подключения к базе данных сам (он вызывал исключение, если был передан уже было открыто соединение). Так как транзакции можно запустить только на открытое соединение, это означало, что единственным способом поместить несколько операций в одну транзакцию пользователя было использовать **TransactionScope** или использовать **ObjectContext.Connection** и начала вызова **Open()** и **BeginTransaction()** непосредственно в возвращенном **EntityConnection** объект. Кроме того вызовы API, которые подключиться к базе данных завершается ошибкой, если транзакция была запущена на самостоятельно базового подключения к базе данных.

NOTE

Ограничение принятие только закрытых подключений был удален в Entity Framework 6. Дополнительные сведения см. в разделе [Управление соединениями](#).

Начиная с EF6 framework теперь предоставляет:

1. **Database.BeginTransaction()** : более простой метод для пользователя для запуска и завершения транзакций, сами в существующих DbContext — позволяя выполнять несколько операций для объединения в той же транзакции и, следовательно все зафиксированные или все откат как один. Он также позволяет пользователю легче указать уровень изоляции для транзакции.
2. **Database.UseTransaction()** : что позволяет использовать транзакцию, в которой был запущен вне платформы Entity Framework DbContext.

Объединение нескольких операций в одну транзакцию, в том же контексте

Database.BeginTransaction() позволяет переопределять – один, принимающий явно [IsolationLevel](#) и которая не принимает аргументы и использует значение по умолчанию IsolationLevel от базового поставщика базы данных. Оба переопределения возвращают **DbContextTransaction** объект, который предоставляет **Commit()** и **Rollback()** методы, выполняющие commit и rollback в базовом хранилище транзакция.

DbContextTransaction должен быть удален после фиксации или отката. Ищете простой способ выполнения этой задачи является **using(...){...}** синтаксис, который автоматически вызовет **Dispose()** при блока using завершения:

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void StartOwnTransactionWithinContext()
        {
            using (var context = new BloggingContext())
            {
                using (var dbContextTransaction = context.Database.BeginTransaction())
                {
                    try
                    {
                        context.Database.ExecuteSqlCommand(
                            @"UPDATE Blogs SET Rating = 5" +
                            " WHERE Name LIKE '%Entity Framework%'"
                        );
                    }

                    var query = context.Posts.Where(p => p.Blog.Rating >= 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }

                    context.SaveChanges();

                    dbContextTransaction.Commit();
                }
                catch (Exception)
                {
                    dbContextTransaction.Rollback();
                }
            }
        }
    }
}

```

NOTE

Запуск транзакции требуется базовое соединение хранилища открытый. Поэтому вызов Database.BeginTransaction() откроет соединение, если он еще не открыт. Если соединение открыто DbContextTransaction затем будут закрыты его при вызове Dispose().

Передача существующей транзакции контекста

Иногда требуется транзакция становится еще шире, в области, а также полностью включает операции на той же базе данных, но вне EF. Для выполнения этой задачи необходимо открыть подключение и начать транзакцию, самостоятельно и затем сообщить EF) для использования подключения к базе данных уже открыт и (б) для использования существующей транзакции для этого соединения.

Для этого необходимо определить и использовать конструктор в классе контекста, который наследуется от одного или конструкторов DbContext, которые принимают логическое i) существующего параметра подключения и contextOwnsConnection ii.

NOTE

Флаг contextOwnsConnection должно быть присвоено значение false при вызове в этом сценарии. Это важно, так как она сообщает об Entity Framework, она не должна закрывать соединение когда он закончила с ним (например, см. в разделе строки 4 ниже):

```
using (var conn = new SqlConnection("..."))
{
    conn.Open();
    using (var context = new BloggingContext(conn, contextOwnsConnection: false))
    {
    }
}
```

Более того, необходимо начать транзакцию, самостоятельно (включая IsolationLevel, если вы хотите избежать значение по умолчанию) и сообщить Entity Framework узнать о существующей транзакции уже запущен на соединение (см. в разделе строки 33 ниже).

Затем вы можете свободно для выполнения операций с базой данных, либо непосредственно на сам SqlConnection, либо на DbContext. Все эти операции выполняются в рамках одной транзакции. Вы отвечаете для фиксации или отката транзакции и для вызова Dispose() на нем, а также для закрытия и удаления подключения к базе данных. Пример:

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void UsingExternalTransaction()
        {
            using (var conn = new SqlConnection("..."))
            {
                conn.Open();

                using (var sqlTxn = conn.BeginTransaction(System.Data.IsolationLevel.Snapshot))
                {
                    try
                    {
                        var sqlCommand = new SqlCommand();
                        sqlCommand.Connection = conn;
                        sqlCommand.Transaction = sqlTxn;
                        sqlCommand.CommandText =
                            @"UPDATE Blogs SET Rating = 5" +
                            " WHERE Name LIKE '%Entity Framework%'";
                        sqlCommand.ExecuteNonQuery();

                        using (var context =
                            new BloggingContext(conn, contextOwnsConnection: false))
                        {
                            context.Database.UseTransaction(sqlTxn);

                            var query = context.Posts.Where(p => p.Blog.Rating >= 5);
                            foreach (var post in query)
                            {
                                post.Title += "[Cool Blog]";
                            }
                            context.SaveChanges();
                        }

                        sqlTxn.Commit();
                    }
                    catch (Exception)
                    {
                        sqlTxn.Rollback();
                    }
                }
            }
        }
    }
}

```

Очистка в операции

Можно передать null Database.UseTransaction() снимите знаний Entity Framework текущей транзакции. Платформа Entity Framework будет ни фиксация, ни откат существующей транзакции, при этом, поэтому следует использовать с осторожностью и только в том случае, если вы точно знаете, это будет выполняться.

Ошибки в UseTransaction

Вы увидите исключения из Database.UseTransaction() при передаче транзакции при:

- Платформа Entity Framework уже существует транзакции
- Платформа Entity Framework уже работает в класс TransactionScope

- Объект соединения в транзакции, передан имеет значение null. То есть транзакция не связан с соединением – обычно это означает, что транзакция уже завершена
- Объект соединения в транзакции, передан не соответствует подключения Entity Framework.

