Contents

Entity Framework
Compare EF Core & EF6
EF6 and EF Core in the Same Application
Porting from EF6 to EF Core
Validate Requirements
Porting an EDMX-Based Model
Porting a Code-Based Model
Entity Framework Core
What's New
EF Core Roadmap
EF Core 2.2 (latest stable release)
EF Core 2.1
EF Core 2.0
EF Core 1.1
EF Core 1.0
Upgrading from previous versions
From 1.0 RC1 to RC2
From 1.0 RC2 to RTM
From 1.x to 2.0
Get Started
Installing EF Core
.NET Core
New Database
ASP.NET Core
New Database
Existing Database
EF Core and Razor Pages
Universal Windows Platform (UWP)
New Database

- .NET Framework
  - New Database
  - Existing Database
- Fundamentals
  - Connection Strings
  - Logging
  - Connection Resiliency
  - Testing
    - Testing with SQLite
    - Testing with InMemory
  - Configuring a DbContext
- Creating a Model
  - Including & Excluding Types
  - Including & Excluding Properties
  - Keys (primary)
  - Generated Values
  - Required/optional properties
  - Maximum Length
  - Concurrency Tokens
  - Shadow Properties
  - Relationships
  - Indexes
  - Alternate Keys
  - Inheritance
  - Backing Fields
  - Value Conversions
  - Data Seeding
  - Entity Type Constructors
  - Owned Entity Types
  - Query Types
  - Alternating models with same DbContext
  - Spatial Data (GIS)

  - Relational Database Modeling
    - Table Mapping
    - Column Mapping
    - Data Types
    - Primary Keys
    - Default Schema
    - Computed Columns
    - Sequences
    - Default Values
    - Indexes
    - Foreign Key Constraints
    - Alternate Keys (Unique Constraints)
    - Inheritance (Relational Database)
- Managing Database Schemas
  - Migrations
    - Team Environments
    - Custom Operations
    - Using a Separate Project
    - Multiple Providers
    - Custom History Table
  - Create and Drop APIs
  - Reverse Engineering (Scaffolding)
- Querying Data
  - Basic Query
  - Loading Related Data
  - Client vs. Server Evaluation
  - Tracking vs. No-Tracking
  - Raw SQL Queries
  - Asynchronous Queries
  - How Query Works
  - Global Query Filters
  - Query Tags

- Saving Data
  - Basic Save
  - Related Data
  - Cascade Delete
  - Concurrency Conflicts
  - Transactions
  - Asynchronous Saving
  - Disconnected Entities
  - Explicit values for generated properties
- Supported .NET Implementations
- Database Providers
  - Microsoft SQL Server
    - Memory-Optimized Tables
  - SQLite
    - SQLite Limitations
  - InMemory (for Testing)
  - Writing a Database Provider
  - Provider-impacting changes
- Tools & Extensions
- Command-Line Reference
  - Package Manager Console (Visual Studio)
  - .NET Core CLI
  - Design-time DbContext Creation
  - Design-time Services
- EF Core API Reference

Entity Framework 6
- What's New
  - Roadmap
  - Past Releases
  - Upgrading To EF6
  - Visual Studio Releases
- Get Started

Fundamentals
- Get Entity Framework
- Working with DbContext
- Understanding Relationships
- Async Query & Save
- Configuration
  - Code-Based
  - Config File
  - Connection Strings
  - Dependency Resolution
- Connection Management
- Connection Resiliency
  - Retry Logic
  - Transaction Commit Failures
- Databinding
  - WinForms
  - WPF
- Disconnected Entities
  - Self Tracking Entities
    - Walkthrough
- Logging & Interception
- Performance
  - Performance Considerations (Whitepaper)
  - Using NGEN
  - Using Pre-Generated Views
- Providers
  - EF6 Provider Model
  - Spatial Support in Providers
- Using Proxies
- Testing with EF6
  - Using Mocking
  - Writing Your Own Test Doubles

- Testability with EF4 (Article)
- Creating a model
  - Using Code First
    - Workflows
      - With a New Database
      - With an Existing Database
    - Data Annotations
    - DbSets
    - Data Types
      - Enums
      - Spatial
    - Conventions
      - Built-In Conventions
      - Custom Conventions
      - Model Conventions
    - Fluent Configuration
      - Relationships
      - Types and Properties
      - Using in Visual Basic
      - Stored Procedure Mapping
    - Migrations
      - Automatic Migrations
      - Working with Existing Databases
      - Customizing Migrations History
      - Using Migrate.exe
      - Migrations in Team Environments
  - Using EF Designer
    - Workflows
      - Model-First
      - Database-First
    - Data types
      - Complex Types

- Enums
- Spatial
- Split Mappings
  - Entity Splitting
  - Table Splitting
- Inheritance Mappings
  - Table per Hierarchy
  - Table per Type
- Mapping Stored Procedures
  - Query
  - Update
- Mapping Relationships
- Multiple Diagrams
- Selecting Runtime Version
- Code Generation
  - Legacy ObjectContext
- Advanced
  - EDMX File Format
  - Defining Query
  - Multiple Result Sets
  - Table-Valued Functions
- Keyboard Shortcuts

Querying Data
- Load Method
- Local Data
- Tracking and No Tracking Queries
- Using Raw SQL Queries
- Querying Related Data

Saving Data
- Change Tracking
  - Auto Detect Changes
  - Entity State

Property Values
Handling Concurrency Conflicts
Using Transactions
Data Validation
Additional Resources
Blogs
Case Studies
Contribute
Getting Help
Glossary
School Sample Database
Tools & Extensions
Licenses
EF5
Chinese Simplified
Chinese Traditional
German
English
Spanish
French
Italian
Japanese
Korean
Russian
EF6
Prerelease
Chinese Simplified
Chinese Traditional
German
English
Spanish
French

- Italian
- Japanese
- Korean
- Russian

 EF6 API Reference

Entity Framework Documentation

Entity Framework

Entity Framework is an object-relational mapper (O/RM) that enables .NET developers to work with a database using .NET objects. It eliminates the need for most of the data-access code that developers usually need to write.

Entity Framework Core

EF Core is a lightweight, extensible, and cross-platform version of Entity Framework.

Entity Framework 6

EF 6 is a tried and tested data access technology with many years of features and stabilization.

Choosing

Find out which version of EF is right for you.

Port to EF Core

Guidance on porting an existing EF 6 application to EF Core.

EF Core
all

EF Core is a lightweight, extensible, and cross-platform version of Entity Framework.

Get Started

Overview

Create a Model

Query Data

Save Data

Tutorials

.NET Framework

.NET Core

ASP.NET Core

UWP

more...

Database providers

SQL Server

MySQL

PostgreSQL

SQLite

more...

API Reference

DbContext

DbSet<TEntity>

more...

EF 6

EF 6 is a tried and tested data access technology with many years of features and stabilization.

Get Started

Learn how to access data with Entity Framework 6.

API Reference

Browse the Entity Framework 6 API, organized by namespace.

Compare EF Core & EF6

11/15/2018 • 4 minutes to read • Edit Online

Entity Framework is an object-relational mapper (O/RM) for .NET. This article compares the two versions: Entity Framework 6 and Entity Framework Core.

Entity Framework 6

Entity Framework 6 (EF6) is a tried and tested data access technology. It was first released in 2008, as part of .NET Framework 3.5 SP1 and Visual Studio 2008 SP1. Starting with the 4.1 release it has shipped as the EntityFramework NuGet package. EF6 runs on the .NET Framework 4.x, which means it runs only on Windows.

EF6 continues to be a supported product, and will continue to see bug fixes and minor improvements.

Entity Framework Core

Entity Framework Core (EF Core) is a complete rewrite of EF6 that was first released in 2016. It ships in Nuget packages, the main one being Microsoft.EntityFrameworkCore. EF Core is a cross-platform product that can run on .NET Core or .NET Framework.

EF Core was designed to provide a developer experience similar to EF6. Most of the top-level APIs remain the same, so EF Core will feel familiar to developers who have used EF6.

Feature comparison

EF Core offers new features that won't be implemented in EF6 (such as alternate keys, batch updates, and mixed client/database evaluation in LINQ queries. But because it's a new code base, it also lacks some features that EF6 has.

The following tables compare the features available in EF Core and EF6. It's a high-level comparison and doesn't list every feature or explain differences between the same feature in different EF versions.

The EF Core column indicates the product version in which the feature first appeared.

Creating a model

FEATURE	EF 6	EF CORE
Basic class mapping	Yes	1.0
Constructors with parameters		2.1
Property value conversions		2.1
Mapped types with no keys (query types)		2.1
Conventions	Yes	1.0
Custom conventions	Yes	1.0 (partial)

FEATURE	EF 6	EF CORE
Data annotations	Yes	1.0
Fluent API	Yes	1.0
Inheritance: Table per hierarchy (TPH)	Yes	1.0
Inheritance: Table per type (TPT)	Yes	
Inheritance: Table per concrete class (TPC)	Yes	
Shadow state properties		1.0
Alternate keys		1.0
Many-to-many without join entity	Yes	
Key generation: Database	Yes	1.0
Key generation: Client		1.0
Complex/owned types	Yes	2.0
Spatial data	Yes	2.2
Graphical visualization of model	Yes	
Graphical model editor	Yes	
Model format: Code	Yes	1.0
Model format: EDMX (XML)	Yes	
Create model from database: Command line	Yes	1.0
Create model from database: VS wizard	Yes	
Update model from database	Partial	
Global query filters		2.0
Table splitting	Yes	2.0
Entity splitting	Yes	
Database scalar function mapping	Poor	2.0
Field mapping		1.1

Querying data

FEATURE	EF6	EF CORE
LINQ queries	Yes	1.0 (in-progress for complex queries)
Readable generated SQL	Poor	1.0
Mixed client/server evaluation		1.0
GroupBy translation	Yes	2.1
Loading related data: Eager	Yes	1.0
Loading related data: Eager loading for derived types		2.1
Loading related data: Lazy	Yes	2.1
Loading related data: Explicit	Yes	1.1
Raw SQL queries: Entity types	Yes	1.0
Raw SQL queries: Non-entity types (query types)	Yes	2.1
Raw SQL queries: Composing with LINQ		1.0
Explicitly compiled queries	Poor	2.0
Text-based query language (Entity SQL)	Yes	

Saving data

FEATURE	EF6	EF CORE
Change tracking: Snapshot	Yes	1.0
Change tracking: Notification	Yes	1.0
Change tracking: Proxies	Yes	
Accessing tracked state	Yes	1.0
Optimistic concurrency	Yes	1.0
Transactions	Yes	1.0
Batching of statements		1.0
Stored procedure mapping	Yes	
Disconnected graph low-level APIs	Poor	1.0
Disconnected graph End-to-end		1.0 (partial)

Other features

FEATURE	EF6	EF CORE
Migrations	Yes	1.0
Database creation/deletion APIs	Yes	1.0
Seed data	Yes	2.1
Connection resiliency	Yes	1.1
Lifecycle hooks (events, interception)	Yes	
Simple Logging (Database.Log)	Yes	
DbContext pooling		2.0

Database providers

FEATURE	EF6	EF CORE
SQL Server	Yes	1.0
MySQL	Yes	1.0
PostgreSQL	Yes	1.0
Oracle	Yes	1.0 (1)
SQLite	Yes	1.0
SQL Server Compact	Yes	1.0 (2)
DB2	Yes	1.0
Firebird	Yes	2.0
Jet (Microsoft Access)		2.0 (2)
In-memory (for testing)		1.0

[1] There is currently a paid provider available for Oracle. A free official provider for Oracle is being worked on.

[2] The SQL Server Compact and Jet providers only work on .NET Framework (not on .NET Core).

.NET implementations

FEATURE	EF6	EF CORE
.NET Framework (Console, WinForms, WPF, ASP.NET)	Yes	1.0
.NET Core (Console, ASP.NET Core)		1.0

FEATURE	EF6	EF CORE
Mono & Xamarin		1.0 (in-progress)
UWP		1.0 (in-progress)

Guidance for new applications

Consider using EF Core for a new application if both of the following conditions are true:

- The app needs the capabilities of .NET Core. For more information, see Choosing between .NET Core and .NET Framework for server apps.
- EF Core supports all of the features that the app requires. If a desired feature is missing, check the EF Core Roadmap to find out if there are plans to support it in the future.

Consider using EF6 if both of the following conditions are true:

- The app will run on Windows and the .NET Framework 4.0 or later.
- EF6 supports all of the features that the app requires.

Guidance for existing EF6 applications

Because of the fundamental changes in EF Core, we do not recommend moving an EF6 application to EF Core unless there is a compelling reason to make the change. If you want to move to EF Core to use new features, make sure you're aware of its limitations. For more information, see Porting from EF6 to EF Core. **The move from EF6 to EF Core is more a port than an upgrade.**

Next steps

For more information, see the documentation:

- Overview - EF Core
- Overview - EF6

Using EF Core and EF6 in the Same Application

8/27/2018 • 2 minutes to read • Edit Online

It is possible to use EF Core and EF6 in the same .NET Framework application or library by installing both NuGet packages.

Some types have the same names in EF Core and EF6 and differ only by namespace, which may complicate using both EF Core and EF6 in the same code file. The ambiguity can be easily removed using namespace alias directives. For example:

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

If you are porting an existing application that has multiple EF models, you can choose to selectively port some of them to EF Core, and continue using EF6 for the others.

Porting from EF6 to EF Core

8/27/2018 • 2 minutes to read • Edit Online

Because of the fundamental changes in EF Core we do not recommend attempting to move an EF6 application to EF Core unless you have a compelling reason to make the change. You should view the move from EF6 to EF Core as a port rather than an upgrade.

Before porting from EF6 to EF Core: Validate your Application's Requirements

8/27/2018 • 2 minutes to read • Edit Online

Before you start the porting process it is important to validate that EF Core meets the data access requirements for your application.

Missing features

Make sure that EF Core has all the features you need to use in your application. See Feature Comparison for a detailed comparison of how the feature set in EF Core compares to EF6. If any required features are missing, ensure that you can compensate for the lack of these features before porting to EF Core.

Behavior changes

This is a non-exhaustive list of some changes in behavior between EF6 and EF Core. It is important to keep these in mind as your port your application as they may change the way your application behaves, but will not show up as compilation errors after swapping to EF Core.

DbSet.Add/Attach and graph behavior

In EF6, calling `DbSet.Add()` on an entity results in a recursive search for all entities referenced in its navigation properties. Any entities that are found, and are not already tracked by the context, are also be marked as added. `DbSet.Attach()` behaves the same, except all entities are marked as unchanged.

EF Core performs a similar recursive search, but with some slightly different rules.

- The root entity is always in the requested state (added for `DbSet.Add` and unchanged for `DbSet.Attach`).
- **For entities that are found during the recursive search of navigation properties:**
  - **If the primary key of the entity is store generated**
    - If the primary key is not set to a value, the state is set to added. The primary key value is considered "not set" if it is assigned the CLR default value for the property type (for example, `0` for `int`, `null` for `string`, etc.).
    - If the primary key is set to a value, the state is set to unchanged.
  - If the primary key is not database generated, the entity is put in the same state as the root.

Code First database initialization

EF6 has a significant amount of magic it performs around selecting the database connection and initializing the database. Some of these rules include:

- If no configuration is performed, EF6 will select a database on SQL Express or LocalDb.
- If a connection string with the same name as the context is in the applications `App/Web.config` file, this connection will be used.
- If the database does not exist, it is created.
- If none of the tables from the model exist in the database, the schema for the current model is added to the database. If migrations are enabled, then they are used to create the database.

- If the database exists and EF6 had previously created the schema, then the schema is checked for compatibility with the current model. An exception is thrown if the model has changed since the schema was created.

EF Core does not perform any of this magic.

- The database connection must be explicitly configured in code.
- No initialization is performed. You must use `DbContext.Database.Migrate()` to apply migrations (or `DbContext.Database.EnsureCreated()` and `EnsureDeleted()` to create/delete the database without using migrations).

Code First table naming convention

EF6 runs the entity class name through a pluralization service to calculate the default table name that the entity is mapped to.

EF Core uses the name of the `DbSet` property that the entity is exposed in on the derived context. If the entity does not have a `DbSet` property, then the class name is used.

Porting an EF6 EDMX-Based Model to EF Core

8/27/2018 • 2 minutes to read • Edit Online

EF Core does not support the EDMX file format for models. The best option to port these models, is to generate a new code-based model from the database for your application.

Install EF Core NuGet packages

Install the `Microsoft.EntityFrameworkCore.Tools` NuGet package.

Regenerate the model

You can now use the reverse engineer functionality to create a model based on your existing database.

Run the following command in Package Manager Console (Tools –> NuGet Package Manager –> Package Manager Console). See Package Manager Console (Visual Studio) for command options to scaffold a subset of tables etc.

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

For example, here is the command to scaffold a model from the Blogging database on your SQL Server LocalDB instance.

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

Remove EF6 model

You would now remove the EF6 model from your application.

It is fine to leave the EF6 NuGet package (EntityFramework) installed, as EF Core and EF6 can be used side-by-side in the same application. However, if you aren't intending to use EF6 in any areas of your application, then uninstalling the package will help give compile errors on pieces of code that need attention.

Update your code

At this point, it's a matter of addressing compilation errors and reviewing code to see if the behavior changes between EF6 and EF Core will impact you.

Test the port

Just because your application compiles, does not mean it is successfully ported to EF Core. You will need to test all areas of your application to ensure that none of the behavior changes have adversely impacted your application.

> **TIP**
>
> See Getting Started with EF Core on ASP.NET Core with an Existing Database for an additional reference on how to work with an existing database,

Porting an EF6 Code-Based Model to EF Core

8/27/2018 • 2 minutes to read • Edit Online

If you've read all the caveats and you are ready to port, then here are some guidelines to help you get started.

Install EF Core NuGet packages

To use EF Core, you install the NuGet package for the database provider you want to use. For example, when targeting SQL Server, you would install `Microsoft.EntityFrameworkCore.SqlServer`. See Database Providers for details.

If you are planning to use migrations, then you should also install the `Microsoft.EntityFrameworkCore.Tools` package.

It is fine to leave the EF6 NuGet package (EntityFramework) installed, as EF Core and EF6 can be used side-by-side in the same application. However, if you aren't intending to use EF6 in any areas of your application, then uninstalling the package will help give compile errors on pieces of code that need attention.

Swap namespaces

Most APIs that you use in EF6 are in the `System.Data.Entity` namespace (and related sub-namespaces). The first code change is to swap to the `Microsoft.EntityFrameworkCore` namespace. You would typically start with your derived context code file and then work out from there, addressing compilation errors as they occur.

Context configuration (connection etc.)

As described in Ensure EF Core Will Work for Your Application, EF Core has less magic around detecting the database to connect to. You will need to override the `OnConfiguring` method on your derived context, and use the database provider specific API to setup the connection to the database.

Most EF6 applications store the connection string in the applications `App/Web.config` file. In EF Core, you read this connection string using the `ConfigurationManager` API. You may need to add a reference to the `System.Configuration` framework assembly to be able to use this 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);
    }
}
```

Update your code

At this point, it's a matter of addressing compilation errors and reviewing code to see if the behavior changes will impact you.

Existing migrations

There isn't really a feasible way to port existing EF6 migrations to EF Core.

If possible, it is best to assume that all previous migrations from EF6 have been applied to the database and then start migrating the schema from that point using EF Core. To do this, you would use the `Add-Migration` command to add a migration once the model is ported to EF Core. You would then remove all code from the `Up` and `Down` methods of the scaffolded migration. Subsequent migrations will compare to the model when that initial migration was scaffolded.

Test the port

Just because your application compiles, does not mean it is successfully ported to EF Core. You will need to test all areas of your application to ensure that none of the behavior changes have adversely impacted your application.

Entity Framework Core

9/9/2018 • 2 minutes to read • Edit Online

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology.

EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.

EF Core supports many database engines, see Database Providers for details.

The Model

With EF Core, data access is performed using a model. A model is made up of entity classes and a derived context that represents a session with the database, allowing you to query and save data. See Creating a Model to learn more.

You can generate a model from an existing database, hand code a model to match your database, or use EF Migrations to create a database from your model (and evolve it as your model changes over time).

```
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; }
    }
}
```

Querying

Instances of your entity classes are retrieved from the database using Language Integrated Query (LINQ). See Querying Data to learn more.

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

Saving Data

Data is created, deleted, and modified in the database using instances of your entity classes. See Saving Data to learn more.

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

Next steps

For introductory tutorials, see Getting Started with Entity Framework Core.

What is new in EF core

12/4/2018 • 2 minutes to read • Edit Online

You can use the links below to learn about new features in each release:

Future releases

- EF Core Roadmap

Recent releases

- EF Core 2.2 (latest stable release)
- EF Core 2.1

Past releases

- EF Core 2.0
- EF Core 1.1
- EF Core 1.0

Entity Framework Core Roadmap

1/6/2019 • 6 minutes to read • Edit Online

> **IMPORTANT**
> Please note that the feature sets and schedules of future releases are always subject to change, and although we will try to keep this page up to date, it may not reflect our latest plans at all times.

EF Core 3.0

With EF Core 2.2 out the door, our main focus is now EF Core 3.0, which will be aligned with the .NET Core 3.0 and ASP.NET 3.0 releases.

We haven't completed any new features yet, so the EF Core 3.0 Preview 1 packages published to the NuGet Gallery on December 2018 only contain bug fixes, minor improvements, and changes we made in preparation for the 3.0 work.

In fact, we still need to refine our release planning for 3.0, to make sure we have the right set of features that can be completed in the allotted time. We will share more information as we get more clarity, but here are some high-level themes and features we itend to work on:

- **LINQ improvements (#12795)**: LINQ enables you to write database queries without leaving your language of choice, taking advantage of rich type information to get IntelliSense and compile-time type checking. But LINQ also enables you to write an unlimited number of complicated queries, and that has always been a huge challenge for LINQ providers. In the first few versions of EF Core, we solved that in part by figuring out what portions of a query could be translated to SQL, and then by allowing the rest of the query to execute in memory on the client. This client-side execution can be desirable in some situations, but in many other cases it can result in inefficient queries that may not be identified until an application is deployed to production. In EF Core 3.0, we are planning to make profound changes to how our LINQ implementation works, and how we test it. The goals are to make it more robust (for example, to avoid breaking queries in patch releases), to be able to translate more expressions correctly into SQL, to generate efficient queries in more cases, and to prevent inefficient queries from going undetected.

- **Cosmos DB support (#8443)**: We're working on a Cosmos DB provider for EF Core, to enable developers familiar with the EF programing model to easily target Azure Cosmos DB as an application database. The goal is to make some of the advantages of Cosmos DB, like global distribution, “always on” availability, elastic scalability, and low latency, even more accessible to .NET developers. The provider will enable most EF Core features, like automatic change tracking, LINQ, and value conversions, against the SQL API in Cosmos DB. We started this effort before EF Core 2.2, and we have made some preview versions of the provider available. The new plan is to continue developing the provider alongside EF Core 3.0.

- **C# 8.0 support (#12047)**: We want our customers to take advantage of some of the new features coming in C# 8.0 like async streams (including await for each) and nullable reference types while using EF Core.

- **Reverse engineering database views into query types (#1679)**: In EF Core 2.1, we added support for query types, which can represent data that can be read from the database, but cannot be updated. Query types are a great fit for mapping database views, so in EF Core 3.0, we would like to automate the creation of query types for database views.

- **Property bag entities (#13610 and #9914)**: This feature is about enabling entities that store data in indexed properties instead of regular properties, and also about being able to use instances of the same .NET class (potentially something as simple as a `Dictionary<string, object>`) to represent different entity types in the same EF Core model. This feature is a stepping stone to support many-to-many relationships without a join entity, which is one of the most requested improvements for EF Core.

- **EF 6.3 on .NET Core (EF6 #271)**: We understand that many existing applications use previous versions of EF, and that porting them to EF Core only to take advantage of .NET Core can sometimes require a significant effort. For that reason, we will be adapting the next version of EF 6 to run on .NET Core 3.0. We are doing this to facilitate porting existing applications with minimal changes. There are going to be some limitations (for example, it will require new providers, spatial support with SQL Server won't be enabled), and there are no new features planned for EF 6.

In the meantime, you can use this query in our issue tracker to see work items tentatively assigned to 3.0.

Schedule

The schedule for EF Core is in-sync with the .NET Core schedule and ASP.NET Core schedule.

Backlog

The Backlog Milestone in our issue tracker contains issues that we expect to work on someday, or that we think someone from the community could tackle. Customers are welcome to submit comments and votes on these issues. Contributors looking to work on any of these issues are encouraged to first start a discussion on how to approach them.

There is never a guarantee that we will work on any given feature in a specific version of EF Core. As in all software projects, priorities, release schedules, and available resources can change at any point. But if we intend to resolve an issue in a specific timeframe, we'll assign it to a release milestone instead of the backlog milestone. We routinely move issues between the backlog and release milestones as part of our release planning process.

We'll likely close an issue if we don't plan to ever address it. But we can reconsider an issue that we previously closed if we get new information about it.

Release planning process

We often get questions about how we choose specific features to go into a particular release. Our backlog certainly doesn't automatically translate into release plans. The presence of a feature in EF6 also doesn't automatically mean that the feature needs to be implemented in EF Core.

It's difficult to detail the whole process we follow to plan a release. Much of it is discussing the specific features, opportunities and priorities, and the process itself also evolves with every release. However, we can summarize the common questions we try to answer when deciding what to work on next:

1. **How many developers we think will use the feature and how much better will it make their applications/experience?** To answer this, we collect feedback from many sources — Comments and votes on issues is one of those sources.

2. **What are the workarounds people can use if we don't implement this feature yet?** For example, many developers can map a join table to work around lack of native many-to-many support. Obviously, not all developers want to do it, but many can, and that counts as a factor in our decision.

3. **Does implementing this feature evolve the architecture of EF Core such that it moves us closer to implementing other features?** We tend to favor features that act as building blocks for other features. For example, property bag entities can help us move towards many-to-many support, and entity constructors enabled our lazy loading support.

4. **Is the feature an extensibility point?** We tend to favor extensibility points over normal features because they enable developers to hook their own behaviors and compensate for any missing functionality.

5. **What is the synergy of the feature when used in combination with other products?** We favor features that enable or significantly improve the experience of using EF Core with other products, such as .NET Core, the latest version of Visual Studio, Microsoft Azure, etc.
6. **What are the skills of the people available to work on a feature, and how to best leverage these resources?** Each member of the EF team and our community contributors has different levels of experience in different areas, so we have to plan accordingly. Even if we wanted to have "all hands on deck" to work on a specific feature like GroupBy translations, or many-to-many, that wouldn't be practical.

As mentioned before, the process evolves on every release. In the future we'll try to add more opportunities for members of the community to provide inputs into our release plans. For example, we would like to make it easier to review draft designs of the features and of the release plan itself.

New features in EF Core 2.2

11/15/2018 • 2 minutes to read • Edit Online

Spatial data support

Spatial data can be used to represent the physical location and shape of objects. Many databases can natively store, index, and query spatial data. Common scenarios include querying for objects within a given distance, and testing if a polygon contains a given location. EF Core 2.2 now supports working with spatial data from various databases using types from the NetTopologySuite (NTS) library.

Spatial data support is implemented as a series of provider-specific extension packages. Each of these packages contributes mappings for NTS types and methods, and the corresponding spatial types and functions in the database. Such provider extensions are now available for SQL Server, SQLite, and PostgreSQL (from the Npgsql project). Spatial types can be used directly with the EF Core in-memory provider without additional extensions.

Once the provider extension is installed, you can add properties of supported types to your entities. For example:

```
using NetTopologySuite.Geometries;

namespace MyApp
{
  public class Friend
  {
    [Key]
    public string Name { get; set; }

    [Required]
    public Point Location { get; set; }
  }
}
```

You can then persist entities with spatial data:

```
using (var context = new MyDbContext())
{
    context.Add(
        new Friend
        {
            Name = "Bill",
            Location = new Point(-122.34877, 47.6233355) {SRID = 4326 }
        });
    context.SaveChanges();
}
```

And you can execute database queries based on spatial data and operations:

```
  var nearestFriends =
      (from f in context.Friends
      orderby f.Location.Distance(myLocation) descending
      select f).Take(5).ToList();
```

For more information on this feature, see the spatial types documentation.

Collections of owned entities

EF Core 2.0 added the ability to model ownership in one-to-one associations. EF Core 2.2 extends the ability to express ownership to one-to-many associations. Ownership helps constrain how entities are used.

For example, owned entities:

- Can only ever appear on navigation properties of other entity types.
- Are automatically loaded, and can only be tracked by a DbContext alongside their owner.

In relational databases, owned collections are mapped to separate tables from the owner, just like regular one-to-many associations. But in document-oriented databases, we plan to nest owned entities (in owned collections or references) within the same document as the owner.

You can use the feature by calling the new OwnsMany() API:

```
modelBuilder.Entity<Customer>().OwnsMany(c => c.Addresses);
```

For more information, see the updated owned entities documentation.

Query tags

This feature simplifies the correlation of LINQ queries in code with generated SQL queries captured in logs.

To take advantage of query tags, you annotate a LINQ query using the new TagWith() method. Using the spatial query from a previous example:

```
  var nearestFriends =
      (from f in context.Friends.TagWith(@"This is my spatial query!")
      orderby f.Location.Distance(myLocation) descending
      select f).Take(5).ToList();
```

This LINQ query will produce the following SQL output:

```
-- This is my spatial query!

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

For more information, see the query tags documentation.

New features in EF Core 2.1

9/13/2018 • 6 minutes to read • Edit Online

Besides numerous bug fixes and small functional and performance enhancements, EF Core 2.1 includes some compelling new features:

Lazy loading

EF Core now contains the necessary building blocks for anyone to author entity classes that can load their navigation properties on demand. We have also created a new package, Microsoft.EntityFrameworkCore.Proxies, that leverages those building blocks to produce lazy loading proxy classes based on minimally modified entity classes (for example, classes with virtual navigation properties).

Read the section on lazy loading for more information about this topic.

Parameters in entity constructors

As one of the required building blocks for lazy loading, we enabled the creation of entities that take parameters in their constructors. You can use parameters to inject property values, lazy loading delegates, and services.

Read the section on entity constructor with parameters for more information about this topic.

Value conversions

Until now, EF Core could only map properties of types natively supported by the underlying database provider. Values were copied back and forth between columns and properties without any transformation. Starting with EF Core 2.1, value conversions can be applied to transform the values obtained from columns before they are applied to properties, and vice versa. We have a number of conversions that can be applied by convention as necessary, as well as an explicit configuration API that allows registering custom conversions between columns and properties. Some of the application of this feature are:

- Storing enums as strings
- Mapping unsigned integers with SQL Server
- Automatic encryption and decryption of property values

Read the section on value conversions for more information about this topic.

LINQ GroupBy translation

Before version 2.1, in EF Core the GroupBy LINQ operator would always be evaluated in memory. We now support translating it to the SQL GROUP BY clause in most common cases.

This example shows a query with GroupBy used to compute various aggregate functions:

```
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 => Amount)
        });
```

The corresponding SQL translation looks like this:

```
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];
```

Data Seeding

With the new release it will be possible to provide initial data to populate a database. Unlike in EF6, seeding data is associated to an entity type as part of the model configuration. Then EF Core migrations can automatically compute what insert, update or delete operations need to be applied when upgrading the database to a new version of the model.

As an example, you can use this to configure seed data for a Post in `OnModelCreating`:

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

Read the section on data seeding for more information about this topic.

Query types

An EF Core model can now include query types. Unlike entity types, query types do not have keys defined on them and cannot be inserted, deleted or updated (that is, they are read-only), but they can be returned directly by queries. Some of the usage scenarios for query types are:

- Mapping to views without primary keys
- Mapping to tables without primary keys
- Mapping to queries defined in the model
- Serving as the return type for `FromSql()` queries

Read the section on query types for more information about this topic.

Include for derived types

It will be now possible to specify navigation properties only defined on derived types when writing expressions for the `Include` method. For the strongly typed version of `Include`, we support using either an explicit cast or the `as` operator. We also now support referencing the names of navigation property defined on derived types in the string version of `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");
```

Read the section on Include with derived types for more information about this topic.

System.Transactions support

We have added the ability to work with System.Transactions features such as TransactionScope. This will work on both .NET Framework and .NET Core when using database providers that support it.

Read the section on System.Transactions for more information about this topic.

Better column ordering in initial migration

Based on customer feedback, we have updated migrations to initially generate columns for tables in the same order as properties are declared in classes. Note that EF Core cannot change order when new members are added after the initial table creation.

Optimization of correlated subqueries

We have improved our query translation to avoid executing "N + 1" SQL queries in many common scenarios in which the usage of a navigation property in the projection leads to joining data from the root query with data from a correlated subquery. The optimization requires buffering the results from the subquery, and we require that you modify the query to opt-in the new behavior.

As an example, the following query normally gets translated into one query for Customers, plus N (where "N" is the number of customers returned) separate queries for Orders:

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

By including `ToList()` in the right place, you indicate that buffering is appropriate for the Orders, which enable the optimization:

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

Note that this query will be translated to only two SQL queries: One for Customers and the next one for Orders.

[Owned] attribute

It is now possible to configure owned entity types by simply annotating the type with `[Owned]` and then making sure the owner entity is added to the model:

```
[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; }
}
```

Command-line tool dotnet-ef included in .NET Core SDK

The *dotnet-ef* commands are now part of the .NET Core SDK, therefore it will no longer be necessary to use DotNetCliToolReference in the project to be able to use migrations or to scaffold a DbContext from an existing database.

See the section on installing the tools for more details on how to enable command line tools for different versions of the .NET Core SDK and EF Core.

Microsoft.EntityFrameworkCore.Abstractions package

The new package contains attributes and interfaces that you can use in your projects to light up EF Core features without taking a dependency on EF Core as a whole. For example, the [Owned] attribute and the ILazyLoader interface are located here.

State change events

New `Tracked` And `StateChanged` events on `ChangeTracker` can be used to write logic that reacts to entities entering the DbContext or changing their state.

Raw SQL parameter analyzer

A new code analyzer is included with EF Core that detects potentially unsafe usages of our raw-SQL APIs, like `FromSql` or `ExecuteSqlCommand`. For example, for the following query, you will see a warning because *minAge* is not parameterized:

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

Database provider compatibility

It is recommended that you use EF Core 2.1 with providers that have been updated or at least tested to work with EF Core 2.1.

> **TIP**
>
> If you find any unexpected incompatibility or any issue in the new features, or if you have feedback on them, please report it using our issue tracker.

New features in EF Core 2.0

8/27/2018 • 9 minutes to read • Edit Online

.NET Standard 2.0

EF Core now targets .NET Standard 2.0, which means it can work with .NET Core 2.0, .NET Framework 4.6.1, and other libraries that implement .NET Standard 2.0. See Supported .NET Implementations for more details on what is supported.

Modeling

Table splitting

It is now possible to map two or more entity types to the same table where the primary key column(s) will be shared and each row will correspond to two or more entities.

To use table splitting an identifying relationship (where foreign key properties form the primary key) must be configured between all of the entity types sharing the table:

```
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");
```

Owned types

An owned entity type can share the same CLR type with another owned entity type, but since it cannot be identified just by the CLR type there must be a navigation to it from another entity type. The entity containing the defining navigation is the owner. When querying the owner the owned types will be included by default.

By convention a shadow primary key will be created for the owned type and it will be mapped to the same table as the owner by using table splitting. This allows to use owned types similarly to how complex types are used in 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; }
}
```

Read the section on owned entity types for more information on this feature.

Model-level query filters

EF Core 2.0 includes a new feature we call Model-level query filters. This feature allows LINQ query predicates (a boolean expression typically passed to the LINQ Where query operator) to be defined directly on Entity Types in the metadata model (usually in OnModelCreating). Such filters are automatically applied to any LINQ queries involving those Entity Types, including Entity Types referenced indirectly, such as through the use of Include or direct navigation property references. Some common applications of this feature are:

- Soft delete - An Entity Types defines an IsDeleted property.
- Multi-tenancy - An Entity Type defines a TenantId property.

Here is a simple example demonstrating the feature for the two scenarios listed above:

```
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 );
    }
}
```

We define a model-level filter that implements multi-tenancy and soft-delete for instances of the `Post` Entity Type. Note the use of a DbContext instance level property: `TenantId`. Model-level filters will use the value from the correct context instance (that is, the context instance that is executing the query).

Filters may be disabled for individual LINQ queries using the IgnoreQueryFilters() operator.

Limitations

- Navigation references are not allowed. This feature may be added based on feedback.

- Filters can only be defined on the root Entity Type of a hierarchy.

Database scalar function mapping

EF Core 2.0 includes an important contribution from Paul Middleton which enables mapping database scalar functions to method stubs so that they can be used in LINQ queries and translated to SQL.

Here is a brief description of how the feature can be used:

Declare a static method on your `DbContext` and annotate it with `DbFunctionAttribute`:

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

Methods like this are automatically registered. Once registered, calls to the method in a LINQ query can be translated to function calls in SQL:

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

A few things to note:

- By convention the name of the method is used as the name of a function (in this case a user defined function) when generating the SQL, but you can override the name and schema during method registration
- Currently only scalar functions are supported
- You must create the mapped function in the database. EF Core migrations will not take care of creating it

Self-contained type configuration for code first

In EF6 it was possible to encapsulate the code first configuration of a specific entity type by deriving from *EntityTypeConfiguration*. In EF Core 2.0 we are bringing this pattern back:

```
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());
```

High Performance

DbContext pooling

The basic pattern for using EF Core in an ASP.NET Core application usually involves registering a custom DbContext type into the dependency injection system and later obtaining instances of that type through constructor parameters in controllers. This means a new instance of the DbContext is created for each requests.

In version 2.0 we are introducing a new way to register custom DbContext types in dependency injection which transparently introduces a pool of reusable DbContext instances. To use DbContext pooling, use the `AddDbContextPool` instead of `AddDbContext` during service registration:

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

If this method is used, at the time a DbContext instance is requested by a controller we will first check if there is an instance available in the pool. Once the request processing finalizes, any state on the instance is reset and the instance is itself returned to the pool.

This is conceptually similar to how connection pooling operates in ADO.NET providers and has the advantage of saving some of the cost of initialization of DbContext instance.

Limitations

The new method introduces a few limitations on what can be done in the `OnConfiguring()` method of the DbContext.

> **WARNING**
> Avoid using DbContext Pooling if you maintain your own state (for example, private fields) in your derived DbContext class that should not be shared across requests. EF Core will only reset the state that is aware of before adding a DbContext instance to the pool.

Explicitly compiled queries

This is the second opt-in performance features designed to offer benefits in high-scale scenarios.

Manual or explicitly compiled query APIs have been available in previous versions of EF and also in LINQ to SQL to allow applications to cache the translation of queries so that they can be computed only once and executed many times.

Although in general EF Core can automatically compile and cache queries based on a hashed representation of the query expressions, this mechanism can be used to obtain a small performance gain by bypassing the computation of the hash and the cache lookup, allowing the application to use an already compiled query through the invocation of a delegate.

```
// 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);
}
```

Change Tracking

Attach can track a graph of new and existing entities

EF Core supports automatic generation of key values through a variety of mechanisms. When using this feature, a value is generated if the key property is the CLR default--usually zero or null. This means that a graph of entities can be passed to `DbContext.Attach` or `DbSet.Attach` and EF Core will mark those entities that have a key already set as `Unchanged` while those entities that do not have a key set will be marked as `Added`. This makes it easy to attach a graph of mixed new and existing entities when using generated keys. `DbContext.Update` and `DbSet.Update` work in the same way, except that entities with a key set are marked as `Modified` instead of `Unchanged`.

Query

Improved LINQ Translation

Enables more queries to successfully execute, with more logic being evaluated in the database (rather than in-memory) and less data unnecessarily being retrieved from the database.

GroupJoin improvements

This work improves the SQL that is generated for group joins. Group joins are most often a result of sub-queries on optional navigation properties.

String interpolation in FromSql and ExecuteSqlCommand

C# 6 introduced String Interpolation, a feature that allows C# expressions to be directly embedded in string literals, providing a nice way of building strings at runtime. In EF Core 2.0 we added special support for interpolated strings to our two primary APIs that accept raw SQL strings: `FromSql` and `ExecuteSqlCommand`. This new support allows C# string interpolation to be used in a 'safe' manner. That is, in a way that protects against common SQL injection mistakes that can occur when dynamically constructing SQL at runtime.

Here is an example:

```
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();
  }
```

In this example, there are two variables embedded in the SQL format string. EF Core will produce the following SQL:

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

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

EF.Functions.Like()

We have added the EF.Functions property which can be used by EF Core or providers to define methods that map to database functions or operators so that those can be invoked in LINQ queries. The first example of such a method is Like():

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

Note that Like() comes with an in-memory implementation, which can be handy when working against an in-memory database or when evaluation of the predicate needs to occur on the client side.

Database management

Pluralization hook for DbContext Scaffolding

EF Core 2.0 introduces a new *IPluralizer* service that is used to singularize entity type names and pluralize DbSet names. The default implementation is a no-op, so this is just a hook where folks can easily plug in their own pluralizer.

Here is what it looks like for a developer to hook in their own pluralizer:

```
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;
    }
}
```

Others

Move ADO.NET SQLite provider to SQLitePCL.raw

This gives us a more robust solution in Microsoft.Data.Sqlite for distributing native SQLite binaries on different platforms.

Only one provider per model

Significantly augments how providers can interact with the model and simplifies how conventions, annotations and fluent APIs work with different providers.

EF Core 2.0 will now build a different IModel for each different provider being used. This is usually transparent to the application. This has facilitated a simplification of lower-level metadata APIs such that any access to *common relational metadata concepts* is always made through a call to `.Relational` instead of `.SqlServer`, `.Sqlite`, etc.

Consolidated Logging and Diagnostics

Logging (based on ILogger) and Diagnostics (based on DiagnosticSource) mechanisms now share more code.

The event IDs for messages sent to an ILogger have changed in 2.0. The event IDs are now unique across EF Core code. These messages now also follow the standard pattern for structured logging used by, for example, MVC.

Logger categories have also changed. There is now a well-known set of categories accessed through DbLoggerCategory.

DiagnosticSource events now use the same event ID names as the corresponding `ILogger` messages.

New features in EF Core 1.1

8/27/2018 • 2 minutes to read • Edit Online

Modelling

Field mapping

Allows you to configure a backing field for a property. This can be useful for read-only properties, or data that has Get/Set methods rather than a property.

Mapping to Memory-Optimized Tables in SQL Server

You can specify that the table an entity is mapped to is memory-optimized. When using EF Core to create and maintain a database based on your model (either with migrations or `Database.EnsureCreated()`), a memory-optimized table will be created for these entities.

Change tracking

Additional change tracking APIs from EF6

Such as `Reload`, `GetModifiedProperties`, `GetDatabaseValues` etc.

Query

Explicit Loading

Allows you to trigger population of a navigation property on an entity that was previously loaded from the database.

DbSet.Find

Provides an easy way to fetch an entity based on its primary key value.

Other

Connection resiliency

Automatically retries failed database commands. This is especially useful when connection to SQL Azure, where transient failures are common.

Simplified service replacement

Makes it easier to replace internal services that EF uses.

Features included in EF Core 1.0

8/27/2018 • 3 minutes to read • Edit Online

Platforms

.NET Framework 4.5.1

Includes Console, WPF, WinForms, ASP.NET 4, etc.

.NET Standard 1.3

Including ASP.NET Core targeting both .NET Framework and .NET Core on Windows, OSX, and Linux.

Modelling

Basic modelling

Based on POCO entities with get/set properties of common scalar types (`int` , `string` , etc.).

Relationships and navigation properties

One-to-many and One-to-zero-or-one relationships can be specified in the model based on a foreign key. Navigation properties of simple collection or reference types can be associated with these relationships.

Built-in conventions

These build an initial model based on the shape of the entity classes.

Fluent API

Allows you to override the `OnModelCreating` method on your context to further configure the model that was discovered by convention.

Data annotations

Are attributes that can be added to your entity classes/properties and will influence the EF model. For example, adding `[Required]` will let EF know that a property is required.

Relational Table mapping

Allows entities to be mapped to tables/columns.

Key value generation

Including client-side generation and database generation.

Database generated values

Allows for values to be generated by the database on insert (default values) or update (computed columns).

Sequences in SQL Server

Allows for sequence objects to be defined in the model.

Unique constraints

Allows for the definition of alternate keys and the ability to define relationships that target that key.

Indexes

Defining indexes in the model automatically introduces indexes in the database. Unique indexes are also supported.

Shadow state properties

Allows for properties to be defined in the model that are not declared and are not stored in the .NET class but can be tracked and updated by EF Core. Commonly used for foreign key properties when exposing these in the object is not desired.