Использование транзакций с другими компонентами

В этом разделе описаны как выше транзакции взаимодействуют с:

- Устойчивость подключений
- Асинхронные методы
- Операции TransactionScope

Устойчивость подключений

Новая функция устойчивости подключений не работает с транзакциями, инициированными пользователем. Дополнительные сведения см. в разделе [стратегий выполнения повторов](#).

Асинхронное программирование

Подход, описанный в предыдущих разделах требуется не Дополнительные параметры или параметры для работы с [асинхронного запроса и сохранения методов](#). Но имейте в виду, что, в зависимости от того, что можно делать в асинхронные методы, это может привести длительные транзакции – что в свою очередь может привести к взаимоблокировкам или блокировки, который является негативно сказаться на производительности приложения в целом.

Операции TransactionScope

До EF6 рекомендованный способ предоставления большего размера область транзакции был использовать объект TransactionScope:

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void UsingTransactionScope()
        {
            using (var scope = new TransactionScope(TransactionScopeOption.Required))
            {
                using (var conn = new SqlConnection("..."))
                {
                    conn.Open();

                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'";
                    sqlCommand.ExecuteNonQuery();

                    using (var context =
                        new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        var query = context.Posts.Where(p => p.Blog.Rating > 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }
                        context.SaveChanges();
                    }
                }

                scope.Complete();
            }
        }
    }
}

```

`SqlConnection` и Entity Framework будет использовать внешнюю транзакцию `TransactionScope` и поэтому выполняться одновременная фиксация.

Начиная с .NET 4.5.1 `TransactionScope` был обновлен для работы с асинхронных методов с помощью [TransactionScopeAsyncFlowOption](#) перечисления:

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        public static void AsyncTransactionScope()
        {
            using (var scope = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
            {
                using (var conn = new SqlConnection("..."))
                {
                    await conn.OpenAsync();

                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'";
                    await sqlCommand.ExecuteNonQueryAsync();

                    using (var context = new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        var query = context.Posts.Where(p => p.Blog.Rating > 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }

                        await context.SaveChangesAsync();
                    }
                }
            }
        }
    }
}

```

По-прежнему имеют определенные ограничения TransactionScope подход:

- Требуется .NET 4.5.1 или более поздней версии для работы с асинхронными методами.
- Он не может использоваться в облачных сценариях кроме случаев, когда у вас есть только одно подключение (облачные сценарии не поддерживают распределенные транзакции).
- Можно сочетать с подходом Database.UseTransaction() из предыдущих разделов.
- Он будет вызывать исключения, если выдавать DDL и не включили распределенные транзакции через службу MSDTC.

Преимущества подхода TransactionScope:

- Он автоматически обновит локальной транзакции до распределенной транзакции, если сделать несколько подключений к заданной базе данных или объединить подключения к одной базе данных с подключением к другой базе данных в той же транзакции (Примечание: необходимо иметь Служба MSDTC разрешать распределенные транзакции для правильной работы).
- Простота написания кода. Если вы предпочитаете транзакции окружающей среды и раздачи с неявно в фоновом режиме, а не явным образом в элемент управлении затем TransactionScope подход может вам подходит лучше.

Таким образом, с помощью новых Database.BeginTransaction() и выше, API-интерфейсы Database.UseTransaction() подход TransactionScope больше не требуется для большинства пользователей.

Если вы продолжите использовать TransactionScope затем следует учитывать ограничения выше. Мы рекомендуем использовать подход, описанный в предыдущих разделах, вместо этого, где это возможно.

Проверка данных

13.09.2018 • 14 minutes to read • [Edit Online](#)

NOTE

EF4.1 и более поздних версий только -функции, интерфейсы API, и т.д., описанных на этой странице появились в версии 4.1 платформы Entity Framework. При использовании более ранней версии, некоторые или все сведения неприменимо

Содержимое на этой странице написана по материалам, а статья задумано, Джули Лерман (<http://thedatafarm.com>).

Платформа Entity Framework предоставляет выполнения разнообразных функций проверки, которые можно веб-канал через пользовательский интерфейс для проверки на стороне клиента или использоваться для проверки на стороне сервера. При использовании кода, во-первых, можно указать с помощью заметки или fluent API конфигурации проверки. Дополнительные проверки и сложнее, можно указать в коде и будет работать ли модель приехал из кода, во-первых, сначала модели или базы данных сначала.

Модель

Я продемонстрирую проверок с парой простых классов: блог и Post.

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public DateTime DateCreated { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

Заметки к данным

Сначала в коде используется заметок, связанных с сборке System.ComponentModel.DataAnnotations как один из способов настройки классов первого кода. Эти заметки относятся те, которые предоставляют правил, таких как обязательные, MaxLength и MinLength. Число клиентских приложениях .NET также распознает такие заметки, например, ASP.NET MVC. Вы можете достичь обе стороны и сервером проверки на стороне клиента с этими заметками. Например можно принудительно свойство Title блог обязательное свойство.

```
[Required]  
public string Title { get; set; }
```

Без дополнительного кода или разметки изменений в приложение существующее приложение MVC будет выполнять проверки на стороне клиента, даже динамическое создание сообщения с использованием имени свойства и заметки.

Create

Blog

Title
 The Title field is required.

BloggerName

DateCreated

В блога back-метод этого создать представления, Entity Framework используется для сохранения в новой записи в базу данных, но проверка на стороне клиента в MVC запускается прежде, чем оно достигнет этого кода.

Проверка на стороне клиента не пуленепробиваема тем не менее. Пользователи могут повлиять на возможности браузера или что еще хуже, злоумышленник может использовать некоторые изобретательности во избежание проверки пользовательского интерфейса. Но платформа Entity Framework также распознает необходимые заметки и его проверки.