Table-Per-Hierarchy inheritance pattern

Allows entities in an inheritance hierarchy to be saved to a single table using a discriminator column to identify they entity type for a given record in the database.

Model validation

Detects invalid patterns in the model and provides helpful error messages.

Change tracking

Snapshot change tracking

Allows changes in entities to be detected automatically by comparing current state against a copy (snapshot) of the original state.

Notification change tracking

Allows your entities to notify the change tracker when property values are modified.

Accessing tracked state

Via `DbContext.Entry` and `DbContext.ChangeTracker`.

Attaching detached entities/graphs

The new `DbContext.AttachGraph` API helps re-attach entities to a context in order to save new/modified entities.

Saving data

Basic save functionality

Allows changes to entity instances to be persisted to the database.

Optimistic Concurrency

Protects against overwriting changes made by another user since data was fetched from the database.

Async SaveChanges

Can free up the current thread to process other requests while the database processes the commands issued from `SaveChanges`.

Database Transactions

Means that `SaveChanges` is always atomic (meaning it either completely succeeds, or no changes are made to the database). There are also transaction related APIs to allow sharing transactions between context instances etc.

Relational: Batching of statements

Provides better performance by batching up multiple INSERT/UPDATE/DELETE commands into a single roundtrip to the database.

Query

Basic LINQ support

Provides the ability to use LINQ to retrieve data from the database.

Mixed client/server evaluation

Enables queries to contain logic that cannot be evaluated in the database, and must therefore be evaluated after the data is retrieved into memory.

NoTracking

Queries enables quicker query execution when the context does not need to monitor for changes to the entity instances (this is useful if the results are read-only).

Eager loading

Provides the `Include` and `ThenInclude` methods to identify related data that should also be fetched when querying.

Async query

Can free up the current thread (and it's associated resources) to process other requests while the database processes the query.

Raw SQL queries

Provides the `DbSet.FromSql` method to use raw SQL queries to fetch data. These queries can also be composed on using LINQ.

Database schema management

Database creation/deletion APIs

Are mostly designed for testing where you want to quickly create/delete the database without using migrations.

Relational database migrations

Allow a relational database schema to evolve overtime as your model changes.

Reverse engineer from database

Scaffolds an EF model based on an existing relational database schema.

Database providers

SQL Server

Connects to Microsoft SQL Server 2008 onwards.

SQLite

Connects to a SQLite 3 database.

In-Memory

Is designed to easily enable testing without connecting to a real database.

3rd party providers

Several providers are available for other database engines. See Database Providers for a complete list.

Upgrading from EF Core 1.0 RC1 to 1.0 RC2

8/27/2018 • 4 minutes to read • Edit Online

This article provides guidance for moving an application built with the RC1 packages to RC2.

Package Names and Versions

Between RC1 and RC2, we changed from "Entity Framework 7" to "Entity Framework Core". You can read more about the reasons for the change in this post by Scott Hanselman. Because of this change, our package names changed from `EntityFramework.*` to `Microsoft.EntityFrameworkCore.*` and our versions from `7.0.0-rc1-final` to `1.0.0-rc2-final` (or `1.0.0-preview1-final` for tooling).

You will need to completely remove the RC1 packages and then install the RC2 ones. Here is the mapping for some common packages.

RC1 PACKAGE	RC2 EQUIVALENT
EntityFramework.MicrosoftSqlServer 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.SqlServer 1.0.0-rc2-final
EntityFramework.SQLite 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.Sqlite 1.0.0-rc2-final
EntityFramework7.Npgsql 3.1.0-rc1-3	NpgSql.EntityFrameworkCore.Postgres
EntityFramework.SqlServerCompact35 7.0.0-rc1-final	EntityFrameworkCore.SqlServerCompact35 1.0.0-rc2-final
EntityFramework.SqlServerCompact40 7.0.0-rc1-final	EntityFrameworkCore.SqlServerCompact40 1.0.0-rc2-final
EntityFramework.InMemory 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.InMemory 1.0.0-rc2-final
EntityFramework.IBMDataServer 7.0.0-beta1	Not yet available for RC2
EntityFramework.Commands 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.Tools 1.0.0-preview1-final
EntityFramework.MicrosoftSqlServer.Design 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.SqlServer.Design 1.0.0-rc2-final

Namespaces

Along with package names, namespaces changed from `Microsoft.Data.Entity.*` to `Microsoft.EntityFrameworkCore.*`. You can handle this change with a find/replace of `using Microsoft.Data.Entity` with `using Microsoft.EntityFrameworkCore`.

Table Naming Convention Changes

A significant functional change we took in RC2 was to use the name of the `DbSet<TEntity>` property for a given entity as the table name it maps to, rather than just the class name. You can read more about this change in the related announcement issue.

For existing RC1 applications, we recommend adding the following code to the start of your `OnModelCreating` method to keep the RC1 naming strategy:

```
foreach (var entity in modelBuilder.Model.GetEntityTypes())
{
    entity.Relational().TableName = entity.DisplayName();
}
```

If you want to adopt the new naming strategy, we would recommend successfully completing the rest of the upgrade steps and then removing the code and creating a migration to apply the table renames.

AddDbContext / Startup.cs Changes (ASP.NET Core Projects Only)

In RC1, you had to add Entity Framework services to the application service provider - in `Startup.ConfigureServices(...)`:

```
services.AddEntityFramework()
  .AddSqlServer()
  .AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

In RC2, you can remove the calls to `AddEntityFramework()`, `AddSqlServer()`, etc.:

```
services.AddDbContext<ApplicationDbContext>(options =>
  options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

You also need to add a constructor, to your derived context, that takes context options and passes them to the base constructor. This is needed because we removed some of the scary magic that snuck them in behind the scenes:

```
public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
}
```

Passing in an IServiceProvider

If you have RC1 code that passes an `IServiceProvider` to the context, this has now moved to `DbContextOptions`, rather than being a separate constructor parameter. Use `DbContextOptionsBuilder.UseInternalServiceProvider(...)` to set the service provider.

Testing

The most common scenario for doing this was to control the scope of an InMemory database when testing. See the updated Testing article for an example of doing this with RC2.

Resolving Internal Services from Application Service Provider (ASP.NET Core Projects Only)

If you have an ASP.NET Core application and you want EF to resolve internal services from the application service provider, there is an overload of `AddDbContext` that allows you to configure this:

```
services.AddEntityFrameworkSqlServer()
  .AddDbContext<ApplicationDbContext>((serviceProvider, options) =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"])
           .UseInternalServiceProvider(serviceProvider)); );
```

WARNING

We recommend allowing EF to internally manage its own services, unless you have a reason to combine the internal EF services into your application service provider. The main reason you may want to do this is to use your application service provider to replace services that EF uses internally

DNX Commands => .NET CLI (ASP.NET Core Projects Only)

If you previously used the `dnx ef` commands for ASP.NET 5 projects, these have now moved to `dotnet ef` commands. The same command syntax still applies. You can use `dotnet ef --help` for syntax information.

The way commands are registered has changed in RC2, due to DNX being replaced by .NET CLI. Commands are now registered in a `tools` section in `project.json`:

```
"tools": {
  "Microsoft.EntityFrameworkCore.Tools": {
    "version": "1.0.0-preview1-final",
    "imports": [
      "portable-net45+win8+dnxcore50",
      "portable-net45+win8"
    ]
  }
}
```

TIP

If you use Visual Studio, you can now use Package Manager Console to run EF commands for ASP.NET Core projects (this was not supported in RC1). You still need to register the commands in the `tools` section of `project.json` to do this.

Package Manager Commands Require PowerShell 5

If you use the Entity Framework commands in Package Manager Console in Visual Studio, then you will need to ensure you have PowerShell 5 installed. This is a temporary requirement that will be removed in the next release (see issue #5327 for more details).

Using "imports" in project.json

Some of EF Core's dependencies do not support .NET Standard yet. EF Core in .NET Standard and .NET Core projects may require adding "imports" to project.json as a temporary workaround.

When adding EF, NuGet restore will display this error message:

```
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)
```

The workaround is to manually import the portable profile "portable-net451+win8". This forces NuGet to treat this binaries that match this provide as a compatible framework with .NET Standard, even though they are not. Although "portable-net451+win8" is not 100% compatible with .NET Standard, it is compatible enough for the transition from PCL to .NET Standard. Imports can be removed when EF's dependencies eventually upgrade to .NET Standard.

Multiple frameworks can be added to "imports" in array syntax. Other imports may be necessary if you add additional libraries to your project.

```
{
  "frameworks": {
    "netcoreapp1.0": {
      "imports": ["dnxcore50", "portable-net451+win8"]
    }
  }
}
```

See Issue #5176.

Upgrading from EF Core 1.0 RC2 to RTM

8/27/2018 • 2 minutes to read • Edit Online

This article provides guidance for moving an application built with the RC2 packages to 1.0.0 RTM.

Package Versions

The names of the top level packages that you would typically install into an application did not change between RC2 and RTM.

You need to upgrade the installed packages to the RTM versions:

- Runtime packages (for example, `Microsoft.EntityFrameworkCore.SqlServer`) changed from `1.0.0-rc2-final` to `1.0.0`.
- The `Microsoft.EntityFrameworkCore.Tools` package changed from `1.0.0-preview1-final` to `1.0.0-preview2-final`. Note that tooling is still pre-release.

Existing migrations may need maxLength added

In RC2, the column definition in a migration looked like `table.Column<string>(nullable: true)` and the length of the column was looked up in some metadata we store in the code behind the migration. In RTM, the length is now included in the scaffolded code `table.Column<string>(maxLength: 450, nullable: true)`.

Any existing migrations that were scaffolded prior to using RTM will not have the `maxLength` argument specified. This means the maximum length supported by the database will be used (`nvarchar(max)` on SQL Server). This may be fine for some columns, but columns that are part of a key, foreign key, or index need to be updated to include a maximum length. By convention, 450 is the maximum length used for keys, foreign keys, and indexed columns. If you have explicitly configured a length in the model, then you should use that length instead.

ASP.NET Identity

This change impacts projects that use ASP.NET Identity and were created from a pre-RTM project template. The project template includes a migration used to create the database. This migration must be edited to specify a maximum length of `256` for the following columns.

- **AspNetRoles**
  - Name
  - NormalizedName
- **AspNetUsers**
  - Email
  - NormalizedEmail
  - NormalizedUserName
  - UserName

Failure to make this change will result in the following exception when the initial migration is applied to a database.

```
System.Data.SqlClient.SqlException (0x80131904): Column 'NormalizedName' in table 'AspNetRoles' is of a type
that is invalid for use as a key column in an index.
```

.NET Core: Remove "imports" in project.json

If you were targeting .NET Core with RC2, you needed to add `imports` to project.json as a temporary workaround for some of EF Core's dependencies not supporting .NET Standard. These can now be removed.

```
{
  "frameworks": {
    "netcoreapp1.0": {
      "imports": ["dnxcore50", "portable-net451+win8"]
    }
  }
}
```

> **NOTE**
>
> As of version 1.0 RTM, the .NET Core SDK no longer supports `project.json` or developing .NET Core applications using Visual Studio 2015. We recommend you migrate from project.json to csproj. If you are using Visual Studio, we recommend you upgrade to Visual Studio 2017.

UWP: Add binding redirects

Attempting to run EF commands on Universal Windows Platform (UWP) projects results in the following error:

```
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.
```

You need to manually add binding redirects to the UWP project. Create a file named `App.config` in the project root folder and add redirects to the correct assembly versions.

```
<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>
```

Upgrading applications from previous versions to EF Core 2.0

9/18/2018 • 7 minutes to read • Edit Online

We have taken the opportunity to significantly refine our existing APIs and behaviors in 2.0. There are a few improvements that can require modifying existing application code, although we believe that for the majority of applications the impact will be low, in most cases requiring just recompilation and minimal guided changes to replace obsolete APIs.

Updating an existing application to EF Core 2.0 may require:

1. Upgrading the target .NET implementation of the application to one that supports .NET Standard 2.0. See Supported .NET Implementations for more details.

2. Identify a provider for the target database which is compatible with EF Core 2.0. See EF Core 2.0 requires a 2.0 database provider below.

3. Upgrading all the EF Core packages (runtime and tooling) to 2.0. Refer to Installing EF Core for more details.

4. Make any necessary code changes to compensate for the breaking changes described in the rest of this document.

ASP.NET Core now includes EF Core

Applications targeting ASP.NET Core 2.0 can use EF Core 2.0 without additional dependencies besides third party database providers. However, applications targeting previous versions of ASP.NET Core need to upgrade to ASP.NET Core 2.0 in order to use EF Core 2.0. For more details on upgrading ASP.NET Core applications to 2.0 see the ASP.NET Core documentation on the subject.

New way of getting application services in ASP.NET Core

The recommended pattern for ASP.NET Core web applications has been updated for 2.0 in a way that broke the design-time logic EF Core used in 1.x. Previously at design-time, EF Core would try to invoke `Startup.ConfigureServices` directly in order to access the application's service provider. In ASP.NET Core 2.0, Configuration is initialized outside of the `Startup` class. Applications using EF Core typically access their connection string from Configuration, so `Startup` by itself is no longer sufficient. If you upgrade an ASP.NET Core 1.x application, you may receive the following error when using the EF Core tools.

> No parameterless constructor was found on 'ApplicationContext'. Either add a parameterless constructor to 'ApplicationContext' or add an implementation of 'IDesignTimeDbContextFactory<ApplicationContext>' in the same assembly as 'ApplicationContext'

A new design-time hook has been added in ASP.NET Core 2.0's default template. The static `Program.BuildWebHost` method enables EF Core to access the application's service provider at design time. If you are upgrading an ASP.NET Core 1.x application, you will need to update the `Program` class to resemble the following.

```
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();
    }
}
```

The adoption of this new pattern when updating applications to 2.0 is highly recommended and is required in order for product features like Entity Framework Core Migrations to work. The other common alternative is to implement *IDesignTimeDbContextFactory<TContext>*.

IDbContextFactory renamed

In order to support diverse application patterns and give users more control over how their `DbContext` is used at design time, we have, in the past, provided the `IDbContextFactory<TContext>` interface. At design-time, the EF Core tools will discover implementations of this interface in your project and use it to create `DbContext` objects.

This interface had a very general name which mislead some users to try re-using it for other `DbContext`-creating scenarios. They were caught off guard when the EF Tools then tried to use their implementation at design-time and caused commands like `Update-Database` or `dotnet ef database update` to fail.

In order to communicate the strong design-time semantics of this interface, we have renamed it to `IDesignTimeDbContextFactory<TContext>`.

For the 2.0 release the `IDbContextFactory<TContext>` still exists but is marked as obsolete.

DbContextFactoryOptions removed

Because of the ASP.NET Core 2.0 changes described above, we found that `DbContextFactoryOptions` was no longer needed on the new `IDesignTimeDbContextFactory<TContext>` interface. Here are the alternatives you should be using instead.

DBCONTEXTFACTORYOPTIONS	ALTERNATIVE
ApplicationBasePath	AppContext.BaseDirectory
ContentRootPath	Directory.GetCurrentDirectory()
EnvironmentName	Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")

Design-time working directory changed

The ASP.NET Core 2.0 changes also required the working directory used by `dotnet ef` to align with the working directory used by Visual Studio when running your application. One observable side effect of this is that SQLite filenames are now relative to the project directory and not the output directory like they used to be.

EF Core 2.0 requires a 2.0 database provider

For EF Core 2.0 we have made many simplifications and improvements in the way database providers work. This means that 1.0.x and 1.1.x providers will not work with EF Core 2.0.

The SQL Server and SQLite providers are shipped by the EF team and 2.0 versions will be available as part of the 2.0 release. The open-source third party providers for SQL Compact, PostgreSQL, and MySQL are being updated for 2.0. For all other providers, please contact the provider writer.

Logging and Diagnostics events have changed

Note: these changes should not impact most application code.

The event IDs for messages sent to an ILogger have changed in 2.0. The event IDs are now unique across EF Core code. These messages now also follow the standard pattern for structured logging used by, for example, MVC.

Logger categories have also changed. There is now a well-known set of categories accessed through DbLoggerCategory.

DiagnosticSource events now use the same event ID names as the corresponding `ILogger` messages. The event payloads are all nominal types derived from EventData.

Event IDs, payload types, and categories are documented in the CoreEventId and the RelationalEventId classes.

IDs have also moved from Microsoft.EntityFrameworkCore.Infraestructure to the new Microsoft.EntityFrameworkCore.Diagnostics namespace.

EF Core relational metadata API changes

EF Core 2.0 will now build a different IModel for each different provider being used. This is usually transparent to the application. This has facilitated a simplification of lower-level metadata APIs such that any access to *common relational metadata concepts* is always made through a call to `.Relational` instead of `.SqlServer`, `.Sqlite`, etc. For example, 1.1.x code like this:

```
var tableName = context.Model.FindEntityType(typeof(User)).SqlServer().TableName;
```

Should now be written like this:

```
var tableName = context.Model.FindEntityType(typeof(User)).Relational().TableName;
```

Instead of using methods like `ForSqlServerToTable`, extension methods are now available to write conditional code based on the current provider in use. For example:

```
modelBuilder.Entity<User>().ToTable(
    Database.IsSqlServer() ? "SqlServerName" : "OtherName");
```

Note that this change only applies to APIs/metadata that is defined for *all* relational providers. The API and metadata remains the same when it is specific to only a single provider. For example, clustered indexes are specific to SQL Sever, so `ForSqlServerIsClustered` and `.SqlServer().IsClustered()` must still be used.

Don't take control of the EF service provider

EF Core uses an internal `IServiceProvider` (a dependency injection container) for its internal implementation. Applications should allow EF Core to create and manage this provider except in special cases. Strongly consider removing any calls to `UseInternalServiceProvider`. If an application does need to call `UseInternalServiceProvider`, then please consider filing an issue so we can investigate other ways to handle your scenario.

Calling `AddEntityFramework`, `AddEntityFrameworkSqlServer`, etc. is not required by application code unless `UseInternalServiceProvider` is also called. Remove any existing calls to `AddEntityFramework` or `AddEntityFrameworkSqlServer`, etc. `AddDbContext` should still be used in the same way as before.

In-memory databases must be named

The global unnamed in-memory database has been removed and instead all in-memory databases must be named. For example:

```
optionsBuilder.UseInMemoryDatabase("MyDatabase");
```

This creates/uses a database with the name “MyDatabase”. If `UseInMemoryDatabase` is called again with the same name, then the same in-memory database will be used, allowing it to be shared by multiple context instances.

Read-only API changes

`IsReadOnlyBeforeSave`, `IsReadOnlyAfterSave`, and `IsStoreGeneratedAlways` have been obsoleted and replaced with BeforeSaveBehavior and AfterSaveBehavior. These behaviors apply to any property (not only store-generated properties) and determine how the value of the property should be used when inserting into a database row (`BeforeSaveBehavior`) or when updating an existing database row (`AfterSaveBehavior`).

Properties marked as ValueGenerated.OnAddOrUpdate (for example, for computed columns) will by default ignore any value currently set on the property. This means that a store-generated value will always be obtained regardless of whether any value has been set or modified on the tracked entity. This can be changed by setting a different `Before\AfterSaveBehavior`.

New ClientSetNull delete behavior

In previous releases, DeleteBehavior.Restrict had a behavior for entities tracked by the context that more closed matched `SetNull` semantics. In EF Core 2.0, a new `ClientSetNull` behavior has been introduced as the default for optional relationships. This behavior has `SetNull` semantics for tracked entities and `Restrict` behavior for databases created using EF Core. In our experience, these are the most expected/useful behaviors for tracked entities and the database. `DeleteBehavior.Restrict` is now honored for tracked entities when set for optional relationships.

Provider design-time packages removed

The `Microsoft.EntityFrameworkCore.Relational.Design` package has been removed. It's contents were consolidated into `Microsoft.EntityFrameworkCore.Relational` and `Microsoft.EntityFrameworkCore.Design`.

This propagates into the provider design-time packages. Those packages (`Microsoft.EntityFrameworkCore.Sqlite.Design`, `Microsoft.EntityFrameworkCore.SqlServer.Design`, etc.) were removed and their contents consolidated into the main provider packages.

To enable `Scaffold-DbContext` or `dotnet ef dbcontext scaffold` in EF Core 2.0, you only need to reference the single provider package:

```
<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" />
```

Getting Started with Entity Framework Core

11/28/2018 • 2 minutes to read • Edit Online

Installing EF Core

A summary of the steps necessary to add EF Core to your application in different platforms and popular IDEs.

Step-by-step Tutorials

These introductory tutorials require no previous knowledge of Entity Framework Core or any particular IDE. They will take you step-by-step through the creation of a simple application that queries and saves data from a database. We have provided tutorials to get you started on various operating systems and application types.

Entity Framework Core can create a model based on an existing database, or create a database for you based on your model. There are tutorials that demonstrate both of these approaches.

- .NET Core Console Apps
  - New Database
- ASP.NET Core Apps
  - New Database
  - Existing Database
  - EF Core and Razor Pages
- Universal Windows Platform (UWP) Apps
  - New Database
- .NET Framework Apps
  - New Database
  - Existing Database

> **NOTE**
>
> These tutorials and the accompanying samples have been updated to use EF Core 2.1. However, in the majority of cases it should be possible to create applications that use previous releases, with minimal modification to the instructions.

Installing Entity Framework Core

12/10/2018 • 5 minutes to read • Edit Online

Prerequisites

- EF Core is a .NET Standard 2.0 library. So EF Core requires a .NET implementation that supports .NET Standard 2.0 to run. EF Core can also be referenced by other .NET Standard 2.0 libraries.
- For example, you can use EF Core to develop apps that target .NET Core. Building .NET Core apps requires the .NET Core SDK. Optionally, you can also use a development environment like Visual Studio, Visual Studio for Mac, or Visual Studio Code. For more information, check Getting Started with .NET Core.
- You can use EF Core to develop applications that target .NET Framework 4.6.1 or later on Windows, using Visual Studio. The latest version of Visual Studio is recommended. If you want to use an older version, like Visual Studio 2015, make sure you upgrade the NuGet client to version 3.6.0 to work with .NET Standard 2.0 libraries.
- EF Core can run on other .NET implementations like Xamarin and .NET Native. But in practice those implementations have runtime limitations that may affect how well EF Core works on your app. For more information, see .NET implementations supported by EF Core.
- Finally, different database providers may require specific database engine versions, .NET implementations, or operating systems. Make sure an EF Core database provider is available that supports the right environment for your application.

Get the Entity Framework Core runtime

To add EF Core to an application, install the NuGet package for the database provider you want to use.

If you're building an ASP.NET Core application, you don't need to install the in-memory and SQL Server providers. Those providers are included in current versions of ASP.NET Core, alongside the EF Core runtime.

To install or update NuGet packages, you can use the [.NET Core command-line interface (CLI), the Visual Studio Package Manager Dialog, or the Visual Studio Package Manager Console.

.NET Core CLI

- Use the following .NET Core CLI command from the operating system's command line to install or update the EF Core SQL Server provider:

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

- You can indicate a specific version in the `dotnet add package` command, using the `-v` modifier. For example, to install EF Core 2.2.0 packages, append `-v 2.2.0` to the command.

For more information, see .NET command-line interface (CLI) tools.

Visual Studio NuGet Package Manager Dialog

- From the Visual Studio menu, select **Project > Manage NuGet Packages**
- Click on the **Browse** or the **Updates** tab
- To install or update the SQL Server provider, select the `Microsoft.EntityFrameworkCore.SqlServer` package, and confirm.

For more information, see NuGet Package Manager Dialog.

Visual Studio NuGet Package Manager Console

- From the Visual Studio menu, select **Tools > NuGet Package Manager > Package Manager Console**
- To install the SQL Server provider, run the following command in the Package Manager Console:

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

- To update the provider, use the `Update-Package` command.
- To specify a specific version, use the `-Version` modifier. For example, to install EF Core 2.2.0 packages, append `-Version 2.2.0` to the commands

For more information, see Package Manager Console.

Get the Entity Framework Core tools

You can install tools to carry out EF Core-related tasks in your project, like creating and applying database migrations, or creating an EF Core model based on an existing database.

Two sets of tools are available:

- The .NET Core command-line interface (CLI) tools can be used on Windows, Linux, or macOS. These commands begin with `dotnet ef`.
- The Package Manager Console (PMC) tools run in Visual Studio on Windows. These commands start with a verb, for example `Add-Migration`, `Update-Database`.

Although you can also use the `dotnet ef` commands from the Package Manager Console, it's recommended to use the Package Manager Console tools when you're using Visual Studio:

- They automatically work with the current project selected in the PMC in Visual Studio, without requiring manually switching directories.
- They automatically open files generated by the commands in Visual Studio after the command is completed.

Get the .NET Core CLI tools

.NET Core CLI tools require the .NET Core SDK, mentioned earlier in Prerequisites.

The `dotnet ef` commands are included in current versions of the .NET Core SDK, but to enable the commands on a specific project, you have to install the `Microsoft.EntityFrameworkCore.Design` package:

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

For ASP.NET Core apps, this package is included automatically.

> **IMPORTANT**
> Always use the version of the tools package that matches the major version of the runtime packages.

Get the Package Manager Console tools

To get the Package Manager Console tools for EF Core, install the `Microsoft.EntityFrameworkCore.Tools` package. For example, from Visual Studio:

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

For ASP.NET Core apps, this package is included automatically.

Upgrading to the latest EF Core

- Any time we release a new version of EF Core, we also release a new version of the providers that are part of the EF Core project, like Microsoft.EntityFrameworkCore.SqlServer, Microsoft.EntityFrameworkCore.Sqlite, and Microsoft.EntityFrameworkCore.InMemory. You can just upgrade to the new version of the provider to get all the improvements.
- EF Core, together with the SQL Server and the in-memory providers are included in current versions of ASP.NET Core. To upgrade an existing ASP.NET Core application to a newer version of EF Core, always upgrade the version of ASP.NET Core.
- If you need to update an application that is using a third-party database provider, always check for an update of the provider that is compatible with the version of EF Core you want to use. For example, database providers for previous versions are not compatible with version 2.0 of the EF Core runtime.
- Third-party providers for EF Core usually don't release patch versions alongside the EF Core runtime. To upgrade an application that uses a third-party provider to a patch version of EF Core, you may need to add a direct reference to individual EF Core runtime components, such as Microsoft.EntityFrameworkCore, and Microsoft.EntityFrameworkCore.Relational.
- If you're upgrading an existing application to the latest version of EF Core, some references to older EF Core packages may need to be removed manually:
  - Database provider design-time packages such as `Microsoft.EntityFrameworkCore.SqlServer.Design` are no longer required or supported from EF Core 2.0 and later, but aren't automatically removed when upgrading the other packages.
  - The .NET CLI tools are included in the .NET SDK since version 2.1, so the reference to that package can be removed from the project file:

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

- Applications that target .NET Framework may need changes to work with .NET Standard 2.0 libraries:
  - Edit the project file and make sure the following entry appears in the initial property group:

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

  - For test projects, also make sure the following entry is present:

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

Getting Started with EF Core on .NET Core

8/27/2018 • 2 minutes to read • Edit Online

These 101 tutorials require no previous knowledge of Entity Framework Core or Visual Studio. They will take you step-by-step through creating a simple .NET Core Console Application that queries and saves data from a database. The tutorials can be completed on any platform supported by .NET Core (Windows, OSX, Linux, etc.).

You can find the .NET Core documentation at docs.microsoft.com/dotnet/articles/core.

Getting Started with EF Core on .NET Core Console App with a New database

10/25/2018 • 3 minutes to read • Edit Online

In this tutorial, you create a .NET Core console app that performs data access against a SQLite database using Entity Framework Core. You use migrations to create the database from the model. See ASP.NET Core - New database for a Visual Studio version using ASP.NET Core MVC.

View this article's sample on GitHub.

Prerequisites

- The .NET Core 2.1 SDK

Create a new project

- Create a new console project:

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

Change the current directory

In subsequent steps, we need to issue `dotnet` commands against the application.

- We change the current directory to the application's directory like this:

```
cd ConsoleApp.SQLite/
```

Install Entity Framework Core

To use EF Core, install the package for the database provider(s) you want to target. This walkthrough uses SQLite. For a list of available providers see Database Providers.

- Install Microsoft.EntityFrameworkCore.Sqlite and Microsoft.EntityFrameworkCore.Design

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

- Run `dotnet restore` to install the new packages.

Create the model

Define a context and entity classes that make up your model.

- Create a new *Model.cs* file with the following contents.

```
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; }
    }
}
```

Tip: In a real application, you put each class in a separate file and put the connection string in a configuration file or environment variable. To keep the tutorial simple, everything is contained in one file.

Create the database

Once you have a model, you use migrations to create a database.

- Run `dotnet ef migrations add InitialCreate` to scaffold a migration and create the initial set of tables for the model.
- Run `dotnet ef database update` to apply the new migration to the database. This command creates the database before applying migrations.

The *blogging.db** SQLite DB is in the project directory.

Use the model

- Open *Program.cs* and replace the contents with the following code:

```
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);
                }
            }
        }
    }
}
```

- Test the app from the console. See the Visual Studio note to run the app from Visual Studio.

  `dotnet run`

  One blog is saved to the database and the details of all blogs are displayed in the console.

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

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

Changing the model:

- If you make changes to the model, you can use the `dotnet ef migrations add` command to scaffold a new migration. Once you have checked the scaffolded code (and made any required changes), you can use the `dotnet ef database update` command to apply the schema changes to the database.
- EF Core uses a `__EFMigrationsHistory` table in the database to keep track of which migrations have already been applied to the database.
- The SQLite database engine doesn't support certain schema changes that are supported by most other relational databases. For example, the `DropColumn` operation is not supported. EF Core Migrations will generate code for these operations. But if you try to apply them to a database or generate a script, EF Core throws exceptions. See SQLite Limitations. For new development, consider dropping the database and creating a new one rather than using migrations when the model changes.

Run from Visual Studio

To run this sample from Visual Studio, you must set the working directory manually to be the root of the project. If you don't set the working directory, the following `Microsoft.Data.Sqlite.SqliteException` is thrown: `SQLite Error 1: 'no such table: Blogs'`.

To set the working directory:

- In **Solution Explorer**, right click the project and then select **Properties**.
- Select the **Debug** tab in the left pane.

- Set **Working directory** to the project directory.
- Save the changes.

Additional Resources

- Tutorial: Get started with EF Core on ASP.NET Core with a new database using SQLite
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Tutorial: Razor Pages with Entity Framework Core in ASP.NET Core

Getting Started with EF Core on ASP.NET Core

8/27/2018 • 2 minutes to read • Edit Online

These 101 tutorials require no previous knowledge of Entity Framework Core or Visual Studio. They will take you step-by-step through creating a simple ASP.NET Core application that queries and saves data from a database. You can choose a tutorial that creates a model based on an existing database, or creates a database for you based on your model.

You can find the ASP.NET Core documentation at Introduction to ASP.NET Core.

> **NOTE**
>
> These tutorials and the accompanying samples have been updated to use EF Core 2.0 (with the exception of the UWP tutorial, that still uses EF Core 1.1). However, in the majority of cases it should be possible to create applications that use previous releases, with minimal modification to the instructions.

Getting Started with EF Core on ASP.NET Core with a New database

1/10/2019 • 6 minutes to read • Edit Online

In this tutorial, you build an ASP.NET Core MVC application that performs basic data access using Entity Framework Core. The tutorial uses migrations to create the database from the data model.

You can follow the tutorial by using Visual Studio 2017 on Windows, or by using the .NET Core CLI on Windows, macOS, or Linux.

View this article's sample on GitHub:

- Visual Studio 2017 with SQL Server
- .NET Core CLI with SQLite.

Prerequisites

Install the following software:

- Visual Studio
- .NET Core CLI

- Visual Studio 2017 version 15.7 or later with these workloads:
  - **ASP.NET and web development** (under **Web & Cloud**)
  - **.NET Core cross-platform development** (under **Other Toolsets**)
- .NET Core 2.1 SDK.

Create a new project

- Visual Studio
- .NET Core CLI

- Open Visual Studio 2017
- **File > New > Project**
- From the left menu, select **Installed > Visual C# > .NET Core**.
- Select **ASP.NET Core Web Application**.
- Enter **EFGetStarted.AspNetCore.NewDb** for the name and click **OK**.
- In the **New ASP.NET Core Web Application** dialog:
  - Make sure that **.NET Core** and **ASP.NET Core 2.1** are selected in the drop-down lists
  - Select the **Web Application (Model-View-Controller)** project template
  - Make sure that **Authentication** is set to **No Authentication**
  - Click **OK**

Warning: If you use **Individual User Accounts** instead of **None** for **Authentication** then an Entity Framework Core model will be added to the project in `Models\IdentityModel.cs`. Using the techniques you learn in this tutorial, you can choose to add a second model, or extend this existing model to contain your entity classes.

Install Entity Framework Core

To install EF Core, you install the package for the EF Core database provider(s) you want to target. For a list of available providers, see Database Providers.

- Visual Studio
- .NET Core CLI

For this tutorial, you don't have to install a provider package because the tutorial uses SQL Server. The SQL Server provider package is included in the Microsoft.AspnetCore.App metapackage.

Create the model

Define a context class and entity classes that make up the model.

- Visual Studio
- .NET Core CLI

- Right-click on the **Models** folder and select **Add > Class**.
- Enter **Model.cs** as the name and click **OK**.
- Replace the contents of the file with the following code:

```
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; }
    }
}
```

A production app would typically put each class in a separate file. For the sake of simplicity, this tutorial puts these classes in one file.

Register the context with dependency injection

Services (such as `BloggingContext`) are registered with dependency injection during application startup.

Components that require these services (such as MVC controllers) are provided these services via constructor parameters or properties.

To make `BloggingContext` available to MVC controllers, register it as a service.

- Visual Studio
- .NET Core CLI

- In **Startup.cs** add the following `using` statements:

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

- Add the following highlighted code to the `ConfigureServices` method:

```
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<BloggingContext>
        (options => options.UseSqlServer(connection));
    // BloggingContext requires
    // using EFGetStarted.AspNetCore.NewDb.Models;
    // UseSqlServer requires
    // using Microsoft.EntityFrameworkCore;
}
```

A production app would typically put the connection string in a configuration file or environment variable. For the sake of simplicity, this tutorial defines it in code. See Connection Strings for more information.

Create the database

The following steps use migrations to create a database.

- Visual Studio
- .NET Core CLI

- **Tools > NuGet Package Manager > Package Manager Console**
- Run the following commands:

```
Add-Migration InitialCreate
Update-Database
```

If you get an error stating `The term 'add-migration' is not recognized as the name of a cmdlet`, close and reopen Visual Studio.

The `Add-Migration` command scaffolds a migration to create the initial set of tables for the model. The

`Update-Database` command creates the database and applies the new migration to it.

Create a controller

Scaffold a controller and views for the `Blog` entity.

- Visual Studio
- .NET Core CLI

- Right-click on the **Controllers** folder in **Solution Explorer** and select **Add > Controller**.
- Select **MVC Controller with views, using Entity Framework** and click **Add**.
- Set **Model class** to **Blog** and **Data context class** to **BloggingContext**.
- Click **Add**.

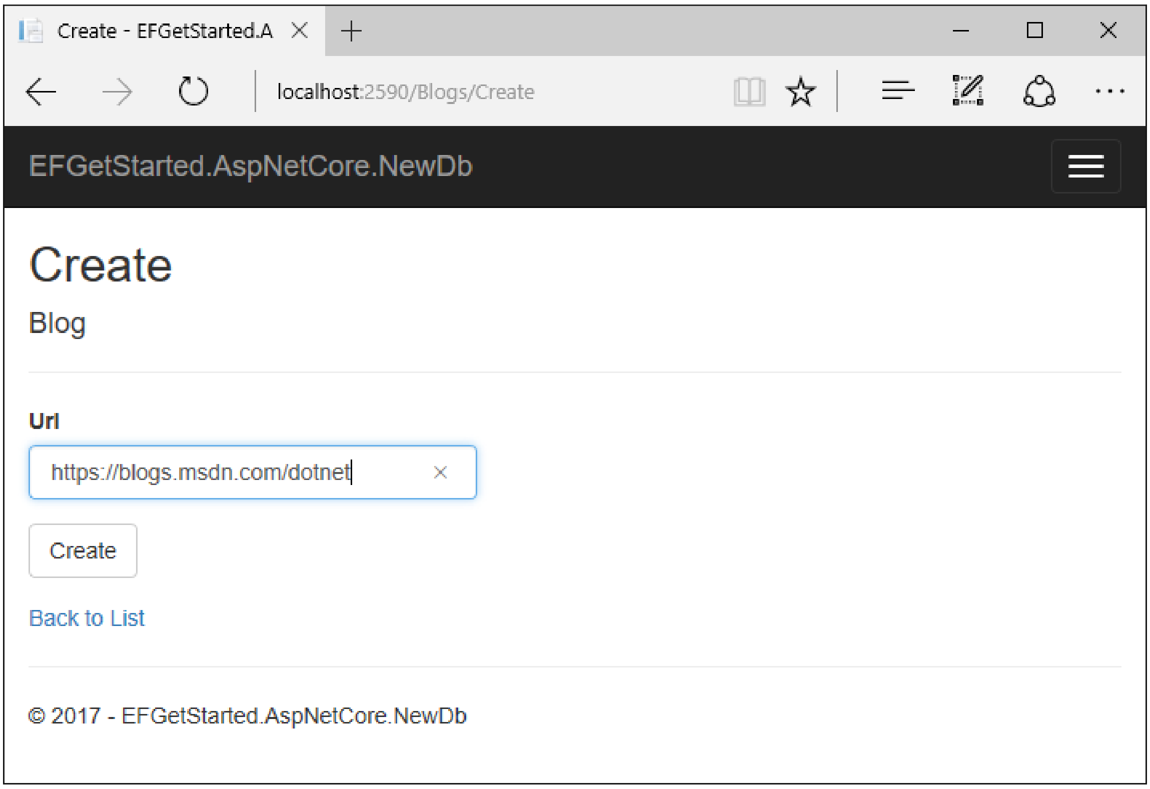
The scaffolding engine creates the following files:

- A controller (*Controllers/BlogsController.cs*)
- Razor views for Create, Delete, Details, Edit, and Index pages (*Views/Blogs/*.cshtml*)

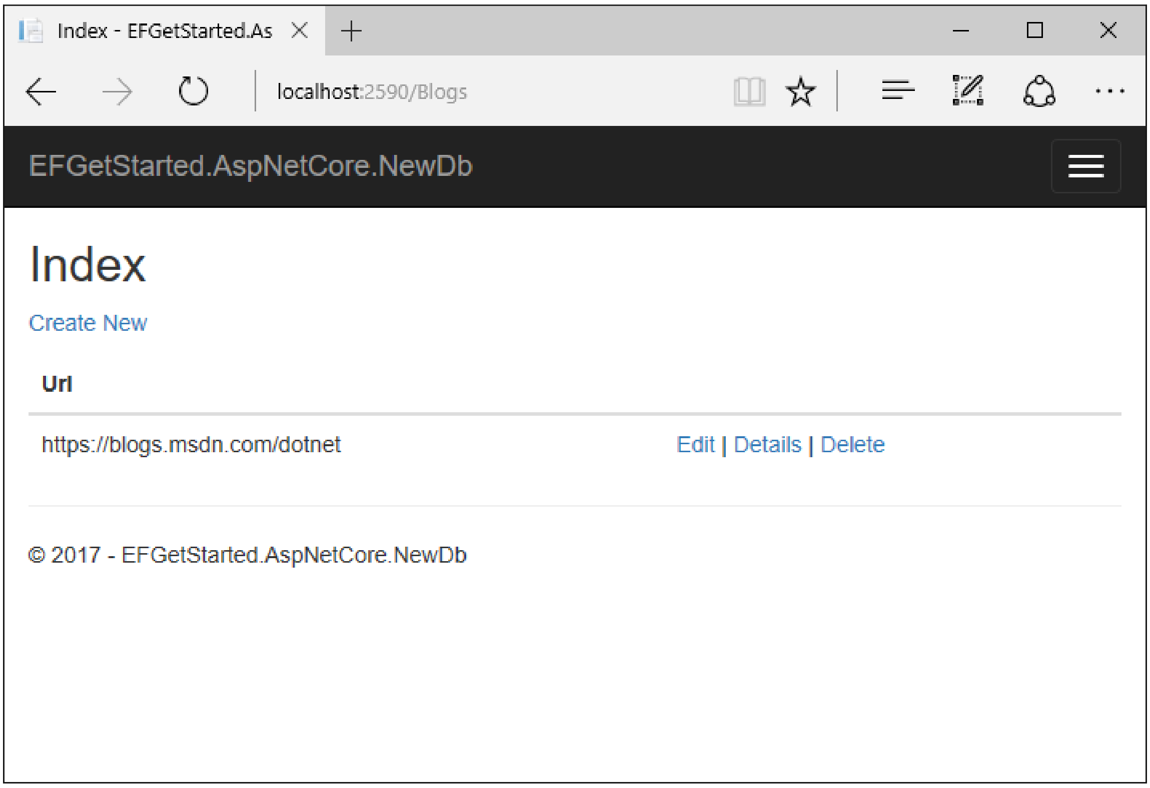
Run the application

- Visual Studio
- .NET Core CLI

- **Debug** > **Start Without Debugging**
- Navigate to `/Blogs`
- Use the **Create New** link to create some blog entries.



- Test the **Details**, **Edit**, and **Delete** links.



Additional Resources

- Tutorial: Get started with EF Core on .NET Core with a new database using SQLite
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Tutorial: Razor Pages with Entity Framework Core in ASP.NET Core

Getting Started with EF Core on ASP.NET Core with an Existing Database

1/6/2019 • 5 minutes to read • Edit Online

In this tutorial, you build an ASP.NET Core MVC application that performs basic data access using Entity Framework Core. You reverse engineer an existing database to create an Entity Framework model.

View this article's sample on GitHub.

Prerequisites

Install the following software:

- Visual Studio 2017 15.7 with these workloads:
  - **ASP.NET and web development** (under **Web & Cloud**)
  - **.NET Core cross-platform development** (under **Other Toolsets**)
- .NET Core 2.1 SDK.

Create Blogging database

This tutorial uses a **Blogging** database on your LocalDb instance as the existing database. If you have already created the **Blogging** database as part of another tutorial, skip these steps.

- Open Visual Studio
- **Tools -> Connect to Database...**
- Select **Microsoft SQL Server** and click **Continue**
- Enter **(localdb)\mssqllocaldb** as the **Server Name**
- Enter **master** as the **Database Name** and click **OK**
- The master database is now displayed under **Data Connections** in **Server Explorer**
- Right-click on the database in **Server Explorer** and select **New Query**
- Copy the script listed below into the query editor
- Right-click on the query editor and select **Execute**

```
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
```

Create a new project

- Open Visual Studio 2017
- **File > New > Project...**
- From the left menu select **Installed > Visual C# > Web**
- Select the **ASP.NET Core Web Application** project template
- Enter **EFGetStarted.AspNetCore.ExistingDb** as the name (it has to match exactly the namespace later used in the code) and click **OK**
- Wait for the **New ASP.NET Core Web Application** dialog to appear
- Make sure that the target framework dropdown is set to **.NET Core**, and the version dropdown is set to **ASP.NET Core 2.1**
- Select the **Web Application (Model-View-Controller)** template
- Ensure that **Authentication** is set to **No Authentication**
- Click **OK**

Install Entity Framework Core

To install EF Core, you install the package for the EF Core database provider(s) you want to target. For a list of available providers see Database Providers.

For this tutorial, you don't have to install a provider package because the tutorial uses SQL Server. The SQL Server provider package is included in the Microsoft.AspnetCore.App metapackage.

Reverse engineer your model

Now it's time to create the EF model based on your existing database.

- **Tools –> NuGet Package Manager –> Package Manager Console**

- Run the following command to create a model from the existing database:

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

If you receive an error stating `The term 'Scaffold-DbContext' is not recognized as the name of a cmdlet`, then close and reopen Visual Studio.

> **TIP**
>
> You can specify which tables you want to generate entities for by adding the `-Tables` argument to the command above. For example, `-Tables Blog,Post`.

The reverse engineer process created entity classes (`Blog.cs` & `Post.cs`) and a derived context (`BloggingContext.cs`) based on the schema of the existing database.

The entity classes are simple C# objects that represent the data you will be querying and saving. Here are the `Blog` and `Post` entity classes:

```
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**
>
> To enable lazy loading, you can make navigation properties `virtual` (Blog.Post and Post.Blog).

The context represents a session with the database and allows you to query and save instances of the entity classes.