Чтобы отключить функцию проверки на стороне клиента в MVC является простой способ проверить это. Это можно сделать в файле web.config приложения MVC. В разделе appSettings имеет ключ для ClientValidationEnabled. Задавать этот параметр в значение false будет препятствовать выполнение проверки пользовательского интерфейса.

```
<appSettings>  
    <add key="ClientValidationEnabled" value="false"/>  
    ...  
</appSettings>
```

Даже при проверке со стороны клиента отключен вы получите отклик в приложении. Сообщение об ошибке «Title это обязательное поле» отображается как. За исключением того, теперь она будет результатом проверки на стороне сервера. Платформа Entity Framework будет выполнять проверку необходимые заметки (прежде чем он даже идея сборки и команда INSERT для отправки в базу данных) и возвращает ошибку для MVC, которая будет отображаться сообщение.

Текущий API

Вы можете использовать код проверки стороне текущий API вместо заметок для получения того же клиента на стороне & сервер первого элемента. Вместо использования требуется, я покажу это с помощью проверки MaxLength.

Fluent API конфигурации применяются в том случае, как код сначала создает модель от классов. Вы можете внедрить конфигураций путем переопределения метода OnModelCreating класса DbContext. Здесь — это конфигурация, указывающего, что свойство BloggerName может быть не более 10 символов.

```

public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>().Property(p => p.BloggerName).HasMaxLength(10);
    }
}

```

Исключение ошибки проверки в зависимости от настроек Fluent API будет не автоматически охватывать пользовательского интерфейса, но можно записать его в код, а затем отреагировать на него соответствующим образом.

Вот некоторые код обработки исключений ошибок в классе BlogController приложения, который фиксирует эту ошибку проверки, при попытке сохранить блог с BloggerName, превышает максимальный предел 10-значный Entity Framework.

```

[HttpPost]
public ActionResult Edit(int id, Blog blog)
{
    try
    {
        db.Entry(blog).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch(DbEntityValidationException ex)
    {
        var error = ex.EntityValidationErrors.First().ValidationErrors.First();
        this.ModelState.AddModelError(error.PropertyName, error.ErrorMessage);
        return View();
    }
}

```

Проверка автоматически не был передан обратно в представление, поэтому используется дополнительный код, который использует ModelState.AddModelError. Это гарантирует, что сведения об ошибке сделать его в представление, в котором будет использовать ValidationMessageFor HtmlHelper для отображения ошибки.

```
@Html.ValidationMessageFor(model => model.BloggerName)
```

IValidatableObject

IValidatableObject — это интерфейс, живет в System.ComponentModel.DataAnnotations. Хотя он не является частью Entity Framework API, вы можете по-прежнему использовать его для проверки на стороне сервера в классах Entity Framework. IValidatableObject предоставляет метод Validate, Entity Framework будет вызывать во время SaveChanges. также можно вызвать самостоятельно любое время, вы хотите проверить классы.

Конфигурации, такие как необходимые и MaxLength выполняют инициализировать по одному полю. В метод Validate может иметь более сложную логику, например, сравнивая два поля.

В следующем примере класс блог была расширена для реализации IValidatableObject, а затем укажите правило, которое не может совпадать с заголовком и BloggerName.

```

public class Blog : IValidatableObject
{
    public int Id { get; set; }
    [Required]
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public DateTime DateCreated { get; set; }
    public virtual ICollection<Post> Posts { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (Title == BloggerName)
        {
            yield return new ValidationResult
                ("Blog Title cannot match Blogger Name", new[] { "Title", "BloggerName" });
        }
    }
}

```

Конструктор ValidationResult принимает строку, которая представляет сообщение об ошибке, а также массив строк, представляющих имена элементов, связанных с проверкой. Так как эта проверка проверяет заголовок и BloggerName, возвращаются как имена свойств.

В отличие от проверки, предоставляемый Fluent API результат этой проверки будет распознаваться модулем представление и обработчик исключений, который я использовал выше, чтобы добавить ошибку в ModelState не требуется. Поскольку я обоих имен свойств в ValidationResult, MVC HtmlHelpers отображает сообщение об ошибке для оба этих свойства.

Edit

Blog

Title
Julie Blog Title cannot match Blogger Name

BloggerName
Julie Blog Title cannot match Blogger Name

DateCreated
3/11/2011 12:00:00 AM

DbContext.ValidateEntity

DbContext имеет переопределяемый метод ValidateEntity. При вызове метода SaveChanges, Entity Framework будет вызывать этот метод для каждой сущности в своем кэше, состояния которых отлично без изменений. Логика проверки можно поместить непосредственно в здесь или даже использовать этот метод для вызова, например, метода Blog.Validate, добавленного в предыдущем разделе.

Ниже приведен пример переопределения ValidateEntity, которое проверяет новых записей, чтобы убедиться, что заголовок post не уже используется. Он сначала проверяет, является ли сущность post, и что добавляется свое состояние. Если это так, он выполняет в базе данных, чтобы увидеть, если уже есть запись с тем же именем. Если уже имеется сообщение, будет создан новый DbEntityValidationResult.

DbEntityValidationResult содержит dbentityentry, который должен и DbValidationErrors из коллекции ICollection для одной сущности. В начале этого метода DbEntityValidationResult создается и затем добавляются все ошибки, обнаруженные в коллекции ValidationErrors.

```

protected override DbEntityValidationResult ValidateEntity (
    System.Data.Entity.Infrastructure.DbEntityEntry entityEntry,
    IDictionary<object, object> items)
{
    var result = new DbEntityValidationResult(entityEntry, new List<DbValidationError>());
    if (entityEntry.Entity is Post && entityEntry.State == EntityState.Added)
    {
        Post post = entityEntry.Entity as Post;
        //check for uniqueness of post title
        if (Posts.Where(p => p.Title == post.Title).Count() > 0)
        {
            result.ValidationErrors.Add(
                new System.Data.Entity.Validation.DbValidationResult("Title",
                "Post title must be unique."));
        }
    }

    if (result.ValidationErrors.Count > 0)
    {
        return result;
    }
    else
    {
        return base.ValidateEntity(entityEntry, items);
    }
}

```

Явного запуска проверки

Вызов SaveChanges запускает все проверки, описанные в этой статье. Но не следует полагаться на SaveChanges. Можно проверить в другом месте в приложении.

DbContext.GetValidationErrors запустит все проверки, определенные заметки или Fluent API, проверки, созданные в IValidatableObject (например, Blog.Validate) и либо в DbContext.ValidateEntity метод.

Следующий код вызовет GetValidationErrors для текущего экземпляра DbContext. ValidationErrors группируются по типу сущности в DbValidationResults. Код сначала проходит через DbValidationResults, возвращаемый методом, а затем проходят через каждый ValidationError внутри.

```

foreach (var validationResults in db.GetValidationErrors())
{
    foreach (var error in validationResults.ValidationErrors)
    {
        Debug.WriteLine(
            "Entity Property: {0}, Error {1}",
            error.PropertyName,
            error.ErrorMessage);
    }
}

```

Дополнительные рекомендации при использовании проверки

Ниже приведены несколько аспектов, которые следует учитывать при использовании Entity Framework проверки.

- Отложенная загрузка отключена во время проверки.
- EF проверит заметок к данным на несопоставленные свойства, которые не сопоставлены со столбцом в базе данных.
- Проверка выполняется после обнаружения изменений во время SaveChanges. При внесении изменений

во время проверки отвечает для уведомления объекта отслеживания изменений.

- `DbUnexpectedValidationException` возникает в том случае, если возникают ошибки во время проверки.
- Аспекты, которые Entity Framework включаются в модель (максимальной длины, т. д.) вызовет проверку, даже в том случае, если нет заметок к данным в классах и (или) конструктор EF вы использовали для создания модели.
- Правила приоритета:
 - Вызовы Fluent API переопределить соответствующие заметки к данным
- Порядок выполнения:
 - Проверка свойства предшествует проверке типа
 - Проверка типа происходят, если свойство проверка прошла успешно
- Если свойство является сложным проверки также будет включать:
 - Проверка уровня свойств в свойства сложного типа
 - Тип уровня проверки в сложном типе, включая проверку `IValidatableObject` сложного типа

Сводка

Проверка API на платформе Entity Framework играет очень хорошо с проверками на стороне клиента в MVC, но не нужно полагаться на проверку на стороне клиента. Платформа Entity Framework позаботится проверки на стороне сервера для `DataAnnotations` или конфигураций, которые вы применили с Fluent API для `code first`.

Вы также узнали ряд точек расширения для настройки поведения с помощью интерфейса `IValidatableObject` или коснитесь в метод `DbContext.ValidateEntity`. И эти последние два средства для проверки доступны через `DbContext`, как при использовании `Code First`, `Model First` или `Database First` рабочего процесса для описания концептуальной модели.

Ресурсы Entity Framework

13.09.2018 • 2 minutes to read • [Edit Online](#)

Здесь вы найдете ссылки на дополнительные ресурсы, связанные с EF, например блоги, сторонних поставщиков, средства и расширения, примеры внедрения и т. д.

Блоги по Entity Framework

13.09.2018 • 2 minutes to read • [Edit Online](#)

Помимо документации по продукту эти блоги могут стать источником полезных сведений в Entity Framework:

Блоги группы разработчиков EF

- [Блог .NET - тега: Платформа Entity Framework](#)
- [Блог ADO.NET \(больше не «используется»\)](#)
- [Блог разработчиков EF \(больше не «используется»\)](#)

Участники блогов группы EF текущими и бывшими

- [Артур Vickers](#)
- [Brice Lambson](#)
- [Диего Вега](#)
- [Роэн Миллер](#)
- [Pawel Kadluczka](#)
- [Джеймс Алекс](#)
- [Michailov Златко](#)

Блоггеры сообщества EF

- [Джули Лерман](#)
- [Шон Вилдермус](#)

Примеры внедрения решения Microsoft для платформы Entity Framework

13.09.2018 • 7 minutes to read • [Edit Online](#)

Примеры внедрения на этой странице выделите несколько реальных рабочих проектов, которые используют Entity Framework.

NOTE

Подробные версиях эти примеры внедрения больше не доступны на веб-сайте Майкрософт. Таким образом будут удалены ссылки.

Epicor

Epicor — компания больших глобального программного обеспечения (с более 400 разработчиков), которая разрабатывает решения планирования ресурсов предприятия (ERP) для компаний в более чем 150 странах. Своего основного продукта, Epicor 9, основан на сервисноориентированной архитектуре (SOA) с помощью .NET Framework. Столкнувшись с многочисленные запросы клиентов с целью поддержки Language Integrated Query (LINQ), а также для снижения нагрузки на серверах SQL серверной части, группа решила обновление до Visual Studio 2010 и .NET Framework 4.0. С помощью Entity Framework 4.0, они смогли достичь этих целей, а также значительно упростить разработку и обслуживание. В частности платформа Entity Framework широкой поддержкой T4 позволили бы им полный контроль над их созданный код и автоматически встраивать функции сохранения производительности, например предварительно скомпилированных запросов и кэширование.

«Мы провели несколько тестов, недавно с существующим кодом, и мы смогли уменьшить запросы к SQL Server 90 процентов. Это из-за ADO.NET Entity Framework 4». — Исследование продукта Эрик Джонсон, вице-президент,

Проверите их решения

После приобретения программной системы, будет трудно поддерживать и переносить по долгосрочного планирования событий, достоверность решения использовать Visual Studio 2010 перезаписи как мощная и несложная в использовании полнофункциональных интернет-приложений на основе Silverlight 4. С помощью .NET RIA Services, они смогут быстро создавать слой служб на основе Entity Framework, избежать дублирования кода и обеспечить общую проверку и логику проверки подлинности между уровнями.