```
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);
       });
   }
}
```

Register your context with dependency injection

The concept of dependency injection is central to ASP.NET Core. Services (such as `BloggingContext`) are registered with dependency injection during application startup. Components that require these services (such as your MVC controllers) are then provided these services via constructor parameters or properties. For more information on dependency injection see the Dependency Injection article on the ASP.NET site.

Register and configure your context in Startup.cs

To make `BloggingContext` available to MVC controllers, register it as a service.

- Open **Startup.cs**
- Add the following `using` statements at the start of the file

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

Now you can use the `AddDbContext(...)` method to register it as a service.

- Locate the `ConfigureServices(...)` method
- Add the following highlighted code to register the context as a service

```
// 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

In a real application you would typically put the connection string in a configuration file or environment variable. For the sake of simplicity, this tutorial has you define it in code. For more information, see Connection Strings.

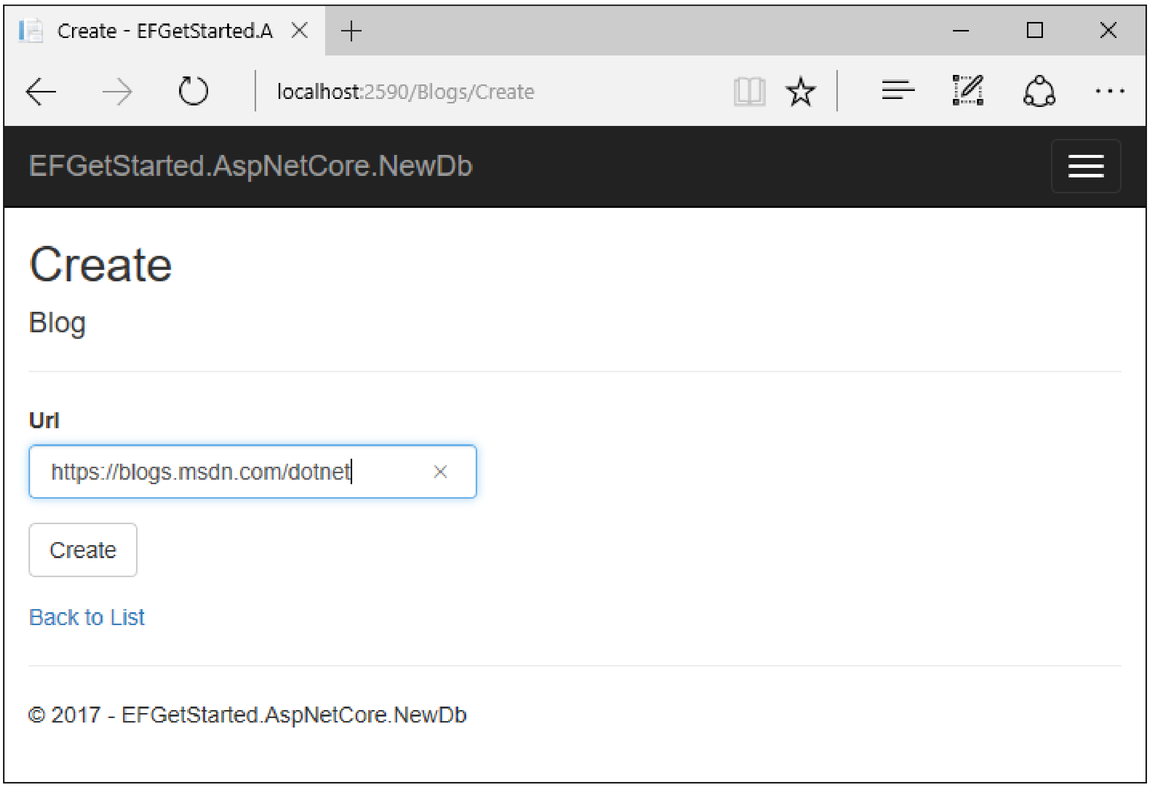
Create a controller and views

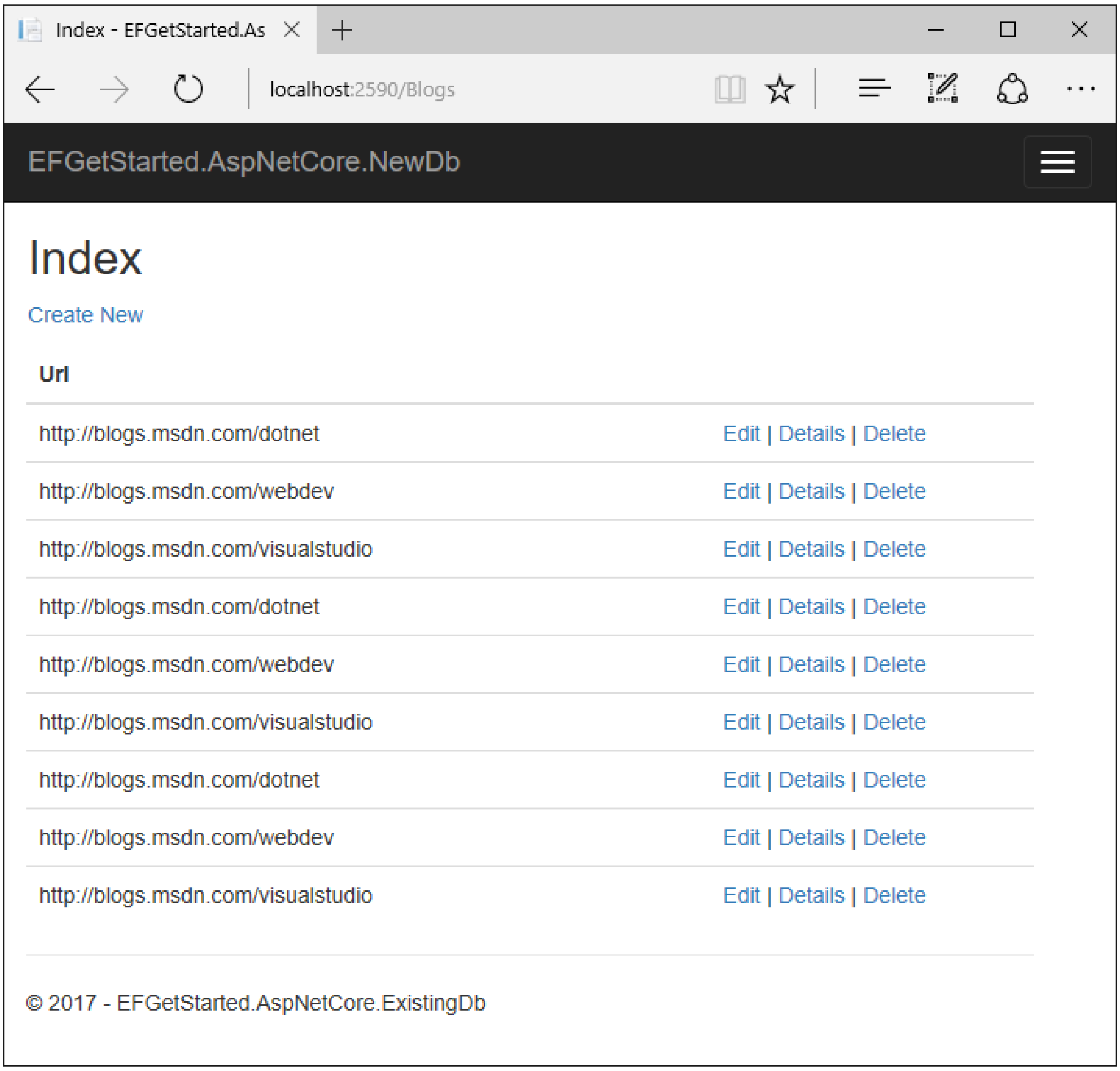
- Right-click on the **Controllers** folder in **Solution Explorer** and select **Add -> Controller...**
- Select **MVC Controller with views, using Entity Framework** and click **Ok**
- Set **Model class** to **Blog** and **Data context class** to **BloggingContext**
- Click **Add**

Run the application

You can now run the application to see it in action.

- **Debug -> Start Without Debugging**
- The application builds and opens in a web browser
- Navigate to `/Blogs`
- Click **Create New**
- Enter a **Url** for the new blog and click **Create**





Next steps

For more information about how to scaffold a context and entity classes, see the following articles:

- Reverse Engineering
- Entity Framework Core tools reference - .NET CLI
- Entity Framework Core tools reference - Package Manager Console

Getting Started with EF Core on Universal Windows Platform (UWP)

9/28/2018 • 2 minutes to read • Edit Online

These 101-level tutorials require no previous knowledge of Entity Framework Core or Visual Studio. They will take you step-by-step through creating a simple Universal Window Platform (UWP) application that queries and saves data from a database using EF Core.

More details about developing UWP applications can be found in the UWP documentation.

Getting Started with EF Core on Universal Windows Platform (UWP) with a New Database

10/25/2018 • 7 minutes to read • Edit Online

In this tutorial, you build a Universal Windows Platform (UWP) application that performs basic data access against a local SQLite database using Entity Framework Core.

View this article's sample on GitHub.

Prerequisites

- Windows 10 Fall Creators Update (10.0; Build 16299) or later.
- Visual Studio 2017 version 15.7 or later with the **Universal Windows Platform Development** workload.
- .NET Core 2.1 SDK or later or later.

> **IMPORTANT**
>
> This tutorial uses Entity Framework Core migrations commands to create and update the schema of the database. These commands don't work directly with UWP projects. For this reason, the application's data model is placed in a shared library project, and a separate .NET Core console application is used to run the commands.

Create a library project to hold the data model

- Open Visual Studio
- **File > New > Project**
- From the left menu select **Installed > Visual C# > .NET Standard**.
- Select the **Class Library (.NET Standard)** template.
- Name the project *Blogging.Model*.
- Name the solution *Blogging*.
- Click **OK**.

Install Entity Framework Core runtime in the data model project

To use EF Core, install the package for the database provider(s) you want to target. This tutorial uses SQLite. For a list of available providers see Database Providers.

- **Tools > NuGet Package Manager > Package Manager Console**.
- Make sure that the library project *Blogging.Model* is selected as the **Default Project** in the Package Manager Console.
- Run `Install-Package Microsoft.EntityFrameworkCore.Sqlite`

Create the data model

Now it's time to define the *DbContext* and entity classes that make up the model.

- Delete *Class1.cs*.
- Create *Model.cs* with the following code:

```
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; }
    }
}
```

Create a new console project to run migrations commands

- In **Solution Explorer**, right-click the solution, and then choose **Add > New Project**.
- From the left menu select **Installed > Visual C# > .NET Core**.
- Select the **Console App (.NET Core)** project template.
- Name the project *Blogging.Migrations.Startup*, and click **OK**.
- Add a project reference from the *Blogging.Migrations.Startup* project to the *Blogging.Model* project.

Install Entity Framework Core tools in the migrations startup project

To enable the EF Core migration commands in the Package Manager Console, install the EF Core tools package in the console application.

- **Tools > NuGet Package Manager > Package Manager Console**
- Run `Install-Package Microsoft.EntityFrameworkCore.Tools -ProjectName Blogging.Migrations.Startup`

Create the initial migration

Create the initial migration, specifying the console application as the startup project.

- Run `Add-Migration InitialCreate -StartupProject Blogging.Migrations.Startup`

This command scaffolds a migration that creates the initial set of database tables for your data model.

Create the UWP project

- In **Solution Explorer**, right-click the solution, and then choose **Add > New Project**.
- From the left menu select **Installed > Visual C# > Windows Universal**.
- Select the **Blank App (Universal Windows)** project template.
- Name the project *Blogging.UWP*, and click **OK**

> **IMPORTANT**
>
> Set the target and minimum versions to at least **Windows 10 Fall Creators Update (10.0; build 16299.0)**. Previous versions of Windows 10 do not support .NET Standard 2.0, which is required by Entity Framework Core.

Add code to create the database on application startup

Since you want the database to be created on the device that the app runs on, add code to apply any pending migrations to the local database on application startup. The first time that the app runs, this will take care of creating the local database.

- Add a project reference from the *Blogging.UWP* project to the *Blogging.Model* project.
- Open *App.xaml.cs*.
- Add the highlighted code to apply any pending migrations.

```
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()
        {
            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**
>
> If you change your model, use the `Add-Migration` command to scaffold a new migration to apply the corresponding changes to the database. Any pending migrations will be applied to the local database on each device when the application starts.
>
> EF Core uses a `__EFMigrationsHistory` table in the database to keep track of which migrations have already been applied to the database.

Use the data model

You can now use EF Core to perform data access.

- Open *MainPage.xaml*.
- Add the page load handler and UI content highlighted below

```
<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>
```

Now add code to wire up the UI with the database

- Open *MainPage.xaml.cs*.
- Add the highlighted code from the following listing:

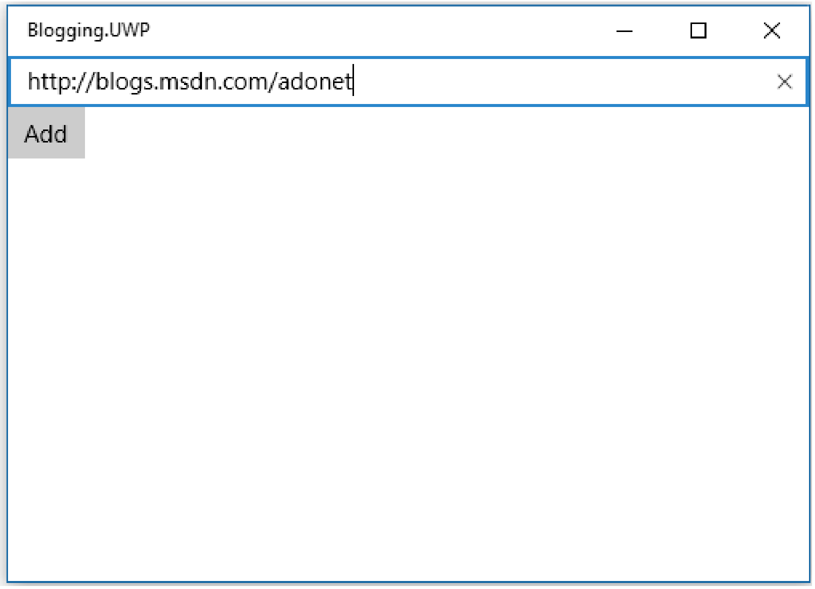
```
using Blogging.Model;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
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();
            }
        }
    }
}
```

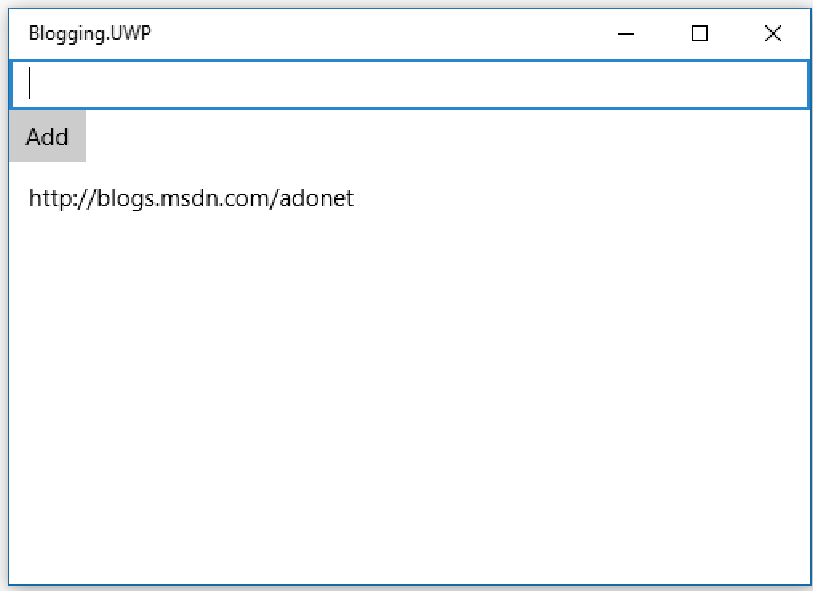
You can now run the application to see it in action.

- In **Solution Explorer**, right-click the *Blogging.UWP* project and then select **Deploy**.
- Set *Blogging.UWP* as the startup project.
- **Debug > Start Without Debugging**

  The app builds and runs.

- Enter a URL and click the **Add** button





Tada! You now have a simple UWP app running Entity Framework Core.

Next steps

For compatibility and performance information that you should know when using EF Core with UWP, see .NET implementations supported by EF Core.

Check out other articles in this documentation to learn more about Entity Framework Core features.

Getting Started with EF Core on .NET Framework

8/27/2018 • 2 minutes to read • Edit Online

These 101-level tutorials require no previous knowledge of Entity Framework Core or Visual Studio. They will take you step-by-step through creating a simple .NET Framework Console Application that queries and saves data from a database. You can choose a tutorial that creates a model based on an existing database, or creates a database for you based on your model.

You can use the techniques learned in these tutorials in any application that targets the .NET Framework, including WPF and WinForms.

> **NOTE**
>
> These tutorials and the accompanying samples have been updated to use EF Core 2.1. However, in the majority of cases it should be possible to create applications that use previous releases, with minimal modification to the instructions.

Getting started with EF Core on .NET Framework with a New Database

8/27/2018 • 3 minutes to read • Edit Online

In this tutorial, you build a console application that performs basic data access against a Microsoft SQL Server database using Entity Framework. You use migrations to create the database from a model.

View this article's sample on GitHub.

Prerequisites

- Visual Studio 2017 version 15.7 or later

Create a new project

- Open Visual Studio 2017
- **File > New > Project...**
- From the left menu select **Installed > Visual C# > Windows Desktop**
- Select the **Console App (.NET Framework)** project template
- Make sure that the project targets **.NET Framework 4.6.1** or later
- Name the project *ConsoleApp.NewDb* and click **OK**

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This tutorial uses SQL Server. For a list of available providers see Database Providers.

- Tools > NuGet Package Manager > Package Manager Console
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

Later in this tutorial you use some Entity Framework Tools to maintain the database. So install the tools package as well.

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

Create the model

Now it's time to define a context and entity classes that make up the model.

- **Project > Add Class...**
- Enter *Model.cs* as the name and click **OK**
- Replace the contents of the file with the following code

```
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

In a real application you would put each class in a separate file and put the connection string in a configuration file or environment variable. For the sake of simplicity, everything is in a single code file for this tutorial.

Create the database

Now that you have a model, you can use migrations to create a database.

- **Tools > NuGet Package Manager > Package Manager Console**
- Run `Add-Migration InitialCreate` to scaffold a migration to create the initial set of tables for the model.
- Run `Update-Database` to apply the new migration to the database. Because the database doesn't exist yet, it will be created before the migration is applied.

TIP

If you make changes to the model, you can use the `Add-Migration` command to scaffold a new migration to make the corresponding schema changes to the database. Once you have checked the scaffolded code (and made any required changes), you can use the `Update-Database` command to apply the changes to the database.

EF uses a `__EFMigrationsHistory` table in the database to keep track of which migrations have already been applied to the database.

Use the model

You can now use the model to perform data access.

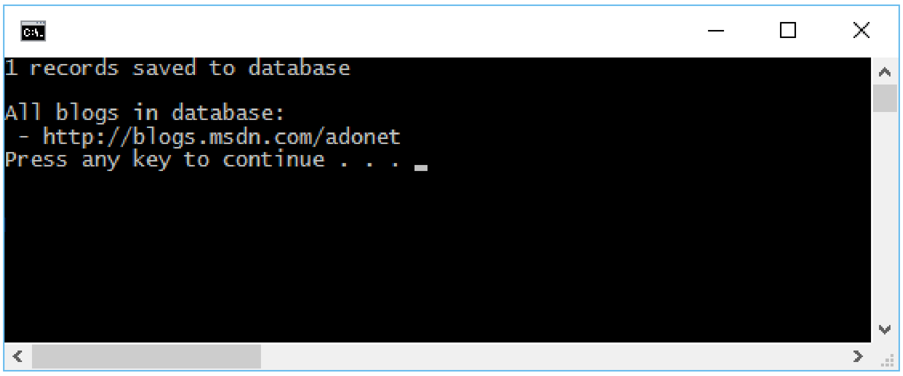
- Open *Program.cs*

- Replace the contents of the file with the following code

```
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);
                }
            }
        }
    }
}
```

- **Debug > Start Without Debugging**

  You see that one blog is saved to the database and then the details of all blogs are printed to the console.



Additional Resources

- EF Core on .NET Framework with an existing database
- EF Core on .NET Core with a new database - SQLite - a cross-platform console EF tutorial.

Getting started with EF Core on .NET Framework with an Existing Database

11/15/2018 • 4 minutes to read • Edit Online

In this tutorial, you build a console application that performs basic data access against a Microsoft SQL Server database using Entity Framework. You create an Entity Framework model by reverse engineering an existing database.

View this article's sample on GitHub.

Prerequisites

- Visual Studio 2017 version 15.7 or later

Create Blogging database

This tutorial uses a **Blogging** database on the LocalDb instance as the existing database. If you have already created the **Blogging** database as part of another tutorial, skip these steps.

- Open Visual Studio
- **Tools > Connect to Database...**
- Select **Microsoft SQL Server** and click **Continue**
- Enter **(localdb)\mssqllocaldb** as the **Server Name**
- Enter **master** as the **Database Name** and click **OK**
- The master database is now displayed under **Data Connections** in **Server Explorer**
- Right-click on the database in **Server Explorer** and select **New Query**
- Copy the script listed below into the query editor
- Right-click on the query editor and select **Execute**

```sql
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
```

Create a new project

- Open Visual Studio 2017
- **File > New > Project...**
- From the left menu select **Installed > Visual C# > Windows Desktop**
- Select the **Console App (.NET Framework)** project template
- Make sure that the project targets **.NET Framework 4.6.1** or later
- Name the project *ConsoleApp.ExistingDb* and click **OK**

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This tutorial uses SQL Server. For a list of available providers see Database Providers.

- **Tools > NuGet Package Manager > Package Manager Console**
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

In the next step, you use some Entity Framework Tools to reverse engineer the database. So install the tools package as well.

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

Reverse engineer the model

Now it's time to create the EF model based on an existing database.

- **Tools –> NuGet Package Manager –> Package Manager Console**

- Run the following command to create a model from the existing database

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

> **TIP**
>
> You can specify the tables to generate entities for by adding the `-Tables` argument to the command above. For example, `-Tables Blog,Post`.

The reverse engineer process created entity classes (`Blog` and `Post`) and a derived context (`BloggingContext`) based on the schema of the existing database.

The entity classes are simple C# objects that represent the data you will be querying and saving. Here are the `Blog` and `Post` entity classes:

```
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**
>
> To enable lazy loading, you can make navigation properties `virtual` (Blog.Post and Post.Blog).

The context represents a session with the database. It has methods that you can use to query and save instances of the entity classes.

```
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);
            });
        }
    }
}
```

Use the model

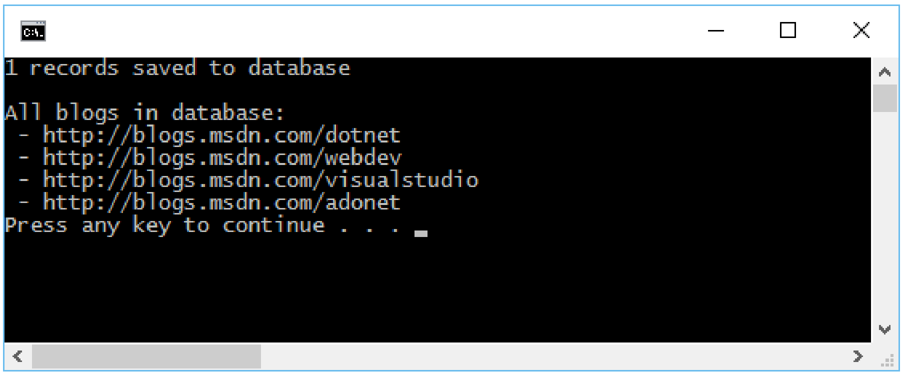
You can now use the model to perform data access.

- Open *Program.cs*
- Replace the contents of the file with the following code

```
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);
                }
            }
        }
    }
}
```

- Debug > Start Without Debugging

  You see that one blog is saved to the database and then the details of all blogs are printed to the console.



Next steps

For more information about how to scaffold a context and entity classes, see the following articles:

- Reverse Engineering
- Entity Framework Core tools reference - .NET CLI
- Entity Framework Core tools reference - Package Manager Console

Connection Strings

10/27/2018 • 2 minutes to read • Edit Online

Most database providers require some form of connection string to connect to the database. Sometimes this connection string contains sensitive information that needs to be protected. You may also need to change the connection string as you move your application between environments, such as development, testing, and production.

.NET Framework Applications

.NET Framework applications, such as WinForms, WPF, Console, and ASP.NET 4, have a tried and tested connection string pattern. The connection string should be added to your applications App.config file (Web.config if you are using ASP.NET). If your connection string contains sensitive information, such as username and password, you can protect the contents of the configuration file using Protected Configuration.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <connectionStrings>
    <add name="BloggingDatabase"
         connectionString="Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;" />
  </connectionStrings>
</configuration>
```

> **TIP**
> The `providerName` setting is not required on EF Core connection strings stored in App.config because the database provider is configured via code.

You can then read the connection string using the `ConfigurationManager` API in your context's `OnConfiguring` method. You may need to add a reference to the `System.Configuration` framework assembly to be able to use this 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);
    }
}
```

Universal Windows Platform (UWP)

Connection strings in a UWP application are typically a SQLite connection that just specifies a local filename. They typically do not contain sensitive information, and do not need to be changed as an application is deployed. As such, these connection strings are usually fine to be left in code, as shown below. If you wish to move them out of code then UWP supports the concept of settings, see the App Settings section of the UWP documentation for details.

```
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

In ASP.NET Core the configuration system is very flexible, and the connection string could be stored in `appsettings.json`, an environment variable, the user secret store, or another configuration source. See the Configuration section of the ASP.NET Core documentation for more details. The following example shows the connection string stored in `appsettings.json`.

```
{
  "ConnectionStrings": {
    "BloggingDatabase": "Server=
(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
  },
}
```

The context is typically configured in `Startup.cs` with the connection string being read from configuration. Note the `GetConnectionString()` method looks for a configuration value whose key is `ConnectionStrings:<connection string name>`. You need to import the Microsoft.Extensions.Configuration namespace to to use this extension method.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

Logging

10/25/2018 • 2 minutes to read • Edit Online

TIP

You can view this article's sample on GitHub.

ASP.NET Core applications

EF Core integrates automatically with the logging mechanisms of ASP.NET Core whenever `AddDbContext` or `AddDbContextPool` is used. Therefore, when using ASP.NET Core, logging should be configured as described in the ASP.NET Core documentation.

Other applications

EF Core logging currently requires an ILoggerFactory which is itself configured with one or more ILoggerProvider. Common providers are shipped in the following packages:

- Microsoft.Extensions.Logging.Console: A simple console logger.
- Microsoft.Extensions.Logging.AzureAppServices: Supports Azure App Services 'Diagnostics logs' and 'Log stream' features.
- Microsoft.Extensions.Logging.Debug: Logs to a debugger monitor using System.Diagnostics.Debug.WriteLine().
- Microsoft.Extensions.Logging.EventLog: Logs to Windows Event Log.
- Microsoft.Extensions.Logging.EventSource: Supports EventSource/EventListener.
- Microsoft.Extensions.Logging.TraceSource: Logs to a trace listener using System.Diagnostics.TraceSource.TraceEvent().

After installing the appropriate package(s), the application should create a singleton/global instance of a LoggerFactory. For example, using the console logger:

```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[] {new ConsoleLoggerProvider((_, __) => true, true)});
```

This singleton/global instance should then be registered with EF Core on the `DbContextOptionsBuilder`. For example:

```
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

It is very important that applications do not create a new ILoggerFactory instance for each context instance. Doing so will result in a memory leak and poor performance.

Filtering what is logged

The easiest way to filter what is logged is to configure it when registering the ILoggerProvider. For example:

```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[]
    {
        new ConsoleLoggerProvider((category, level)
            => category == DbLoggerCategory.Database.Command.Name
               && level == LogLevel.Information, true)
    });
```

In this example, the log is filtered to return only messages:

- in the 'Microsoft.EntityFrameworkCore.Database.Command' category
- at the 'Information' level

For EF Core, logger categories are defined in the `DbLoggerCategory` class to make it easy to find categories, but these resolve to simple strings.

More details on the underlying logging infrastructure can be found in the ASP.NET Core logging documentation.

Connection Resiliency

10/25/2018 • 5 minutes to read • Edit Online

Connection resiliency automatically retries failed database commands. The feature can be used with any database by supplying an "execution strategy", which encapsulates the logic necessary to detect failures and retry commands. EF Core providers can supply execution strategies tailored to their specific database failure conditions and optimal retry policies.

As an example, the SQL Server provider includes an execution strategy that is specifically tailored to SQL Server (including SQL Azure). It is aware of the exception types that can be retried and has sensible defaults for maximum retries, delay between retries, etc.

An execution strategy is specified when configuring the options for your context. This is typically in the `OnConfiguring` method of your derived context, or in `Startup.cs` for an ASP.NET Core application.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=
(localdb)\mssqllocaldb;Database=EFMiscellanous.ConnectionResiliency;Trusted_Connection=True;ConnectRetryCount=0
",
            options => options.EnableRetryOnFailure());
}
```

Custom execution strategy

There is a mechanism to register a custom execution strategy of your own if you wish to change any of the defaults.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseMyProvider(
            "<connection string>",
            options => options.ExecutionStrategy(...));
}
```

Execution strategies and transactions

An execution strategy that automatically retries on failures needs to be able to play back each operation in a retry block that fails. When retries are enabled, each operation you perform via EF Core becomes its own retriable operation. That is, each query and each call to `SaveChanges()` will be retried as a unit if a transient failure occurs.

However, if your code initiates a transaction using `BeginTransaction()` you are defining your own group of operations that need to be treated as a unit, and everything inside the transaction would need to be played back shall a failure occur. You will receive an exception like the following if you attempt to do this when using an execution strategy:

> InvalidOperationException: The configured execution strategy 'SqlServerRetryingExecutionStrategy' does not support user initiated transactions. Use the execution strategy returned by 'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in the transaction as a retriable unit.

The solution is to manually invoke the execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again.

```
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();
            }
        }
    });
}
```

This approach can also be used with ambient transactions.

```
using (var context1 = new BloggingContext())
{
    context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });

    var strategy = context1.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var context2 = new BloggingContext())
        {
            using (var transaction = new TransactionScope())
            {
                context2.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context2.SaveChanges();

                context1.SaveChanges();

                transaction.Complete();
            }
        }
    });
}
```

Transaction commit failure and the idempotency issue

In general, when there is a connection failure the current transaction is rolled back. However, if the connection is dropped while the transaction is being committed the resulting state of the transaction is unknown. See this blog post for more details.

By default, the execution strategy will retry the operation as if the transaction was rolled back, but if it's not the case this will result in an exception if the new database state is incompatible or could lead to **data corruption** if the operation does not rely on a particular state, for example when inserting a new row with auto-generated key values.

There are several ways to deal with this.

Option 1 - Do (almost) nothing

The likelihood of a connection failure during transaction commit is low so it may be acceptable for your application to just fail if this condition actually occurs.

However, you need to avoid using store-generated keys in order to ensure that an exception is thrown instead of adding a duplicate row. Consider using a client-generated GUID value or a client-side value generator.

Option 2 - Rebuild application state

1. Discard the current `DbContext`.
2. Create a new `DbContext` and restore the state of your application from the database.
3. Inform the user that the last operation might not have been completed successfully.

Option 3 - Add state verification

For most of the operations that change the database state it is possible to add code that checks whether it succeeded. EF provides an extension method to make this easier - `IExecutionStrategy.ExecuteInTransaction`.

This method begins and commits a transaction and also accepts a function in the `verifySucceeded` parameter that is invoked when a transient error occurs during the transaction commit.

```
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**
>
> Here `SaveChanges` is invoked with `acceptAllChangesOnSuccess` set to `false` to avoid changing the state of the `Blog` entity to `Unchanged` if `SaveChanges` succeeds. This allows to retry the same operation if the commit fails and the transaction is rolled back.

Option 4 - Manually track the transaction

If you need to use store-generated keys or need a generic way of handling commit failures that doesn't depend on the operation performed each transaction could be assigned an ID that is checked when the commit fails.

1. Add a table to the database used to track the status of the transactions.
2. Insert a row into the table at the beginning of each transaction.
3. If the connection fails during the commit, check for the presence of the corresponding row in the database.
4. If the commit is successful, delete the corresponding row to avoid the growth of the table.

```
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

Make sure that the context used for the verification has an execution strategy defined as the connection is likely to fail again during verification if it failed during transaction commit.

Testing

8/27/2018 • 2 minutes to read • Edit Online

You may want to test components using something that approximates connecting to the real database, without the overhead of actual database I/O operations.

There are two main options for doing this:

- SQLite in-memory mode allows you to write efficient tests against a provider that behaves like a relational database.
- The InMemory provider is a lightweight provider that has minimal dependencies, but does not always behave like a relational database.

Testing with SQLite

8/27/2018 • 3 minutes to read • Edit Online

SQLite has an in-memory mode that allows you to use SQLite to write tests against a relational database, without the overhead of actual database operations.

TIP

You can view this article's sample on GitHub

Example testing scenario

Consider the following service that allows application code to perform some operations related to blogs. Internally it uses a `DbContext` that connects to a SQL Server database. It would be useful to swap this context to connect to an in-memory SQLite database so that we can write efficient tests for this service without having to modify the code, or do a lot of work to create a test double of the context.

```
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();
        }
    }
}
```

Get your context ready

Avoid configuring two database providers

In your tests you are going to externally configure the context to use the InMemory provider. If you are configuring a database provider by overriding `OnConfiguring` in your context, then you need to add some conditional code to ensure that you only configure the database provider if one has not already been configured.

TIP

If you are using ASP.NET Core, then you should not need this code since your database provider is configured outside of the context (in Startup.cs).

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}
```

Add a constructor for testing

The simplest way to enable testing against a different database is to modify your context to expose a constructor that accepts a `DbContextOptions<TContext>`.

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

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

TIP

`DbContextOptions<TContext>` tells the context all of its settings, such as which database to connect to. This is the same object that is built by running the OnConfiguring method in your context.

Writing tests

The key to testing with this provider is the ability to tell the context to use SQLite, and control the scope of the in-memory database. The scope of the database is controlled by opening and closing the connection. The database is scoped to the duration that the connection is open. Typically you want a clean database for each test method.

```
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
            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();
            }
        }
    }
}
```

Testing with InMemory

8/27/2018 • 3 minutes to read • Edit Online

The InMemory provider is useful when you want to test components using something that approximates connecting to the real database, without the overhead of actual database operations.

> **TIP**
>
> You can view this article's sample on GitHub.

InMemory is not a relational database

EF Core database providers do not have to be relational databases. InMemory is designed to be a general purpose database for testing, and is not designed to mimic a relational database.

Some examples of this include:

- InMemory will allow you to save data that would violate referential integrity constraints in a relational database.
- If you use DefaultValueSql(string) for a property in your model, this is a relational database API and will have no effect when running against InMemory.
- Concurrency via Timestamp/row version (`[Timestamp]` or `IsRowVersion`) is not supported. No DbUpdateConcurrencyException will be thrown if an update is done using an old concurrency token.

> **TIP**
>
> For many test purposes these differences will not matter. However, if you want to test against something that behaves more like a true relational database, then consider using SQLite in-memory mode.

Example testing scenario

Consider the following service that allows application code to perform some operations related to blogs. Internally it uses a `DbContext` that connects to a SQL Server database. It would be useful to swap this context to connect to an InMemory database so that we can write efficient tests for this service without having to modify the code, or do a lot of work to create a test double of the context.

```
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();
        }
    }
}
```

Get your context ready

Avoid configuring two database providers

In your tests you are going to externally configure the context to use the InMemory provider. If you are configuring a database provider by overriding `OnConfiguring` in your context, then you need to add some conditional code to ensure that you only configure the database provider if one has not already been configured.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}
```

> **TIP**
>
> If you are using ASP.NET Core, then you should not need this code since your database provider is already configured outside of the context (in Startup.cs).

Add a constructor for testing

The simplest way to enable testing against a different database is to modify your context to expose a constructor that accepts a `DbContextOptions<TContext>`.

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

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

> **TIP**
>
> `DbContextOptions<TContext>` tells the context all of its settings, such as which database to connect to. This is the same object that is built by running the OnConfiguring method in your context.

Writing tests

The key to testing with this provider is the ability to tell the context to use the InMemory provider, and control the scope of the in-memory database. Typically you want a clean database for each test method.

Here is an example of a test class that uses the InMemory database. Each test method specifies a unique database name, meaning each method has its own InMemory database.

> **TIP**
>
> To use the `.UseInMemoryDatabase()` extension method, reference the NuGet package `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());
            }
        }
    }
}
```

Configuring a DbContext

11/3/2018 • 3 minutes to read • Edit Online

This article shows basic patterns for configuring a `DbContext` via a `DbContextOptions` to connect to a database using a specific EF Core provider and optional behaviors.

Design-time DbContext configuration

EF Core design-time tools such as migrations need to be able to discover and create a working instance of a `DbContext` type in order to gather details about the application's entity types and how they map to a database schema. This process can be automatic as long as the tool can easily create the `DbContext` in such a way that it will be configured similarly to how it would be configured at run-time.

While any pattern that provides the necessary configuration information to the `DbContext` can work at run-time, tools that require using a `DbContext` at design-time can only work with a limited number of patterns. These are covered in more detail in the Design-Time Context Creation section.

Configuring DbContextOptions

`DbContext` must have an instance of `DbContextOptions` in order to perform any work. The `DbContextOptions` instance carries configuration information such as:

- The database provider to use, typically selected by invoking a method such as `UseSqlServer` or `UseSqlite`. These extension methods require the corresponding provider package, such as `Microsoft.EntityFrameworkCore.SqlServer` or `Microsoft.EntityFrameworkCore.Sqlite`. The methods are defined in the `Microsoft.EntityFrameworkCore` namespace.
- Any necessary connection string or identifier of the database instance, typically passed as an argument to the provider selection method mentioned above
- Any provider-level optional behavior selectors, typically also chained inside the call to the provider selection method
- Any general EF Core behavior selectors, typically chained after or before the provider selector method

The following example configures the `DbContextOptions` to use the SQL Server provider, a connection contained in the `connectionString` variable, a provider-level command timeout, and an EF Core behavior selector that makes all queries executed in the `DbContext` no-tracking by default:

```
optionsBuilder
    .UseSqlServer(connectionString, providerOptions=>providerOptions.CommandTimeout(60))
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
```

> **NOTE**
>
> Provider selector methods and other behavior selector methods mentioned above are extension methods on `DbContextOptions` or provider-specific option classes. In order to have access to these extension methods you may need to have a namespace (typically `Microsoft.EntityFrameworkCore`) in scope and include additional package dependencies in the project.

The `DbContextOptions` can be supplied to the `DbContext` by overriding the `OnConfiguring` method or externally via a constructor argument.

If both are used, `OnConfiguring` is applied last and can overwrite options supplied to the constructor argument.

Constructor argument

Context code with constructor:

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

    public DbSet<Blog> Blogs { get; set; }
}
```

> **TIP**
>
> The base constructor of DbContext also accepts the non-generic version of `DbContextOptions`, but using the non-generic version is not recommended for applications with multiple context types.

Application code to initialize from constructor argument:

```
var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))
{
  // do stuff
}
```

OnConfiguring

Context code with `OnConfiguring`:

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

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

Application code to initialize a `DbContext` that uses `OnConfiguring`:

```
using (var context = new BloggingContext())
{
  // do stuff
}
```

> **TIP**
>
> This approach does not lend itself to testing, unless the tests target the full database.

Using DbContext with dependency injection

EF Core supports using `DbContext` with a dependency injection container. Your DbContext type can be added to the service container by using the `AddDbContext<TContext>` method.

`AddDbContext<TContext>` will make both your DbContext type, `TContext`, and the corresponding `DbContextOptions<TContext>` available for injection from the service container.

See more reading below for additional information on dependency injection.

Adding the `Dbcontext` to dependency injection:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data Source=blog.db"));
}
```

This requires adding a constructor argument to your DbContext type that accepts `DbContextOptions<TContext>`.

Context code:

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

    public DbSet<Blog> Blogs { get; set; }
}
```

Application code (in ASP.NET Core):

```
public class MyController
{
    private readonly BloggingContext _context;

    public MyController(BloggingContext context)
    {
      _context = context;
    }

    ...
}
```

Application code (using ServiceProvider directly, less common):

```
using (var context = serviceProvider.GetService<BloggingContext>())
{
  // do stuff
}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>();
```

More reading

- Read Getting Started on ASP.NET Core for more information on using EF with ASP.NET Core.
- Read Dependency Injection to learn more about using DI.
- Read Testing for more information.

Creating and configuring a Model

9/13/2018 • 2 minutes to read • Edit Online

Entity Framework uses a set of conventions to build a model based on the shape of your entity classes. You can specify additional configuration to supplement and/or override what was discovered by convention.

This article covers configuration that can be applied to a model targeting any data store and that which can be applied when targeting any relational database. Providers may also enable configuration that is specific to a particular data store. For documentation on provider specific configuration see the Database Providers section.

> **TIP**
> You can view this article's sample on GitHub.

Use fluent API to configure a model

You can override the `OnModelCreating` method in your derived context and use the `ModelBuilder API` to configure your model. This is the most powerful method of configuration and allows configuration to be specified without modifying your entity classes. Fluent API configuration has the highest precedence and will override conventions and data annotations.

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

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

Use data annotations to configure a model

You can also apply attributes (known as Data Annotations) to your classes and properties. Data annotations will override conventions, but will be overridden by Fluent API configuration.

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

Including & Excluding Types

8/27/2018 • 2 minutes to read • Edit Online

Including a type in the model means that EF has metadata about that type and will attempt to read and write instances from/to the database.

Conventions

By convention, types that are exposed in `DbSet` properties on your context are included in your model. In addition, types that are mentioned in the `OnModelCreating` method are also included. Finally, any types that are found by recursively exploring the navigation properties of discovered types are also included in the model.

For example, in the following code listing all three types are discovered:

- `Blog` because it is exposed in a `DbSet` property on the context
- `Post` because it is discovered via the `Blog.Posts` navigation property
- `AuditEntry` because it is mentioned in `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; }
}
```

Data Annotations

You can use Data Annotations to exclude a type from the model.

```
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; }
}
```

Fluent API

You can use the Fluent API to exclude a type from the model.

```
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; }
}
```

Including & Excluding Properties

8/27/2018 • 2 minutes to read • Edit Online

Including a property in the model means that EF has metadata about that property and will attempt to read and write values from/to the database.

Conventions

By convention, public properties with a getter and a setter will be included in the model.

Data Annotations

You can use Data Annotations to exclude a property from the model.

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

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

Fluent API

You can use the Fluent API to exclude a property from the model.

```
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; }
}
```

Keys (primary)

8/27/2018 • 2 minutes to read • Edit Online

A key serves as the primary unique identifier for each entity instance. When using a relational database this maps to the concept of a *primary key*. You can also configure a unique identifier that is not the primary key (see Alternate Keys for more information).

Conventions

By convention, a property named `Id` or `<type name>Id` will be configured as the key of an entity.

```
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; }
}
```

Data Annotations

You can use Data Annotations to configure a single property to be the key of an entity.

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

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

Fluent API

You can use the Fluent API to configure a single property to be the key of an entity.

```
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; }
}
```

You can also use the Fluent API to configure multiple properties to be the key of an entity (known as a composite key). Composite keys can only be configured using the Fluent API - conventions will never setup a composite key and you can not use Data Annotations to configure one.

```
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; }
}
```

Generated Values

9/9/2018 • 4 minutes to read • Edit Online

Value Generation Patterns

There are three value generation patterns that can be used for properties:

- No value generation
- Value generated on add
- Value generated on add or update

No value generation

No value generation means that you will always supply a valid value to be saved to the database. This valid value must be assigned to new entities before they are added to the context.

Value generated on add

Value generated on add means that a value is generated for new entities.

Depending on the database provider being used, values may be generated client side by EF or in the database. If the value is generated by the database, then EF may assign a temporary value when you add the entity to the context. This temporary value will then be replaced by the database generated value during `SaveChanges()`.

If you add an entity to the context that has a value assigned to the property, then EF will attempt to insert that value rather than generating a new one. A property is considered to have a value assigned if it is not assigned the CLR default value (`null` for `string`, `0` for `int`, `Guid.Empty` for `Guid`, etc.). For more information, see Explicit values for generated properties.

> **WARNING**
> How the value is generated for added entities will depend on the database provider being used. Database providers may automatically setup value generation for some property types, but others may require you to manually setup how the value is generated.
>
> For example, when using SQL Server, values will be automatically generated for `GUID` properties (using the SQL Server sequential GUID algorithm). However, if you specify that a `DateTime` property is generated on add, then you must setup a way for the values to be generated. One way to do this, is to configure a default value of `GETDATE()`, see Default Values.

Value generated on add or update

Value generated on add or update means that a new value is generated every time the record is saved (insert or update).

Like `value generated on add`, if you specify a value for the property on a newly added instance of an entity, that value will be inserted rather than a value being generated. It is also possible to set an explicit value when updating. For more information, see Explicit values for generated properties.

WARNING

How the value is generated for added and updated entities will depend on the database provider being used. Database providers may automatically setup value generation for some property types, while others will require you to manually setup how the value is generated.

For example, when using SQL Server, `byte[]` properties that are set as generated on add or update and marked as concurrency tokens, will be setup with the `rowversion` data type - so that values will be generated in the database. However, if you specify that a `DateTime` property is generated on add or update, then you must setup a way for the values to be generated. One way to do this, is to configure a default value of `GETDATE()` (see Default Values) to generate values for new rows. You could then use a database trigger to generate values during updates (such as the following example trigger).

```
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
```

Conventions

By convention, non-composite primary keys of type short, int, long, or Guid will be setup to have values generated on add. All other properties will be setup with no value generation.

Data Annotations

No value generation (Data Annotations)

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

Value generated on add (Data Annotations)

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

> **WARNING**
>
> This just lets EF know that values are generated for added entities, it does not guarantee that EF will setup the actual mechanism to generate values. See Value generated on add section for more details.

Value generated on add or update (Data Annotations)

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

> **WARNING**
>
> This just lets EF know that values are generated for added or updated entities, it does not guarantee that EF will setup the actual mechanism to generate values. See Value generated on add or update section for more details.

Fluent API

You can use the Fluent API to change the value generation pattern for a given property.

No value generation (Fluent API)

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

Value generated on add (Fluent API)

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

> **WARNING**
>
> `ValueGeneratedOnAdd()` just lets EF know that values are generated for added entities, it does not guarantee that EF will setup the actual mechanism to generate values. See Value generated on add section for more details.

Value generated on add or update (Fluent API)

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

> **WARNING**
>
> This just lets EF know that values are generated for added or updated entities, it does not guarantee that EF will setup the actual mechanism to generate values. See Value generated on add or update section for more details.

Required and Optional Properties

8/27/2018 • 2 minutes to read • Edit Online

A property is considered optional if it is valid for it to contain `null`. If `null` is not a valid value to be assigned to a property then it is considered to be a required property.

Conventions

By convention, a property whose CLR type can contain null will be configured as optional (`string`, `int?`, `byte[]`, etc.). Properties whose CLR type cannot contain null will be configured as required (`int`, `decimal`, `bool`, etc.).

> **NOTE**
>
> A property whose CLR type cannot contain null cannot be configured as optional. The property will always be considered required by Entity Framework.

Data Annotations

You can use Data Annotations to indicate that a property is required.

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

Fluent API

You can use the Fluent API to indicate that a property is required.

```
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; }
}
```

Maximum Length

8/27/2018 • 2 minutes to read • Edit Online

Configuring a maximum length provides a hint to the data store about the appropriate data type to use for a given property. Maximum length only applies to array data types, such as `string` and `byte[]`.

> **NOTE**
>
> Entity Framework does not do any validation of maximum length before passing data to the provider. It is up to the provider or data store to validate if appropriate. For example, when targeting SQL Server, exceeding the maximum length will result in an exception as the data type of the underlying column will not allow excess data to be stored.

Conventions

By convention, it is left up to the database provider to choose an appropriate data type for properties. For properties that have a length, the database provider will generally choose a data type that allows for the longest length of data. For example, Microsoft SQL Server will use `nvarchar(max)` for `string` properties (or `nvarchar(450)` if the column is used as a key).

Data Annotations

You can use the Data Annotations to configure a maximum length for a property. In this example, targeting SQL Server this would result in the `nvarchar(500)` data type being used.

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

Fluent API

You can use the Fluent API to configure a maximum length for a property. In this example, targeting SQL Server this would result in the `nvarchar(500)` data type being used.

```
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; }
}
```

Concurrency Tokens

8/27/2018 • 2 minutes to read • Edit Online

NOTE

This page documents how to configure concurrency tokens. See Handling Concurrency Conflicts for a detailed explanation of how concurrency control works on EF Core and examples of how to handle concurrency conflicts in your application.

Properties configured as concurrency tokens are used to implement optimistic concurrency control.

Conventions

By convention, properties are never configured as concurrency tokens.

Data Annotations

You can use the Data Annotations to configure a property as a concurrency token.

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

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

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

Fluent API

You can use the Fluent API to configure a property as a concurrency token.

```
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; }
}
```

Timestamp/row version

A timestamp is a property where a new value is generated by the database every time a row is inserted or updated. The property is also treated as a concurrency token. This ensures you will get an exception if anyone else has modified a row that you are trying to update since you queried for the data.

How this is achieved is up to the database provider being used. For SQL Server, timestamp is usually used on a *byte[]* property, which will be setup as a *ROWVERSION* column in the database.

Conventions

By convention, properties are never configured as timestamps.

Data Annotations

You can use Data Annotations to configure a property as a timestamp.

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

    public string Url { get; set; }

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

Fluent API

You can use the Fluent API to configure a property as a timestamp.

```
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; }
}
```

Shadow Properties

8/27/2018 • 2 minutes to read • Edit Online

Shadow properties are properties that are not defined in your .NET entity class but are defined for that entity type in the EF Core model. The value and state of these properties is maintained purely in the Change Tracker.

Shadow properties are useful when there is data in the database that should not be exposed on the mapped entity types. They are most often used for foreign key properties, where the relationship between two entities is represented by a foreign key value in the database, but the relationship is managed on the entity types using navigation properties between the entity types.

Shadow property values can be obtained and changed through the `ChangeTracker` API.

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

Shadow properties can be referenced in LINQ queries via the `EF.Property` static method.

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

Conventions

Shadow properties can be created by convention when a relationship is discovered but no foreign key property is found in the dependent entity class. In this case, a shadow foreign key property will be introduced. The shadow foreign key property will be named `<navigation property name><principal key property name>` (the navigation on the dependent entity, which points to the principal entity, is used for the naming). If the principal key property name includes the name of the navigation property, then the name will just be `<principal key property name>`. If there is no navigation property on the dependent entity, then the principal type name is used in its place.

For example, the following code listing will result in a `BlogId` shadow property being introduced to the `Post` entity.

```
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; }
}
```

Data Annotations

Shadow properties can not be created with data annotations.

Fluent API

You can use the Fluent API to configure shadow properties. Once you have called the string overload of `Property` you can chain any of the configuration calls you would for other properties.

If the name supplied to the `Property` method matches the name of an existing property (a shadow property or one defined on the entity class), then the code will configure that existing property rather than introducing a new shadow property.

```
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; }
}
```

Relationships

8/27/2018 • 15 minutes to read • Edit Online

A relationship defines how two entities relate to each other. In a relational database, this is represented by a foreign key constraint.

> **NOTE**
>
> Most of the samples in this article use a one-to-many relationship to demonstrate concepts. For examples of one-to-one and many-to-many relationships see the Other Relationship Patterns section at the end of the article.

Definition of Terms

There are a number of terms used to describe relationships

- **Dependent entity:** This is the entity that contains the foreign key property(s). Sometimes referred to as the 'child' of the relationship.
- **Principal entity:** This is the entity that contains the primary/alternate key property(s). Sometimes referred to as the 'parent' of the relationship.
- **Foreign key:** The property(s) in the dependent entity that is used to store the values of the principal key property that the entity is related to.
- **Principal key:** The property(s) that uniquely identifies the principal entity. This may be the primary key or an alternate key.
- **Navigation property:** A property defined on the principal and/or dependent entity that contains a reference(s) to the related entity(s).
  - **Collection navigation property:** A navigation property that contains references to many related entities.
  - **Reference navigation property:** A navigation property that holds a reference to a single related entity.
  - **Inverse navigation property:** When discussing a particular navigation property, this term refers to the navigation property on the other end of the relationship.

The following code listing shows a one-to-many relationship between `Blog` and `Post`

- `Post` is the dependent entity
- `Blog` is the principal entity
- `Post.BlogId` is the foreign key
- `Blog.BlogId` is the principal key (in this case it is a primary key rather than an alternate key)
- `Post.Blog` is a reference navigation property
- `Blog.Posts` is a collection navigation property
- `Post.Blog` is the inverse navigation property of `Blog.Posts` (and vice versa)

```
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; }
}
```

Conventions

By convention, a relationship will be created when there is a navigation property discovered on a type. A property is considered a navigation property if the type it points to can not be mapped as a scalar type by the current database provider.

> **NOTE**
>
> Relationships that are discovered by convention will always target the primary key of the principal entity. To target an alternate key, additional configuration must be performed using the Fluent API.

Fully Defined Relationships

The most common pattern for relationships is to have navigation properties defined on both ends of the relationship and a foreign key property defined in the dependent entity class.

- If a pair of navigation properties is found between two types, then they will be configured as inverse navigation properties of the same relationship.
- If the dependent entity contains a property named `<primary key property name>`, `<navigation property name><primary key property name>`, or `<principal entity name><primary key property name>` then it will be configured as the foreign key.

```
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**
>
> If there are multiple navigation properties defined between two types (that is, more than one distinct pair of navigations that point to each other), then no relationships will be created by convention and you will need to manually configure them to identify how the navigation properties pair up.

No Foreign Key Property

While it is recommended to have a foreign key property defined in the dependent entity class, it is not required. If no foreign key property is found, a shadow foreign key property will be introduced with the name `<navigation property name><principal key property name>` (see Shadow Properties for more information).

```
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; }
}
```

Single Navigation Property

Including just one navigation property (no inverse navigation, and no foreign key property) is enough to have a relationship defined by convention. You can also have a single navigation property and a foreign key property.

```
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; }
}
```

Cascade Delete

By convention, cascade delete will be set to *Cascade* for required relationships and *ClientSetNull* for optional relationships. *Cascade* means dependent entities are also deleted. *ClientSetNull* means that dependent entities that are not loaded into memory will remain unchanged and must be manually deleted, or updated to point to a valid principal entity. For entities that are loaded into memory, EF Core will attempt to set the foreign key properties to null.

See the Required and Optional Relationships section for the difference between required and optional relationships.

See Cascade Delete for more details about the different delete behaviors and the defaults used by convention.

Data Annotations

There are two data annotations that can be used to configure relationships, `[ForeignKey]` and `[InverseProperty]`.

[ForeignKey]

You can use the Data Annotations to configure which property should be used as the foreign key property for a given relationship. This is typically done when the foreign key property is not discovered by convention.

```
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**
> The `[ForeignKey]` annotation can be placed on either navigation property in the relationship. It does not need to go on the navigation property in the dependent entity class.

[InverseProperty]

You can use the Data Annotations to configure how navigation properties on the dependent and principal entities pair up. This is typically done when there is more than one pair of navigation properties between two entity types.

```
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; }
}
```

Fluent API

To configure a relationship in the Fluent API, you start by identifying the navigation properties that make up the relationship. `HasOne` or `HasMany` identifies the navigation property on the entity type you are beginning the configuration on. You then chain a call to `WithOne` or `WithMany` to identify the inverse navigation. `HasOne`/`WithOne` are used for reference navigation properties and `HasMany`/`WithMany` are used for collection navigation properties.

```
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; }
}
```

Single Navigation Property

If you only have one navigation property then there are parameterless overloads of `WithOne` and `WithMany`. This indicates that there is conceptually a reference or collection on the other end of the relationship, but there is no navigation property included in the entity class.

```
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; }
}
```

Foreign Key

You can use the Fluent API to configure which property should be used as the foreign key property for a given relationship.

```
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; }
}
```

The following code listing shows how to configure a composite foreign key.

```
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; }
}
```

You can use the string overload of `HasForeignKey(...)` to configure a shadow property as a foreign key (see Shadow Properties for more information). We recommend explicitly adding the shadow property to the model before using it as a foreign key (as shown below).

```
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; }
}
```

Principal Key

If you want the foreign key to reference a property other than the primary key, you can use the Fluent API to configure the principal key property for the relationship. The property that you configure as the principal key will automatically be setup as an alternate key (see Alternate Keys for more information).

```
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; }
}
```

The following code listing shows how to configure a composite principal key.

```
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

The order in which you specify principal key properties must match the order in which they are specified for the foreign key.

Required and Optional Relationships

You can use the Fluent API to configure whether the relationship is required or optional. Ultimately this controls whether the foreign key property is required or optional. This is most useful when you are using a shadow state foreign key. If you have a foreign key property in your entity class then the requiredness of the relationship is determined based on whether the foreign key property is required or optional (see Required and Optional properties for more information).

```
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; }
}
```

Cascade Delete

You can use the Fluent API to configure the cascade delete behavior for a given relationship explicitly.

See Cascade Delete on the Saving Data section for a detailed discussion of each option.

```
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; }
}
```

Other Relationship Patterns

One-to-one

One to one relationships have a reference navigation property on both sides. They follow the same conventions as one-to-many relationships, but a unique index is introduced on the foreign key property to ensure only one dependent is related to each principal.

```
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 will choose one of the entities to be the dependent based on its ability to detect a foreign key property. If the wrong entity is chosen as the dependent, you can use the Fluent API to correct this.

When configuring the relationship with the Fluent API, you use the `HasOne` and `WithOne` methods.

When configuring the foreign key you need to specify the dependent entity type - notice the generic parameter provided to `HasForeignKey` in the listing below. In a one-to-many relationship it is clear that the entity with the reference navigation is the dependent and the one with the collection is the principal. But this is not so in a one-to-one relationship - hence the need to explicitly define it.

```
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; }
}
```

Many-to-many

Many-to-many relationships without an entity class to represent the join table are not yet supported. However, you can represent a many-to-many relationship by including an entity class for the join table and mapping two separate one-to-many relationships.

```
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; }
}
```

Indexes

8/27/2018 • 2 minutes to read • Edit Online

Indexes are a common concept across many data stores. While their implementation in the data store may vary, they are used to make lookups based on a column (or set of columns) more efficient.

Conventions

By convention, an index is created in each property (or set of properties) that are used as a foreign key.

Data Annotations

Indexes can not be created using data annotations.

Fluent API

You can use the Fluent API to specify an index on a single property. By default, indexes are non-unique.

```
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; }
}
```

You can also specify that an index should be unique, meaning that no two entities can have the same value(s) for the given property(s).

```
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .IsUnique();
```

You can also specify an index over more than one column.

```
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

There is only one index per distinct set of properties. If you use the Fluent API to configure an index on a set of properties that already has an index defined, either by convention or previous configuration, then you will be changing the definition of that index. This is useful if you want to further configure an index that was created by convention.

Alternate Keys

8/27/2018 • 2 minutes to read • Edit Online

An alternate key serves as an alternate unique identifier for each entity instance in addition to the primary key. Alternate keys can be used as the target of a relationship. When using a relational database this maps to the concept of a unique index/constraint on the alternate key column(s) and one or more foreign key constraints that reference the column(s).

> **TIP**
> If you just want to enforce uniqueness of a column then you want a unique index rather than an alternate key, see Indexes. In EF, alternate keys provide greater functionality than unique indexes because they can be used as the target of a foreign key.

Alternate keys are typically introduced for you when needed and you do not need to manually configure them. See Conventions for more details.

Conventions

By convention, an alternate key is introduced for you when you identify a property, that is not the primary key, as the target of a relationship.

```
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; }
}
```

Data Annotations

Alternate keys can not be configured using Data Annotations.

Fluent API

You can use the Fluent API to configure a single property to be an alternate key.

```
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; }
}
```

You can also use the Fluent API to configure multiple properties to be an alternate key (known as a composite alternate key).

```
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; }
}
```

Inheritance

8/27/2018 • 2 minutes to read • Edit Online

Inheritance in the EF model is used to control how inheritance in the entity classes is represented in the database.

Conventions

By convention, it is up to the database provider to determine how inheritance will be represented in the database. See Inheritance (Relational Database) for how this is handled with a relational database provider.

EF will only setup inheritance if two or more inherited types are explicitly included in the model. EF will not scan for base or derived types that were not otherwise included in the model. You can include types in the model by exposing a *DbSet* for each type in the inheritance hierarchy.

```
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; }
}
```

If you don't want to expose a *DbSet* for one or more entities in the hierarchy, you can use the Fluent API to ensure they are included in the model. And if you don't rely on conventions you can specify the base type explicitly using `HasBaseType`.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RssBlog>().HasBaseType<Blog>();
    }
}
```

NOTE

You can use `.HasBaseType((Type)null)` to remove an entity type from the hierarchy.

Data Annotations

You cannot use Data Annotations to configure inheritance.

Fluent API

The Fluent API for inheritance depends on the database provider you are using. See Inheritance (Relational Database) for the configuration you can perform for a relational database provider.

Backing Fields

8/27/2018 • 3 minutes to read • Edit Online

> **NOTE**
> This feature is new in EF Core 1.1.

Backing fields allow EF to read and/or write to a field rather than a property. This can be useful when encapsulation in the class is being used to restrict the use of and/or enhance the semantics around access to the data by application code, but the value should be read from and/or written to the database without using those restrictions/enhancements.

Conventions

By convention, the following fields will be discovered as backing fields for a given property (listed in precedence order). Fields are only discovered for properties that are included in the model. For more information on which properties are included in the model, see Including & Excluding Properties.

- `_<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; }
    }
}
```

When a backing field is configured, EF will write directly to that field when materializing entity instances from the database (rather than using the property setter). If EF needs to read or write the value at other times, it will use the property if possible. For example, if EF needs to update the value for a property, it will use the property setter if one is defined. If the property is read-only, then it will write to the field.

Data Annotations

Backing fields cannot be configured with data annotations.

Fluent API

You can use the Fluent API to configure a backing field for a property.

```
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;
    }
}
```

Controlling when the field is used

You can configure when EF uses the field or property. See the PropertyAccessMode enum for the supported options.

```
modelBuilder.Entity<Blog>()
    .Property(b => b.Url)
    .HasField("_validatedUrl")
    .UsePropertyAccessMode(PropertyAccessMode.Field);
```

Fields without a property

You can also create a conceptual property in your model that does not have a corresponding CLR property in the entity class, but instead uses a field to store the data in the entity. This is different from Shadow Properties, where the data is stored in the change tracker. This would typically be used if the entity class uses methods to get/set values.

You can give EF the name of the field in the `Property(...)` API. If there is no property with the given name, then EF will look for a field.

```csharp
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;
    }
}
```

You can also choose to give the property a name, other than the field name. This name is then used when creating the model, most notably it will be used for the column name that is mapped to in the database.

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property<string>("Url")
        .HasField("_validatedUrl");
}
```

When there is no property in the entity class, you can use the `EF.Property(...)` method in a LINQ query to refer to the property that is conceptually part of the model.

```csharp
var blogs = db.blogs.OrderBy(b => EF.Property<string>(b, "Url"));
```

Value Conversions

8/28/2018 • 4 minutes to read • Edit Online

NOTE

This feature is new in EF Core 2.1.

Value converters allow property values to be converted when reading from or writing to the database. This conversion can be from one value to another of the same type (for example, encrypting strings) or from a value of one type to a value of another type (for example, converting enum values to and from strings in the database.)

Fundamentals

Value converters are specified in terms of a `ModelClrType` and a `ProviderClrType`. The model type is the .NET type of the property in the entity type. The provider type is the .NET type understood by the database provider. For example, to save enums as strings in the database, the model type is the type of the enum, and the provider type is `String`. These two types can be the same.

Conversions are defined using two `Func` expression trees: one from `ModelClrType` to `ProviderClrType` and the other from `ProviderClrType` to `ModelClrType`. Expression trees are used so that they can be compiled into the database access code for efficient conversions. For complex conversions, the expression tree may be a simple call to a method that performs the conversion.

Configuring a value converter

Value conversions are defined on properties in the `OnModelCreating` of your `DbContext`. For example, consider an enum and entity type defined as:

```
public class Rider
{
    public int Id { get; set; }
    public EquineBeast Mount { get; set; }
}

public enum EquineBeast
{
    Donkey,
    Mule,
    Horse,
    Unicorn
}
```

Then conversions can be defined in `OnModelCreating` to store the enum values as strings (for example, "Donkey", "Mule", ...) in the database:

```
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

A `null` value will never be passed to a value converter. This makes the implementation of conversions easier and allows them to be shared amongst nullable and non-nullable properties.

The ValueConverter class

Calling `HasConversion` as shown above will create a `ValueConverter` instance and set it on the property. The `ValueConverter` can instead be created explicitly. For example:

```
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);
```

This can be useful when multiple properties use the same conversion.

NOTE

There is currently no way to specify in one place that every property of a given type must use the same value converter. This feature will be considered for a future release.

Built-in converters

EF Core ships with a set of pre-defined `ValueConverter` classes, found in the `Microsoft.EntityFrameworkCore.Storage.ValueConversion` namespace. These are:

- `BoolToZeroOneConverter` - Bool to zero and one
- `BoolToStringConverter` - Bool to strings such as "Y" and "N"
- `BoolToTwoValuesConverter` - Bool to any two values
- `BytesToStringConverter` - Byte array to Base64-encoded string
- `CastingConverter` - Conversions that require only a type cast
- `CharToStringConverter` - Char to single character string
- `DateTimeOffsetToBinaryConverter` - DateTimeOffset to binary-encoded 64-bit value
- `DateTimeOffsetToBytesConverter` - DateTimeOffset to byte array
- `DateTimeOffsetToStringConverter` - DateTimeOffset to string
- `DateTimeToBinaryConverter` - DateTime to 64-bit value including DateTimeKind
- `DateTimeToStringConverter` - DateTime to string

- `DateTimeToTicksConverter` - DateTime to ticks
- `EnumToNumberConverter` - Enum to underlying number
- `EnumToStringConverter` - Enum to string
- `GuidToBytesConverter` - Guid to byte array
- `GuidToStringConverter` - Guid to string
- `NumberToBytesConverter` - Any numerical value to byte array
- `NumberToStringConverter` - Any numerical value to string
- `StringToBytesConverter` - String to UTF8 bytes
- `TimeSpanToStringConverter` - TimeSpan to string
- `TimeSpanToTicksConverter` - TimeSpan to ticks

Notice that `EnumToStringConverter` is included in this list. This means that there is no need to specify the conversion explicitly, as shown above. Instead, just use the built-in converter:

```
var converter = new EnumToStringConverter<EquineBeast>();

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

Note that all the built-in converters are stateless and so a single instance can be safely shared by multiple properties.

Pre-defined conversions

For common conversions for which a built-in converter exists there is no need to specify the converter explicitly. Instead, just configure which provider type should be used and EF will automatically use the appropriate built-in converter. Enum to string conversions are used as an example above, but EF will actually do this automatically if the provider type is configured:

```
modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion<string>();
```

The same thing can be achieved by explicitly specifying the column type. For example, if the entity type is defined like so:

```
public class Rider
{
    public int Id { get; set; }

    [Column(TypeName = "nvarchar(24)")]
    public EquineBeast Mount { get; set; }
}
```

Then the enum values will be saved as strings in the database without any further configuration in `OnModelCreating`.

Limitations

There are a few known current limitations of the value conversion system:

- As noted above, `null` cannot be converted.
- There is currently no way to spread a conversion of one property to multiple columns or vice-versa.
- Use of value conversions may impact the ability of EF Core to translate expressions to SQL. A warning will be logged for such cases. Removal of these limitations is being considered for a future release.

Data Seeding

1/15/2019 • 3 minutes to read • Edit Online

Data seeding is the process of populating a database with an initial set of data.

There are several ways this can be accomplished in EF Core:

- Model seed data
- Manual migration customization
- Custom initialization logic

Model seed data

> **NOTE**
>
> This feature is new in EF Core 2.1.

Unlike in EF6, in EF Core, seeding data can be associated with an entity type as part of the model configuration. Then EF Core migrations can automatically compute what insert, update or delete operations need to be applied when upgrading the database to a new version of the model.

> **NOTE**
>
> Migrations only considers model changes when determining what operation should be performed to get the seed data into the desired state. Thus any changes to the data performed outside of migrations might be lost or cause an error.

As an example, this will configure seed data for a `Blog` in `OnModelCreating`:

```
modelBuilder.Entity<Blog>().HasData(new Blog {BlogId = 1, Url = "http://sample.com"});
```

To add entities that have a relationship the foreign key values need to be specified:

```
modelBuilder.Entity<Post>().HasData(
    new Post() { BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1" });
```

If the entity type has any properties in shadow state an anonymous class can be used to provide the values:

```
modelBuilder.Entity<Post>().HasData(
    new { BlogId = 1, PostId = 2, Title = "Second post", Content = "Test 2" });
```

Owned entity types can be seeded in a similar fashion:

```
modelBuilder.Entity<Post>().OwnsOne(p => p.AuthorName).HasData(
    new { PostId = 1, First = "Andriy", Last = "Svyryd" },
    new { PostId = 2, First = "Diego", Last = "Vega" });
```

See the full sample project for more context.

Once the data has been added to the model, migrations should be used to apply the changes.

> **TIP**
>
> If you need to apply migrations as part of an automated deployment you can create a SQL script that can be previewed before execution.

Alternatively, you can use `context.Database.EnsureCreated()` to create a new database containing the seed data, for example for a test database or when using the in-memory provider or any non-relation database. Note that if the database already exists, `EnsureCreated()` will neither update the schema nor seed data in the database. For relational databases you shouldn't call `EnsureCreated()` if you plan to use Migrations.

Limitations of model seed data

This type of seed data is managed by migrations and the script to update the data that's already in the database needs to be generated without connecting to the database. This imposes some restrictions:

- The primary key value needs to be specified even if it's usually generated by the database. It will be used to detect data changes between migrations.
- Previously seeded data will be removed if the primary key is changed in any way.

Therefore this feature is most useful for static data that's not expected to change outside of migrations and does not depend on anything else in the database, for example ZIP codes.

If your scenario includes any of the following it is recommended to use custom initialization logic described in the last section:

- Temporary data for testing
- Data that depends on database state
- Data that needs key values to be generated by the database, including entities that use alternate keys as the identity
- Data that requires custom transformation (that is not handled by value conversions), such as some password hashing
- Data that requires calls to external API, such as ASP.NET Core Identity roles and users creation

Manual migration customization

When a migration is added the changes to the data specified with `HasData` are transformed to calls to `InsertData()`, `UpdateData()`, and `DeleteData()`. One way of working around some of the limitations of `HasData` is to manually add these calls or custom operations to the migration instead.

```
migrationBuilder.InsertData(
    table: "Blogs",
    columns: new[] { "Url" },
    values: new object[] { "http://generated.com" });
```

Custom initialization logic

A straightforward and powerful way to perform data seeding is to use `DbContext.SaveChanges()` before the main application logic begins execution.

```
using (var context = new DataSeedingContext())
{
    context.Database.EnsureCreated();

    var testBlog = context.Blogs.FirstOrDefault(b => b.Url == "http://test.com");
    if (testBlog == null)
    {
        context.Blogs.Add(new Blog { Url = "http://test.com" });
    }
    context.SaveChanges();
}
```

WARNING

The seeding code should not be part of the normal app execution as this can cause concurrency issues when multiple instances are running and would also require the app having permission to modify the database schema.

Depending on the constraints of your deployment the initialization code can be executed in different ways:

- Running the initialization app locally
- Deploying the initialization app with the main app, invoking the initialization routine and disabling or removing the initialization app.

This can usually be automated by using publish profiles.

Entity types with constructors

8/28/2018 • 6 minutes to read • Edit Online

NOTE

This feature is new in EF Core 2.1.

Starting with EF Core 2.1, it is now possible to define a constructor with parameters and have EF Core call this constructor when creating an instance of the entity. The constructor parameters can be bound to mapped properties, or to various kinds of services to facilitate behaviors like lazy-loading.

NOTE

As of EF Core 2.1, all constructor binding is by convention. Configuration of specific constructors to use is planned for a future release.

Binding to mapped properties

Consider a typical Blog/Post model:

```
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; }
}
```

When EF Core creates instances of these types, such as for the results of a query, it will first call the default parameterless constructor and then set each property to the value from the database. However, if EF Core finds a parameterized constructor with parameter names and types that match those of mapped properties, then it will instead call the parameterized constructor with values for those properties and will not set each property explicitly. For example:

```
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; }
}
```

Some things to note:

- Not all properties need to have constructor parameters. For example, the Post.Content property is not set by any constructor parameter, so EF Core will set it after calling the constructor in the normal way.
- The parameter types and names must match property types and names, except that properties can be Pascal-cased while the parameters are camel-cased.
- EF Core cannot set navigation properties (such as Blog or Posts above) using a constructor.
- The constructor can be public, private, or have any other accessibility.

Read-only properties

Once properties are being set via the constructor it can make sense to make some of them read-only. EF Core supports this, but there are some things to look out for:

- Properties without setters are not mapped by convention. (Doing so tends to map properties that should not be mapped, such as computed properties.)
- Using automatically generated key values requires a key property that is read-write, since the key value needs to be set by the key generator when inserting new entities.

An easy way to avoid these things is to use private setters. For example:

```
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 sees a property with a private setter as read-write, which means that all properties are mapped as before and the key can still be store-generated.

An alternative to using private setters is to make properties really read-only and add more explicit mapping in OnModelCreating. Likewise, some properties can be removed completely and replaced with only fields. For example, consider these entity types:

```
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; }
}
```

And this configuration in 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);
        });
}
```

Things to note:

- The key "property" is now a field. It is not a `readonly` field so that store-generated keys can be used.
- The other properties are read-only properties set only in the constructor.
- If the primary key value is only ever set by EF or read from the database, then there is no need to include it in the constructor. This leaves the key "property" as a simple field and makes it clear that it should not be set explicitly when creating new blogs or posts.

> **NOTE**
>
> This code will result in compiler warning '169' indicating that the field is never used. This can be ignored since in reality EF Core is using the field in an extralinguistic manner.

Injecting services

EF Core can also inject "services" into an entity type's constructor. For example, the following can be injected:

- `DbContext` - the current context instance, which can also be typed as your derived DbContext type
- `ILazyLoader` - the lazy-loading service--see the lazy-loading documentation for more details
- `Action<object, string>` - a lazy-loading delegate--see the lazy-loading documentation for more details
- `IEntityType` - the EF Core metadata associated with this entity type

> **NOTE**
>
> As of EF Core 2.1, only services known by EF Core can be injected. Support for injecting application services is being considered for a future release.

For example, an injected DbContext can be used to selectively access the database to obtain information about related entities without loading them all. In the example below this is used to obtain the number of posts in a blog without loading the posts:

```
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; }
}
```

A few things to notice about this:

- The constructor is private, since it is only ever called by EF Core, and there is another public constructor for general use.
- The code using the injected service (that is, the context) is defensive against it being `null` to handle cases where EF Core is not creating the instance.
- Because service is stored in a read/write property it will be reset when the entity is attached to a new context instance.

> **WARNING**
>
> Injecting the DbContext like this is often considered an anti-pattern since it couples your entity types directly to EF Core. Carefully consider all options before using service injection like this.

Owned Entity Types

1/7/2019 • 6 minutes to read • Edit Online

NOTE

This feature is new in EF Core 2.0.

EF Core allows you to model entity types that can only ever appear on navigation properties of other entity types. These are called *owned entity types*. The entity containing an owned entity type is its *owner*.

Explicit configuration

Owned entity types are never included by EF Core in the model by convention. You can use the `OwnsOne` method in `OnModelCreating` or annotate the type with `OwnedAttribute` (new in EF Core 2.1) to configure the type as an owned type.

In this example, `StreetAddress` is a type with no identity property. It is used as a property of the Order type to specify the shipping address for a particular order.

We can use the `OwnedAttribute` to treat it as an owned entity when referenced from another entity type:

```
[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; }
}
```

It is also possible to use the `OwnsOne` method in `OnModelCreating` to specify that the `ShippingAddress` property is an Owned Entity of the `Order` entity type and to configure additional facets if needed.

```
modelBuilder.Entity<Order>().OwnsOne(p => p.ShippingAddress);
```

If the `ShippingAddress` property is private in the `Order` type, you can use the string version of the `OwnsOne` method:

```
modelBuilder.Entity<Order>().OwnsOne(typeof(StreetAddress), "ShippingAddress");
```

See the full sample project for more context.

Implicit keys

Owned types configured with `OwnsOne` or discovered through a reference navigation always have a one-to-one relationship with the owner, therefore they don't need their own key values as the foreign key values are unique. In the previous example, the `StreetAddress` type does not need to define a key property.

In order to understand how EF Core tracks these objects, it is useful to think that a primary key is created as a shadow property for the owned type. The value of the key of an instance of the owned type will be the same as the value of the key of the owner instance.

Collections of owned types

> **NOTE**
>
> This feature is new in EF Core 2.2.

To configure a collection of owned types `OwnsMany` should be used in `OnModelCreating`. However the primary key will not be configured by convention, so it needs to be specified explicitly. It is common to use a complex key for these type of entities incorporating the foreign key to the owner and an additional unique property that can also be in shadow state:

```
modelBuilder.Entity<Distributor>().OwnsMany(p => p.ShippingCenters, a =>
{
    a.HasForeignKey("DistributorId");
    a.Property<int>("Id");
    a.HasKey("DistributorId", "Id");
});
```

Mapping owned types with table splitting

When using relational databases, by convention reference owned types are mapped to the same table as the owner. This requires splitting the table in two: some columns will be used to store the data of the owner, and some columns will be used to store data of the owned entity. This is a common feature known as table splitting.

> **TIP**
>
> Owned types stored with table splitting can be used similarly to how complex types are used in EF6.

By convention, EF Core will name the database columns for the properties of the owned entity type following the pattern *Navigation_OwnedEntityProperty*. Therefore the `StreetAddress` properties will appear in the 'Orders' table with the names 'ShippingAddress_Street' and 'ShippingAddress_City'.

You can append the `HasColumnName` method to rename those columns:

```
modelBuilder.Entity<Order>().OwnsOne(
    o => o.ShippingAddress,
    sa =>
    {
        sa.Property(p => p.Street).HasColumnName("ShipsToStreet");
        sa.Property(p => p.City).HasColumnName("ShipsToCity");
    });
```

Sharing the same .NET type among multiple owned types

An owned entity type can be of the same .NET type as another owned entity type, therefore the .NET type may not be enough to identify an owned type.

In those cases, the property pointing from the owner to the owned entity becomes the *defining navigation* of the owned entity type. From the perspective of EF Core, the defining navigation is part of the type's identity alongside the .NET type.

For example, in the following class `ShippingAddress` and `BillingAddress` are both of the same .NET type, `StreetAddress`:

```
public class OrderDetails
{
    public DetailedOrder Order { get; set; }
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

In order to understand how EF Core will distinguish tracked instances of these objects, it may be useful to think that the defining navigation has become part of the key of the instance alongside the value of the key of the owner and the .NET type of the owned type.

Nested owned types

In this example `OrderDetails` owns `BillingAddress` and `ShippingAddress`, which are both `StreetAddress` types. Then `OrderDetails` is owned by the `DetailedOrder` type.

```
public class DetailedOrder
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
    public OrderStatus Status { get; set; }
}
```

```
public enum OrderStatus
{
    Pending,
    Shipped
}
```

In addition to nested owned types, an owned type can reference a regular entity, it can be either the owner or a different entity as long as the owned entity is on the dependent side. This capability sets owned entity types apart from complex types in EF6.

```
public class OrderDetails
{
    public DetailedOrder Order { get; set; }
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

It is possible to chain the `OwnsOne` method in a fluent call to configure this model:

```
modelBuilder.Entity<DetailedOrder>().OwnsOne(p => p.OrderDetails, od =>
{
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
});
```

It is also possible to achieve the same thing using `OwnedAttribute` on both `OrderDetails` and `StreetAdress`.

Storing owned types in separate tables

Also unlike EF6 complex types, owned types can be stored in a separate table from the owner. In order to override the convention that maps an owned type to the same table as the owner, you can simply call `ToTable` and provide a different table name. The following example will map `OrderDetails` and its two addresses to a separate table from `DetailedOrder`:

```
modelBuilder.Entity<DetailedOrder>().OwnsOne(p => p.OrderDetails, od =>
{
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
    od.ToTable("OrderDetails");
});
```

Querying owned types

When querying the owner the owned types will be included by default. It is not necessary to use the `Include` method, even if the owned types are stored in a separate table. Based on the model described before, the following query will get `Order`, `OrderDetails` and the two owned `StreetAddresses` from the database:

```
var order = context.DetailedOrders.First(o => o.Status == OrderStatus.Pending);
Console.WriteLine($"First pending order will ship to: {order.OrderDetails.ShippingAddress.City}");
```

Limitations

Some of these limitations are fundamental to how owned entity types work, but some others are restrictions that we may be able to remove in future releases:

By-design restrictions

- You cannot create a `DbSet<T>` for an owned type
- You cannot call `Entity<T>()` with an owned type on `ModelBuilder`

Current shortcomings

- Inheritance hierarchies that include owned entity types are not supported
- Reference navigations to owned entity types cannot be null unless they are explicitly mapped to a separate table from the owner
- Instances of owned entity types cannot be shared by multiple owners (this is a well-known scenario for value objects that cannot be implemented using owned entity types)

Shortcomings in previous versions

- In EF Core 2.0, navigations to owned entity types cannot be declared in derived entity types unless the owned entities are explicitly mapped to a separate table from the owner hierarchy. This limitation has been removed in EF Core 2.1
- In EF Core 2.0 and 2.1 only reference navigations to owned types were supported. This limitation has been removed in EF Core 2.2

Query Types

12/7/2018 • 3 minutes to read • Edit Online

> **NOTE**
> This feature is new in EF Core 2.1

In addition to entity types, an EF Core model can contain *query types*, which can be used to carry out database queries against data that isn't mapped to entity types.

Compare query types to entity types

Query types are like entity types in that they:

- Can be added to the model either in `OnModelCreating` or via a "set" property on a derived *DbContext*.
- Support many of the same mapping capabilities, like inheritance mapping and navigation properties. On relational stores, they can configure the target database objects and columns via fluent API methods or data annotations.

However, they are different from entity types in that they:

- Do not require a key to be defined.
- Are never tracked for changes on the *DbContext* and therefore are never inserted, updated or deleted on the database.
- Are never discovered by convention.
- Only support a subset of navigation mapping capabilities - Specifically:
  - They may never act as the principal end of a relationship.
  - They can only contain reference navigation properties pointing to entities.
  - Entities cannot contain navigation properties to query types.
- Are addressed on the *ModelBuilder* using the `Query` method rather than the `Entity` method.
- Are mapped on the *DbContext* through properties of type `DbQuery<T>` rather than `DbSet<T>`
- Are mapped to database objects using the `ToView` method, rather than `ToTable`.
- May be mapped to a *defining query* - A defining query is a secondary query declared in the model that acts a data source for a query type.

Usage scenarios

Some of the main usage scenarios for query types are:

- Serving as the return type for ad hoc `FromSql()` queries.
- Mapping to database views.
- Mapping to tables that do not have a primary key defined.
- Mapping to queries defined in the model.

Mapping to database objects

Mapping a query type to a database object is achieved using the `ToView` fluent API. From the perspective of EF Core, the database object specified in this method is a *view*, meaning that it is treated as a read-only query source and cannot be the target of update, insert or delete operations. However, this does not mean that the database object is actually required to be a database view - It can alternatively be a database table that will be treated as read-only. Conversely, for entity types, EF Core assumes that a database object specified in the `ToTable` method can be treated as a *table*, meaning that it can be used as a query source but also targeted by update, delete and insert operations. In fact, you can specify the name of a database view in `ToTable` and everything should work fine as long as the view is configured to be updatable on the database.

Example

The following example shows how to use Query Type to query a database view.

> **TIP**
>
> You can view this article's sample on GitHub.

First, we define a simple Blog and Post model:

```
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; }
}
```

Next, we define a simple database view that will allow us to query the number of posts associated with each blog:

```
db.Database.ExecuteSqlCommand(
    @"CREATE VIEW View_BlogPostCounts AS
        SELECT b.Name, Count(p.PostId) as PostCount
        FROM Blogs b
        JOIN Posts p on p.BlogId = b.BlogId
        GROUP BY b.Name");
```

Next, we define a class to hold the result from the database view:

```
public class BlogPostsCount
{
    public string BlogName { get; set; }
    public int PostCount { get; set; }
}
```

Next, we configure the query type in *OnModelCreating* using the `modelBuilder.Query<T>` API. We use standard fluent configuration APIs to configure the mapping for the Query Type:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Query<BlogPostsCount>().ToView("View_BlogPostCounts")
        .Property(v => v.BlogName).HasColumnName("Name");
}
```

Finally, we can query the database view in the standard way:

```
var postCounts = db.BlogPostCounts.ToList();

foreach (var postCount in postCounts)
{
    Console.WriteLine($"{postCount.BlogName} has {postCount.PostCount} posts.");
    Console.WriteLine();
}
```

> **TIP**
>
> Note we have also defined a context level query property (DbQuery) to act as a root for queries against this type.

Alternating between multiple models with the same DbContext type

8/27/2018 • 2 minutes to read • Edit Online

The model built in `OnModelCreating` could use a property on the context to change how the model is built. For example it could be used to exclude a certain property:

```
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

However if you tried doing the above without additional changes you would get the same model every time a new context is created for any value of `IgnoreIntProperty`. This is caused by the model caching mechanism EF uses to improve the performance by only invoking `OnModelCreating` once and caching the model.

By default EF assumes that for any given context type the model will be the same. To accomplish this the default implementation of `IModelCacheKeyFactory` returns a key that just contains the context type. To change this you need to replace the `IModelCacheKeyFactory` service. The new implementation needs to return an object that can be compared to other model keys using the `Equals` method that takes into account all the variables that affect the model:

```
public class DynamicModelCacheKeyFactory : IModelCacheKeyFactory
{
    public object Create(DbContext context)
    {
        if (context is DynamicContext dynamicContext)
        {
            return (context.GetType(), dynamicContext.IgnoreIntProperty);
        }
        return context.GetType();
    }
}
```

Spatial Data

11/16/2018 • 7 minutes to read • Edit Online

NOTE

This feature is new in EF Core 2.2.

Spatial data represents the physical location and the shape of objects. Many databases provide support for this type of data so it can be indexed and queried alongside other data. Common scenarios include querying for objects within a given distance from a location, or selecting the object whose border contains a given location. EF Core supports mapping to spatial data types using the NetTopologySuite spatial library.

Installing

In order to use spatial data with EF Core, you need to install the appropriate supporting NuGet package. Which package you need to install depends on the provider you're using.

EF CORE PROVIDER	SPATIAL NUGET PACKAGE
Microsoft.EntityFrameworkCore.SqlServer	Microsoft.EntityFrameworkCore.SqlServer.NetTopologySuite
Microsoft.EntityFrameworkCore.Sqlite	Microsoft.EntityFrameworkCore.Sqlite.NetTopologySuite
Microsoft.EntityFrameworkCore.InMemory	NetTopologySuite
Npgsql.EntityFrameworkCore.PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL.NetTopologySuite

Reverse engineering

The spatial NuGet packages also enable reverse engineering models with spatial properties, but you need to install the package ***before*** running `Scaffold-DbContext` or `dotnet ef dbcontext scaffold`. If you don't, you'll receive warnings about not finding type mappings for the columns and the columns will be skipped.

NetTopologySuite (NTS)

NetTopologySuite is a spatial library for .NET. EF Core enables mapping to spatial data types in the database by using NTS types in your model.

To enable mapping to spatial types via NTS, call the UseNetTopologySuite method on the provider's DbContext options builder. For example, with SQL Server you'd call it like this.

```
optionsBuilder.UseSqlServer(
    @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=WideWorldImporters",
    x => x.UseNetTopologySuite());
```

There are several spatial data types. Which type you use depends on the types of shapes you want to allow. Here is the hierarchy of NTS types that you can use for properties in your model. They're located within the `NetTopologySuite.Geometries` namespace. Corresponding interfaces in the GeoAPI package (`GeoAPI.Geometries` namespace) can also be used.

- Geometry
  - Point
  - LineString
  - Polygon
  - GeometryCollection
    - MultiPoint
    - MultiLineString
    - MultiPolygon

> **WARNING**
>
> CircularString, CompoundCurve, and CurePolygon aren't supported by NTS.

Using the base Geometry type allows any type of shape to be specified by the property.