«Мы были проданы на платформе Entity Framework когда она впервые появилась и Entity Framework 4 оказался еще лучше. Улучшенные средства и проще управлять файлах EDMX, определяющие концептуальной модели, модели хранения и сопоставления между этими моделями... С помощью Entity Framework, я могу получить этот уровень доступа к данным, работа в течение дня и постройте его, как я перейду. Entity Framework является нашей де-факто доступа к данным; Я не знаю, почему любой пользователь не используется.» — Джо McBride, старший разработчик

Отображение NEC решения Америки

NEC было вводить рынка для цифровой рекламы основе на месте с решением, чтобы воспользоваться рекламодателей и владельцы сети и увеличить свой собственный доходов. Для этого оно запущено пару веб-

приложения, автоматизировать ручные процессы, необходимые в традиционных ad кампании. Узлы, созданные с помощью ASP.NET, Silverlight 3, AJAX и WCF, а также Entity Framework в слое доступа к данным для взаимодействия с SQL Server 2008.

«С SQL Server, мы полагаем, мы можем получить пропускную способность, нам нужно было обслуживать рекламодателей и сетей с информацией в режиме реального времени и надежности, чтобы гарантировать, что информация в наших критически важных приложений всегда будет доступен»- Corcoran Майк Директор по ИТ

Измерения Дарвин

С помощью технологии широкий диапазон корпорации Майкрософт, в команде в Дарвин решил написать Evolver - на портале online аватар, потребители могут использовать для создания аватары впечатляющие, реалистичные средства для использования в играх, анимации и социальные сети страниц. Преимущества производительности платформы Entity Framework, и который извлекает компонентов, таких как Windows Workflow Foundation (WF) и Windows Server AppFabric (масштабируемых приложений в оперативной памяти кэша) группа смогла доставить это удивительные продукт в 35% меньше время разработки. Несмотря на наличие члены команды разбиты на несколько стран, команда, использующая гибкого процесса разработки с помощью еженедельных выпусках.

«Мы стараемся не создавать технологии ради самих технологий. Как начальный очень важно, что мы используем технологию, которая экономит время и деньги. .NET был хороший выбор для быстрой и выгодной разработки.» — Захар Olsen, архитектор

Столовое

С более чем 15 лет опыта в разработке кассового терминала (POS) решений для малого и среднего размера ресторан групп разработчиков на столовое заключении для улучшения своих продуктов с большим количеством возможностей корпоративного уровня для привлечения большего размера Ресторан цепочки. Используя последнюю версию средства разработки корпорации Майкрософт, они могли создавать новое решение, четыре раза быстрее, чем прежде. Основные новые возможности, такие как LINQ и Entity Framework стала проще для перемещения из Crystal Reports в SQL Server 2008 и службы SQL Server Reporting Services (SSRS) для хранения своих данных и создания отчетов.

«Управление дату действия является ключом к успеху столовое — и вот почему мы решили внедрить SQL Reporting». -Николас Romanidis, директор по ИТ и разработчиков программного обеспечения

Участвовать в Entity Framework 6

13.09.2018 • 2 minutes to read • [Edit Online](#)

Entity Framework 6 разрабатывается на основе модели с открытым исходным кодом на сайте GitHub.

Несмотря на то, что является основной темой группы разработки Entity Framework в корпорации Майкрософт о добавлении новых функций в Entity Framework Core и не ожидается, что все основные функции, добавляемых к Entity Framework 6, по-прежнему мы принимаем работы.

Для публикаций, продукт, запустите в [участию вики-страницы в нашем репозитории GitHub](#).

Для публикаций, документации, запустите чтения [рекомендации вклад](#) в наш репозиторий документации.

Получить справку по использованию Entity Framework

13.09.2018 • 2 minutes to read • [Edit Online](#)



Вопросы, связанные с помощью EF

Лучший способ получить справку по использованию Entity Framework — [опубликуйте вопрос на Stack Overflow](#) с помощью **платформа entity framework** тега.

Если вы не знакомы с платформой Stack Overflow, не забудьте [Познакомьтесь с рекомендациями о вопросах](#). В частности не используйте Stack Overflow, чтобы сообщить об ошибках, задавать вопросы план или предложить новые функции.

Отчеты об ошибках и запросы функций

Если найдены ошибки, которая должна быть устранена, функция, вы бы хотели см. в разделе реализованного или вопрос не удалось найти ответ на вопрос о проблеме на [репозитория EF6 GitHub](#).

Глоссарий Entity Framework

10.10.2018 • 5 minutes to read • [Edit Online](#)

Code First

Создание модели Entity Framework с помощью кода. Модели можно ориентироваться и существующей базы данных, или новую базу данных.

Контекст

Класс, представляющий сеанс с базой данных, позволяя запрашивать и сохранять данные. Контекст является производным от класса DbContext или ObjectContext.

Соглашение об (Code First)

Правило, которое используют Entity Framework для определения вида модели из классов.

Сначала базы данных

Создание модели Entity Framework, в конструкторе EF, нацеленный существующей базы данных.

Безотложная загрузка

Шаблон загрузка связанных данных, где запрос для одного типа сущности также загружает связанные сущности как часть запроса.

Конструктор EF

Визуальный конструктор в Visual Studio, можно создать модель Entity Framework с помощью полей и строк.

Объект

Класс или объект, представляющий данные приложения, такие как клиенты, продукты и заказы.

EDM (модель данных с использованием сущностей)

Модели, описывающий сущности и связи между ними. EF использует модель EDM для описания концептуальной модели, для которого программы для разработчиков. EDM строится на модели отношений сущностей, представленных аварийного восстановления. Питером Ченом. Модель EDM, первоначально был разработан с основной задачей функции становятся модели общих данных всей технологий разработки и серверных технологий от корпорации Майкрософт. Модель EDM также используется как часть протокола OData.

Явная загрузка

Шаблон загрузка связанных данных, где связанные объекты загружаются путем вызова API.

Текущий API

Интерфейс API, который может использоваться для настройки модели Code First.

Ассоциации внешнего ключа

Ассоциация между сущностями, где свойство, которое представляет внешний ключ включен в классе зависимой сущности. Например продукт содержит свойства CategoryId.

Идентифицирующее отношение

Отношение, в котором первичный ключ основной сущности является частью первичного ключа зависимой сущности. В таком отношении зависимая сущность не может существовать без основной.