The following entity classes could be used to map to tables in the Wide World Importers sample database.

```
[Table("Cities", Schema = "Application"))]
class City
{
    public int CityID { get; set; }

    public string CityName { get; set; }

    public IPoint Location { get; set; }
}

[Table("Countries", Schema = "Application"))]
class Country
{
    public int CountryID { get; set; }

    public string CountryName { get; set; }

    // Database includes both Polygon and MultiPolygon values
    public IGeometry Border { get; set; }
}
```

Creating values

You can use constructors to create geometry objects; however, NTS recommends using a geometry factory instead. This lets you specify a default SRID (the spatial reference system used by the coordinates) and gives you control over more advanced things like the precision model (used during calculations) and the coordinate sequence (determines which ordinates--dimensions and measures--are available).

```
var geometryFactory = NtsGeometryServices.Instance.CreateGeometryFactory(srid: 4326);
var currentLocation = geometryFactory.CreatePoint(-122.121512, 47.6739882);
```

> **NOTE**
>
> 4326 refers to WGS 84, a standard used in GPS and other geographic systems.

Longitude and Latitude

Coordinates in NTS are in terms of X and Y values. To represent longitude and latitude, use X for longitude and Y for latitude. Note that this is **backwards** from the `latitude, longitude` format in which you typically see these values.

SRID Ignored during client operations

NTS ignores SRID values during operations. It assumes a planar coordinate system. This means that if you specify coordinates in terms of longitude and latitude, some client-evaluated values like distance, length, and area will be in degrees, not meters. For more meaningful values, you first need to project the coordinates to another coordinate system using a library like ProjNet4GeoAPI before calculating these values.

If an operation is server-evaluated by EF Core via SQL, the result's unit will be determined by the database.

Here is an example of using ProjNet4GeoAPI to calculate the distance between two cities.

```
static class GeometryExtensions
{
    static readonly IGeometryServices _geometryServices = NtsGeometryServices.Instance;
    static readonly ICoordinateSystemServices _coordinateSystemServices
        = new CoordinateSystemServices(
            new CoordinateSystemFactory(),
            new CoordinateTransformationFactory(),
            new Dictionary<int, string>
            {
                // Coordinate systems:

                // (3857 and 4326 included automatically)

                // This coordinate system covers the area of our data.
                // Different data requires a different coordinate system.
                [2855] =
                @"
                    PROJCS[""NAD83(HARN) / Washington North"",
                        GEOGCS[""NAD83(HARN)"",
                            DATUM[""NAD83_High_Accuracy_Regional_Network"",
                                SPHEROID[""GRS 1980"",6378137,298.257222101,
                                    AUTHORITY[""EPSG"",""7019""]],
                                AUTHORITY[""EPSG"",""6152""]],
                            PRIMEM[""Greenwich"",0,
                                AUTHORITY[""EPSG"",""8901""]],
                            UNIT[""degree"",0.01745329251994328,
                                AUTHORITY[""EPSG"",""9122""]],
                            AUTHORITY[""EPSG"",""4152""]],
                        PROJECTION[""Lambert_Conformal_Conic_2SP""],
                        PARAMETER[""standard_parallel_1"",48.73333333333333],
                        PARAMETER[""standard_parallel_2"",47.5],
                        PARAMETER[""latitude_of_origin"",47],
                        PARAMETER[""central_meridian"",-120.8333333333333],
                        PARAMETER[""false_easting"",500000],
                        PARAMETER[""false_northing"",0],
                        UNIT[""metre"",1,
                            AUTHORITY[""EPSG"",""9001""]],
                        AUTHORITY[""EPSG"",""2855""]]
                "
            });

    public static IGeometry ProjectTo(this IGeometry geometry, int srid)
    {
        var geometryFactory = _geometryServices.CreateGeometryFactory(srid);
        var transformation = _coordinateSystemServices.CreateTransformation(geometry.SRID, srid);

        return GeometryTransform.TransformGeometry(
            geometryFactory,
            geometry,
            transformation.MathTransform);
    }
}
```

```
var seattle = new Point(-122.333056, 47.609722) { SRID = 4326 };
var redmond = new Point(-122.123889, 47.669444) { SRID = 4326 };

var distance = seattle.ProjectTo(2855).Distance(redmond.ProjectTo(2855));
```

Querying Data

In LINQ, the NTS methods and properties available as database functions will be translated to SQL. For example, the Distance and Contains methods are translated in the following queries. The table at the end of this article shows which members are supported by various EF Core providers.

```
var nearestCity = db.Cities
    .OrderBy(c => c.Location.Distance(currentLocation))
    .FirstOrDefault();

var currentCountry = db.Countries
    .FirstOrDefault(c => c.Border.Contains(currentLocation));
```

SQL Server

If you're using SQL Server, there are some additional things you should be aware of.

Geography or geometry

By default, spatial properties are mapped to `geography` columns in SQL Server. To use `geometry`, configure the column type in your model.

Geography polygon rings

When using the `geography` column type, SQL Server imposes additional requirements on the exterior ring (or shell) and interior rings (or holes). The exterior ring must be oriented counterclockwise and the interior rings clockwise. NTS validates this before sending values to the database.

FullGlobe

SQL Server has a non-standard geometry type to represent the full globe when using the `geography` column type. It also has a way to represent polygons based on the full globe (without an exterior ring). Neither of these are supported by NTS.

> **WARNING**
>
> FullGlobe and polygons based on it aren't supported by NTS.

SQLite

Here is some additional information for those using SQLite.

Installing SpatiaLite

On Windows, the native mod_spatialite library is distributed as a NuGet package dependency. Other platforms need to install it separately. This is typically done using a software package manager. For example, you can use APT on Ubuntu and Homebrew on MacOS.

```
# Ubuntu
apt-get install libsqlite3-mod-spatialite

# macOS
brew install libspatialite
```

Configuring SRID

In SpatiaLite, columns need to specify an SRID per column. The default SRID is `0`. Specify a different SRID using the ForSqliteHasSrid method.

```
modelBuilder.Entity<City>().Property(c => c.Location)
    .ForSqliteHasSrid(4326);
```

Dimension

Similar to SRID, a column's dimension (or ordinates) is also specified as part of the column. The default ordinates are X and Y. Enable additional ordinates (Z and M) using the ForSqliteHasDimension method.

```
modelBuilder.Entity<City>().Property(c => c.Location)
    .ForSqliteHasDimension(Ordinates.XYZ);
```

Translated Operations

This table shows which NTS members are translated into SQL by each EF Core provider.

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQLITE	NPGSQL
Geometry.Area	✔	✔	✔	✔
Geometry.AsBinary()	✔	✔	✔	✔
Geometry.AsText()	✔	✔	✔	✔
Geometry.Boundary	✔		✔	✔
Geometry.Buffer(double)	✔	✔	✔	✔
Geometry.Buffer(double, int)			✔	
Geometry.Centroid	✔		✔	✔
Geometry.Contains(Geometry)	✔	✔	✔	✔
Geometry.ConvexHull()	✔	✔	✔	✔
Geometry.CoveredBy(Geometry)			✔	✔

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQLITE	NPGSQL
Geometry.Covers(Geometry)			✔	✔
Geometry.Crosses(Geometry)	✔		✔	✔
Geometry.Difference(Geometry)	✔	✔	✔	✔
Geometry.Dimension	✔	✔	✔	✔
Geometry.Disjoint(Geometry)	✔	✔	✔	✔
Geometry.Distance(Geometry)	✔	✔	✔	✔
Geometry.Envelope	✔		✔	✔
Geometry.EqualsExact(Geometry)				✔
Geometry.EqualsTopologically(Geometry)	✔	✔	✔	✔
Geometry.GeometryType	✔	✔	✔	✔
Geometry.GetGeometryN(int)	✔		✔	✔
Geometry.InteriorPoint	✔		✔	
Geometry.Intersection(Geometry)	✔	✔	✔	✔
Geometry.Intersects(Geometry)	✔	✔	✔	✔
Geometry.IsEmpty	✔	✔	✔	✔
Geometry.IsSimple	✔		✔	✔
Geometry.IsValid	✔	✔	✔	✔
Geometry.IsWithinDistance(Geometry, double)	✔		✔	
Geometry.Length	✔	✔	✔	✔

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQLITE	NPGSQL
Geometry.NumGeometries	✔	✔	✔	✔
Geometry.NumPoints	✔	✔	✔	✔
Geometry.OgcGeometryType	✔	✔	✔	
Geometry.Overlaps(Geometry)	✔	✔	✔	✔
Geometry.PointOnSurface	✔		✔	✔
Geometry.Relate(Geometry, string)	✔		✔	✔
Geometry.Reverse()			✔	✔
Geometry.SRID	✔	✔	✔	✔
Geometry.SymmetricDifference(Geometry)	✔	✔	✔	✔
Geometry.ToBinary()	✔	✔	✔	✔
Geometry.ToText()	✔	✔	✔	✔
Geometry.Touches(Geometry)	✔		✔	✔
Geometry.Union()			✔	
Geometry.Union(Geometry)	✔	✔	✔	✔
Geometry.Within(Geometry)	✔	✔	✔	✔
GeometryCollection.Count	✔	✔	✔	✔
GeometryCollection[int]	✔	✔	✔	✔
LineString.Count	✔	✔	✔	✔
LineString.EndPoint	✔	✔	✔	✔
LineString.GetPointN(int)	✔	✔	✔	✔

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQLITE	NPGSQL
LineString.IsClosed	✔	✔	✔	✔
LineString.IsRing	✔		✔	✔
LineString.StartPoint	✔	✔	✔	✔
MultiLineString.IsClosed	✔	✔	✔	✔
Point.M	✔	✔	✔	✔
Point.X	✔	✔	✔	✔
Point.Y	✔	✔	✔	✔
Point.Z	✔	✔	✔	✔
Polygon.ExteriorRing	✔	✔	✔	✔
Polygon.GetInteriorRingN(int)	✔	✔	✔	✔
Polygon.NumInteriorRings	✔	✔	✔	✔

Additional resources

- Spatial Data in SQL Server
- SpatiaLite Homepage
- Npgsql Spatial Documentation
- PostGIS Documentation

Relational Database Modeling

8/27/2018 • 2 minutes to read • Edit Online

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

Table Mapping

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

Table mapping identifies which table data should be queried from and saved to in the database.

Conventions

By convention, each entity will be set up to map to a table with the same name as the `DbSet<TEntity>` property that exposes the entity on the derived context. If no `DbSet<TEntity>` is included for the given entity, the class name is used.

Data Annotations

You can use Data Annotations to configure the table that a type maps to.

```
using System.ComponentModel.DataAnnotations.Schema;
```

```
[Table("blogs")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

You can also specify a schema that the table belongs to.

```
[Table("blogs", Schema = "blogging")]
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Fluent API

You can use the Fluent API to configure the table that a type maps to.

```
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; }
}
```

You can also specify a schema that the table belongs to.

```
        modelBuilder.Entity<Blog>()
            .ToTable("blogs", schema: "blogging");
```

Column Mapping

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

Column mapping identifies which column data should be queried from and saved to in the database.

Conventions

By convention, each property will be set up to map to a column with the same name as the property.

Data Annotations

You can use Data Annotations to configure the column to which a property is mapped.

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

Fluent API

You can use the Fluent API to configure the column to which a property is mapped.

```
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; }
}
```

Data Types

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

Data type refers to the database specific type of the column to which a property is mapped.

Conventions

By convention, the database provider selects a data type based on the CLR type of the property. It also takes into account other metadata, such as the configured Maximum Length, whether the property is part of a primary key, etc.

For example, SQL Server uses `datetime2(7)` for `DateTime` properties, and `nvarchar(max)` for `string` properties (or `nvarchar(450)` for `string` properties that are used as a key).

Data Annotations

You can use Data Annotations to specify an exact data type for a column.

For example the following code configures `Url` as a non-unicode string with maximum length of `200` and `Rating` as decimal with precision of `5` and scale of `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; }
}
```

Fluent API

You can also use the Fluent API to specify the same data types for the columns.

```
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; }
}
```

Primary Keys

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

A primary key constraint is introduced for the key of each entity type.

Conventions

By convention, the primary key in the database will be named `PK_<type name>`.

Data Annotations

No relational database specific aspects of a primary key can be configured using Data Annotations.

Fluent API

You can use the Fluent API to configure the name of the primary key constraint in the database.

```
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; }
}
```

Default Schema

8/27/2018 • 2 minutes to read • Edit Online

> **NOTE**
> The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

The default schema is the database schema that objects will be created in if a schema is not explicitly configured for that object.

Conventions

By convention, the database provider will choose the most appropriate default schema. For example, Microsoft SQL Server will use the `dbo` schema and SQLite will not use a schema (since schemas are not supported in SQLite).

Data Annotations

You can not set the default schema using Data Annotations.

Fluent API

You can use the Fluent API to specify a default schema.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasDefaultSchema("blogging");
    }
}
```

Computed Columns

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

A computed column is a column whose value is calculated in the database. A computed column can use other columns in the table to calculate its value.

Conventions

By convention, computed columns are not created in the model.

Data Annotations

Computed columns can not be configured with Data Annotations.

Fluent API

You can use the Fluent API to specify that a property should map to a computed column.

```
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; }
}
```

Sequences

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

A sequence generates a sequential numeric values in the database. Sequences are not associated with a specific table.

Conventions

By convention, sequences are not introduced in to the model.

Data Annotations

You can not configure a sequence using Data Annotations.

Fluent API

You can use the Fluent API to create a sequence in the model.

```
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; }
}
```

You can also configure additional aspect of the sequence, such as its schema, start value, and increment.

```
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);
    }
}
```

Once a sequence is introduced, you can use it to generate values for properties in your model. For example, you can use Default Values to insert the next value from the sequence.

```
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; }
}
```

Default Values

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

The default value of a column is the value that will be inserted if a new row is inserted but no value is specified for the column.

Conventions

By convention, a default value is not configured.

Data Annotations

You can not set a default value using Data Annotations.

Fluent API

You can use the Fluent API to specify the default value for a property.

```
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; }
}
```

You can also specify a SQL fragment that is used to calculate the default value.

```
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; }
}
```

Indexes

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

An index in a relational database maps to the same concept as an index in the core of Entity Framework.

Conventions

By convention, indexes are named `IX_<type name>_<property name>`. For composite indexes `<property name>` becomes an underscore separated list of property names.

Data Annotations

Indexes can not be configured using Data Annotations.

Fluent API

You can use the Fluent API to configure the name of an index.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .HasName("Index_Url");
    }
}

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

You can also specify a filter.

```
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; }
}
```

When using the SQL Server provider EF adds a 'IS NOT NULL' filter for all nullable columns that are part of a unique index. To override this convention you can supply a `null` value.

```
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; }
}
```

Foreign Key Constraints

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

A foreign key constraint is introduced for each relationship in the model.

Conventions

By convention, foreign key constraints are named `FK_<dependent type name>_<principal type name>_<foreign key property name>`. For composite foreign keys `<foreign key property name>` becomes an underscore separated list of foreign key property names.

Data Annotations

Foreign key constraint names cannot be configured using data annotations.

Fluent API

You can use the Fluent API to configure the foreign key constraint name for a relationship.

```
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; }
}
```

Alternate Keys (Unique Constraints)

8/27/2018 • 2 minutes to read • Edit Online

NOTE

The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

A unique constraint is introduced for each alternate key in the model.

Conventions

By convention, the index and constraint that are introduced for an alternate key will be named `AK_<type name>_<property name>`. For composite alternate keys `<property name>` becomes an underscore separated list of property names.

Data Annotations

Unique constraints can not be configured using Data Annotations.

Fluent API

You can use the Fluent API to configure the index and constraint name for an alternate key.

```
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; }
}
```

Inheritance (Relational Database)

8/27/2018 • 2 minutes to read • Edit Online

> **NOTE**
>
> The configuration in this section is applicable to relational databases in general. The extension methods shown here will become available when you install a relational database provider (due to the shared *Microsoft.EntityFrameworkCore.Relational* package).

Inheritance in the EF model is used to control how inheritance in the entity classes is represented in the database.

> **NOTE**
>
> Currently, only the table-per-hierarchy (TPH) pattern is implemented in EF Core. Other common patterns like table-per-type (TPT) and table-per-concrete-type (TPC) are not yet available.

Conventions

By convention, inheritance will be mapped using the table-per-hierarchy (TPH) pattern. TPH uses a single table to store the data for all types in the hierarchy. A discriminator column is used to identify which type each row represents.

EF Core will only setup inheritance if two or more inherited types are explicitly included in the model (see Inheritance for more details).

Below is an example showing a simple inheritance scenario and the data stored in a relational database table using the TPH pattern. The *Discriminator* column identifies which type of *Blog* is stored in each row.

```
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

Data Annotations

You cannot use Data Annotations to configure inheritance.

Fluent API

You can use the Fluent API to configure the name and type of the discriminator column and the values that are used to identify each type in the hierarchy.

```
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; }
}
```

Configuring the discriminator property

In the examples above, the discriminator is created as a shadow property on the base entity of the hierarchy. Since it is a property in the model, it can be configured just like other properties. For example, to set the max length when the default, by-convention discriminator is being used:

```
modelBuilder.Entity<Blog>()
    .Property("Discriminator")
    .HasMaxLength(200);
```

The discriminator can also be mapped to an actual CLR property in your entity. For example:

```
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; }
}
```

Combining these two things together it is possible to both map the discriminator to a real property and configure it:

```
modelBuilder.Entity<Blog>(b =>
{
    b.HasDiscriminator<string>("BlogType");

    b.Property(e => e.BlogType)
        .HasMaxLength(200)
        .HasColumnName("blog_type");
});
```

Managing Database Schemas

8/27/2018 • 2 minutes to read • Edit Online

EF Core provides two primary ways of keeping your EF Core model and database schema in sync. To choose between the two, decide whether your EF Core model or the database schema is the source of truth.

If you want your EF Core model to be the source of truth, use Migrations. As you make changes to your EF Core model, this approach incrementally applies the corresponding schema changes to your database so that it remains compatible with your EF Core model.

Use Reverse Engineering if you want your database schema to be the source of truth. This approach allows you to scaffold a DbContext and the entity type classes by reverse engineering your database schema into an EF Core model.

> **NOTE**
> The create and drop APIs can also create the database schema from your EF Core model. However, they are primarily for testing, prototyping, and other scenarios where dropping the database is acceptable.

Migrations

10/6/2018 • 5 minutes to read • Edit Online

A data model changes during development and gets out of sync with the database. You can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data. The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

Migrations includes command-line tools and APIs that help with the following tasks:

- Create a migration. Generate code that can update the database to sync it with a set of model changes.
- Update the database. Apply pending migrations to update the database schema.
- Customize migration code. Sometimes the generated code needs to be modified or supplemented.
- Remove a migration. Delete the generated code.
- Revert a migration. Undo the database changes.
- Generate SQL scripts. You might need a script to update a production database or to troubleshoot migration code.
- Apply migrations at runtime. When design-time updates and running scripts aren't the best options, call the `Migrate()` method.

Install the tools

Install the command-line tools:

- For Visual Studio, we recommend the Package Manager Console tools.
- For other development environments, choose the .NET Core CLI tools.

Create a migration

After you've defined your initial model, it's time to create the database. To add an initial migration, run the following command.

```
Add-Migration InitialCreate
```

```
dotnet ef migrations add InitialCreate
```

Three files are added to your project under the **Migrations** directory:

- **00000000000000_InitialCreate.cs**--The main migrations file. Contains the operations necessary to apply the migration (in `Up()`) and to revert it (in `Down()`).
- **00000000000000_InitialCreate.Designer.cs**--The migrations metadata file. Contains information used by EF.
- **MyContextModelSnapshot.cs**--A snapshot of your current model. Used to determine what changed when adding the next migration.

The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

> **TIP**
>
> You are free to move Migrations files and change their namespace. New migrations are created as siblings of the last migration.

Update the database

Next, apply the migration to the database to create the schema.

```
Update-Database
```

```
dotnet ef database update
```

Customize migration code

After making changes to your EF Core model, the database schema might be out of sync. To bring it up to date, add another migration. The migration name can be used like a commit message in a version control system. For example, you might choose a name like *AddProductReviews* if the change is a new entity class for reviews.

```
Add-Migration AddProductReviews
```

```
dotnet ef migrations add AddProductReviews
```

Once the migration is scaffolded (code generated for it), review the code for accuracy and add, remove or modify any operations required to apply it correctly.

For example, a migration might contain the following operations:

```
migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");

migrationBuilder.AddColumn<string>(
    name: "Name",
    table: "Customer",
    nullable: true);
```

While these operations make the database schema compatible, they don't preserve the existing customer names. To make it better, rewrite it as follows.

```
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

The migration scaffolding process warns when an operation might result in data loss (like dropping a column). If you see that warning, be especially sure to review the migrations code for accuracy.

Apply the migration to the database using the appropriate command.

```
Update-Database
```

```
dotnet ef database update
```

Empty migrations

Sometimes it's useful to add a migration without making any model changes. In this case, adding a new migration creates code files with empty classes. You can customize this migration to perform operations that don't directly relate to the EF Core model. Some things you might want to manage this way are:

- Full-Text Search
- Functions
- Stored procedures
- Triggers
- Views

Remove a migration

Sometimes you add a migration and realize you need to make additional changes to your EF Core model before applying it. To remove the last migration, use this command.

```
Remove-Migration
```

```
dotnet ef migrations remove
```

After removing the migration, you can make the additional model changes and add it again.

Revert a migration

If you already applied a migration (or several migrations) to the database but need to revert it, you can use the same command to apply migrations, but specify the name of the migration you want to roll back to.

```
Update-Database LastGoodMigration
```

```
dotnet ef database update LastGoodMigration
```

Generate SQL scripts

When debugging your migrations or deploying them to a production database, it's useful to generate a SQL script. The script can then be further reviewed for accuracy and tuned to fit the needs of a production database. The script can also be used in conjunction with a deployment technology. The basic command is as follows.

```
Script-Migration
```

```
dotnet ef migrations script
```

There are several options to this command.

The **from** migration should be the last migration applied to the database before running the script. If no migrations have been applied, specify `0` (this is the default).

The **to** migration is the last migration that will be applied to the database after running the script. This defaults to the last migration in your project.

An **idempotent** script can optionally be generated. This script only applies migrations if they haven't already been applied to the database. This is useful if you don't exactly know what the last migration applied to the database was or if you are deploying to multiple databases that may each be at a different migration.

Apply migrations at runtime

Some apps may want to apply migrations at runtime during startup or first run. Do this using the `Migrate()` method.

This method builds on top of the `IMigrator` service, which can be used for more advanced scenarios. Use `DbContext.GetService<IMigrator>()` to access it.

```
myDbContext.Database.Migrate();
```

WARNING

- This approach isn't for everyone. While it's great for apps with a local database, most applications will require more robust deployment strategy like generating SQL scripts.
- Don't call `EnsureCreated()` before `Migrate()`. `EnsureCreated()` bypasses Migrations to create the schema, which causes `Migrate()` to fail.

Next steps

For more information, see Entity Framework Core tools reference - EF Core.

Migrations in Team Environments

8/27/2018 • 2 minutes to read • Edit Online

When working with Migrations in team environments, pay extra attention to the model snapshot file. This file can tell you if your teammate's migration merges cleanly with yours or if you need to resolve a conflict by re-creating your migration before sharing it.

Merging

When you merge migrations from your teammates, you may get conflicts in your model snapshot file. If both changes are unrelated, the merge is trivial and the two migrations can coexist. For example, you may get a merge conflict in the customer entity type configuration that looks like this:

```
<<<<<<< Mine
b.Property<bool>("Deactivated");
=======
b.Property<int>("LoyaltyPoints");
>>>>>>> Theirs
```

Since both of these properties need to exist in the final model, complete the merge by adding both properties. In many cases, your version control system may automatically merge such changes for you.

```
b.Property<bool>("Deactivated");
b.Property<int>("LoyaltyPoints");
```

In these cases, your migration and your teammate's migration are independent of each other. Since either of them could be applied first, you don't need to make any additional changes to your migration before sharing it with your team.

Resolving conflicts

Sometimes you encounter a true conflict when merging the model snapshot model. For example, you and your teammate may each have renamed the same property.

```
<<<<<<< Mine
b.Property<string>("Username");
=======
b.Property<string>("Alias");
>>>>>>> Theirs
```

If you encounter this kind of conflict, resolve it by re-creating your migration. Follow these steps:

1. Abort the merge and rollback to your working directory before the merge
2. Remove your migration (but keep your model changes)
3. Merge your teammate's changes into your working directory
4. Re-add your migration

After doing this, the two migrations can be applied in the correct order. Their migration is applied first, renaming the column to *Alias*, thereafter your migration renames it to *Username*.

Your migration can safely be shared with the rest of the team.

Custom Migrations Operations

11/6/2018 • 2 minutes to read • Edit Online

The MigrationBuilder API allows you to perform many different kinds of operations during a migration, but it's far from exhaustive. However, the API is also extensible allowing you to define your own operations. There are two ways to extend the API: Using the `Sql()` method, or by defining custom `MigrationOperation` objects.

To illustrate, let's look at implementing an operation that creates a database user using each approach. In our migrations, we want to enable writing the following code:

```
migrationBuilder.CreateUser("SQLUser1", "Password");
```

Using MigrationBuilder.Sql()

The easiest way to implement a custom operation is to define an extension method that calls `MigrationBuilder.Sql()`. Here is an example that generates the appropriate Transact-SQL.

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
    => migrationBuilder.Sql($"CREATE USER {name} WITH PASSWORD '{password}';");
```

If your migrations need to support multiple database providers, you can use the `MigrationBuilder.ActiveProvider` property. Here's an example supporting both Microsoft SQL Server and 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;
}
```

This approach only works if you know every provider where your custom operation will be applied.

Using a MigrationOperation

To decouple the custom operation from the SQL, you can define your own `MigrationOperation` to represent it. The operation is then passed to the provider so it can determine the appropriate SQL to generate.

```
class CreateUserOperation : MigrationOperation
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

With this approach, the extension method just needs to add one of these operations to `MigrationBuilder.Operations`.

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
{
    migrationBuilder.Operations.Add(
        new CreateUserOperation
        {
            Name = name,
            Password = password
        });

    return migrationBuilder;
}
```

This approach requires each provider to know how to generate SQL for this operation in their `IMigrationsSqlGenerator` service. Here is an example overriding the SQL Server's generator to handle the new operation.

```
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();
    }
}
```

Replace the default migrations sql generator service with the updated one.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IMigrationsSqlGenerator, MyMigrationsSqlGenerator>();
```

Using a Separate Project

9/28/2018 • 2 minutes to read • Edit Online

You may want to store your migrations in a different assembly than the one containing your `DbContext`. You can also use this strategy to maintain multiple sets of migrations, for example, one for development and another for release-to-release upgrades.

To do this...

1. Create a new class library.
2. Add a reference to your DbContext assembly.
3. Move the migrations and model snapshot files to the class library.

   > **TIP**
   > If you have no existing migrations, generate one in the project containing the DbContext then move it. This is important because if the migrations assembly does not contain an existing migration, the Add-Migration command will be unable to find the DbContext.

4. Configure the migrations assembly:

   ```
   options.UseSqlServer(
       connectionString,
       x => x.MigrationsAssembly("MyApp.Migrations"));
   ```

5. Add a reference to your migrations assembly from the startup assembly.
   - If this causes a circular dependency, update the output path of the class library:

     ```
     <PropertyGroup>
       <OutputPath>..\MyStartupProject\bin\$(Configuration)\</OutputPath>
     </PropertyGroup>
     ```

If you did everything correctly, you should be able to add new migrations to the project.

```
Add-Migration NewMigration -Project MyApp.Migrations
```

```
dotnet ef migrations add NewMigration --project MyApp.Migrations
```

Migrations with Multiple Providers

9/13/2018 • 2 minutes to read • Edit Online

The EF Core Tools only scaffold migrations for the active provider. Sometimes, however, you may want to use more than one provider (for example Microsoft SQL Server and SQLite) with your DbContext. There are two ways to handle this with Migrations. You can maintain two sets of migrations--one for each provider--or merge them into a single set that can work on both.

Two migration sets

In the first approach, you generate two migrations for each model change.

One way to do this is to put each migration set in a separate assembly and manually switch the active provider (and migrations assembly) between adding the two migrations.

Another approach that makes working with the tools easier is to create a new type that derives from your DbContext and overrides the active provider. This type is used at design time when adding or applying migrations.

```
class MySqliteDbContext : MyDbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=my.db");
}
```

NOTE

Since each migration set uses its own DbContext types, this approach doesn't require using a separate migrations assembly.

When adding new migration, specify the context types.

```
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

You don't need to specify the output directory for subsequent migrations since they are created as siblings to the last one.

One migration set

If you don't like having two sets of migrations, you can manually combine them into a single set that can be applied to both providers.

Annotations can coexist since a provider ignores any annotations that it doesn't understand. For example, a primary key column that works with both Microsoft SQL Server and SQLite might look like this.

```
Id = table.Column<int>(nullable: false)
    .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn)
    .Annotation("Sqlite:Autoincrement", true),
```

If operations can only be applied on one provider (or they're differently between providers), use the `ActiveProvider` property to tell which provider is active.

```
if (migrationBuilder.ActiveProvider == "Microsoft.EntityFrameworkCore.SqlServer")
{
    migrationBuilder.CreateSequence(
        name: "EntityFrameworkHiLoSequence");
}
```

Custom Migrations History Table

9/13/2018 • 2 minutes to read • Edit Online

By default, EF Core keeps track of which migrations have been applied to the database by recording them in a table named `__EFMigrationsHistory`. For various reasons, you may want to customize this table to better suit your needs.

> **IMPORTANT**
> If you customize the Migrations history table *after* applying migrations, you are responsible for updating the existing table in the database.

Schema and table name

You can change the schema and table name using the `MigrationsHistoryTable()` method in `OnConfiguring()` (or `ConfigureServices()` on ASP.NET Core). Here is an example using the SQL Server EF Core provider.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlServer(
        connectionString,
        x => x.MigrationsHistoryTable("__MyMigrationsHistory", "mySchema"));
```

Other changes

To configure additional aspects of the table, override and replace the provider-specific `IHistoryRepository` service. Here is an example of changing the MigrationId column name to *Id* on SQL Server.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IHistoryRepository, MyHistoryRepository>();
```

> **WARNING**
> `SqlServerHistoryRepository` is inside an internal namespace and may change in future releases.

```
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");
    }
}
```

Create and Drop APIs

11/15/2018 • 2 minutes to read • Edit Online

The EnsureCreated and EnsureDeleted methods provide a lightweight alternative to Migrations for managing the database schema. These methods are useful in scenarios when the data is transient and can be dropped when the schema changes. For example during prototyping, in tests, or for local caches.

Some providers (especially non-relational ones) don't support Migrations. For these providers, EnsureCreated is often the easiest way to initialize the database schema.

> **WARNING**
> EnsureCreated and Migrations don't work well together. If you're using Migrations, don't use EnsureCreated to initialize the schema.

Transitioning from EnsureCreated to Migrations is not a seamless experience. The simplest way to do it is to drop the database and re-create it using Migrations. If you anticipate using migrations in the future, it's best to just start with Migrations instead of using EnsureCreated.

EnsureDeleted

The EnsureDeleted method will drop the database if it exists. If you don't have the appropriate permissions, an exception is thrown.

```
// Drop the database if it exists
dbContext.Database.EnsureDeleted();
```

EnsureCreated

EnsureCreated will create the database if it doesn't exist and initialize the database schema. If any tables exist (including tables for another DbContext class), the schema won't be initialized.

```
// Create the database if it doesn't exist
dbContext.Database.EnsureCreated();
```

> **TIP**
> Async versions of these methods are also available.

SQL Script

To get the SQL used by EnsureCreated, you can use the GenerateCreateScript method.

```
var sql = dbContext.Database.GenerateCreateScript();
```

Multiple DbContext classes

EnsureCreated only works when no tables are present in the database. If needed, you can write your own check to see if the schema needs to be initialized, and use the underlying IRelationalDatabaseCreator service to initialize the schema.

```
// TODO: Check whether the schema needs to be initialized

// Initialize the schema for this DbContext
var databaseCreator = dbContext.GetService<IRelationalDatabaseCreator>();
databaseCreator.CreateTables();
```

Reverse Engineering

11/15/2018 • 5 minutes to read • Edit Online

Reverse engineering is the process of scaffolding entity type classes and a DbContext class based on a database schema. It can be performed using the `Scaffold-DbContext` command of the EF Core Package Manager Console (PMC) tools or the `dotnet ef dbcontext scaffold` command of the .NET Command-line Interface (CLI) tools.

Installing

Before reverse engineering, you'll need to install either the PMC tools (Visual Studio only) or the CLI tools. See links for details.

You'll also need to install an appropriate database provider for the database schema you want to reverse engineer.

Connection string

The first argument to the command is a connection string to the database. The tools will use this connection string to read the database schema.

How you quote and escape the connection string depends on which shell you are using to execute the command. Refer to your shell's documentation for specifics. For example, PowerShell requires you to escape the `$` character, but not `\`.

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook'
Microsoft.EntityFrameworkCore.SqlServer
```

```
dotnet ef dbcontext scaffold "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook"
Microsoft.EntityFrameworkCore.SqlServer
```

Configuration and User Secrets

If you have an ASP.NET Core project, you can use the `Name=<connection-string>` syntax to read the connection string from configuration.

This works well with the Secret Manager tool to keep your database password separate from your codebase.

```
dotnet user-secrets set ConnectionStrings.Chinook "Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Chinook"
dotnet ef dbcontext scaffold Name=Chinook Microsoft.EntityFrameworkCore.SqlServer
```

Provider name

The second argument is the provider name. The provider name is typically the same as the provider's NuGet package name.

Specifying tables

All tables in the database schema are reverse engineered into entity types by default. You can limit which tables are reverse engineered by specifying schemas and tables.

The `-Schemas` parameter in PMC and the `--schema` option in the CLI can be used to include every table within a schema.

`-Tables` (PMC) and `--table` (CLI) can be used to include specific tables.

To include multiple tables in PMC, use an array.

```
Scaffold-DbContext ... -Tables Artist, Album
```

To include multiple tables in the CLI, specify the option multiple times.

```
dotnet ef dbcontext scaffold ... --table Artist --table Album
```

Preserving names

Table and column names are fixed up to better match the .NET naming conventions for types and properties by default. Specifying the `-UseDatabaseNames` switch in PMC or the `--use-database-names` option in the CLI will disable this behavior preserving the original database names as much as possible. Invalid .NET identifiers will still be fixed and synthesized names like navigation properties will still conform to .NET naming conventions.

Fluent API or Data Annotations

Entity types are configured using the Fluent API by default. Specify `-DataAnnotations` (PMC) or `--data-annotations` (CLI) to instead use data annotations when possible.

For example, using the Fluent API will scaffold the this.

```
entity.Property(e => e.Title)
    .IsRequired()
    .HasMaxLength(160);
```

While using Data Annotations will scaffold this.

```
[Required]
[StringLength(160)]
public string Title { get; set; }
```

DbContext name

The scaffolded DbContext class name will be the name of the database suffixed with *Context* by default. To specify a different one, use `-Context` in PMC and `--context` in the CLI.

Directories and namespaces

The entity classes and a DbContext class are scaffolded into the project's root directory and use the project's default namespace. You can specify the directory where classes are scaffolded using `-OutputDir` (PMC) or `--output-dir` (CLI). The namespace will be the root namespace plus the names of any subdirectories under the project's root directory.

You can also use `-ContextDir` (PMC) and `--context-dir` (CLI) to scaffold the DbContext class into a separate directory from the entity type classes.

```
Scaffold-DbContext ... -ContextDir Data -OutputDir Models
```

```
dotnet ef dbcontext scaffold ... --context-dir Data --output-dir Models
```

How it works

Reverse engineering starts by reading the database schema. It reads information about tables, columns, constraints, and indexes.

Next, it uses the schema information to create an EF Core model. Tables are used to create entity types; columns are used to create properties; and foreign keys are used to create relationships.

Finally, the model is used to generate code. The corresponding entity type classes, Fluent API, and data annotations are scaffolded in order to re-create the same model from your app.

What doesn't work

Not everything about a model can be represented using a database schema. For example, information about **inheritance hierarchies**, **owned types**, and **table splitting** are not present in the database schema. Because of this, these constructs will never be reverse engineered.

In addition, **some column types** may not be supported by the EF Core provider. These columns won't be included in the model.

EF Core requires every entity type to have a key. Tables, however, aren't required to specify a primary key.**Tables without a primary key** are currently not reverse engineered.

You can define **concurrency tokens** in an EF Core model to prevent two users from updating the same entity at the same time. Some databases have a special type to represent this type of column (for example, rowversion in SQL Server) in which case we can reverse engineer this information; however, other concurrency tokens will not be reverse engineered.

Customizing the model

The code generated by EF Core is your code. Feel free to change it. It will only be regenerated if you reverse engineer the same model again. The scaffolded code represents *one* model that can be used to access the database, but it's certainly not the *only* model that can be used.

Customize the entity type classes and DbContext class to fit your needs. For example, you may choose to rename types and properties, introduce inheritance hierarchies, or split a table into to multiple entities. You can also remove non-unique indexes, unused sequences and navigation properties, optional scalar properties, and constraint names from the model.

You can also add additional constructors, methods, properties, etc. using another partial class in a separate file. This approach works even when you intend to reverse engineer the model again.

Updating the model

After making changes to the database, you may need to update your EF Core model to reflect those changes. If the database changes are simple, it may be easiest just to manually make the changes to your EF Core model. For example, renaming a table or column, removing a column, or updating a column's type are trivial changes to make in code.

More significant changes, however, are not as easy make manually. One common workflow is to reverse engineer the model from the database again using `-Force` (PMC) or `--force` (CLI) to overwrite the existing model with an updated one.

Another commonly requested feature is the ability to update the model from the database while preserving customization like renames, type hierarchies, etc. Use issue #831 to track the progress of this feature.

> **WARNING**
> If you reverse engineer the model from the database again, any changes you've made to the files will be lost.

Querying Data

8/27/2018 • 2 minutes to read • Edit Online

Entity Framework Core uses Language Integrated Query (LINQ) to query data from the database. LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries based on your derived context and entity classes. A representation of the LINQ query is passed to the database provider, to be translated in database-specific query language (for example, SQL for a relational database). For more detailed information on how a query is processed, see How Query Works.

Basic Queries

8/27/2018 • 2 minutes to read • Edit Online

Learn how to load entities from the database using Language Integrated Query (LINQ).

TIP

You can view this article's sample on GitHub.

101 LINQ samples

This page shows a few examples to achieve common tasks with Entity Framework Core. For an extensive set of samples showing what is possible with LINQ, see 101 LINQ Samples.

Loading all data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

Loading a single entity

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Filtering

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Loading Related Data

8/27/2018 • 9 minutes to read • Edit Online

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three common O/RM patterns used to load related data.

- **Eager loading** means that the related data is loaded from the database as part of the initial query.
- **Explicit loading** means that the related data is explicitly loaded from the database at a later time.
- **Lazy loading** means that the related data is transparently loaded from the database when the navigation property is accessed.

> **TIP**
> You can view this article's sample on GitHub.

Eager loading

You can use the `Include` method to specify related data to be included in query results. In the following example, the blogs that are returned in the results will have their `Posts` property populated with the related posts.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

> **TIP**
> Entity Framework Core will automatically fix-up navigation properties to any other entities that were previously loaded into the context instance. So even if you don't explicitly include the data for a navigation property, the property may still be populated if some or all of the related entities were previously loaded.

You can include related data from multiple relationships in a single query.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
}
```

Including multiple levels

You can drill down through relationships to include multiple levels of related data using the `ThenInclude` method. The following example loads all blogs, their related posts, and the author of each post.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
        .ToList();
}
```

> **NOTE**
>
> Current versions of Visual Studio offer incorrect code completion options and can cause correct expressions to be flagged with syntax errors when using the `ThenInclude` method after a collection navigation property. This is a symptom of an IntelliSense bug tracked at https://github.com/dotnet/roslyn/issues/8237. It is safe to ignore these spurious syntax errors as long as the code is correct and can be compiled successfully.

You can chain multiple calls to `ThenInclude` to continue including further levels of related data.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
            .ThenInclude(post => post.Author)
                .ThenInclude(author => author.Photo)
        .ToList();
}
```

You can combine all of this to include related data from multiple levels and multiple roots in the same query.

```
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();
}
```

You may want to include multiple related entities for one of the entities that is being included. For example, when querying `Blog`s, you include `Posts` and then want to include both the `Author` and `Tags` of the `Posts`. To do this, you need to specify each include path starting at the root. For example, `Blog -> Posts -> Author` and `Blog -> Posts -> Tags`. This does not mean you will get redundant joins, in most cases EF will consolidate the joins when generating 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();
}
```

Include on derived types

You can include related data from navigations defined only on a derived type using `Include` and `ThenInclude`.

Given the following model:

```
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; }
}
```

Contents of `School` navigation of all People who are Students can be eagerly loaded using a number of patterns:

- using cast

  ```
  context.People.Include(person => ((Student)person).School).ToList()
  ```

- using `as` operator

  ```
  context.People.Include(person => (person as Student).School).ToList()
  ```

- using overload of `Include` that takes parameter of type `string`

  ```
  context.People.Include("Student").ToList()
  ```

Ignored includes

If you change the query so that it no longer returns instances of the entity type that the query began with, then the include operators are ignored.

In the following example, the include operators are based on the `Blog`, but then the `Select` operator is used to change the query to return an anonymous type. In this case, the include operators have no effect.

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Select(blog => new
        {
            Id = blog.BlogId,
            Url = blog.Url
        })
        .ToList();
}
```

By default, EF Core will log a warning when include operators are ignored. See Logging for more information on viewing logging output. You can change the behavior when an include operator is ignored to either throw or do nothing. This is done when setting up the options for your context - typically in `DbContext.OnConfiguring`, or in `Startup.cs` if you are using 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));
}
```

Explicit loading

NOTE

This feature was introduced in EF Core 1.1.

You can explicitly load a navigation property via the `DbContext.Entry(...)` API.

```
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();
}
```

You can also explicitly load a navigation property by executing a separate query that returns the related entities. If change tracking is enabled, then when loading an entity, EF Core will automatically set the navigation properties of the newly-loaded entitiy to refer to any entities already loaded, and set the navigation properties of the already-loaded entities to refer to the newly-loaded entity.

Querying related entities

You can also get a LINQ query that represents the contents of a navigation property.

This allows you to do things such as running an aggregate operator over the related entities without loading them into memory.

```
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();
}
```

You can also filter which related entities are loaded into memory.

```
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();
}
```

Lazy loading

> **NOTE**
>
> This feature was introduced in EF Core 2.1.

The simplest way to use lazy-loading is by installing the Microsoft.EntityFrameworkCore.Proxies package and enabling it with a call to `UseLazyLoadingProxies`. For example:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

Or when using AddDbContext:

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
          .UseSqlServer(myConnectionString));
```

EF Core will then enable lazy loading for any navigation property that can be overridden--that is, it must be `virtual` and on a class that can be inherited from. For example, in the following entities, the `Post.Blog` and `Blog.Posts` navigation properties will be lazy-loaded.

```
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; }
}
```

Lazy loading without proxies

Lazy-loading proxies work by injecting the `ILazyLoader` service into an entity, as described in Entity Type Constructors. For example:

```
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;
    }
}
```

This doesn't require entity types to be inherited from or navigation properties to be virtual, and allows entity instances created with `new` to lazy-load once attached to a context. However, it requires a reference to the `ILazyLoader` service, which is defined in the Microsoft.EntityFrameworkCore.Abstractions package. This package contains a minimal set of types so that there is very little impact in depending on it. However, to completely avoid depending on any EF Core packages in the entity types, it is possible to inject the `ILazyLoader.Load` method as a delegate. For example:

```csharp
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;
    }
}
```

The code above uses a `Load` extension method to make using the delegate a bit cleaner:

```
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

The constructor parameter for the lazy-loading delegate must be called "lazyLoader". Configuration to use a different name than this is planned for a future release.

Related data and serialization

Because EF Core will automatically fix-up navigation properties, you can end up with cycles in your object graph. For example, loading a blog and its related posts will result in a blog object that references a collection of posts. Each of those posts will have a reference back to the blog.

Some serialization frameworks do not allow such cycles. For example, Json.NET will throw the following exception if a cycle is encountered.

> Newtonsoft.Json.JsonSerializationException: Self referencing loop detected for property 'Blog' with type 'MyApplication.Models.Blog'.

If you are using ASP.NET Core, you can configure Json.NET to ignore cycles that it finds in the object graph. This is done in the `ConfigureServices(...)` method in `Startup.cs`.

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddMvc()
        .AddJsonOptions(
            options => options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );

    ...
}
```

Client vs. Server Evaluation

9/9/2018 • 2 minutes to read • Edit Online

Entity Framework Core supports parts of the query being evaluated on the client and parts of it being pushed to the database. It is up to the database provider to determine which parts of the query will be evaluated in the database.

> **TIP**
>
> You can view this article's sample on GitHub.