Независимое сопоставление

Ассоциации между сущностями там, где отсутствует свойство, представляющее внешний ключ в классе зависимой сущности. Например класс Product содержит связь категории, но нет свойства CategoryId.

Платформа Entity Framework отслеживает состояние ассоциации независимо от состояния сущностей в две ассоциации.

Отложенная загрузка

Шаблон загрузки связанных данных, где связанные объекты автоматически загружаются при обращении к свойству навигации.

Model First

Создание модели Entity Framework, в конструкторе EF, который затем используется для создания новой базы данных.

Свойство навигации

Свойство сущности, которая ссылается на другую сущность. Например продукт содержит свойство навигации категория и категория содержит свойства навигации продуктов.

POCO

Управляющий объект Plain Old CLR. Простой пользовательский класс, который не имеет зависимостей на любой платформе. В контексте EF, класс сущностей, который не является производным от класса EntityObject, реализующий интерфейсы или содержит все атрибуты, определенные в EF. Такие классы сущностей, которые связаны с инфраструктурой сохранения, также называются «сохраняемость».

Обратная связь

Противоположный конец связи, например, продукт. Категория и категория. Продукт.

Сущности с самостоятельным отслеживанием

Сущность, построенная на основе шаблона создания кода, которые помогут в разработке N-уровневых проектов.

Таблица на конкретный тип (TPC)

Метод сопоставления наследования, который сопоставлен каждого неабстрактный тип в иерархии для разделения таблицы в базе данных.

Одна таблица на иерархию (ТРИ)

Метод сопоставления наследования, где все типы в иерархии сопоставляются ту же таблицу в базе данных. Связан дискриминатора столбцов используется для определения, какой тип каждой строки.

Одна таблица на тип (ТРТ)

Метод сопоставления наследования, где общие свойства всех типов в иерархии, сопоставляются ту же таблицу в базе данных, но свойства, уникальные для каждого типа сопоставляются с отдельной таблицей.

Обнаружение типа

Процесс определения типов, которые должны быть частью модели Entity Framework.

Образец базы данных School

13.09.2018 • 15 minutes to read • [Edit Online](#)

В этом разделе содержатся схема и данные для базы данных School. Образец базы данных School используется в различных местах в документации по Entity Framework.

NOTE

Сервер базы данных, который устанавливается вместе с Visual Studio отличается в зависимости от версии Visual Studio, использовать. См. в разделе [выпуски Visual Studio](#) Дополнительные сведения о том, что для использования.

Ниже приведены шаги по созданию базы данных.

- Открытие Visual Studio
- **Представление -> обозревателя серверов**
- Щелкните правой кнопкой мыши **подключения к данным -> добавить соединение...**
- Если вы не подключились к базе данных с помощью обозревателя сервера прежде, чем вам нужно будет выбрать **Microsoft SQL Server** как источник данных
- Подключение к LocalDB или SQL Express, в зависимости от того, какой из них установки
- Введите **School** имя базы данных
- Выберите **OK** и вам нужно будет Если вы хотите создать новую базу данных, выберите **Да**
- Новая база данных появится в обозревателе серверов
- Если вы используете Visual Studio 2012 или более поздней версии
 - Щелкните правой кнопкой мыши, в базе данных в обозревателе серверов и выберите **новый запрос**
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Execute**
- Если вы используете Visual Studio 2010
 - Выберите **данных -> Transact SQL редактора -> новое соединение запроса...**
 - Введите **. \SQLEXPRESS** имя сервера и нажмите кнопку **OK**
 - Выберите **STESample** базы данных в раскрывающемся вниз в верхней части редактора запросов
 - Скопируйте следующий запрос SQL в новый запрос, а затем щелкните правой кнопкой мыши запрос и выберите **Выполнение SQL**

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

-- Create the Department table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Department]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Department]([DepartmentID] [int] NOT NULL,
[Name] [nvarchar](50) NOT NULL,
[Budget] [money] NOT NULL,
[StartDate] [datetime] NOT NULL,
[Administrator] [int] NULL,
CONSTRAINT [PK_Department1] PRIMARY KEY CLUSTERED
```

```

(
[DepartmentID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the Person table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Person]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Person]([PersonID] [int] IDENTITY(1,1) NOT NULL,
[LastName] [nvarchar](50) NOT NULL,
[FirstName] [nvarchar](50) NOT NULL,
[HireDate] [datetime] NULL,
[EnrollmentDate] [datetime] NULL,
[Discriminator] [nvarchar](50) NOT NULL,
CONSTRAINT [PK_School.Student] PRIMARY KEY CLUSTERED
(
[PersonID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OnsiteCourse table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OnsiteCourse]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OnsiteCourse]([CourseID] [int] NOT NULL,
[Location] [nvarchar](50) NOT NULL,
[Days] [nvarchar](50) NOT NULL,
[Time] [smalldatetime] NOT NULL,
CONSTRAINT [PK_OnsiteCourse] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OnlineCourse table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OnlineCourse]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OnlineCourse]([CourseID] [int] NOT NULL,
[URL] [nvarchar](100) NOT NULL,
CONSTRAINT [PK_OnlineCourse] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

--Create the StudentGrade table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[StudentGrade]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[StudentGrade]([EnrollmentID] [int] IDENTITY(1,1) NOT NULL,
[CourseID] [int] NOT NULL,
[StudentID] [int] NOT NULL,
[Grade] [decimal](3, 2) NULL,
CONSTRAINT [PK_StudentGrade] PRIMARY KEY CLUSTERED
(
[EnrollmentID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

```

```

-- Create the CourseInstructor table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[CourseInstructor]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[CourseInstructor]([CourseID] [int] NOT NULL,
[PersonID] [int] NOT NULL,
CONSTRAINT [PK_CourseInstructor] PRIMARY KEY CLUSTERED
(
[CourseID] ASC,
[PersonID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the Course table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Course]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Course]([CourseID] [int] NOT NULL,
[Title] [nvarchar](100) NOT NULL,
[Credits] [int] NOT NULL,
[DepartmentID] [int] NOT NULL,
CONSTRAINT [PK_School.Course] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OfficeAssignment table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OfficeAssignment]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OfficeAssignment]([InstructorID] [int] NOT NULL,
[Location] [nvarchar](50) NOT NULL,
[Timestamp] [timestamp] NOT NULL,
CONSTRAINT [PK_OfficeAssignment] PRIMARY KEY CLUSTERED
(
[InstructorID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Define the relationship between OnsiteCourse and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OnsiteCourse_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[OnsiteCourse]'))
ALTER TABLE [dbo].[OnsiteCourse] WITH CHECK ADD
CONSTRAINT [FK_OnsiteCourse_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO

ALTER TABLE [dbo].[OnsiteCourse] CHECK
CONSTRAINT [FK_OnsiteCourse_Course]
GO

-- Define the relationship between OnlineCourse and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OnlineCourse_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[OnlineCourse]'))
ALTER TABLE [dbo].[OnlineCourse] WITH CHECK ADD
CONSTRAINT [FK_OnlineCourse_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO

ALTER TABLE [dbo].[OnlineCourse] CHECK
CONSTRAINT [FK_OnlineCourse_Course]

```

```

CONSTRAINT [FK_OnlineCourse_Course]
GO

-- Define the relationship between StudentGrade and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_StudentGrade_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[StudentGrade]'))
ALTER TABLE [dbo].[StudentGrade] WITH CHECK ADD
CONSTRAINT [FK_StudentGrade_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO

ALTER TABLE [dbo].[StudentGrade] CHECK
CONSTRAINT [FK_StudentGrade_Course]
GO

--Define the relationship between StudentGrade and Student.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_StudentGrade_Student]')
AND parent_object_id = OBJECT_ID(N'[dbo].[StudentGrade]'))
ALTER TABLE [dbo].[StudentGrade] WITH CHECK ADD
CONSTRAINT [FK_StudentGrade_Student] FOREIGN KEY([StudentID])
REFERENCES [dbo].[Person] ([PersonID])
GO

ALTER TABLE [dbo].[StudentGrade] CHECK
CONSTRAINT [FK_StudentGrade_Student]
GO

-- Define the relationship between CourseInstructor and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_CourseInstructor_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[CourseInstructor]'))
ALTER TABLE [dbo].[CourseInstructor] WITH CHECK ADD
CONSTRAINT [FK_CourseInstructor_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO

ALTER TABLE [dbo].[CourseInstructor] CHECK
CONSTRAINT [FK_CourseInstructor_Course]
GO

-- Define the relationship between CourseInstructor and Person.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_CourseInstructor_Person]')
AND parent_object_id = OBJECT_ID(N'[dbo].[CourseInstructor]'))
ALTER TABLE [dbo].[CourseInstructor] WITH CHECK ADD
CONSTRAINT [FK_CourseInstructor_Person] FOREIGN KEY([PersonID])
REFERENCES [dbo].[Person] ([PersonID])
GO

ALTER TABLE [dbo].[CourseInstructor] CHECK
CONSTRAINT [FK_CourseInstructor_Person]
GO

-- Define the relationship between Course and Department.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_Course_Department]')
AND parent_object_id = OBJECT_ID(N'[dbo].[Course]'))
ALTER TABLE [dbo].[Course] WITH CHECK ADD
CONSTRAINT [FK_Course_Department] FOREIGN KEY([DepartmentID])
REFERENCES [dbo].[Department] ([DepartmentID])
GO

ALTER TABLE [dbo].[Course] CHECK CONSTRAINT [FK_Course_Department]
GO

--Define the relationship between OfficeAssignment and Person.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OfficeAssignment_Person]')
AND parent_object_id = OBJECT_ID(N'[dbo].[OfficeAssignment]'))
ALTER TABLE [dbo].[OfficeAssignment] WITH CHECK ADD
CONSTRAINT [FK_OfficeAssignment_Person] FOREIGN KEY([InstructorID])
REFERENCES [dbo].[Person] ([PersonID])

```

```

GO
ALTER TABLE [dbo].[OfficeAssignment] CHECK
CONSTRAINT [FK_OfficeAssignment_Person]
GO

-- Create InsertOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[InsertOfficeAssignment]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[InsertOfficeAssignment]
@InstructorID int,
@Location nvarchar(50)
AS
INSERT INTO dbo.OfficeAssignment (InstructorID, Location)
VALUES (@InstructorID, @Location);
IF @@ROWCOUNT > 0
BEGIN
SELECT [Timestamp] FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
END
'
END
GO

--Create the UpdateOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[UpdateOfficeAssignment]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[UpdateOfficeAssignment]
@InstructorID int,
@Location nvarchar(50),
@OrigTimestamp timestamp
AS
UPDATE OfficeAssignment SET Location=@Location
WHERE InstructorID=@InstructorID AND [Timestamp]=@OrigTimestamp;
IF @@ROWCOUNT > 0
BEGIN
SELECT [Timestamp] FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
END
'
END
GO

-- Create the DeleteOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[DeleteOfficeAssignment]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[DeleteOfficeAssignment]
@InstructorID int
AS
DELETE FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
'
END
GO

-- Create the DeletePerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[DeletePerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'

```

```

CREATE PROCEDURE [dbo].[DeletePerson]
@PersonID int
AS
DELETE FROM Person WHERE PersonID = @PersonID;
'

END
GO

-- Create the UpdatePerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[UpdatePerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[UpdatePerson]
@PersonID int,
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime,
@EnrollmentDate datetime,
@Discriminator nvarchar(50)
AS
UPDATE Person SET LastName=@LastName,
FirstName=@FirstName,
HireDate=@HireDate,
EnrollmentDate=@EnrollmentDate,
Discriminator=@Discriminator
WHERE PersonID=@PersonID;
'

END
GO

-- Create the InsertPerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[InsertPerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[InsertPerson]
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime,
@EnrollmentDate datetime,
@Discriminator nvarchar(50)
AS
INSERT INTO dbo.Person (LastName,
FirstName,
HireDate,
EnrollmentDate,
Discriminator)
VALUES (@LastName,
@FirstName,
@HireDate,
@EnrollmentDate,
@Discriminator);
SELECT SCOPE_IDENTITY() as NewPersonID;
'