Client evaluation

In the following example a helper method is used to standardize URLs for blogs that are returned from a SQL Server database. Because the SQL Server provider has no insight into how this method is implemented, it is not possible to translate it into SQL. All other aspects of the query are evaluated in the database, but passing the returned `URL` through this method is performed on the client.

```
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;
}
```

Client evaluation performance issues

While client evaluation can be very useful, in some instances it can result in poor performance. Consider the following query, where the helper method is now used in a filter. Because this can't be performed in the database, all the data is pulled into memory and then the filter is applied on the client. Depending on the amount of data, and how much of that data is filtered out, this could result in poor performance.

```
var blogs = context.Blogs
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

Client evaluation logging

By default, EF Core will log a warning when client evaluation is performed. See Logging for more information on viewing logging output.

Optional behavior: throw an exception for client evaluation

You can change the behavior when client evaluation occurs to either throw or do nothing. This is done when setting up the options for your context - typically in `DbContext.OnConfiguring`, or in `Startup.cs` if you are using 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));
}
```

Tracking vs. No-Tracking Queries

8/27/2018 • 2 minutes to read • Edit Online

Tracking behavior controls whether or not Entity Framework Core will keep information about an entity instance in its change tracker. If an entity is tracked, any changes detected in the entity will be persisted to the database during `SaveChanges()`. Entity Framework Core will also fix-up navigation properties between entities that are obtained from a tracking query and entities that were previously loaded into the DbContext instance.

> **TIP**
>
> You can view this article's sample on GitHub.

Tracking queries

By default, queries that return entity types are tracking. This means you can make changes to those entity instances and have those changes persisted by `SaveChanges()`.

In the following example, the change to the blogs rating will be detected and persisted to the database during `SaveChanges()`.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
    blog.Rating = 5;
    context.SaveChanges();
}
```

No-tracking queries

No tracking queries are useful when the results are used in a read-only scenario. They are quicker to execute because there is no need to setup change tracking information.

You can swap an individual query to be no-tracking:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .AsNoTracking()
        .ToList();
}
```

You can also change the default tracking behavior at the context instance level:

```
using (var context = new BloggingContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var blogs = context.Blogs.ToList();
}
```

> **NOTE**
>
> No tracking queries still perform identity resolution within the excuting query. If the result set contains the same entity multiple times, the same instance of the entity class will be returned for each occurrence in the result set. However, weak references are used to keep track of entities that have already been returned. If a previous result with the same identity goes out of scope, and garbage collection runs, you may get a new entity instance. For more information, see How Query Works.

Tracking and projections

Even if the result type of the query isn't an entity type, if the result contains entity types they will still be tracked by default. In the following query, which returns an anonymous type, the instances of `Blog` in the result set will be tracked.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Blog = b,
                Posts = b.Posts.Count()
            });
}
```

If the result set does not contain any entity types, then no tracking is performed. In the following query, which returns an anonymous type with some of the values from the entity (but no instances of the actual entity type), there is no tracking performed.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Id = b.BlogId,
                Url = b.Url
            });
}
```

Raw SQL Queries

1/6/2019 • 3 minutes to read • Edit Online

Entity Framework Core allows you to drop down to raw SQL queries when working with a relational database. This can be useful if the query you want to perform can't be expressed using LINQ, or if using a LINQ query is resulting in inefficient SQL queries. Raw SQL queries can return entity types or, starting with EF Core 2.1, query types that are part of your model.

> **TIP**
>
> You can view this article's sample on GitHub.

Limitations

There are a few limitations to be aware of when using raw SQL queries:

- The SQL query must return data for all properties of the entity or query type.
- The column names in the result set must match the column names that properties are mapped to. Note this is different from EF6 where property/column mapping was ignored for raw SQL queries and result set column names had to match the property names.
- The SQL query cannot contain related data. However, in many cases you can compose on top of the query using the `Include` operator to return related data (see Including related data).
- `SELECT` statements passed to this method should generally be composable: If EF Core needs to evaluate additional query operators on the server (for example, to translate LINQ operators applied after `FromSql`), the supplied SQL will be treated as a subquery. This means that the SQL passed should not contain any characters or options that are not valid on a subquery, such as:
  - a trailing semicolon
  - On SQL Server, a trailing query-level hint (for example, `OPTION (HASH JOIN)`)
  - On SQL Server, an `ORDER BY` clause that is not accompanied of `TOP 100 PERCENT` in the `SELECT` clause
- SQL statements other than `SELECT` are recognized automatically as non-composable. As a consequence, the full results of stored procedures are always returned to the client and any LINQ operators applied after `FromSql` are evaluated in-memory.

Basic raw SQL queries

You can use the *FromSql* extension method to begin a LINQ query based on a raw SQL query.

```
var blogs = context.Blogs
    .FromSql("SELECT * FROM dbo.Blogs")
    .ToList();
```

Raw SQL queries can be used to execute a stored procedure.

```
var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

Passing parameters

As with any API that accepts SQL, it is important to parameterize any user input to protect against a SQL injection attack. You can include parameter placeholders in the SQL query string and then supply parameter values as additional arguments. Any parameter values you supply will automatically be converted to a `DbParameter`.

The following example passes a single parameter to a stored procedure. While this may look like `String.Format` syntax, the supplied value is wrapped in a parameter and the generated parameter name inserted where the `{0}` placeholder was specified.

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

This is the same query but using string interpolation syntax, which is supported in EF Core 2.0 and above:

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
    .ToList();
```

You can also construct a DbParameter and supply it as a parameter value. This allows you to use named parameters in the SQL query string.

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser @user", user)
    .ToList();
```

Composing with LINQ

If the SQL query can be composed on in the database, then you can compose on top of the initial raw SQL query using LINQ operators. SQL queries that can be composed on begin with the `SELECT` keyword.

The following example uses a raw SQL query that selects from a Table-Valued Function (TVF) and then composes on it using LINQ to perform filtering and sorting.

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```

Including related data

Composing with LINQ operators can be used to include related data in the query.

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Include(b => b.Posts)
    .ToList();
```

WARNING

Always use parameterization for raw SQL queries: APIs that accept a raw SQL string such as `FromSql` and `ExecuteSqlCommand` allow values to be easily passed as parameters. In addition to validating user input, always use parameterization for any values used in a raw SQL query/command. If you are using string concatenation to dynamically build any part of the query string then you are responsible for validating any input to protect against SQL injection attacks.

Asynchronous Queries

8/27/2018 • 2 minutes to read • Edit Online

Asynchronous queries avoid blocking a thread while the query is executed in the database. This can be useful to avoid freezing the UI of a thick-client application. Asynchronous operations can also increase throughput in a web application, where the thread can be freed up to service other requests while the database operation completes. For more information, see Asynchronous Programming in C#.

> **WARNING**
>
> EF Core does not support multiple parallel operations being run on the same context instance. You should always wait for an operation to complete before beginning the next operation. This is typically done by using the `await` keyword on each asynchronous operation.

Entity Framework Core provides a set of asynchronous extension methods that can be used as an alternative to the LINQ methods that cause a query to be executed and results returned. Examples include `ToListAsync()`, `ToArrayAsync()`, `SingleAsync()`, etc. There are not async versions of LINQ operators such as `Where(...)`, `OrderBy(...)`, etc. because these methods only build up the LINQ expression tree and do not cause the query to be executed in the database.

> **IMPORTANT**
>
> The EF Core async extension methods are defined in the `Microsoft.EntityFrameworkCore` namespace. This namespace must be imported for the methods to be available.

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

How Queries Work

9/27/2018 • 2 minutes to read • Edit Online

Entity Framework Core uses Language Integrated Query (LINQ) to query data from the database. LINQ allows you to use C# (or your .NET language of choice) to write strongly typed queries based on your derived context and entity classes.

The life of a query

The following is a high level overview of the process each query goes through.

1. The LINQ query is processed by Entity Framework Core to build a representation that is ready to be processed by the database provider
   a. The result is cached so that this processing does not need to be done every time the query is executed
2. The result is passed to the database provider
   a. The database provider identifies which parts of the query can be evaluated in the database
   b. These parts of the query are translated to database specific query language (for example, SQL for a relational database)
   c. One or more queries are sent to the database and the result set returned (results are values from the database, not entity instances)
3. For each item in the result set
   a. If this is a tracking query, EF checks if the data represents an entity already in the change tracker for the context instance
      - If so, the existing entity is returned
      - If not, a new entity is created, change tracking is setup, and the new entity is returned
   b. If this is a no-tracking query, EF checks if the data represents an entity already in the result set for this query
      - If so, the existing entity is returned [(1)]
      - If not, a new entity is created and returned

[(1)] No tracking queries use weak references to keep track of entities that have already been returned. If a previous result with the same identity goes out of scope, and garbage collection runs, you may get a new entity instance.

When queries are executed

When you call LINQ operators, you are simply building up an in-memory representation of the query. The query is only sent to the database when the results are consumed.

The most common operations that result in the query being sent to the database are:

- Iterating the results in a `for` loop
- Using an operator such as `ToList`, `ToArray`, `Single`, `Count`
- Databinding the results of a query to a UI

WARNING

Always validate user input: While EF Core protects against SQL injection attacks by using parameters and escaping literals in queries, it does not validate inputs. Appropriate validation, per the application's requirements, should be performed before values from untrusted sources are used in LINQ queries, assigned to entity properties, or passed to other EF Core APIs. This includes any user input used to dynamically construct queries. Even when using LINQ, if you are accepting user input to build expressions, you need to make sure that only intended expressions can be constructed.

Global Query Filters

12/7/2018 • 2 minutes to read • Edit Online

Global query filters are LINQ query predicates (a boolean expression typically passed to the LINQ *Where* query operator) applied to Entity Types in the metadata model (usually in *OnModelCreating*). Such filters are automatically applied to any LINQ queries involving those Entity Types, including Entity Types referenced indirectly, such as through the use of Include or direct navigation property references. Some common applications of this feature are:

- **Soft delete** - An Entity Type defines an *IsDeleted* property.
- **Multi-tenancy** - An Entity Type defines a *TenantId* property.

Example

The following example shows how to use Global Query Filters to implement soft-delete and multi-tenancy query behaviors in a simple blogging model.

> **TIP**
>
> You can view this article's sample on GitHub.

First, define the entities:

```
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; }
}
```

Note the declaration of a *_tenantId* field on the *Blog* entity. This will be used to associate each Blog instance with a specific tenant. Also defined is an *IsDeleted* property on the *Post* entity type. This is used to keep track of whether a *Post* instance has been "soft-deleted". That is, the instance is marked as deleted without physically removing the underlying data.

Next, configure the query filters in *OnModelCreating* using the `HasQueryFilter` API.

```
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);
}
```

The predicate expressions passed to the *HasQueryFilter* calls will now automatically be applied to any LINQ queries for those types.

> **TIP**
> Note the use of a DbContext instance level field: `_tenantId` used to set the current tenant. Model-level filters will use the value from the correct context instance (that is, the instance that is executing the query).

Disabling Filters

Filters may be disabled for individual LINQ queries by using the `IgnoreQueryFilters()` operator.

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

Limitations

Global query filters have the following limitations:

- Filters cannot contain references to navigation properties.
- Filters can only be defined for the root Entity Type of an inheritance hierarchy.

Query tags

11/15/2018 • 2 minutes to read • Edit Online

NOTE

This feature is new in EF Core 2.2.

This feature helps correlate LINQ queries in code with generated SQL queries captured in logs. You annotate a LINQ query using the new `TagWith()` method:

```
var nearestFriends =
    (from f in context.Friends.TagWith("This is my spatial query!")
    orderby f.Location.Distance(myLocation) descending
    select f).Take(5).ToList();
```

This LINQ query is translated to the following SQL statement:

```
-- This is my spatial query!

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

It's possible to call `TagWith()` many times on the same query. Query tags are cumulative. For example, given the following methods:

```
IQueryable<Friend> GetNearestFriends(Point myLocation) =>
    from f in context.Friends.TagWith("GetNearestFriends")
    orderby f.Location.Distance(myLocation) descending
    select f;

IQueryable<T> Limit<T>(IQueryable<T> source, int limit) =>
    source.TagWith("Limit").Take(limit);
```

The following query:

```
var results = Limit(GetNearestFriends(myLocation), 25).ToList();
```

Translates to:

```
-- GetNearestFriends

-- Limit

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

It's also possible to use multi-line strings as query tags. For example:

```
var results = Limit(GetNearestFriends(myLocation), 25).TagWith(
@"This is a multi-line
string").ToList();
```

Produces the following SQL:

```
-- GetNearestFriends

-- Limit

-- This is a multi-line
-- string

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

Known limitations

Query tags aren't parameterizable: EF Core always treats query tags in the LINQ query as string literals that are included in the generated SQL. Compiled queries that take query tags as parameters aren't allowed.

Saving Data

8/27/2018 • 2 minutes to read • Edit Online

Each context instance has a `ChangeTracker` that is responsible for keeping track of changes that need to be written to the database. As you make changes to instances of your entity classes, these changes are recorded in the `ChangeTracker` and then written to the database when you call `SaveChanges`. The database provider is responsible for translating the changes into database-specific operations (for example, `INSERT`, `UPDATE`, and `DELETE` commands for a relational database).

Basic Save

8/27/2018 • 2 minutes to read • Edit Online

Learn how to add, modify, and remove data using your context and entity classes.

> **TIP**
>
> You can view this article's sample on GitHub.

Adding Data

Use the *DbSet.Add* method to add new instances of your entity classes. The data will be inserted in the database when you call *SaveChanges*.

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

> **TIP**
>
> The Add, Attach, and Update methods all work on the full graph of entities passed to them, as described in the Related Data section. Alternately, the EntityEntry.State property can be used to set the state of just a single entity. For example, `context.Entry(blog).State = EntityState.Modified`.

Updating Data

EF will automatically detect changes made to an existing entity that is tracked by the context. This includes entities that you load/query from the database, and entities that were previously added and saved to the database.

Simply modify the values assigned to properties and then call *SaveChanges*.

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

Deleting Data

Use the *DbSet.Remove* method to delete instances of your entity classes.

If the entity already exists in the database, it will be deleted during *SaveChanges*. If the entity has not yet been saved to the database (that is, it is tracked as added) then it will be removed from the context and will no longer be inserted when *SaveChanges* is called.

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

Multiple Operations in a single SaveChanges

You can combine multiple Add/Update/Remove operations into a single call to *SaveChanges*.

> **NOTE**
>
> For most database providers, *SaveChanges* is transactional. This means all the operations will either succeed or fail and the operations will never be left partially applied.

```
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();
}
```

Saving Related Data

8/27/2018 • 2 minutes to read • Edit Online

In addition to isolated entities, you can also make use of the relationships defined in your model.

> **TIP**
>
> You can view this article's sample on GitHub.

Adding a graph of new entities

If you create several new related entities, adding one of them to the context will cause the others to be added too.

In the following example, the blog and three related posts are all inserted into the database. The posts are found and added, because they are reachable via the `Blog.Posts` navigation property.

```
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**
>
> Use the EntityEntry.State property to set the state of just a single entity. For example, `context.Entry(blog).State = EntityState.Modified`.

Adding a related entity

If you reference a new entity from the navigation property of an entity that is already tracked by the context, the entity will be discovered and inserted into the database.

In the following example, the `post` entity is inserted because it is added to the `Posts` property of the `blog` entity which was fetched from the database.

```
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();
}
```

Changing relationships

If you change the navigation property of an entity, the corresponding changes will be made to the foreign key column in the database.

In the following example, the `post` entity is updated to belong to the new `blog` entity because its `Blog` navigation property is set to point to `blog`. Note that `blog` will also be inserted into the database because it is a new entity that is referenced by the navigation property of an entity that is already tracked by the context (`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();
}
```

Removing relationships

You can remove a relationship by setting a reference navigation to `null`, or removing the related entity from a collection navigation.

Removing a relationship can have side effects on the dependent entity, according to the cascade delete behavior configured in the relationship.

By default, for required relationships, a cascade delete behavior is configured and the child/dependent entity will be deleted from the database. For optional relationships, cascade delete is not configured by default, but the foreign key property will be set to null.

See Required and Optional Relationships to learn about how the requiredness of relationships can be configured.

See Cascade Delete for more details on how cascade delete behaviors work, how they can be configured explicitly and how they are selected by convention.

In the following example, a cascade delete is configured on the relationship between `Blog` and `Post`, so the `post` entity is deleted from the database.

```
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();
}
```

Cascade Delete

9/11/2018 • 13 minutes to read • Edit Online

Cascade delete is commonly used in database terminology to describe a characteristic that allows the deletion of a row to automatically trigger the deletion of related rows. A closely related concept also covered by EF Core delete behaviors is the automatic deletion of a child entity when it's relationship to a parent has been severed--this is commonly known as "deleting orphans".

EF Core implements several different delete behaviors and allows for the configuration of the delete behaviors of individual relationships. EF Core also implements conventions that automatically configure useful default delete behaviors for each relationship based on the requiredness of the relationship.

Delete behaviors

Delete behaviors are defined in the *DeleteBehavior* enumerator type and can be passed to the *OnDelete* fluent API to control whether the deletion of a principal/parent entity or the severing of the relationship to dependent/child entities should have a side effect on the dependent/child entities.

There are three actions EF can take when a principal/parent entity is deleted or the relationship to the child is severed:

- The child/dependent can be deleted
- The child's foreign key values can be set to null
- The child remains unchanged

> **NOTE**
> The delete behavior configured in the EF Core model is only applied when the principal entity is deleted using EF Core and the dependent entities are loaded in memory (that is, for tracked dependents). A corresponding cascade behavior needs to be setup in the database to ensure data that is not being tracked by the context has the necessary action applied. If you use EF Core to create the database, this cascade behavior will be setup for you.

For the second action above, setting a foreign key value to null is not valid if foreign key is not nullable. (A non-nullable foreign key is equivalent to a required relationship.) In these cases, EF Core tracks that the foreign key property has been marked as null until SaveChanges is called, at which time an exception is thrown because the change cannot be persisted to the database. This is similar to getting a constraint violation from the database.

There are four delete behaviors, as listed in the tables below.

Optional relationships

For optional relationships (nullable foreign key) it *is* possible to save a null foreign key value, which results in the following effects:

BEHAVIOR NAME	EFFECT ON DEPENDENT/CHILD IN MEMORY	EFFECT ON DEPENDENT/CHILD IN DATABASE
Cascade	Entities are deleted	Entities are deleted
ClientSetNull (Default)	Foreign key properties are set to null	None

BEHAVIOR NAME	EFFECT ON DEPENDENT/CHILD IN MEMORY	EFFECT ON DEPENDENT/CHILD IN DATABASE
SetNull	Foreign key properties are set to null	Foreign key properties are set to null
Restrict	None	None

Required relationships

For required relationships (non-nullable foreign key) it is *not* possible to save a null foreign key value, which results in the following effects:

BEHAVIOR NAME	EFFECT ON DEPENDENT/CHILD IN MEMORY	EFFECT ON DEPENDENT/CHILD IN DATABASE
Cascade (Default)	Entities are deleted	Entities are deleted
ClientSetNull	SaveChanges throws	None
SetNull	SaveChanges throws	SaveChanges throws
Restrict	None	None

In the tables above, *None* can result in a constraint violation. For example, if a principal/child entity is deleted but no action is taken to change the foreign key of a dependent/child, then the database will likely throw on SaveChanges due to a foreign constraint violation.

At a high level:

- If you have entities that cannot exist without a parent, and you want EF to take care for deleting the children automatically, then use *Cascade*.
  - Entities that cannot exist without a parent usually make use of required relationships, for which *Cascade* is the default.
- If you have entities that may or may not have a parent, and you want EF to take care of nulling out the foreign key for you, then use *ClientSetNull*
  - Entities that can exist without a parent usually make use of optional relationships, for which *ClientSetNull* is the default.
  - If you want the database to also try to propagate null values to child foreign keys even when the child entity is not loaded, then use *SetNull*. However, note that the database must support this, and configuring the database like this can result in other restrictions, which in practice often makes this option impractical. This is why *SetNull* is not the default.
- If you don't want EF Core to ever delete an entity automatically or null out the foreign key automatically, then use *Restrict*. Note that this requires that your code keep child entities and their foreign key values in sync manually otherwise constraint exceptions will be thrown.

> **NOTE**
>
> In EF Core, unlike EF6, cascading effects do not happen immediately, but instead only when SaveChanges is called.

> **NOTE**
>
> **Changes in EF Core 2.0:** In previous releases, *Restrict* would cause optional foreign key properties in tracked dependent entities to be set to null, and was the default delete behavior for optional relationships. In EF Core 2.0, the *ClientSetNull* was introduced to represent that behavior and became the default for optional relationships. The behavior of *Restrict* was adjusted to never have any side effects on dependent entities.

Entity deletion examples

The code below is part of a sample that can be downloaded and run. The sample shows what happens for each delete behavior for both optional and required relationships when a parent entity is deleted.

```
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)}");
}
```

Let's walk through each variation to understand what is happening.

DeleteBehavior.Cascade with required or optional relationship

```
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.
```

- Blog is marked as Deleted
- Posts initially remain Unchanged since cascades do not happen until SaveChanges
- SaveChanges sends deletes for both dependents/children (posts) and then the principal/parent (blog)
- After saving, all entities are detached since they have now been deleted from the database

DeleteBehavior.ClientSetNull or DeleteBehavior.SetNull with required relationship

```
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.
```

- Blog is marked as Deleted
- Posts initially remain Unchanged since cascades do not happen until SaveChanges
- SaveChanges attempts to set the post FK to null, but this fails because the FK is not nullable

DeleteBehavior.ClientSetNull or DeleteBehavior.SetNull with optional relationship

```
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.
```

- Blog is marked as Deleted
- Posts initially remain Unchanged since cascades do not happen until SaveChanges
- SaveChanges attempts sets the FK of both dependents/children (posts) to null before deleting the principal/parent (blog)
- After saving, the principal/parent (blog) is deleted, but the dependents/children (posts) are still tracked
- The tracked dependents/children (posts) now have null FK values and their reference to the deleted principal/parent (blog) has been removed

DeleteBehavior.Restrict with required or optional relationship

```
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.
```

- Blog is marked as Deleted
- Posts initially remain Unchanged since cascades do not happen until SaveChanges
- Since *Restrict* tells EF to not automatically set the FK to null, it remains non-null and SaveChanges throws without saving

Delete orphans examples

The code below is part of a sample that can be downloaded and run. The sample shows what happens for each delete behavior for both optional and required relationships when the relationship between a parent/principal and its children/dependents is severed. In this example, the relationship is severed by removing the dependents/children (posts) from the collection navigation property on the principal/parent (blog). However, the behavior is the same if the reference from dependent/child to principal/parent is instead nulled out.

```
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)}");
}
```

Let's walk through each variation to understand what is happening.

DeleteBehavior.Cascade with required or optional relationship

```
  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.
```

- Posts are marked as Modified because severing the relationship caused the FK to be marked as null
  - If the FK is not nullable, then the actual value will not change even though it is marked as null
- SaveChanges sends deletes for dependents/children (posts)
- After saving, the dependents/children (posts) are detached since they have now been deleted from the database

DeleteBehavior.ClientSetNull or DeleteBehavior.SetNull with required relationship

```
  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.
```

- Posts are marked as Modified because severing the relationship caused the FK to be marked as null
  - If the FK is not nullable, then the actual value will not change even though it is marked as null
- SaveChanges attempts to set the post FK to null, but this fails because the FK is not nullable

DeleteBehavior.ClientSetNull or DeleteBehavior.SetNull with optional relationship

```
  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.
```

- Posts are marked as Modified because severing the relationship caused the FK to be marked as null
  - If the FK is not nullable, then the actual value will not change even though it is marked as null
- SaveChanges sets the FK of both dependents/children (posts) to null
- After saving, the dependents/children (posts) now have null FK values and their reference to the deleted principal/parent (blog) has been removed

DeleteBehavior.Restrict with required or optional relationship

```
  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.
```

- Posts are marked as Modified because severing the relationship caused the FK to be marked as null
  - If the FK is not nullable, then the actual value will not change even though it is marked as null
- Since *Restrict* tells EF to not automatically set the FK to null, it remains non-null and SaveChanges throws without saving

Cascading to untracked entities

When you call *SaveChanges*, the cascade delete rules will be applied to any entities that are being tracked by the context. This is the situation in all the examples shown above, which is why SQL was generated to delete both the principal/parent (blog) and all the dependents/children (posts):

```
    DELETE FROM [Posts] WHERE [PostId] = 1
    DELETE FROM [Posts] WHERE [PostId] = 2
    DELETE FROM [Blogs] WHERE [BlogId] = 1
```

If only the principal is loaded--for example, when a query is made for a blog without an `Include(b => b.Posts)` to also include posts--then SaveChanges will only generate SQL to delete the principal/parent:

```
    DELETE FROM [Blogs] WHERE [BlogId] = 1
```

The dependents/children (posts) will only be deleted if the database has a corresponding cascade behavior configured. If you use EF to create the database, this cascade behavior will be setup for you.

Handling Concurrency Conflicts

8/27/2018 • 3 minutes to read • Edit Online

NOTE

This page documents how concurrency works in EF Core and how to handle concurrency conflicts in your application. See Concurrency Tokens for details on how to configure concurrency tokens in your model.

TIP

You can view this article's sample on GitHub.

Database concurrency refers to situations in which multiple processes or users access or change the same data in a database at the same time. *Concurrency control* refers to specific mechanisms used to ensure data consistency in presence of concurrent changes.

EF Core implements *optimistic concurrency control*, meaning that it will let multiple processes or users make changes independently without the overhead of synchronization or locking. In the ideal situation, these changes will not interfere with each other and therefore will be able to succeed. In the worst case scenario, two or more processes will attempt to make conflicting changes, and only one of them should succeed.

How concurrency control works in EF Core

Properties configured as concurrency tokens are used to implement optimistic concurrency control: whenever an update or delete operation is performed during `SaveChanges`, the value of the concurrency token on the database is compared against the original value read by EF Core.

- If the values match, the operation can complete.
- If the values do not match, EF Core assumes that another user has performed a conflicting operation and aborts the current transaction.

The situation when another user has performed an operation that conflicts with the current operation is known as *concurrency conflict*.

Database providers are responsible for implementing the comparison of concurrency token values.

On relational databases EF Core includes a check for the value of the concurrency token in the `WHERE` clause of any `UPDATE` or `DELETE` statements. After executing the statements, EF Core reads the number of rows that were affected.

If no rows are affected, a concurrency conflict is detected, and EF Core throws `DbUpdateConcurrencyException`.

For example, we may want to configure `LastName` on `Person` to be a concurrency token. Then any update operation on Person will include the concurrency check in the `WHERE` clause:

```
UPDATE [Person] SET [FirstName] = @p1
WHERE [PersonId] = @p0 AND [LastName] = @p2;
```

Resolving concurrency conflicts

Continuing with the previous example, if one user tries to save some changes to a `Person`, but another user has already changed the `LastName`, then an exception will be thrown.

At this point, the application could simply inform the user that the update was not successful due to conflicting changes and move on. But it may be desirable to prompt the user to ensure this record still represents the same actual person and to retry the operation.

This process is an example of *resolving a concurrency conflict*.

Resolving a concurrency conflict involves merging the pending changes from the current `DbContext` with the values in the database. What values get merged will vary based on the application and may be directed by user input.

There are three sets of values available to help resolve a concurrency conflict:

- **Current values** are the values that the application was attempting to write to the database.
- **Original values** are the values that were originally retrieved from the database, before any edits were made.
- **Database values** are the values currently stored in the database.

The general approach to handle a concurrency conflicts is:

1. Catch `DbUpdateConcurrencyException` during `SaveChanges`.
2. Use `DbUpdateConcurrencyException.Entries` to prepare a new set of changes for the affected entities.
3. Refresh the original values of the concurrency token to reflect the current values in the database.
4. Retry the process until no conflicts occur.

In the following example, `Person.FirstName` and `Person.LastName` are setup as concurrency tokens. There is a `// TODO:` comment in the location where you include application specific logic to choose the value to be saved.

```
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);
                }
            }
        }
    }
}
```

Using Transactions

1/11/2019 • 5 minutes to read • Edit Online

Transactions allow several database operations to be processed in an atomic manner. If the transaction is committed, all of the operations are successfully applied to the database. If the transaction is rolled back, none of the operations are applied to the database.

TIP

You can view this article's sample on GitHub.

Default transaction behavior

By default, if the database provider supports transactions, all changes in a single call to `SaveChanges()` are applied in a transaction. If any of the changes fail, then the transaction is rolled back and none of the changes are applied to the database. This means that `SaveChanges()` is guaranteed to either completely succeed, or leave the database unmodified if an error occurs.

For most applications, this default behavior is sufficient. You should only manually control transactions if your application requirements deem it necessary.

Controlling transactions

You can use the `DbContext.Database` API to begin, commit, and rollback transactions. The following example shows two `SaveChanges()` operations and a LINQ query being executed in a single transaction.

Not all database providers support transactions. Some providers may throw or no-op when transaction APIs are called.

```
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
        }
    }
}
```

Cross-context transaction (relational databases only)

You can also share a transaction across multiple context instances. This functionality is only available when using a relational database provider because it requires the use of `DbTransaction` and `DbConnection`, which are specific to relational databases.

To share a transaction, the contexts must share both a `DbConnection` and a `DbTransaction`.

Allow connection to be externally provided

Sharing a `DbConnection` requires the ability to pass a connection into a context when constructing it.

The easiest way to allow `DbConnection` to be externally provided, is to stop using the `DbContext.OnConfiguring` method to configure the context and externally create `DbContextOptions` and pass them to the context constructor.

> **TIP**
> `DbContextOptionsBuilder` is the API you used in `DbContext.OnConfiguring` to configure the context, you are now going to use it externally to create `DbContextOptions`.

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

    public DbSet<Blog> Blogs { get; set; }
}
```

An alternative is to keep using `DbContext.OnConfiguring`, but accept a `DbConnection` that is saved and then used in `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);
    }
}
```

Share connection and transaction

You can now create multiple context instances that share the same connection. Then use the `DbContext.Database.UseTransaction(DbTransaction)` API to enlist both contexts in the same transaction.

```
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
        }
    }
}
```

Using external DbTransactions (relational databases only)

If you are using multiple data access technologies to access a relational database, you may want to share a transaction between operations performed by these different technologies.

The following example, shows how to perform an ADO.NET SqlClient operation and an Entity Framework Core operation in the same transaction.

```
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
        }
    }
}
```

Using System.Transactions

> **NOTE**
>
> This feature is new in EF Core 2.1.

It is possible to use ambient transactions if you need to coordinate across a larger scope.

```
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
        }
    }
}
```

It is also possible to enlist in an explicit transaction.

```
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
    }
}
```

Limitations of System.Transactions

1. EF Core relies on database providers to implement support for System.Transactions. Although support is quite common among ADO.NET providers for .NET Framework, the API has only been recently added to .NET Core and hence support is not as widespread. If a provider does not implement support for System.Transactions, it is possible that calls to these APIs will be completely ignored. SqlClient for .NET Core does support it from 2.1 onwards. SqlClient for .NET Core 2.0 will throw an exception if you attempt to use the feature.

> **IMPORTANT**
>
> It is recommended that you test that the API behaves correctly with your provider before you rely on it for managing transactions. You are encouraged to contact the maintainer of the database provider if it does not.

2. As of version 2.1, the System.Transactions implementation in .NET Core does not include support for distributed transactions, therefore you cannot use `TransactionScope` or `CommittableTransaction` to coordinate transactions across multiple resource managers.

Asynchronous Saving

8/27/2018 • 2 minutes to read • Edit Online

Asynchronous saving avoids blocking a thread while the changes are written to the database. This can be useful to avoid freezing the UI of a thick-client application. Asynchronous operations can also increase throughput in a web application, where the thread can be freed up to service other requests while the database operation completes. For more information, see Asynchronous Programming in C#.

WARNING

EF Core does not support multiple parallel operations being run on the same context instance. You should always wait for an operation to complete before beginning the next operation. This is typically done by using the `await` keyword on each asynchronous operation.

Entity Framework Core provides `DbContext.SaveChangesAsync()` as an asynchronous alternative to `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();
    }
}
```

Disconnected entities

8/27/2018 • 8 minutes to read • Edit Online

A DbContext instance will automatically track entities returned from the database. Changes made to these entities will then be detected when SaveChanges is called and the database will be updated as needed. See Basic Save and Related Data for details.

However, sometimes entities are queried using one context instance and then saved using a different instance. This often happens in "disconnected" scenarios such as a web application where the entities are queried, sent to the client, modified, sent back to the server in a request, and then saved. In this case, the second context instance needs to know whether the entities are new (should be inserted) or existing (should be updated).

> **TIP**
>
> You can view this article's sample on GitHub.

> **TIP**
>
> EF Core can only track one instance of any entity with a given primary key value. The best way to avoid this being an issue is to use a short-lived context for each unit-of-work such that the context starts empty, has entities attached to it, saves those entities, and then the context is disposed and discarded.

Identifying new entities

Client identifies new entities

The simplest case to deal with is when the client informs the server whether the entity is new or existing. For example, often the request to insert a new entity is different from the request to update an existing entity.

The remainder of this section covers the cases where it necessary to determine in some other way whether to insert or update.

With auto-generated keys

The value of an automatically generated key can often be used to determine whether an entity needs to be inserted or updated. If the key has not been set (that is, it still has the CLR default value of null, zero, etc.), then the entity must be new and needs inserting. On the other hand, if the key value has been set, then it must have already been previously saved and now needs updating. In other words, if the key has a value, then the entity was queried, sent to the client, and has now come back to be updated.

It is easy to check for an unset key when the entity type is known:

```
public static bool IsItNew(Blog blog)
    => blog.BlogId == 0;
```

However, EF also has a built-in way to do this for any entity type and key type:

```
public static bool IsItNew(DbContext context, object entity)
    => !context.Entry(entity).IsKeySet;
```

> **TIP**
>
> Keys are set as soon as entities are tracked by the context, even if the entity is in the Added state. This helps when traversing a graph of entities and deciding what to do with each, such as when using the TrackGraph API. The key value should only be used in the way shown here *before* any call is made to track the entity.

With other keys

Some other mechanism is needed to identify new entities when key values are not generated automatically. There are two general approaches to this:

- Query for the entity
- Pass a flag from the client

To query for the entity, just use the Find method:

```
public static bool IsItNew(BloggingContext context, Blog blog)
    => context.Blogs.Find(blog.BlogId) == null;
```

It is beyond the scope of this document to show the full code for passing a flag from a client. In a web app, it usually means making different requests for different actions, or passing some state in the request then extracting it in the controller.

Saving single entities

If it is known whether or not an insert or update is needed, then either Add or Update can be used appropriately:

```
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();
}
```

However, if the entity uses auto-generated key values, then the Update method can be used for both cases:

```
public static void InsertOrUpdate(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

The Update method normally marks the entity for update, not insert. However, if the entity has a auto-generated key, and no key value has been set, then the entity is instead automatically marked for insert.

> **TIP**
>
> This behavior was introduced in EF Core 2.0. For earlier releases it is always necessary to explicitly choose either Add or Update.

If the entity is not using auto-generated keys, then the application must decide whether the entity should be inserted or updated: For example:

```
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();
}
```

The steps here are:

- If Find returns null, then the database doesn't already contain the blog with this ID, so we call Add mark it for insertion.
- If Find returns an entity, then it exists in the database and the context is now tracking the existing entity
  - We then use SetValues to set the values for all properties on this entity to those that came from the client.
  - The SetValues call will mark the entity to be updated as needed.

> **TIP**
> SetValues will only mark as modified the properties that have different values to those in the tracked entity. This means that when the update is sent, only those columns that have actually changed will be updated. (And if nothing has changed, then no update will be sent at all.)

Working with graphs

Identity resolution

As noted above, EF Core can only track one instance of any entity with a given primary key value. When working with graphs the graph should ideally be created such that this invariant is maintained, and the context should be used for only one unit-of-work. If the graph does contain duplicates, then it will be necessary to process the graph before sending it to EF to consolidate multiple instances into one. This may not be trivial where instances have conflicting values and relationships, so consolidating duplicates should be done as soon as possible in your application pipeline to avoid conflict resolution.

All new/all existing entities

An example of working with graphs is inserting or updating a blog together with its collection of associated posts. If all the entities in the graph should be inserted, or all should be updated, then the process is the same as described above for single entities. For example, a graph of blogs and posts created like this:

```
var blog = new Blog
{
    Url = "http://sample.com",
    Posts = new List<Post>
    {
        new Post {Title = "Post 1"},
        new Post {Title = "Post 2"},
    }
};
```

can be inserted like this:

```
public static void InsertGraph(DbContext context, object rootEntity)
{
    context.Add(rootEntity);
    context.SaveChanges();
}
```

The call to Add will mark the blog and all the posts to be inserted.

Likewise, if all the entities in a graph need to be updated, then Update can be used:

```
public static void UpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

The blog and all its posts will be marked to be updated.

Mix of new and existing entities

With auto-generated keys, Update can again be used for both inserts and updates, even if the graph contains a mix of entities that require inserting and those that require updating:

```
public static void InsertOrUpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

Update will mark any entity in the graph, blog or post, for insertion if it does not have a key value set, while all other entities are marked for update.

As before, when not using auto-generated keys, a query and some processing can be used:

```
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();
}
```

Handling deletes

Delete can be tricky to handle since often the absence of an entity means that it should be deleted. One way to deal with this is to use "soft deletes" such that the entity is marked as deleted rather than actually being deleted. Deletes then becomes the same as updates. Soft deletes can be implemented in using query filters.

For true deletes, a common pattern is to use an extension of the query pattern to perform what is essentially a graph diff. For example:

```
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

Internally, Add, Attach, and Update use graph-traversal with a determination made for each entity as to whether it should be marked as Added (to insert), Modified (to update), Unchanged (do nothing), or Deleted (to delete). This mechanism is exposed via the TrackGraph API. For example, let's assume that when the client sends back a graph of entities it sets some flag on each entity indicating how it should be handled. TrackGraph can then be used to process this flag:

```
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();
}
```

The flags are only shown as part of the entity for simplicity of the example. Typically the flags would be part of a DTO or some other state included in the request.

Setting Explicit Values for Generated Properties

8/27/2018 • 3 minutes to read • Edit Online

A generated property is a property whose value is generated (either by EF or the database) when the entity is added and/or updated. See Generated Properties for more information.

There may be situations where you want to set an explicit value for a generated property, rather than having one generated.

TIP

You can view this article's sample on GitHub.

The model

The model used in this article contains a single `Employee` entity.

```
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; }
}
```

Saving an explicit value during add

The `Employee.EmploymentStarted` property is configured to have values generated by the database for new entities (using a default value).

```
modelBuilder.Entity<Employee>()
    .Property(b => b.EmploymentStarted)
    .HasDefaultValueSql("CONVERT(date, GETDATE())");
```

The following code inserts two employees into the database.

- For the first, no value is assigned to `Employee.EmploymentStarted` property, so it remains set to the CLR default value for `DateTime`.
- For the second, we have set an explicit value of `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);
    }
}
```

Output shows that the database generated a value for the first employee and our explicit value was used for the second.

```
1: John Doe, 1/26/2017 12:00:00 AM
2: Jane Doe, 1/1/2000 12:00:00 AM
```

Explicit values into SQL Server IDENTITY columns

By convention the `Employee.EmployeeId` property is a store generated `IDENTITY` column.

For most situations, the approach shown above will work for key properties. However, to insert explicit values into a SQL Server `IDENTITY` column, you need to manually enable `IDENTITY_INSERT` before calling `SaveChanges()`.

> **NOTE**
>
> We have a feature request on our backlog to do this automatically within the SQL Server provider.

```
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);
    }
}
```

Output shows that the supplied ids were saved to the database.

```
100: John Doe
101: Jane Doe
```

Setting an explicit value during update

The `Employee.LastPayRaise` property is configured to have values generated by the database during updates.

```
modelBuilder.Entity<Employee>()
    .Property(b => b.LastPayRaise)
    .ValueGeneratedOnAddOrUpdate();

modelBuilder.Entity<Employee>()
    .Property(b => b.LastPayRaise)
    .Metadata.AfterSaveBehavior = PropertySaveBehavior.Ignore;
```

> **NOTE**
>
> By default, EF Core will throw an exception if you try to save an explicit value for a property that is configured to be generated during update. To avoid this, you need to drop down to the lower level metadata API and set the `AfterSaveBehavior` (as shown above).

> **NOTE**
>
> **Changes in EF Core 2.0:** In previous releases the after-save behavior was controlled through the `IsReadOnlyAfterSave` flag. This flag has been obsoleted and replaced by `AfterSaveBehavior`.

There is also a trigger in the database to generate values for the `LastPayRaise` column during `UPDATE` operations.

```
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
```

The following code increases the salary of two employees in the database.

- For the first, no value is assigned to `Employee.LastPayRaise` property, so it remains set to null.
- For the second, we have set an explicit value of one week ago (back dating the pay raise).

```
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);
    }
}
```

Output shows that the database generated a value for the first employee and our explicit value was used for the second.