END
GO

-- Create GetStudentGrades stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[GetStudentGrades]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[GetStudentGrades]
@studentID int
AS

```

```

SELECT EnrollmentID, Grade, CourseID, StudentID FROM dbo.StudentGrade
WHERE StudentID = @StudentID
'
END
GO

-- Create GetDepartmentName stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[GetDepartmentName]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[GetDepartmentName]
@ID int,
@Name nvarchar(50) OUTPUT
AS
SELECT @Name = Name FROM Department
WHERE DepartmentID = @ID
'
END
GO

-- Insert data into the Person table.
USE School
GO
SET IDENTITY_INSERT dbo.Person ON
GO
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (1, 'Abercrombie', 'Kim', '1995-03-11', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (2, 'Barzdukas', 'Gytis', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (3, 'Justice', 'Peggy', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (4, 'Fakhouri', 'Fadi', '2002-08-06', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (5, 'Harui', 'Roger', '1998-07-01', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (6, 'Li', 'Yan', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (7, 'Norman', 'Laura', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (8, 'Olivotto', 'Nino', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (9, 'Tang', 'Wayne', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (10, 'Alonso', 'Meredith', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (11, 'Lopez', 'Sophia', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (12, 'Browning', 'Meredith', null, '2000-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (13, 'Anand', 'Arturo', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (14, 'Walker', 'Alexandra', null, '2000-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (15, 'Powell', 'Carson', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (16, 'Jai', 'Damien', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (17, 'Carlson', 'Robyn', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (18, 'Zheng', 'Roger', '2004-02-12', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (19, 'Bryant', 'Carson', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (20, 'Suarez', 'Robyn', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (21, 'Holt', 'Roger', null, '2004-09-01', 'Student');

```

```

INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (22, 'Alexander', 'Carson', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (23, 'Morgan', 'Isaiah', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (24, 'Martin', 'Randall', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (25, 'Kapoor', 'Candace', '2001-01-15', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (26, 'Rogers', 'Cody', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (27, 'Serrano', 'Stacy', '1999-06-01', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (28, 'White', 'Anthony', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (29, 'Griffin', 'Rachel', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (30, 'Shan', 'Alicia', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (31, 'Stewart', 'Jasmine', '1997-10-12', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (32, 'Xu', 'Kristen', '2001-07-23', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (33, 'Gao', 'Erica', null, '2003-01-30', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (34, 'Van Houten', 'Roger', '2000-12-07', null, 'Instructor');
GO
SET IDENTITY_INSERT dbo.Person OFF
GO

-- Insert data into the Department table.
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (1, 'Engineering', 350000.00, '2007-09-01', 2);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (2, 'English', 120000.00, '2007-09-01', 6);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (4, 'Economics', 200000.00, '2007-09-01', 4);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (7, 'Mathematics', 250000.00, '2007-09-01', 3);
GO

-- Insert data into the Course table.
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1050, 'Chemistry', 4, 1);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1061, 'Physics', 4, 1);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1045, 'Calculus', 4, 7);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2030, 'Poetry', 2, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2021, 'Composition', 3, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2042, 'Literature', 4, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4022, 'Microeconomics', 3, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4041, 'Macroeconomics', 3, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4061, 'Quantitative', 2, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (3141, 'Trigonometry', 4, 7);
GO

-- Insert data into the OnlineCourse table.
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES ('2020', 'http://www.finertschool.net/Poetry');

```

```

VALUES (2050, 'http://www.fineartschool.net/poetry'),
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (2021, 'http://www.fineartschool.net/Composition');
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (4041, 'http://www.fineartschool.net/Macroeconomics');
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (3141, 'http://www.fineartschool.net/Trigonometry');

--Insert data into OnsiteCourse table.
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1050, '123 Smith', 'MTWH', '11:30');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1061, '234 Smith', 'TWHF', '13:15');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1045, '121 Smith', 'MWHF', '15:30');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (4061, '22 Williams', 'TH', '11:15');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (2042, '225 Adams', 'MTWH', '11:00');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (4022, '23 Williams', 'MWF', '9:00');

-- Insert data into the CourseInstructor table.
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1050, 1);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1061, 31);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1045, 5);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2030, 4);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2021, 27);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2042, 25);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4022, 18);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4041, 32);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4061, 34);
GO

--Insert data into the OfficeAssignment table.
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (1, '17 Smith');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (4, '29 Adams');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (5, '37 Williams');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (18, '143 Smith');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (25, '57 Adams');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (27, '271 Williams');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (31, '131 Smith');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (32, '203 Williams');
INSERT INTO dbo.OfficeAssignment(InstructorID, Location)
VALUES (34, '213 Smith');

-- Insert data into the StudentGrade table.
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 2, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2030, 2, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 2, 3);

```

```
VALUES (2021, 3, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2030, 3, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 6, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 6, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 7, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 7, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 8, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 8, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 9, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 10, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 11, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 12, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 12, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 14, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 13, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 13, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 14, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 15, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 16, 2);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 17, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 19, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 20, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 21, 2);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 22, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 22, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 22, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 23, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1045, 23, 1.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 24, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 25, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 26, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 26, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 27, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1045, 28, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 28, 3.5);
```

```
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 29, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 30, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 30, 4);
GO
```

Средства платформы Entity Framework и расширения

13.09.2018 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Расширения создаются с помощью разнообразных источников и не сохраняется как часть платформы Entity Framework. Выбирая стороннее расширение, обязательно оцените качество, лицензирование, совместимость, поддержку и другие показатели на соответствие вашим требованиям.

Платформа Entity Framework была популярных O/RM в течение многих лет. Ниже приведены некоторые примеры платные и бесплатные инструменты и расширения, разработанные для него.

- [EF Power Tools Community Edition](#)
- [Profiler EF](#)
- [Profiler ORM](#)
- [LINQPad](#)
- [LLBLGen Pro](#)
- [Средства Huagati DBML/EDMX](#)
- [Entity Developer](#)