```
1: John Doe, 1/26/2017 12:00:00 AM
2: Jane Doe, 1/19/2017 12:00:00 AM
```

.NET implementations supported by EF Core

10/25/2018 • 2 minutes to read • Edit Online

We want EF Core to be available anywhere you can write .NET code, and we're still working towards that goal. While EF Core's support on .NET Core and .NET Framework is covered by automated testing and many applications known to be using it successfully, Mono, Xamarin and UWP have some issues.

Overview

The following table provides guidance for each .NET implementation:

.NET IMPLEMENTATION	STATUS	EF CORE 1.X REQUIREMENTS	EF CORE 2.X REQUIREMENTS [1]
.NET Core (ASP.NET Core, Console, etc.)	Fully supported and recommended	.NET Core SDK 1.x	.NET Core SDK 2.x
.NET Framework (WinForms, WPF, ASP.NET, Console, etc.)	Fully supported and recommended. EF6 also available [2]	.NET Framework 4.5.1	.NET Framework 4.6.1
Mono & Xamarin	In progress [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
Universal Windows Platform	EF Core 2.0.1 recommended [4]	.NET Core UWP 5.x package	.NET Core UWP 6.x package

(1) EF Core 2.0 targets and therefore requires .NET implementations that support .NET Standard 2.0.

(2) See Compare EF Core & EF6 to choose the right technology.

(3) There are issues and known limitations with Xamarin which may prevent some applications developed using EF Core 2.0 from working correctly. Check the list of active issues for workarounds.

(4) See the Universal Windows Platform section of this article.

Universal Windows Platform

Earlier versions of EF Core and .NET UWP had numerous compatibility issues, especially with applications compiled with the .NET Native toolchain. The new .NET UWP version adds support for .NET Standard 2.0 and contains .NET Native 2.0, which fixes most of the compatibility issues previously reported. EF Core 2.0.1 has been tested more thoroughly with UWP but testing is not automated.

When using EF Core on UWP:

- To optimize query performance, avoid anonymous types in LINQ queries. Deploying a UWP application to the app store requires an application to be compiled with .NET Native. Queries with anonymous types have worse performance on .NET Native.
- To optimize `SaveChanges()` performance, use ChangeTrackingStrategy.ChangingAndChangedNotifications and implement INotifyPropertyChanged, INotifyPropertyChanging, and INotifyCollectionChanged in your entity types.

Report issues

For any combination that doesn’t work as expected, we encourage creating new issues on the EF Core issue tracker. For Xamarin-specific issues use the issue tracker for Xamarin.Android or Xamarin.iOS.

Database Providers

12/4/2018 • 3 minutes to read • Edit Online

Entity Framework Core can access many different databases through plug-in libraries called database providers.

Current providers

IMPORTANT

EF Core providers are built by a variety of sources. Not all providers are maintained as part of the Entity Framework Core Project. When considering a provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements. Also make sure you review each provider's documentation for detailed version compatibility information.

NUGET PACKAGE	SUPPORTED DATABASE ENGINES	MAINTAINER / VENDOR	NOTES / REQUIREMENTS	USEFUL LINKS
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2008 onwards	EF Core Project (Microsoft)		docs
Microsoft.EntityFrameworkCore.Sqlite	SQLite 3.7 onwards	EF Core Project (Microsoft)		docs
Microsoft.EntityFrameworkCore.InMemory	EF Core in-memory database	EF Core Project (Microsoft)	For testing only	docs
Microsoft.EntityFrameworkCore.Cosmos	Azure Cosmos DB SQL API	EF Core Project (Microsoft)	Preview only	blog
Npgsql.EntityFrameworkCore.PostgreSQL	PostgreSQL	Npgsql Development Team		docs
Pomelo.EntityFrameworkCore.MySql	MySQL, MariaDB	Pomelo Foundation Project		readme
Pomelo.EntityFrameworkCore.MyCat	MyCAT Server	Pomelo Foundation Project	Prerelease only	readme
EntityFrameworkCore.SqlServerCompact40	SQL Server Compact 4.0	Erik Ejlskov Jensen	.NET Framework	wiki
EntityFrameworkCore.SqlServerCompact35	SQL Server Compact 3.5	Erik Ejlskov Jensen	.NET Framework	wiki
EntityFrameworkCore.Jet	Microsoft Access files	Bubi	.NET Framework	readme
MySql.Data.EntityFrameworkCore	MySQL	MySQL project (Oracle)		docs

NUGET PACKAGE	SUPPORTED DATABASE ENGINES	MAINTAINER / VENDOR	NOTES / REQUIREMENTS	USEFUL LINKS
FirebirdSql.EntityFrameworkCore.Firebird	Firebird 2.5 and 3.x	Jiří Činčura		docs
EntityFrameworkCore.FirebirdSql	Firebird 2.5 and 3.x	Rafael Almeida		wiki
IBM.EntityFrameworkCore	Db2, Informix	IBM	Windows version	blog
IBM.EntityFrameworkCore-lnx	Db2, Informix	IBM	Linux version	blog
IBM.EntityFrameworkCore-osx	Db2, Informix	IBM	macOS version	blog
EntityFrameworkCore.OpenEdge	Progress OpenEdge	Alex Wiese		readme
Devart.Data.Oracle.EFCore	Oracle 9.2.0.4 onwards	DevArt	Paid	docs
Devart.Data.PostgreSql.EFCore	PostgreSQL 8.0 onwards	DevArt	Paid	docs
Devart.Data.SQLite.EFCore	SQLite 3 onwards	DevArt	Paid	docs
Devart.Data.MySql.EFCore	MySQL 5 onwards	DevArt	Paid	docs

Future Providers

Cosmos DB

We have been developing an EF Core provider for the SQL API in Cosmos DB. This will be the first complete document-oriented database provider we have produced, and the learnings from this exercise are going to inform improvements in the design of future releases of EF Core and possibly other non-relational providers. A preview is available on the NuGet Gallery.

Oracle

The Oracle .NET team has announced they are planning to release a first-party provider for EF Core approximately in the third quarter of 2018. See their statement of direction for .NET Core and Entity Framework Core for more information. Please direct any questions about this provider, including the release timeline, to the Oracle Community Site.

In the meanwhile, the EF team has produced a sample EF Core provider for Oracle databases. The purpose of the project is not to produce an EF Core provider owned by Microsoft. We started the project to identify gaps in EF Core's relational and base functionality which we needed to address in order to better support Oracle. It should also help Oracle or third parties jumpstart the development of Oracle providers for EF Core.

We will consider contributions that improve the sample implementation. We would also welcome and encourage a community effort to create an open-source Oracle provider for EF Core, using the sample as a starting point.

Adding a database provider to your application

Most database providers for EF Core are distributed as NuGet packages. This means they can be installed using the `dotnet` tool in the command line:

```
dotnet add package provider_package_name
```

Or in Visual Studio, using NuGet's Package Manager Console:

```
install-package provider_package_name
```

Once installed, you will configure the provider in your `DbContext`, either in the `OnConfiguring` method or in the `AddDbContext` method if you are using a dependency injection container. For example, the following line configures the SQL Server provider with the passed connection string:

```
optionsBuilder.UseSqlServer(
    "Server=(localdb)\mssqllocaldb;Database=MyDatabase;Trusted_Connection=True;");
```

Database providers can extend EF Core to enable functionality unique to specific databases. Some concepts are common to most databases, and are included in the primary EF Core components. Such concepts include expressing queries in LINQ, transactions, and tracking changes to objects once they are loaded from the database. Some concepts are specific to a particular provider. For example, the SQL Server provider allows you to configure memory-optimized tables (a feature specific to SQL Server). Other concepts are specific to a class of providers. For example, EF Core providers for relational databases build on the common `Microsoft.EntityFrameworkCore.Relational` library, which provides APIs for configuring table and column mappings, foreign key constraints, etc. Providers are usually distributed as NuGet packages.

> **IMPORTANT**
>
> When a new patch version of EF Core is released, it often includes updates to the `Microsoft.EntityFrameworkCore.Relational` package. When you add a relational database provider, this package becomes a transitive dependency of your application. But many providers are released independently from EF Core and may not be updated to depend on the newer patch version of that package. In order to make sure you will get all bug fixes, it is recommended that you add the patch version of `Microsoft.EntityFrameworkCore.Relational` as a direct dependency of your application.

Microsoft SQL Server EF Core Database Provider

8/27/2018 • 2 minutes to read • Edit Online

This database provider allows Entity Framework Core to be used with Microsoft SQL Server (including SQL Azure). The provider is maintained as part of the Entity Framework Core Project.

Install

Install the Microsoft.EntityFrameworkCore.SqlServer NuGet package.

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

Get Started

The following resources will help you get started with this provider.

- Getting Started on .NET Framework (Console, WinForms, WPF, etc.)
- Getting Started on ASP.NET Core
- UnicornStore Sample Application

Supported Database Engines

- Microsoft SQL Server (2008 onwards)

Supported Platforms

- .NET Framework (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)

```
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).
```

Memory-Optimized Tables support in SQL Server EF Core Database Provider

8/27/2018 • 2 minutes to read • Edit Online

> **NOTE**
>
> This feature was introduced in EF Core 1.1.

Memory-Optimized Tables are a feature of SQL Server where the entire table resides in memory. A second copy of the table data is maintained on disk, but only for durability purposes. Data in memory-optimized tables is only read from disk during database recovery. For example, after a server restart.

Configuring a memory-optimized table

You can specify that the table an entity is mapped to is memory-optimized. When using EF Core to create and maintain a database based on your model (either with migrations or `Database.EnsureCreated()`), a memory-optimized table will be created for these entities.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ForSqlServerIsMemoryOptimized();
}
```

SQLite EF Core Database Provider

8/27/2018 • 2 minutes to read • Edit Online

This database provider allows Entity Framework Core to be used with SQLite. The provider is maintained as part of the Entity Framework Core project.

Install

Install the Microsoft.EntityFrameworkCore.Sqlite NuGet package.

```
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

Get Started

The following resources will help you get started with this provider.

- Local SQLite on UWP
- .NET Core Application to New SQLite Database
- Unicorn Clicker Sample Application
- Unicorn Packer Sample Application

Supported Database Engines

- SQLite (3.7 onwards)

Supported Platforms

- .NET Framework (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)
- Universal Windows Platform

Limitations

See SQLite Limitations for some important limitations of the SQLite provider.

SQLite EF Core Database Provider Limitations

8/27/2018 • 2 minutes to read • Edit Online

The SQLite provider has a number of migrations limitations. Most of these limitations are a result of limitations in the underlying SQLite database engine and are not specific to EF.

Modeling limitations

The common relational library (shared by Entity Framework relational database providers) defines APIs for modelling concepts that are common to most relational database engines. A couple of these concepts are not supported by the SQLite provider.

- Schemas
- Sequences

Migrations limitations

The SQLite database engine does not support a number of schema operations that are supported by the majority of other relational databases. If you attempt to apply one of the unsupported operations to a SQLite database then a `NotSupportedException` will be thrown.

OPERATION	SUPPORTED?	REQUIRES VERSION
AddColumn	✔	1.0
AddForeignKey	✗	
AddPrimaryKey	✗	
AddUniqueConstraint	✗	
AlterColumn	✗	
CreateIndex	✔	1.0
CreateTable	✔	1.0
DropColumn	✗	
DropForeignKey	✗	
DropIndex	✔	1.0
DropPrimaryKey	✗	
DropTable	✔	1.0
DropUniqueConstraint	✗	

OPERATION	SUPPORTED?	REQUIRES VERSION
RenameColumn	✗	
RenameIndex	✔	2.1
RenameTable	✔	1.0
EnsureSchema	✔ (no-op)	2.0
DropSchema	✔ (no-op)	2.0
Insert	✔	2.0
Update	✔	2.0
Delete	✔	2.0

Migrations limitations workaround

You can workaround some of these limitations by manually writing code in your migrations to perform a table rebuild. A table rebuild involves renaming the existing table, creating a new table, copying data to the new table, and dropping the old table. You will need to use the `Sql(string)` method to perform some of these steps.

See Making Other Kinds Of Table Schema Changes in the SQLite documentation for more details.

In the future, EF may support some of these operations by using the table rebuild approach under the covers. You can track this feature on our GitHub project.

EF Core In-Memory Database Provider

8/27/2018 • 2 minutes to read • Edit Online

This database provider allows Entity Framework Core to be used with an in-memory database. This can be useful for testing, although the SQLite provider in in-memory mode may be a more appropriate test replacement for relational databases. The provider is maintained as part of the Entity Framework Core Project.

Install

Install the Microsoft.EntityFrameworkCore.InMemory NuGet package.

```
Install-Package Microsoft.EntityFrameworkCore.InMemory
```

Get Started

The following resources will help you get started with this provider.

- Testing with InMemory
- UnicornStore Sample Application Tests

Supported Database Engines

- Built-in in-memory database (designed for testing purposes only)

Supported Platforms

- .NET Framework (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)
- Universal Windows Platform

Writing a Database Provider

8/27/2018 • 2 minutes to read • Edit Online

For information about writing an Entity Framework Core database provider, see So you want to write an EF Core provider by Arthur Vickers.

> **NOTE**
>
> These posts have not been updated since EF Core 1.1 and there have been significant changes since that time Issue 681 is tracking updates to this documentation.

The EF Core codebase is open source and contains several database providers that can be used as a reference. You can find the source code at https://github.com/aspnet/EntityFrameworkCore. It may also be helpful to look at the code for commonly used third-party providers, such as Npgsql, Pomelo MySQL, and SQL Server Compact. In particular, these projects are setup to extend from and run functional tests that we publish on NuGet. This kind of setup is strongly recommended.

Keeping up-to-date with provider changes

Starting with work after the 2.1 release, we have created a log of changes that may need corresponding changes in provider code. This is intended to help when updating an existing provider to work with a new version of EF Core.

Prior to 2.1, we used the `providers-beware` and `providers-fyi` labels on our GitHub issues and pull requests for a similar purpose. We will continiue to use these lables on issues to give an indication which work items in a given release may also require work to be done in providers. A `providers-beware` label typically means that the implementation of an work item may break providers, while a `providers-fyi` label typically means that providers will not be broken, but code may need to be changed anyway, for example, to enable new functionality.

Suggested naming of third party providers

We suggest using the following naming for NuGet packages. This is consistent with the names of packages delivered by the EF Core team.

`<Optional project/company name>.EntityFrameworkCore.<Database engine name>`

For example:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Npgsql.EntityFrameworkCore.PostgreSQL`
- `EntityFrameworkCore.SqlServerCompact40`

Provider-impacting changes

10/25/2018 • 2 minutes to read • Edit Online

This page contains links to pull requests made on the EF Core repo that may require authors of other database providers to react. The intention is to provide a starting point for authors of existing third-party database providers when updating their provider to a new version.

We are starting this log with changes from 2.1 to 2.2. Prior to 2.1 we used the `providers-beware` and `providers-fyi` labels on our issues and pull requests.

2.1 ---> 2.2

Test-only changes

- https://github.com/aspnet/EntityFrameworkCore/pull/12057 - Allow customizable SQL delimeters in tests
  - Test changes that allow non-strict floating point comparisons in BuiltInDataTypesTestBase
  - Test changes that allow query tests to be re-used with different SQL delimeters
- https://github.com/aspnet/EntityFrameworkCore/pull/12072 - Add DbFunction tests to the relational specification tests
  - Such that these tests can be run against all database providers
- https://github.com/aspnet/EntityFrameworkCore/pull/12362 - Async test cleanup
  - Remove `Wait` calls, unneeded async, and renamed some test methods
- https://github.com/aspnet/EntityFrameworkCore/pull/12666 - Unify logging test infrastructure
  - Added `CreateListLoggerFactory` and removed some previous logging infrastructure, which will require providers using these tests to react
- https://github.com/aspnet/EntityFrameworkCore/pull/12500 - Run more query tests both synchronously and asynchronously
  - Test names and factoring has changed, which will require providers using these tests to react
- https://github.com/aspnet/EntityFrameworkCore/pull/12766 - Renaming navigations in the ComplexNavigations model
  - Providers using these tests may need to react
- https://github.com/aspnet/EntityFrameworkCore/pull/12141 - Return the context to the pool instead of disposing in functional tests
  - This change includes some test refactoring which may require providers to react

Test and product code changes

- https://github.com/aspnet/EntityFrameworkCore/pull/12109 - Consolidate RelationalTypeMapping.Clone methods
  - Changes in 2.1 to the RelationalTypeMapping allowed for a simplification in derived classes. We don't believe this was breaking to providers, but providers can take advantage of this change in their derived type mapping classes.
- https://github.com/aspnet/EntityFrameworkCore/pull/12069 - Tagged or named queries
  - Adds infrastructure for tagging LINQ queries and having those tags show up as comments in the SQL. This may require providers to react in SQL generation.
- https://github.com/aspnet/EntityFrameworkCore/pull/13115 - Support spatial data via NTS
  - Allows type mappings and member translators to be registered outside of the provider
    - Providers must call base.FindMapping() in their ITypeMappingSource implementation for it to

work
  - Follow this pattern to add spatial support to your provider that is consistent across providers.
- https://github.com/aspnet/EntityFrameworkCore/pull/13199 - Add enhanced debugging for service provider creation
  - Allows DbContextOptionsExtensions to implement a new interface that can help people understand why the internal service provider is being re-built
- https://github.com/aspnet/EntityFrameworkCore/pull/13289 - Adds CanConnect API for use by health checks
  - This PR adds the concept of `CanConnect` which will be used by ASP.NET Core health checks to determine if the database is available. By default, the relational implementation just calls `Exist`, but providers can implement something different if necessary. Non-relational providers will need to implement the new API in order for the health check to be usable.
- https://github.com/aspnet/EntityFrameworkCore/pull/13306 - Update base RelationalTypeMapping to not set DbParameter Size
  - Stop setting Size by default since it can cause truncation. Providers may need to add their own logic if Size needs to be set.
- https://github.com/aspnet/EntityFrameworkCore/pull/13372 - RevEng: Always specify column type for decimal columns
  - Always configure column type for decimal columns in scaffolded code rather than configuring by convention.
  - Providers should not require any changes on their end.
- https://github.com/aspnet/EntityFrameworkCore/pull/13469 - Adds CaseExpression for generating SQL CASE expressions
- https://github.com/aspnet/EntityFrameworkCore/pull/13648 - Adds the ability to specify type mappings on SqlFunctionExpression to improve store type inference of arguments and results.

EF Core Tools & Extensions

1/7/2019 • 3 minutes to read • Edit Online

These tools and extensions provide additional functionality for Entity Framework Core 2.0 and later.

> **IMPORTANT**
> Extensions are built by a variety of sources and aren't maintained as part of the Entity Framework Core project. When considering a third party extension, be sure to evaluate its quality, licensing, compatibility, support, etc. to ensure it meets your requirements.

Tools

LLBLGen Pro

LLBLGen Pro is an entity modeling solution with support for Entity Framework and Entity Framework Core. It lets you easily define your entity model and map it to your database, using database first or model first, so you can get started writing queries right away.

Website

Devart Entity Developer

Entity Developer is a powerful ORM designer for ADO.NET Entity Framework, NHibernate, LinqConnect, Telerik Data Access, and LINQ to SQL. It supports designing EF Core models visually, using model first or database first approaches, and C# or Visual Basic code generation.

Website

EF Core Power Tools

EF Core Power Tools is a Visual Studio 2017 extension that exposes various EF Core design-time tasks in a simple user interface. It includes reverse engineering of DbContext and entity classes from existing databases and SQL Server DACPACs, management of database migrations, and model visualizations.

GitHub wiki

Entity Framework Visual Editor

Entity Framework Visual Editor is a Visual Studio 2017 extension that adds an ORM designer for visual design of EF 6, and EF Core classes. Code is generated using T4 templates so can be customized to suit any needs. It supports inheritance, unidirectional and bidirectional associations, enumerations, and the ability to color-code your classes and add text blocks to explain potentially arcane parts of your design.

Marketplace

CatFactory

CatFactory is a scaffolding engine for .NET Core that can automate the generation of DbContext classes, entities, mapping configurations, and repository classes from a SQL Server database.

GitHub repository

LoreSoft's Entity Framework Core Generator

Entity Framework Core Generator (efg) is a .NET Core CLI tool that can generate EF Core models from an existing database, much like `dotnet ef dbcontext scaffold`, but it also supports safe code regeneration via region replacement or by parsing mapping files. This tool supports generating view models, validation, and object mapper code.

Tutorial Documentation

Extensions

Microsoft.EntityFrameworkCore.AutoHistory

A plugin library that enables automatically recording the data changes performed by EF Core into a history table.

GitHub repository

Microsoft.EntityFrameworkCore.DynamicLinq

A .NET Core / .NET Standard port of System.Linq.Dynamic that includes async support with EF Core. System.Linq.Dynamic originated as a Microsoft sample that shows how to construct LINQ queries dynamically from string expressions rather than code.

GitHub repository

EFSecondLevelCache.Core

An extension that enables storing the results of EF Core queries into a second-level cache, so that subsequent executions of the same queries can avoid accessing the database and retrieve the data directly from the cache.

GitHub repository

EntityFrameworkCore.PrimaryKey

This library allows retrieving the values of primary key (including composite keys) from any entity as a dictionary.

GitHub repository

EntityFrameworkCore.TypedOriginalValues

This library enables strongly typed access to the original values of entity properties.

GitHub repository

Geco

Geco (Generator Console) is a simple code generator based on a console project, that runs on .NET Core and uses C# interpolated strings for code generation. Geco includes a reverse model generator for EF Core with support for pluralization, singularization, and editable templates. It also provides a seed data script generator, a script runner, and a database cleaner.

GitHub repository

LinqKit.Microsoft.EntityFrameworkCore

LinqKit.Microsoft.EntityFrameworkCore is an EF Core-compatible version of the LINQKit library. LINQKit is a free set of extensions for LINQ to SQL and Entity Framework power users. It enables advanced functionality like dynamic building of predicate expressions, and using expression variables in subqueries.

GitHub repository

NeinLinq.EntityFrameworkCore

NeinLinq extends LINQ providers such as Entity Framework to enable reusing functions, rewriting queries, and building dynamic queries using translatable predicates and selectors.

GitHub repository

Microsoft.EntityFrameworkCore.UnitOfWork

A plugin for Microsoft.EntityFrameworkCore to support repository, unit of work patterns, and multiple databases with distributed transaction supported.

GitHub repository

EFCore.BulkExtensions

EF Core extensions for Bulk operations (Insert, Update, Delete).

GitHub repository

Bricelam.EntityFrameworkCore.Pluralizer

Adds design-time pluralization to EF Core.

GitHub repository

PomeloFoundation/Pomelo.EntityFrameworkCore.Extensions.ToSql

A simple extension method that obtains the SQL statement EF Core would generate for a given LINQ query in simple scenarios. The ToSql method is limited to simple scenarios because EF Core can generate more than one SQL statement for a single LINQ query, and different SQL statements depending on parameter values.

GitHub repository

Toolbelt.EntityFrameworkCore.IndexAttribute

Revival of [Index] attribute for EF Core (with extension for model building).

GitHub repository

EfCore.InMemoryHelpers

Provides a wrapper around the EF Core In-Memory Database Provider. Makes it act more like a relational provider.

GitHub repository

EFCore.TemporalSupport

An implementation of temporal support for EF Core.

GitHub repository

EntityFrameworkCore.Cacheable

A high-performance second-level query cache for EF Core.

GitHub repository

Entity Framework Core tools reference

9/28/2018 • 2 minutes to read • Edit Online

The Entity Framework Core tools help with design-time development tasks. They're primarily used to manage Migrations and to scaffold a `DbContext` and entity types by reverse engineering the schema of a database.

- The EF Core Package Manager Console tools) run in the Package Manager Console in Visual Studio. These tools work with both .NET Framework and .NET Core projects.
- The EF Core .NET command-line interface (CLI) tools are an extension to the cross-platform .NET Core CLI tools. These tools require a .NET Core SDK project (one with `Sdk="Microsoft.NET.Sdk"` or similar in the project file).

Both tools expose the same functionality. If you're developing in Visual Studio, we recommend using the **Package Manager Console** tools since they provide a more integrated experience.

Next steps

- EF Core Package Manager Console tools reference
- EF Core .NET CLI tools reference

Entity Framework Core tools reference - Package Manager Console in Visual Studio

11/15/2018 • 8 minutes to read • Edit Online

The Package Manager Console (PMC) tools for Entity Framework Core perform design-time development tasks. For example, they create migrations, apply migrations, and generate code for a model based on an existing database. The commands run inside of Visual Studio using the Package Manager Console. These tools work with both .NET Framework and .NET Core projects.

If you aren't using Visual Studio, we recommend the EF Core Command-line Tools instead. The CLI tools are cross-platform and run inside a command prompt.

Installing the tools

The procedures for installing and updating the tools differ between ASP.NET Core 2.1+ and earlier versions or other project types.

ASP.NET Core version 2.1 and later

The tools are automatically included in an ASP.NET Core 2.1+ project because the `Microsoft.EntityFrameworkCore.Tools` package is included in the Microsoft.AspNetCore.App metapackage.

Therefore, you don't have to do anything to install the tools, but you do have to:

- Restore packages before using the tools in a new project.
- Install a package to update the tools to a newer version.

To make sure that you're getting the latest version of the tools, we recommend that you also do the following step:

- Edit your *.csproj* file and add a line specifying the latest version of the Microsoft.EntityFrameworkCore.Tools package. For example, the *.csproj* file might include an `ItemGroup` that looks like this:

```
<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>
```

Update the tools when you get a message like the following example:

> The EF Core tools version '2.1.1-rtm-30846' is older than that of the runtime '2.1.3-rtm-32065'. Update the tools for the latest features and bug fixes.

To update the tools:

- Install the latest .NET Core SDK.
- Update Visual Studio to the latest version.
- Edit the *.csproj* file so that it includes a package reference to the latest tools package, as shown earlier.

Other versions and project types

Install the Package Manager Console tools by running the following command in **Package Manager Console**:

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

Update the tools by running the following command in **Package Manager Console**.

```
Update-Package Microsoft.EntityFrameworkCore.Tools
```

Verify the installation

Verify that the tools are installed by running this command:

```
Get-Help about_EntityFrameworkCore
```

The output looks like this (it doesn't tell you which version of the tools you're using):

```

                     _/\__
               ---==/    \\
         ___  ___   |.    \|\
        | __|| __|  |  )   \\\
        | _| | _|   \_/ |  //|\\
        |___||_|       /   \\\/\\

TOPIC
    about_EntityFrameworkCore

SHORT DESCRIPTION
    Provides information about the Entity Framework Core Package Manager Console Tools.

<A list of available commands follows, omitted here.>
```

Using the tools

Before using the tools:

- Understand the difference between target and startup project.
- Learn how to use the tools with .NET Standard class libraries.
- For ASP.NET Core projects, set the environment.

Target and startup project

The commands refer to a *project* and a *startup project*.

- The *project* is also known as the *target project* because it's where the commands add or remove files. By default, the **Default project** selected in **Package Manager Console** is the target project. You can specify a different project as target project by using the `--project` option.
- The *startup project* is the one that the tools build and run. The tools have to execute application code at design time to get information about the project, such as the database connection string and the configuration of the model. By default, the **Startup Project** in **Solution Explorer** is the startup project. You can specify a different project as startup project by using the `--startup-project` option.

The startup project and target project are often the same project. A typical scenario where they are separate projects is when:

- The EF Core context and entity classes are in a .NET Core class library.

- A .NET Core console app or web app references the class library.

It's also possible to put migrations code in a class library separate from the EF Core context.

Other target frameworks

The Package Manager Console tools work with .NET Core or .NET Framework projects. Apps that have the EF Core model in a .NET Standard class library might not have a .NET Core or .NET Framework project. For example, this is true of Xamarin and Universal Windows Platform apps. In such cases, you can create a .NET Core or .NET Framework console app project whose only purpose is to act as startup project for the tools. The project can be a dummy project with no real code — it is only needed to provide a target for the tooling.

Why is a dummy project required? As mentioned earlier, the tools have to execute application code at design time. To do that, they need to use the .NET Core or .NET Framework runtime. When the EF Core model is in a project that targets .NET Core or .NET Framework, the EF Core tools borrow the runtime from the project. They can't do that if the EF Core model is in a .NET Standard class library. The .NET Standard is not an actual .NET implementation; it's a specification of a set of APIs that .NET implementations must support. Therefore .NET Standard is not sufficient for the EF Core tools to execute application code. The dummy project you create to use as startup project provides a concrete target platform into which the tools can load the .NET Standard class library.

ASP.NET Core environment

To specify the environment for ASP.NET Core projects, set **env:ASPNETCORE_ENVIRONMENT** before running commands.

Common parameters

The following table shows parameters that are common to all of the EF Core commands:

PARAMETER	DESCRIPTION
-Context <String>	The `DbContext` class to use. Class name only or fully qualified with namespaces. If this parameter is omitted, EF Core finds the context class. If there are multiple context classes, this parameter is required.
-Project <String>	The target project. If this parameter is omitted, the **Default project** for **Package Manager Console** is used as the target project.
-StartupProject <String>	The startup project. If this parameter is omitted, the **Startup project** in **Solution properties** is used as the target project.
-Verbose	Show verbose output.

To show help information about a command, use PowerShell's `Get-Help` command.

> **TIP**
>
> The Context, Project, and StartupProject parameters support tab-expansion.

Add-Migration

Adds a new migration.

Parameters:

PARAMETER	DESCRIPTION
-Name <String>	The name of the migration. This is a positional parameter and is required.
-OutputDir <String>	The directory (and sub-namespace) to use. Paths are relative to the target project directory. Defaults to "Migrations".

Drop-Database

Drops the database.

Parameters:

PARAMETER	DESCRIPTION
-WhatIf	Show which database would be dropped, but don't drop it.

Get-DbContext

Lists available `DbContext` types.

Remove-Migration

Removes the last migration (rolls back the code changes that were done for the migration).

Parameters:

PARAMETER	DESCRIPTION
-Force	Revert the migration (roll back the changes that were applied to the database).

Scaffold-DbContext

Generates code for a `DbContext` and entity types for a database. In order for `Scaffold-DbContext` to generate an entity type, the database table must have a primary key.

Parameters:

PARAMETER	DESCRIPTION
-Connection <String>	The connection string to the database. For ASP.NET Core 2.x projects, the value can be *name=<name of connection string>*. In that case the name comes from the configuration sources that are set up for the project. This is a positional parameter and is required.
-Provider <String>	The provider to use. Typically this is the name of the NuGet package, for example: `Microsoft.EntityFrameworkCore.SqlServer`. This is a positional parameter and is required.
-OutputDir <String>	The directory to put files in. Paths are relative to the project directory.

PARAMETER	DESCRIPTION
-ContextDir <String>	The directory to put the `DbContext` file in. Paths are relative to the project directory.
-Context <String>	The name of the `DbContext` class to generate.
-Schemas <String[]>	The schemas of tables to generate entity types for. If this parameter is omitted, all schemas are included.
-Tables <String[]>	The tables to generate entity types for. If this parameter is omitted, all tables are included.
-DataAnnotations	Use attributes to configure the model (where possible). If this parameter is omitted, only the fluent API is used.
-UseDatabaseNames	Use table and column names exactly as they appear in the database. If this parameter is omitted, database names are changed to more closely conform to C# name style conventions.
-Force	Overwrite existing files.

Example:

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

Example that scaffolds only selected tables and creates the context in a separate folder with a specified name:

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables "Blog","Post" -ContextDir Context -Context
BlogContext
```

Script-Migration

Generates a SQL script that applies all of the changes from one selected migration to another selected migration.

Parameters:

PARAMETER	DESCRIPTION
-*From* <String>	The starting migration. Migrations may be identified by name or by ID. The number 0 is a special case that means *before the first migration*. Defaults to 0.
-*To* <String>	The ending migration. Defaults to the last migration.
-Idempotent	Generate a script that can be used on a database at any migration.

PARAMETER	DESCRIPTION
-Output <String>	The file to write the result to. IF this parameter is omitted, the file is created with a generated name in the same folder as the app's runtime files are created, for example: */obj/Debug/netcoreapp2.1/ghbkztfz.sql/*.

> **TIP**
> The To, From, and Output parameters support tab-expansion.

The following example creates a script for the InitialCreate migration, using the migration name.

```
Script-Migration -To InitialCreate
```

The following example creates a script for all migrations after the InitialCreate migration, using the migration ID.

```
Script-Migration -From 20180904195021_InitialCreate
```

Update-Database

Updates the database to the last migration or to a specified migration.

PARAMETER	DESCRIPTION
-*Migration* <String>	The target migration. Migrations may be identified by name or by ID. The number 0 is a special case that means *before the first migration* and causes all migrations to be reverted. If no migration is specified, the command defaults to the last migration.

> **TIP**
> The Migration parameter supports tab-expansion.

The following example reverts all migrations.

```
Update-Database -Migration 0
```

The following examples update the database to a specified migration. The first uses the migration name and the second uses the migration ID:

```
Update-Database -Migration InitialCreate
Update-Database -Migration 20180904195021_InitialCreate
```

Additional resources

- Migrations
- Reverse Engineering

Entity Framework Core tools reference - .NET CLI

11/15/2018 • 8 minutes to read • Edit Online

The command-line interface (CLI) tools for Entity Framework Core perform design-time development tasks. For example, they create migrations, apply migrations, and generate code for a model based on an existing database. The commands are an extension to the cross-platform dotnet command, which is part of the .NET Core SDK. These tools work with .NET Core projects.

If you're using Visual Studio, we recommend the Package Manager Console tools instead:

- They automatically work with the current project selected in the **Package Manager Console** without requiring that you manually switch directories.
- They automatically open files generated by a command after the command is completed.

Installing the tools

The installation procedure depends on project type and version:

- ASP.NET Core version 2.1 and later
- EF Core 2.x
- EF Core 1.x

ASP.NET Core 2.1+

- Install the current .NET Core SDK. The SDK has to be installed even if you have the latest version of Visual Studio 2017.

  This is all that is needed for ASP.NET Core 2.1+ because the `Microsoft.EntityFrameworkCore.Design` package is included in the Microsoft.AspNetCore.App metapackage.

EF Core 2.x (not ASP.NET Core)

The `dotnet ef` commands are included in the .NET Core SDK, but to enable the commands you have to install the `Microsoft.EntityFrameworkCore.Design` package.

- Install the current .NET Core SDK. The SDK has to be installed even if you have the latest version of Visual Studio 2017.
- Install the latest stable `Microsoft.EntityFrameworkCore.Design` package.

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

EF Core 1.x

- Install the .NET Core SDK version 2.1.200. Later versions are not compatible with CLI tools for EF Core 1.0 and 1.1.
- Configure the application to use the 2.1.200 SDK version by modifying its global.json file. This file is normally included in the solution directory (one above the project).
- Edit the project file and add `Microsoft.EntityFrameworkCore.Tools.DotNet` as a `DotNetCliToolReference` item. Specify the latest 1.x version, for example: 1.1.6. See the project file example at the end of this section.
- Install the latest 1.x version of the `Microsoft.EntityFrameworkCore.Design` package, for example:

```
dotnet add package Microsoft.EntityFrameworkCore.Design -v 1.1.6
```

With both package references added, the project file looks something like this:

```
<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>
```

A package reference with `PrivateAssets="All"` isn't exposed to projects that reference this project. This restriction is especially useful for packages that are typically only used during development.

Verify installation

Run the following commands to verify that EF Core CLI tools are correctly installed:

```
dotnet restore
dotnet ef
```

The output from the command identifies the version of the tools in use:

```

                     _/\__
               ---==/    \\
         ___  ___   |.    \|\
        | __|| __|  |  )   \\\
        | _| | _|   \_/ |  //|\\
        |___||_|       /   \\\/\\

Entity Framework Core .NET Command-line Tools 2.1.3-rtm-32065

<Usage documentation follows, not shown.>
```

Using the tools

Before using the tools, you might have to create a startup project or set the environment.

Target project and startup project

The commands refer to a *project* and a *startup project*.

- The *project* is also known as the *target project* because it's where the commands add or remove files. By default, the project in the current directory is the target project. You can specify a different project as target project by using the `--project` option.
- The *startup project* is the one that the tools build and run. The tools have to execute application code at design time to get information about the project, such as the database connection string and the configuration of the model. By default, the project in the current directory is the startup project. You can

specify a different project as startup project by using the `--startup-project` option.

The startup project and target project are often the same project. A typical scenario where they are separate projects is when:

- The EF Core context and entity classes are in a .NET Core class library.
- A .NET Core console app or web app references the class library.

It's also possible to put migrations code in a class library separate from the EF Core context.

Other target frameworks

The CLI tools work with .NET Core projects and .NET Framework projects. Apps that have the EF Core model in a .NET Standard class library might not have a .NET Core or .NET Framework project. For example, this is true of Xamarin and Universal Windows Platform apps. In such cases, you can create a .NET Core console app project whose only purpose is to act as startup project for the tools. The project can be a dummy project with no real code — it is only needed to provide a target for the tooling.

Why is a dummy project required? As mentioned earlier, the tools have to execute application code at design time. To do that, they need to use the .NET Core runtime. When the EF Core model is in a project that targets .NET Core or .NET Framework, the EF Core tools borrow the runtime from the project. They can't do that if the EF Core model is in a .NET Standard class library. The .NET Standard is not an actual .NET implementation; it's a specification of a set of APIs that .NET implementations must support. Therefore .NET Standard is not sufficient for the EF Core tools to execute application code. The dummy project you create to use as startup project provides a concrete target platform into which the tools can load the .NET Standard class library.

ASP.NET Core environment

To specify the environment for ASP.NET Core projects, set the **ASPNETCORE_ENVIRONMENT** environment variable before running commands.

Common options

	OPTION	DESCRIPTION
	`--json`	Show JSON output.
`-c`	`--context <DBCONTEXT>`	The `DbContext` class to use. Class name only or fully qualified with namespaces. If this option is omitted, EF Core will find the context class. If there are multiple context classes, this option is required.
`-p`	`--project <PROJECT>`	Relative path to the project folder of the target project. Default value is the current folder.
`-s`	`--startup-project <PROJECT>`	Relative path to the project folder of the startup project. Default value is the current folder.
	`--framework <FRAMEWORK>`	The Target Framework Moniker for the target framework. Use when the project file specifies multiple target frameworks, and you want to select one of them.

	OPTION	DESCRIPTION
	`--configuration <CONFIGURATION>`	The build configuration, for example: `Debug` or `Release`.
	`--runtime <IDENTIFIER>`	The identifier of the target runtime to restore packages for. For a list of Runtime Identifiers (RIDs), see the RID catalog.
`-h`	`--help`	Show help information.
`-v`	`--verbose`	Show verbose output.
	`--no-color`	Don't colorize output.
	`--prefix-output`	Prefix output with level.

dotnet ef database drop

Drops the database.

Options:

	OPTION	DESCRIPTION
`-f`	`--force`	Don't confirm.
	`--dry-run`	Show which database would be dropped, but don't drop it.

dotnet ef database update

Updates the database to the last migration or to a specified migration.

Arguments:

ARGUMENT	DESCRIPTION
`<MIGRATION>`	The target migration. Migrations may be identified by name or by ID. The number 0 is a special case that means *before the first migration* and causes all migrations to be reverted. If no migration is specified, the command defaults to the last migration.

The following examples update the database to a specified migration. The first uses the migration name and the second uses the migration ID:

```
dotnet ef database update InitialCreate
dotnet ef database update 20180904195021_InitialCreate
```

dotnet ef dbcontext info

Gets information about a `DbContext` type.

dotnet ef dbcontext list

Lists available `DbContext` types.

dotnet ef dbcontext scaffold

Generates code for a `DbContext` and entity types for a database. In order for this command to generate an entity type, the database table must have a primary key.

Arguments:

ARGUMENT	DESCRIPTION
`<CONNECTION>`	The connection string to the database. For ASP.NET Core 2.x projects, the value can be *name=<name of connection string>*. In that case the name comes from the configuration sources that are set up for the project.
`<PROVIDER>`	The provider to use. Typically this is the name of the NuGet package, for example: `Microsoft.EntityFrameworkCore.SqlServer`.

Options:

	OPTION	DESCRIPTION
-d	`--data-annotations`	Use attributes to configure the model (where possible). If this option is omitted, only the fluent API is used.
`-c`	`--context <NAME>`	The name of the `DbContext` class to generate.
	`--context-dir <PATH>`	The directory to put the `DbContext` class file in. Paths are relative to the project directory. Namespaces are derived from the folder names.
`-f`	`--force`	Overwrite existing files.
`-o`	`--output-dir <PATH>`	The directory to put entity class files in. Paths are relative to the project directory.
	`--schema <SCHEMA_NAME>...`	The schemas of tables to generate entity types for. To specify multiple schemas, repeat `--schema` for each one. If this option is omitted, all schemas are included.
`-t`	`--table <TABLE_NAME>`...	The tables to generate entity types for. To specify multiple tables, repeat `-t` or `--table` for each one. If this option is omitted, all tables are included.

	OPTION	DESCRIPTION
	`--use-database-names`	Use table and column names exactly as they appear in the database. If this option is omitted, database names are changed to more closely conform to C# name style conventions.

The following example scaffolds all schemas and tables and puts the new files in the *Models* folder.

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -o Models
```

The following example scaffolds only selected tables and creates the context in a separate folder with a specified name:

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"
Microsoft.EntityFrameworkCore.SqlServer -o Models -t Blog -t Post --context-dir Context -c BlogContext
```

dotnet ef migrations add

Adds a new migration.

Arguments:

ARGUMENT	DESCRIPTION
`<NAME>`	The name of the migration.

Options:

	OPTION	DESCRIPTION
`-o`	`--output-dir <PATH>`	The directory (and sub-namespace) to use. Paths are relative to the project directory. Defaults to "Migrations".

dotnet ef migrations list

Lists available migrations.

dotnet ef migrations remove

Removes the last migration (rolls back the code changes that were done for the migration).

Options:

	OPTION	DESCRIPTION
`-f`	`--force`	Revert the migration (roll back the changes that were applied to the database).

dotnet ef migrations script

Generates a SQL script from migrations.

Arguments:

ARGUMENT	DESCRIPTION
`<FROM>`	The starting migration. Migrations may be identified by name or by ID. The number 0 is a special case that means *before the first migration*. Defaults to 0.
`<TO>`	The ending migration. Defaults to the last migration.

Options:

	OPTION	DESCRIPTION
`-o`	`--output <FILE>`	The file to write the script to.
`-i`	`--idempotent`	Generate a script that can be used on a database at any migration.

The following example creates a script for the InitialCreate migration:

```
dotnet ef migrations script 0 InitialCreate
```

The following example creates a script for all migrations after the InitialCreate migration.

```
dotnet ef migrations script 20180904195021_InitialCreate
```

Additional resources

- Migrations
- Reverse Engineering

Design-time DbContext Creation

8/27/2018 • 2 minutes to read • Edit Online

Some of the EF Core Tools commands (for example, the Migrations commands) require a derived `DbContext` instance to be created at design time in order to gather details about the application's entity types and how they map to a database schema. In most cases, it is desirable that the `DbContext` thereby created is configured in a similar way to how it would be configured at run time.

There are various ways the tools try to create the `DbContext`:

From application services

If your startup project is an ASP.NET Core app, the tools try to obtain the DbContext object from the application's service provider.

The tools first try to obtain the service provider by invoking `Program.BuildWebHost()` and accessing the `IWebHost.Services` property.

> **NOTE**
>
> When you create a new ASP.NET Core 2.0 application, this hook is included by default. In previous versions of EF Core and ASP.NET Core, the tools try to invoke `Startup.ConfigureServices` directly in order to obtain the application's service provider, but this pattern no longer works correctly in ASP.NET Core 2.0 applications. If you are upgrading an ASP.NET Core 1.x application to 2.0, you can modify your `Program` class to follow the new pattern.

The `DbContext` itself and any dependencies in its constructor need to be registered as services in the application's service provider. This can be easily achieved by having a constructor on the `DbContext` that takes an instance of `DbContextOptions<TContext>` as an argument and using the `AddDbContext<TContext>` method.

Using a constructor with no parameters

If the DbContext can't be obtained from the application service provider, the tools look for the derived `DbContext` type inside the project. Then they try to create an instance using a constructor with no parameters. This can be the default constructor if the `DbContext` is configured using the `OnConfiguring` method.

From a design-time factory

You can also tell the tools how to create your DbContext by implementing the `IDesignTimeDbContextFactory<TContext>` interface: If a class implementing this interface is found in either the same project as the derived `DbContext` or in the application's startup project, the tools bypass the other ways of creating the DbContext and use the design-time factory instead.

```
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

The `args` parameter is currently unused. There is an issue tracking the ability to specify design-time arguments from the tools.

A design-time factory can be especially useful if you need to configure the DbContext differently for design time than at run time, if the `DbContext` constructor takes additional parameters are not registered in DI, if you are not using DI at all, or if for some reason you prefer not to have a `BuildWebHost` method in your ASP.NET Core application's `Main` class.

Design-time services

8/27/2018 • 2 minutes to read • Edit Online

Some services used by the tools are only used at design time. These services are managed separately from EF Core's runtime services to prevent them from being deployed with your app. To override one of these services (for example the service to generate migration files), add an implementation of `IDesignTimeServices` to your startup project.

```
class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
        => services.AddSingleton<IMigrationsCodeGenerator, MyMigrationsCodeGenerator>()
}
```

Entity Framework 6

10/14/2018 • 2 minutes to read • Edit Online

Entity Framework 6 (EF6) is a tried and tested object-relational mapper (O/RM) for .NET with many years of feature development and stabilization.

As an O/RM, EF6 reduces the impedance mismatch between the relational and object-oriented worlds, enabling developers to write applications that interact with data stored in relational databases using strongly-typed .NET objects that represent the application's domain, and eliminating the need for a large portion of the data access "plumbing" code that they usually need to write.

EF6 implements many popular O/RM features:

- Mapping of POCO entity classes which do not depend on any EF types
- Automatic change tracking
- Identity resolution and Unit of Work
- Eager, lazy and explicit loading
- Translation of strongly-typed queries using LINQ (Language INtegrated Query)
- Rich mapping capabilities, including support for:
  - One-to-one, one-to-many and many-to-many relationships
  - Inheritance (table per hierarchy, table per type and table per concrete class)
  - Complex types
  - Stored procedures
- A visual designer to create entity models.
- A "Code First" experience to create entity models by writing code.
- Models can either be generated from existing databases and then hand-edited, or they can be created from scratch and then used to generate new databases.
- Integration with .NET Framework application models, including ASP.NET, and through databinding, with WPF and WinForms.
- Database connectivity based on ADO.NET and numerous providers available to connect to SQL Server, Oracle, MySQL, SQLite, PostgreSQL, DB2, etc.

Should I use EF6 or EF Core?

EF Core is a more modern, lightweight and extensible version of Entity Framework that has very similar capabilities and benefits to EF6. EF Core is a complete rewrite and contains many new features not available in EF6, although it also still lacks some of the most advanced mapping capabilities of EF6. Consider using EF Core in new applications if the feature set matches your requirements. Compare EF Core & EF6 examines this choice in greater detail.

Get Started

Add the EntityFramework NuGet package to your project or install the Entity Framework Tools for Visual Studio. Then watch videos, read tutorials, and advanced documentation to help you make the most of EF6.

Past Entity Framework Versions

This is the documentation for the latest version of Entity Framework 6, although much of it also applies to past releases. Check out What's New and Past Releases for a complete list of EF releases and the features they introduced.

What's New in EF6

9/13/2018 • 2 minutes to read • Edit Online

We highly recommend that you use the latest released version of Entity Framework to ensure you get the latest features and the highest stability. However, we realize that you may need to use a previous version, or that you may want to experiment with new improvements in the latest pre-release. To install specific versions of EF, see Get Entity Framework.

This page documents the features that are included on each new release.

Recent releases

EF Tools Update in Visual Studio 2017 15.7

In May 2018, we released an updated version of the EF Tools as part of Visual Studio 2017 15.7. It includes improvements for some common pain points:

- Fixes for several user interface accessibility bugs
- Workaround for SQL Server performance regression when generating models from existing databases #4
- Support for updating models for larger models on SQL Server #185

Another improvement in this this new version of EF Tools is that it installs the EF 6.2 runtime when creating a model in a new project. With older versions of Visual Studio, it is possible to use the EF 6.2 runtime (as well as any past version of EF) by installing the corresponding version of the NuGet package.

EF 6.2 Runtime

The EF 6.2 runtime was released to NuGet in October of 2017. Thanks in great part to the efforts our community of open source contributors, EF 6.2 includes numerous bugs fixes and product enhancements.

Here is a brief list of the most important changes affecting the EF 6.2 runtime:

- Reduce start up time by loading finished code first models from a persistent cache #275
- Fluent API to define indexes #274
- DbFunctions.Like() to enable writing LINQ queries that translate to LIKE in SQL #241
- Migrate.exe now supports -script option #240
- EF6 can now work with key values generated by a sequence in SQL Server #165
- Update list of transient errors for SQL Azure Execution Strategy #83
- Bug: Retrying queries or SQL commands fails with "The SqlParameter is already contained by another SqlParameterCollection" #81
- Bug: Evaluation of DbQuery.ToString() frequently times out in the debugger #73

Future Releases

For information on future version of EF6, please look at our Roadmap.

Past Releases

The Past Releases page contains an archive of all previous versions of EF and the major features that were introduced on each release.

Future Versions of Entity Framework

10/25/2018 • 2 minutes to read • Edit Online

Here you can find information on upcoming versions of Entity Framework. While most of the focus of the EF team is nowadays on adding new features and improvements to EF Core, we plan to still fix important bugs, implement small improvements, and incorporate community contributions in the EF6 codebase.

Post-EF 6.2 releases

The roadmap of post-EF 6.2 releases is still being developed. More information will be posted here once available.

Staying Up To Date

Besides this page, new releases are usually announced on the .NET team blog and our Twitter account, @efmagicunicorns.

Past Releases of Entity Framework

9/18/2018 • 12 minutes to read • Edit Online

The first version of Entity Framework was released in 2008, as part of .NET Framework 3.5 SP1 and Visual Studio 2008 SP1.

Starting with the EF4.1 release it has shipped as the EntityFramework NuGet Package - currently one of the most popular packages on NuGet.org.

Between versions 4.1 and 5.0, the EntityFramework NuGet package extended the EF libraries that shipped as part of .NET Framework.

Starting with version 6, EF became an open source project and also moved completely out of band form the .NET Framework. This means that when you add the EntityFramework version 6 NuGet package to an application, you are getting a complete copy of the EF library that does not depend on the EF bits that ship as part of .NET Framework. This helped somewhat accelerate the pace of development and delivery of new features.

In June 2016, we released EF Core 1.0. EF Core is based on a new codebase and is designed as a more lightweight and extensible version of EF. Currently EF Core is the main focus of development for the Entity Framework Team at Microsoft. This means there are no new major features planned for EF6. However EF6 is still maintained as an open source project and a supported Microsoft product.

Here is the list of past releases, in reverse chronological order, with information on the new features that were introduced in each release.

EF 6.1.3

The EF 6.1.3 runtime was released to NuGet in October of 2015. This release contains only fixes to high-priority defects and regressions reported on the 6.1.2 release. The fixes include:

- Query: Regression in EF 6.1.2: OUTER APPLY introduced and more complex queries for 1:1 relationships and “let” clause
- TPT problem with hiding base class property in inherited class
- DbMigration.Sql fails when the word ‘go’ is contained in the text
- Create compatibility flag for UnionAll and Intersect flattening support
- Query with multiple Includes does not work in 6.1.2 (working in 6.1.1)
- “You have an error in your SQL syntax” exception after upgrading from EF 6.1.1 to 6.1.2

EF 6.1.2

The EF 6.1.2 runtime was released to NuGet in December of 2014. This version is mostly about bug fixes. We also accepted a couple of noteworthy changes from members of the community:

- **Query cache parameters can be configured from the app/web.configuration file**

```
<entityFramework>
  <queryCache size='1000' cleaningIntervalInSeconds='-1'/>
</entityFramework>
```

- **SqlFile and SqlResource methods on DbMigration** allow you to run a SQL script stored as a file or embedded resource.

EF 6.1.1

The EF 6.1.1 runtime was released to NuGet in June of 2014. This version contains fixes for issues that a number of people have encountered. Among others:

- Designer: Error opening EF5 edmx with decimal precision in EF6 designer
- Default instance detection logic for LocalDB doesn't work with SQL Server 2014

EF 6.1.0

The EF 6.1.0 runtime was released to NuGet in March of 2014. This minor update includes a significant number of new features:

- **Tooling consolidation** provides a consistent way to create a new EF model. This feature extends the ADO.NET Entity Data Model wizard to support creating Code First models, including reverse engineering from an existing database. These features were previously available in Beta quality in the EF Power Tools.
- **Handling of transaction commit failures** provides the CommitFailureHandler which makes use of the newly introduced ability to intercept transaction operations. The CommitFailureHandler allows automatic recovery from connection failures whilst committing a transaction.
- **IndexAttribute** allows indexes to be specified by placing an `[Index]` attribute on a property (or properties) in your Code First model. Code First will then create a corresponding index in the database.
- **The public mapping API** provides access to the information EF has on how properties and types are mapped to columns and tables in the database. In past releases this API was internal.
- **Ability to configure interceptors via the App/Web.config file** allows interceptors to be added without recompiling the application.
- **System.Data.Entity.Infrastructure.Interception.DatabaseLogger**is a new interceptor that makes it easy to log all database operations to a file. In combination with the previous feature, this allows you to easily switch on logging of database operations for a deployed application, without the need to recompile.
- **Migrations model change detection** has been improved so that scaffolded migrations are more accurate; performance of the change detection process has also been enhanced.
- **Performance improvements** including reduced database operations during initialization, optimizations for null equality comparison in LINQ queries, faster view generation (model creation) in more scenarios, and more efficient materialization of tracked entities with multiple associations.

EF 6.0.2

The EF 6.0.2 runtime was released to NuGet in December of 2013. This patch release is limited to fixing issues that were introduced in the EF6 release (regressions in performance/behavior since EF5).

EF 6.0.1

The EF 6.0.1 runtime was released to NuGet in October of 2013 simultaneously with EF 6.0.0, because the latter was embedded in a version of Visual Studio that had locked down a few months before. This patch release is limited to fixing issues that were introduced in the EF6 release (regressions in performance/behavior since EF5). The most notable changes were to fix some performance issues during warm-up for EF models. This was important as warm-up performance was an area of focus in EF6 and these issues were negating some of the other performance gains made in EF6.

EF 6.0

The EF 6.0.0 runtime was released to NuGet in October of 2013. This is the first version in which a complete EF runtime is included in the EntityFramework NuGet Package which does not depend on the EF bits that are part of the .NET Framework. Moving the remaining parts of the runtime to the NuGet package required a number of breaking change for existing code. See the section on Upgrading to Entity Framework 6 for more details on the manual steps required to upgrade.

This release includes numerous new features. The following features work for models created with Code First or the EF Designer:

- **Async Query and Save** adds support for the task-based asynchronous patterns that were introduced in .NET 4.5.
- **Connection Resiliency** enables automatic recovery from transient connection failures.
- **Code-Based Configuration** gives you the option of performing configuration – that was traditionally performed in a config file – in code.
- **Dependency Resolution** introduces support for the Service Locator pattern and we've factored out some pieces of functionality that can be replaced with custom implementations.
- **Interception/SQL logging** provides low-level building blocks for interception of EF operations with simple SQL logging built on top.
- **Testability improvements** make it easier to create test doubles for DbContext and DbSet when using a mocking framework or writing your own test doubles.
- **DbContext can now be created with a DbConnection that is already opened** which enables scenarios where it would be helpful if the connection could be open when creating the context (such as sharing a connection between components where you can not guarantee the state of the connection).
- **Improved Transaction Support** provides support for a transaction external to the framework as well as improved ways of creating a transaction within the Framework.
- **Enums, Spatial and Better Performance on .NET 4.0** - By moving the core components that used to be in the .NET Framework into the EF NuGet package we are now able to offer enum support, spatial data types and the performance improvements from EF5 on .NET 4.0.
- **Improved performance of Enumerable.Contains in LINQ queries**.
- **Improved warm up time (view generation)**, especially for large models.
- **Pluggable Pluralization & Singularization Service**.
- **Custom implementations of Equals or GetHashCode** on entity classes are now supported.
- **DbSet.AddRange/RemoveRange** provides an optimized way to add or remove multiple entities from a set.
- **DbChangeTracker.HasChanges** provides an easy and efficient way to see if there are any pending changes to be saved to the database.
- **SqlCeFunctions** provides a SQL Compact equivalent to the SqlFunctions.

The following features apply to Code First only:

- **Custom Code First Conventions** allow write your own conventions to help avoid repetitive configuration. We provide a simple API for lightweight conventions as well as some more complex building blocks to allow you to author more complicated conventions.
- **Code First Mapping to Insert/Update/Delete Stored Procedures** is now supported.
- **Idempotent migrations scripts** allow you to generate a SQL script that can upgrade a database at any version up to the latest version.
- **Configurable Migrations History Table** allows you to customize the definition of the migrations history table. This is particularly useful for database providers that require the appropriate data types etc. to be specified for the Migrations History table to work correctly.
- **Multiple Contexts per Database** removes the previous limitation of one Code First model per database when using Migrations or when Code First automatically created the database for you.
- **DbModelBuilder.HasDefaultSchema** is a new Code First API that allows the default database schema for a Code First model to be configured in one place. Previously the Code First default schema was hard-coded to "dbo" and the only way to configure the schema to which a table belonged was via the ToTable API.
- **DbModelBuilder.Configurations.AddFromAssembly method** allows you to easily add all configuration

classes defined in an assembly when you are using configuration classes with the Code First Fluent API.
- **Custom Migrations Operations** enabled you to add additional operations to be used in your code-based migrations.
- **Default transaction isolation level is changed to READ_COMMITTED_SNAPSHOT** for databases created using Code First, allowing for more scalability and fewer deadlocks.
- **Entity and complex types can now be nestedinside classes**. |

EF 5.0

The EF 5.0.0 runtime was released to NuGet in August of 2012. This release introduces some new features including enum support, table-valued functions, spatial data types and various performance improvements.

The Entity Framework Designer in Visual Studio 2012 also introduces support for multiple-diagrams per model, coloring of shapes on the design surface and batch import of stored procedures.

Here is a list of content we put together specifically for the EF 5 release.

- EF 5 Release Post
- New Features in EF5
  - Enum Support in Code First
  - Enum Support in EF Designer
  - Spatial Data Types in Code First
  - Spatial Data Types in EF Designer
  - Provider Support for Spatial Types
  - Table-Valued Functions
  - Multiple Diagrams per Model
- Setting up your model
  - Creating a Model
  - Connections and Models
  - Performance Considerations
  - Working with Microsoft SQL Azure
  - Configuration File Settings
  - Glossary
  - Code First
    - Code First to a new database (walkthrough and video)
    - Code First to an existing database (walkthrough and video)
    - Conventions
    - Data Annotations
    - Fluent API - Configuring/Mapping Properties & Types
    - Fluent API - Configuring Relationships
    - Fluent API with VB.NET
    - Code First Migrations
    - Automatic Code First Migrations
    - Migrate.exe
    - Defining DbSets
  - EF Designer
    - Model First (walkthrough and video)
    - Database First (walkthrough and video)
    - Complex Types

    - Associations/Relationships
    - TPT Inheritance Pattern
    - TPH Inheritance Pattern
    - Query with Stored Procedures
    - Stored Procedures with Multiple Result Sets
    - Insert, Update & Delete with Stored Procedures
    - Map an Entity to Multiple Tables (Entity Splitting)
    - Map Multiple Entities to One Table (Table Splitting)
    - Defining Queries
    - Code Generation Templates
    - Reverting to ObjectContext
- Using Your Model
  - Working with DbContext
  - Querying/Finding Entities
  - Working with Relationships
  - Loading Related Entities
  - Working with Local Data
  - N-Tier Applications
  - Raw SQL Queries
  - Optimistic Concurrency Patterns
  - Working with Proxies
  - Automatic Detect Changes
  - No-Tracking Queries
  - The Load Method
  - Add/Attach and Entity States
  - Working with Property Values
  - Data Binding with WPF (Windows Presentation Foundation)
  - Data Binding with WinForms (Windows Forms)

EF 4.3.1

The EF 4.3.1 runtime was released to NuGet in February 2012 shortly after EF 4.3.0. This patch release included some bug fixes to the EF 4.3 release and introduced better LocalDB support for customers using EF 4.3 with Visual Studio 2012.

Here is a list of content we put together specifically for the EF 4.3.1 release, most of the content provided for EF 4.1 still applies to EF 4.3 as well.

- EF 4.3.1 Release Blog Post

EF 4.3

The EF 4.3.0 runtime was released to NuGet in February of 2012. This release included the new Code First Migrations feature that allows a database created by Code First to be incrementally changed as your Code First model evolves.

Here is a list of content we put together specifically for the EF 4.3 release, most of the content provided for EF 4.1 still applies to EF 4.3 as well:

- EF 4.3 Release Post
- EF 4.3 Code-Based Migrations Walkthrough

- EF 4.3 Automatic Migrations Walkthrough

EF 4.2

The EF 4.2.0 runtime was released to NuGet in November of 2011. This release includes bug fixes to the EF 4.1.1 release. Because this release only included bug fixes it could have been the EF 4.1.2 patch release but we opted to move to 4.2 to allow us to move away from the date based patch version numbers we used in the 4.1.x releases and adopt the Semantic Versioning standard for semantic versioning.

Here is a list of content we put together specifically for the EF 4.2 release, the content provided for EF 4.1 still applies to EF 4.2 as well.

- EF 4.2 Release Post
- Code First Walkthrough
- Model & Database First Walkthrough

EF 4.1.1

The EF 4.1.10715 runtime was released to NuGet in July of 2011. In addition to bug fixes this patch release introduced some components to make it easier for design time tooling to work with a Code First model. These components are used by Code First Migrations (included in EF 4.3) and the EF Power Tools.

You'll notice that the strange version number 4.1.10715 of the package. We used to use date based patch versions before we decided to adopt Semantic Versioning. Think of this version as EF 4.1 patch 1 (or EF 4.1.1).

Here is a list of content we put together for the 4.1.1 release:

- EF 4.1.1 Release Post

EF 4.1

The EF 4.1.10331 runtime was the first to be published on NuGet, in April of 2011. This release included the simplified DbContext API and the Code First workflow.

You will notice the strange version number, 4.1.10331, which should really have been 4.1. In addition there is a 4.1.10311 version which should have been 4.1.0-rc (the 'rc' stands for 'release candidate'). We used to use date based patch versions before we decided to adopt Semantic Versioning.

Here is a list of content we put together for the 4.1 release. Much of it still applies to later releases of Entity Framework:

- EF 4.1 Release Post
- Code First Walkthrough
- Model & Database First Walkthrough
- SQL Azure Federations and the Entity Framework

EF 4.0

This release was included in .NET Framework 4 and Visual Studio 2010, in April of 2010. Important new features in this release included POCO support, foreign key mapping, lazy loading, testability improvements, customizable code generation and the Model First workflow.

Although it was the second release of Entity Framework, it was named EF 4 to align with the .NET Framework version that it shipped with. After this release, we started making Entity Framework available on NuGet and adopted semantic versioning since we were no longer tied to the .NET Framework Version.

Note that some subsequent versions of .NET Framework have shipped with significant updates to the included EF bits. In fact, many of the new features of EF 5.0 were implemented as improvements on these bits. However, in order to rationalize the versioning story for EF, we continue to refer to the EF bits that are part of the .NET Framework as the EF 4.0 runtime, while all newer versions consist of the EntityFramework NuGet Package.

EF 3.5

The initial version of Entity Framework was included in .NET 3.5 Service Pack 1 and Visual Studio 2008 SP1, released in August of 2008. This release provided basic O/RM support using the Database First workflow.

Upgrading to Entity Framework 6

12/10/2018 • 3 minutes to read • Edit Online

In previous versions of EF the code was split between core libraries (primarily System.Data.Entity.dll) shipped as part of the .NET Framework and out-of-band (OOB) libraries (primarily EntityFramework.dll) shipped in a NuGet package. EF6 takes the code from the core libraries and incorporates it into the OOB libraries. This was necessary in order to allow EF to be made open source and for it to be able to evolve at a different pace from .NET Framework. The consequence of this is that applications will need to be rebuilt against the moved types.

This should be straightforward for applications that make use of DbContext as shipped in EF 4.1 and later. A little more work is required for applications that make use of ObjectContext but it still isn't hard to do.

Here is a checklist of the things you need to do to upgrade an existing application to EF6.

1. Install the EF6 NuGet package

You need to upgrade to the new Entity Framework 6 runtime.

1. Right-click on your project and select **Manage NuGet Packages...**
2. Under the **Online** tab select **EntityFramework** and click **Install**

> **NOTE**
> If a previous version of the EntityFramework NuGet package was installed this will upgrade it to EF6.

Alternatively, you can run the following command from Package Manager Console:

```
Install-Package EntityFramework
```

2. Ensure that assembly references to System.Data.Entity.dll are removed

Installing the EF6 NuGet package should automatically remove any references to System.Data.Entity from your project for you.

3. Swap any EF Designer (EDMX) models to use EF 6.x code generation

If you have any models created with the EF Designer, you will need to update the code generation templates to generate EF6 compatible code.

> **NOTE**
> There are currently only EF 6.x DbContext Generator templates available for Visual Studio 2012 and 2013.

1. Delete existing code-generation templates. These files will typically be named **<edmx_file_name>.tt** and **<edmx_file_name>.Context.tt** and be nested under your edmx file in Solution Explorer. You can select the templates in Solution Explorer and press the **Del** key to delete them.

> **NOTE**
>
> In Web Site projects the templates will not be nested under your edmx file, but listed alongside it in Solution Explorer.

> **NOTE**
>
> In VB.NET projects you will need to enable 'Show All Files' to be able to see the nested template files.

2. Add the appropriate EF 6.x code generation template. Open your model in the EF Designer, right-click on the design surface and select **Add Code Generation Item...**

   - If you are using the DbContext API (recommended) then **EF 6.x DbContext Generator** will be available under the **Data** tab.

     > **NOTE**
     >
     > If you are using Visual Studio 2012, you will need to install the EF 6 Tools to have this template. See Get Entity Framework for details.

   - If you are using the ObjectContext API then you will need to select the **Online** tab and search for **EF 6.x EntityObject Generator**.

3. If you applied any customizations to the code generation templates you will need to re-apply them to the updated templates.

4. Update namespaces for any core EF types being used

The namespaces for DbContext and Code First types have not changed. This means for many applications that use EF 4.1 or later you will not need to change anything.

Types like ObjectContext that were previously in System.Data.Entity.dll have been moved to new namespaces. This means you may need to update your *using* or *Import* directives to build against EF6.

The general rule for namespace changes is that any type in System.Data.* is moved to System.Data.Entity.Core.*. In other words, just insert **Entity.Core.** after System.Data. For example:

- 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

These types are in the *Core* namespaces because they are not used directly for most DbContext-based applications. Some types that were part of System.Data.Entity.dll are still used commonly and directly for DbContext-based applications and so have not been moved into the *Core* namespaces. These are:

- System.Data.EntityState => System.Data.**Entity**.EntityState
- System.Data.Objects.DataClasses.EdmFunctionAttribute => System.Data.**Entity.DbFunctionAttribute**

  > **NOTE**
  >
  > This class has been renamed; a class with the old name still exists and works, but it now marked as obsolete.

- System.Data.Objects.EntityFunctions => System.Data.**Entity.DbFunctions**

> **NOTE**
> This class has been renamed; a class with the old name still exists and works, but it now marked as obsolete.)

- Spatial classes (for example, DbGeography, DbGeometry) have moved from System.Data.Spatial => System.Data.**Entity**.Spatial

> **NOTE**
> Some types in the System.Data namespace are in System.Data.dll which is not an EF assembly. These types have not moved and so their namespaces remain unchanged.

Visual Studio Releases

10/25/2018 • 3 minutes to read • Edit Online

We recommend to always use the latest version of Visual Studio because it contains the latest tools for .NET, NuGet, and Entity Framework. In fact, the various samples and walkthroughs across the Entity Framework documentation assume that you are using a recent version of Visual Studio.

It is possible however to use older versions of Visual Studio with different versions of Entity Framework as long as you take into account some differences:

Visual Studio 2017 15.7 and newer

- This version of Visual Studio includes the latest release of Entity Framework tools and the EF 6.2 runtime, and does not require additional setup steps. See What's New for more details on these releases.
- Adding Entity Framework to new projects using the EF tools will automatically add the EF 6.2 NuGet package. You can manually install or upgrade to any EF NuGet package available online.
- By default, the SQL Server instance available with this version of Visual Studio is a LocalDB instance called MSSQLLocalDB. The server section of connection string you should use is "(localdb)\MSSQLLocalDB". Remember to use a verbatim string prefixed with `@` or double back-slashes "\\" when specifying a connection string in C# code.

Visual Studio 2015 to Visual Studio 2017 15.6

- These versions of Visual Studio include Entity Framework tools and runtime 6.1.3. See Past Releases for more details on these releases.
- Adding Entity Framework to new projects using the EF tools will automatically add the EF 6.1.3 NuGet package. You can manually install or upgrade to any EF NuGet package available online.
- By default, the SQL Server instance available with this version of Visual Studio is a LocalDB instance called MSSQLLocalDB. The server section of connection string you should use is "(localdb)\MSSQLLocalDB". Remember to use a verbatim string prefixed with `@` or double back-slashes "\\" when specifying a connection string in C# code.

Visual Studio 2013

- This version of Visual Studio includes and older version of Entity Framework tools and runtime. It is recommended that you upgrade to Entity Framework Tools 6.1.3, using the installer available in the Microsoft Download Center. See Past Releases for more details on these releases.
- Adding Entity Framework to new projects using the upgraded EF tools will automatically add the EF 6.1.3 NuGet package. You can manually install or upgrade to any EF NuGet package available online.
- By default, the SQL Server instance available with this version of Visual Studio is a LocalDB instance called MSSQLLocalDB. The server section of connection string you should use is "(localdb)\MSSQLLocalDB". Remember to use a verbatim string prefixed with `@` or double back-slashes "\\" when specifying a connection string in C# code.

Visual Studio 2012

- This version of Visual Studio includes and older version of Entity Framework tools and runtime. It is recommended that you upgrade to Entity Framework Tools 6.1.3, using the installer available in the Microsoft Download Center. See Past Releases for more details on these releases.

- Adding Entity Framework to new projects using the upgraded EF tools will automatically add the EF 6.1.3 NuGet package. You can manually install or upgrade to any EF NuGet package available online.
- By default, the SQL Server instance available with this version of Visual Studio is a LocalDB instance called v11.0. The server section of connection string you should use is "(localdb)\v11.0". Remember to use a verbatim string prefixed with `@` or double back-slashes "\\" when specifying a connection string in C# code.

Visual Studio 2010

- The version of Entity Framework Tools available with this version of Visual Studio is not compatible with the Entity Framework 6 runtime and cannot be upgraded.
- By default, the Entity Framework tools will add Entity Framework 4.0 to your projects. In order to create applications using any newer versions of EF, you will first need to install the NuGet Package Manager extension.
- By default, all code generation in the version of EF tools is based on EntityObject and Entity Framework 4. We recommend that you switch the code generation to be based on DbContext and Entity Framework 5, by installing the DbContext code generation templates for C# or Visual Basic.
- Once you have installed the NuGet Package Manager extensions, you can manually install or upgrade to any EF NuGet package available online and use EF6 with Code First, which does not require a designer.
- By default, the SQL Server instance available with this version of Visual Studio is SQL Server Express named SQLEXPRESS. The server section of connection string you should use is ".\SQLEXPRESS". Remember to use a verbatim string prefixed with `@` or double back-slashes "\\" when specifying a connection string in C# code.

Get started with Entity Framework 6

9/13/2018 • 2 minutes to read • Edit Online

This guide contains a collection of links to selected documentation articles, walkthroughs and videos that can help you get started quickly.

Fundamentals

- Get Entity Framework

  Here you will learn how to add Entity Framework to your applications and, if you want to use the EF Designer, make sure you get it installed in Visual Studio.

- Creating a Model: Code First, the EF Designer, and the EF Workflows

  Do you prefer to specify your EF model writing code or drawing boxes and lines? Are you going to use EF to map your objects to an existing database or would you like EF to create a database tailored for your objects? Here your learn about two different approaches to use EF6: EF Designer and Code First. Make sure you follow the discussion and watch the video about the difference.

- Working with DbContext

  DbContext is the first and most important EF type that you need to learn how to use. It serves as the launchpad for database queries and keeps track of changes you make to objects so that they can be persisted back to the database.

- Ask a Question

  Find out how to get help from the experts and contribute your own answers to the community.

- Contribute

  Entity Framework 6 uses an open development model. Find out how you can help make EF even better by visiting our GitHub repository.

Code First resources

- Code First to an Existing Database Workflow
- Code First to a New Database Workflow
- Mapping Enums Using Code First
- Mapping Spatial Types Using Code First
- Writing Custom Code First Conventions
- Using Code First Fluent Configuration with Visual Basic
- Code First Migrations
- Code First Migrations in Team Environments
- Automatic Code First Migrations (This is no longer recommended)

EF Designer resources

- Database First Workflow
- Model First Workflow
- Mapping Enums

- Mapping Spatial Types
- Table-Per Hierarchy Inheritance Mapping
- Table-Per Type Inheritance Mapping
- Stored Procedure Mapping for Updates
- Stored Procedure Mapping for Query
- Entity Splitting
- Table Splitting
- Defining Query (Advanced)
- Table-Valued Functions (Advanced)

Other resources

- Async Query and Save
- Databinding with WinForms
- Databinding with WPF
- Disconnected scenarios with Self-Tracking Entities (This is no longer recommended)

Entity Framework 6 fundamentals

9/13/2018 • 2 minutes to read • Edit Online

Topics under this section describe various basic aspects of working with EF6.

Get Entity Framework

9/13/2018 • 2 minutes to read • Edit Online

Entity Framework is made up of the EF Tools for Visual Studio and the EF Runtime.

EF Tools for Visual Studio

The Entity Framework Tools for Visual Studio include the EF Designer and the EF Model Wizard and are required for the database first and model first workflows. EF Tools are included in all recent versions of Visual Studio. If you perform a custom install of Visual Studio you will need to ensure that the item "Entity Framework 6 Tools" is selected by either choosing a workload that includes it or by selecting it as an individual component.

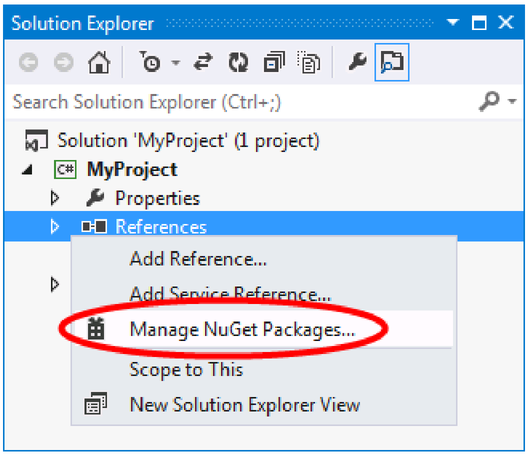
For some past versions of Visual Studio, updated EF Tools are available as a download. See Visual Studio Versions for guidance on how to get the latest version of EF Tools available for your version of Visual Studio.

EF Runtime

The latest version of Entity Framework is available as the EntityFramework NuGet package. If you are not familiar with the NuGet Package Manager, we encourage you to read the NuGet Overview.

Installing the EF NuGet Package

You can install the EntityFramework package by right-clicking on the **References** folder of your project and selecting **Manage NuGet Packages...**



Installing from Package Manager Console

Alternatively, you can install EntityFramework by running the following command in the Package Manager Console.

```
Install-Package EntityFramework
```

Installing a specific version of EF

From EF 4.1 onwards, new versions of the EF runtime have been released as the EntityFramework NuGet Package. Any of those versions can be added to a .NET Framework-based project by running the following command in Visual Studio's Package Manager Console:

```
Install-Package EntityFramework -Version <number>
```

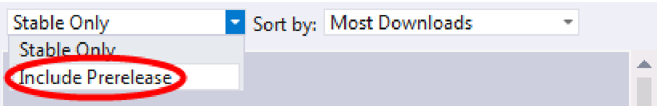
Note that `<number>` represents the specific version of EF to install. For example, 6.2.0 is the version of number for EF 6.2.

EF runtimes before 4.1 were part of .NET Framework and cannot be installed separately.

Installing the Latest Preview

The above methods will give you the latest fully supported release of Entity Framework. There are often prerelease versions of Entity Framework available that we would love you to try out and give us feedback on.

To install the latest preview of EntityFramework you can select **Include Prerelease** in the Manage NuGet Packages window. If no prerelease versions are available you will automatically get the latest fully supported version of Entity Framework.



Alternatively, you can run the following command in the Package Manager Console.

```
Install-Package EntityFramework -Pre
```

Working with DbContext

9/13/2018 • 3 minutes to read • Edit Online

In order to use Entity Framework to query, insert, update, and delete data using .NET objects, you first need to Create a Model which maps the entities and relationships that are defined in your model to tables in a database.

Once you have a model, the primary class your application interacts with is `System.Data.Entity.DbContext` (often referred to as the context class). You can use a DbContext associated to a model to:

- Write and execute queries
- Materialize query results as entity objects
- Track changes that are made to those objects
- Persist object changes back on the database
- Bind objects in memory to UI controls

This page gives some guidance on how to manage the context class.

Defining a DbContext derived class

The recommended way to work with context is to define a class that derives from DbContext and exposes DbSet properties that represent collections of the specified entities in the context. If you are working with the EF Designer, the context will be generated for you. If you are working with Code First, you will typically write the context yourself.

```
public class ProductContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

Once you have a context, you would query for, add (using `Add` or `Attach` methods) or remove (using `Remove`) entities in the context through these properties. Accessing a `DbSet` property on a context object represent a starting query that returns all entities of the specified type. Note that just accessing a property will not execute the query. A query is executed when:

- It is enumerated by a `foreach` (C#) or `For Each` (Visual Basic) statement.
- It is enumerated by a collection operation such as `ToArray`, `ToDictionary`, or `ToList`.
- LINQ operators such as `First` or `Any` are specified in the outermost part of the query.
- One of the following methods are called: the `Load` extension method, `DbEntityEntry.Reload`, `Database.ExecuteSqlCommand`, and `DbSet<T>.Find`, if an entity with the specified key is not found already loaded in the context.

Lifetime

The lifetime of the context begins when the instance is created and ends when the instance is either disposed or garbage-collected. Use **using** if you want all the resources that the context controls to be disposed at the end of the block. When you use **using**, the compiler automatically creates a try/finally block and calls dispose in the **finally** block.

```
public void UseProducts()
{
    using (var context = new ProductContext())
    {
        // Perform data access using the context
    }
}
```

Here are some general guidelines when deciding on the lifetime of the context:

- When working with Web applications, use a context instance per request.
- When working with Windows Presentation Foundation (WPF) or Windows Forms, use a context instance per form. This lets you use change-tracking functionality that context provides.
- If the context instance is created by a dependency injection container, it is usually the responsibility of the container to dispose the context.
- If the context is created in application code, remember to dispose of the context when it is no longer required.
- When working with long-running context consider the following:
  - As you load more objects and their references into memory, the memory consumption of the context may increase rapidly. This may cause performance issues.
  - The context is not thread-safe, therefore it should not be shared across multiple threads doing work on it concurrently.
  - If an exception causes the context to be in an unrecoverable state, the whole application may terminate.
  - The chances of running into concurrency-related issues increase as the gap between the time when the data is queried and updated grows.

Connections

By default, the context manages connections to the database. The context opens and closes connections as needed. For example, the context opens a connection to execute a query, and then closes the connection when all the result sets have been processed.

There are cases when you want to have more control over when the connection opens and closes. For example, when working with SQL Server Compact, it is often recommended to maintain a separate open connection to the database for the lifetime of the application to improve performance. You can manage this process manually by using the `Connection` property.

Relationships, navigation properties and foreign keys

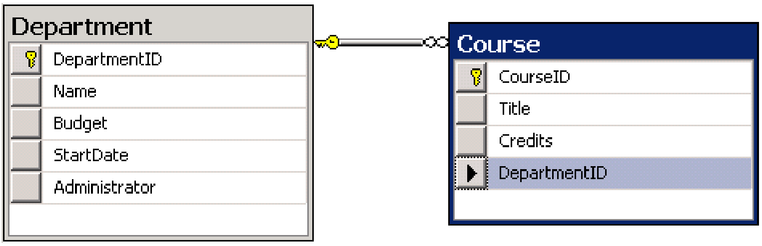
10/14/2018 • 9 minutes to read • Edit Online

This topic gives an overview of how Entity Framework manages relationships between entities. It also gives some guidance on how to map and manipulate relationships.

Relationships in EF

In relational databases, relationships (also called associations) between tables are defined through foreign keys. A foreign key (FK) is a column or combination of columns that is used to establish and enforce a link between the data in two tables. There are generally three types of relationships: one-to-one, one-to-many, and many-to-many. In a one-to-many relationship, the foreign key is defined on the table that represents the many end of the relationship. The many-to-many relationship involves defining a third table (called a junction or join table), whose primary key is composed of the foreign keys from both related tables. In a one-to-one relationship, the primary key acts additionally as a foreign key and there is no separate foreign key column for either table.

The following image shows two tables that participate in one-to-many relationship. The **Course** table is the dependent table because it contains the **DepartmentID** column that links it to the **Department** table.



In Entity Framework, an entity can be related to other entities through an association or relationship. Each relationship contains two ends that describe the entity type and the multiplicity of the type (one, zero-or-one, or many) for the two entities in that relationship. The relationship may be governed by a referential constraint, which describes which end in the relationship is a principal role and which is a dependent role.

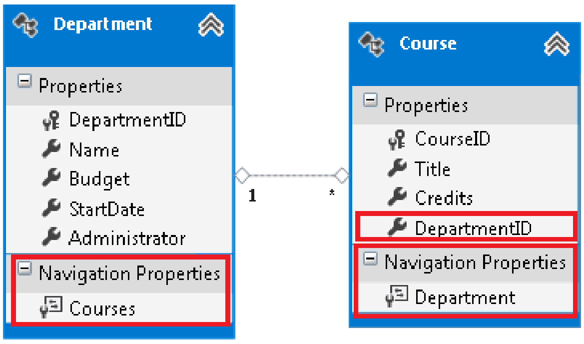
Navigation properties provide a way to navigate an association between two entity types. Every object can have a navigation property for every relationship in which it participates. Navigation properties allow you to navigate and manage relationships in both directions, returning either a reference object (if the multiplicity is either one or zero-or-one) or a collection (if the multiplicity is many). You may also choose to have one-way navigation, in which case you define the navigation property on only one of the types that participates in the relationship and not on both.

It is recommended to include properties in the model that map to foreign keys in the database. With foreign key properties included, you can create or change a relationship by modifying the foreign key value on a dependent object. This kind of association is called a foreign key association. Using foreign keys is even more essential when working with disconnected entities. Note, that when working with 1-to-1 or 1-to-0..1 relationships, there is no separate foreign key column, the primary key property acts as the foreign key and is always included in the model.

When foreign key columns are not included in the model, the association information is managed as an independent object. Relationships are tracked through object references instead of foreign key properties. This type of association is called an *independent association*. The most common way to modify an *independent association* is to modify the navigation properties that are generated for each entity that participates in the association.

You can choose to use one or both types of associations in your model. However, if you have a pure many-to-many relationship that is connected by a join table that contains only foreign keys, the EF will use an independent association to manage such many-to-many relationship.

The following image shows a conceptual model that was created with the Entity Framework Designer. The model contains two entities that participate in one-to-many relationship. Both entities have navigation properties. **Course** is the depend entity and has the **DepartmentID** foreign key property defined.



The following code snippet shows the same model that was created with 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; }
}
```

Configuring or mapping relationships

The rest of this page covers how to access and manipulate data using relationships. For information on setting up relationships in your model, see the following pages.

- To configure relationships in Code First, see Data Annotations and Fluent API – Relationships.
- To configure relationships using the Entity Framework Designer, see Relationships with the EF Designer.

Creating and modifying relationships

In a *foreign key association*, when you change the relationship, the state of a dependent object with an `EntityState.Unchanged` state changes to `EntityState.Modified`. In an independent relationship, changing the relationship does not update the state of the dependent object.

The following examples show how to use the foreign key properties and navigation properties to associate the related objects. With foreign key associations, you can use either method to change, create, or modify relationships. With independent associations, you cannot use the foreign key property.

- By assigning a new value to a foreign key property, as in the following example.

```
course.DepartmentID = newCourse.DepartmentID;
```

- The following code removes a relationship by setting the foreign key to **null**. Note, that the foreign key property must be nullable.

```
course.DepartmentID = null;
```

> **NOTE**
>
> If the reference is in the added state (in this example, the course object), the reference navigation property will not be synchronized with the key values of a new object until SaveChanges is called. Synchronization does not occur because the object context does not contain permanent keys for added objects until they are saved. If you must have new objects fully synchronized as soon as you set the relationship, use one of the following methods.*

- By assigning a new object to a navigation property. The following code creates a relationship between a course and a `department`. If the objects are attached to the context, the `course` is also added to the `department.Courses` collection, and the corresponding foreign key property on the `course` object is set to the key property value of the department.

```
course.Department = department;
```

- To delete the relationship, set the navigation property to `null`. If you are working with Entity Framework that is based on .NET 4.0, then the related end needs to be loaded before you set it to null. For example:

```
context.Entry(course).Reference(c => c.Department).Load();
course.Department = null;
```

  Starting with Entity Framework 5.0, that is based on .NET 4.5, you can set the relationship to null without loading the related end. You can also set the current value to null using the following method.

```
context.Entry(course).Reference(c => c.Department).CurrentValue = null;
```

- By deleting or adding an object in an entity collection. For example, you can add an object of type `Course` to the `department.Courses` collection. This operation creates a relationship between a particular **course** and a particular `department`. If the objects are attached to the context, the department reference and the foreign key property on the **course** object will be set to the appropriate `department`.

```
department.Courses.Add(newCourse);
```

- By using the `ChangeRelationshipState` method to change the state of the specified relationship between two entity objects. This method is most commonly used when working with N-Tier applications and an *independent association* (it cannot be used with a foreign key association). Also, to use this method you must drop down to `ObjectContext`, as shown in the example below.
  In the following example, there is a many-to-many relationship between Instructors and Courses. Calling the `ChangeRelationshipState` method and passing the `EntityState.Added` parameter, lets the `SchoolContext` know that a relationship has been added between the two objects:

```
((IObjectContextAdapter)context).ObjectContext.
  ObjectStateManager.
  ChangeRelationshipState(course, instructor, c => c.Instructor, EntityState.Added);
```

Note that if you are updating (not just adding) a relationship, you must delete the old relationship after adding the new one:

```
((IObjectContextAdapter)context).ObjectContext.
  ObjectStateManager.
  ChangeRelationshipState(course, oldInstructor, c => c.Instructor, EntityState.Deleted);
```

Synchronizing the changes between the foreign keys and navigation properties

When you change the relationship of the objects attached to the context by using one of the methods described above, Entity Framework needs to keep foreign keys, references, and collections in sync. Entity Framework automatically manages this synchronization (also known as relationship fix-up) for the POCO entities with proxies. For more information, see Working with Proxies.

If you are using POCO entities without proxies, you must make sure that the **DetectChanges** method is called to synchronize the related objects in the context. Note, that the following APIs automatically trigger a **DetectChanges** call.

- `DbSet.Add`
- `DbSet.AddRange`
- `DbSet.Remove`
- `DbSet.RemoveRange`
- `DbSet.Find`
- `DbSet.Local`
- `DbContext.SaveChanges`
- `DbSet.Attach`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`
- Executing a LINQ query against a `DbSet`

Loading related objects

In Entity Framework you use most commonly use the navigation properties to load entities that are related to the returned entity by the defined association. For more information, see Loading Related Objects.

> **NOTE**
>
> In a foreign key association, when you load a related end of a dependent object, the related object will be loaded based on the foreign key value of the dependent that is currently in memory:

```
// 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();
```

In an independent association, the related end of a dependent object is queried based on the foreign key value that is currently in the database. However, if the relationship was modified, and the reference property on the dependent object points to a different principal object that is loaded in the object context, Entity Framework will try to create a relationship as it is defined on the client.

Managing concurrency

In both foreign key and independent associations, concurrency checks are based on the entity keys and other entity properties that are defined in the model. When using the EF Designer to create a model, set the `ConcurrencyMode` attribute to **fixed** to specify that the property should be checked for concurrency. When using Code First to define a model, use the `ConcurrencyCheck` annotation on properties that you want to be checked for concurrency. When working with Code First you can also use the `TimeStamp` annotation to specify that the property should be checked for concurrency. You can have only one timestamp property in a given class. Code First maps this property to a non-nullable field in the database.

We recommend that you always use the foreign key association when working with entities that participate in concurrency checking and resolution.

For more information, see Handling Concurrency Conflicts.

Working with overlapping Keys

Overlapping keys are composite keys where some properties in the key are also part of another key in the entity. You cannot have an overlapping key in an independent association. To change a foreign key association that includes overlapping keys, we recommend that you modify the foreign key values instead of using the object references.

Async query and save

1/6/2019 • 6 minutes to read • Edit Online

> **NOTE**
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

EF6 introduced support for asynchronous query and save using the async and await keywords that were introduced in .NET 4.5. While not all applications may benefit from asynchrony, it can be used to improve client responsiveness and server scalability when handling long-running, network or I/O-bound tasks.

When to really use async

The purpose of this walkthrough is to introduce the async concepts in a way that makes it easy to observe the difference between asynchronous and synchronous program execution. This walkthrough is not intended to illustrate any of the key scenarios where async programming provides benefits.

Async programming is primarily focused on freeing up the current managed thread (thread running .NET code) to do other work while it waits for an operation that does not require any compute time from a managed thread. For example, whilst the database engine is processing a query there is nothing to be done by .NET code.

In client applications (WinForms, WPF, etc.) the current thread can be used to keep the UI responsive while the async operation is performed. In server applications (ASP.NET etc.) the thread can be used to process other incoming requests - this can reduce memory usage and/or increase throughput of the server.

In most applications using async will have no noticeable benefits and even could be detrimental. Use tests, profiling and common sense to measure the impact of async in your particular scenario before committing to it.

Here are some more resources to learn about async:

- Brandon Bray's overview of async/await in .NET 4.5
- Asynchronous Programming pages in the MSDN Library
- How to Build ASP.NET Web Applications Using Async (includes a demo of increased server throughput)

Create the model

We'll be using the Code First workflow to create our model and generate the database, however the asynchronous functionality will work with all EF models including those created with the EF Designer.

- Create a Console Application and call it **AsyncDemo**
- Add the EntityFramework NuGet package
  - In Solution Explorer, right-click on the **AsyncDemo** project
  - Select **Manage NuGet Packages...**
  - In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package
  - Click **Install**
- Add a **Model.cs** class with the following implementation

```
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; }
    }
}
```

Create a synchronous program

Now that we have an EF model, let's write some code that uses it to perform some data access.

- Replace the contents of **Program.cs** with the following code

```
using System;
using System.Linq;

namespace AsyncDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            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.");

            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)
                });
                Console.WriteLine("Calling SaveChanges.");
                db.SaveChanges();
                Console.WriteLine("SaveChanges completed.");

                // Query for all blogs ordered by name
                Console.WriteLine("Executing query.");
                var blogs = (from b in db.Blogs
                            orderby b.Name
                            select b).ToList();

                // Write all blogs out to Console
                Console.WriteLine("Query completed with following results:");
                foreach (var blog in blogs)
                {
                    Console.WriteLine(" " + blog.Name);
                }
            }
        }
    }
}
```

This code calls the **PerformDatabaseOperations** method which saves a new **Blog** to the database and then retrieves all **Blogs** from the database and prints them to the **Console**. After this, the program writes a quote of the day to the **Console**.

Since the code is synchronous, we can observe the following execution flow when we run the program:

1. **SaveChanges** begins to push the new **Blog** to the database
2. **SaveChanges** completes
3. Query for all **Blogs** is sent to the database
4. Query returns and results are written to **Console**
5. Quote of the day is written to **Console**

Making it asynchronous

Now that we have our program up and running, we can begin making use of the new async and await keywords. We've made the following changes to Program.cs

1. Line 2: The using statement for the **System.Data.Entity** namespace gives us access to the EF async extension methods.
2. Line 4: The using statement for the **System.Threading.Tasks** namespace allows us to use the **Task** type.
3. Line 12 & 18: We are capturing as task that monitors the progress of **PerformSomeDatabaseOperations** (line 12) and then blocking program execution for this task to complete once all the work for the program is done (line 18).
4. Line 25: We've update **PerformSomeDatabaseOperations** to be marked as **async** and return a **Task**.
5. Line 35: We're now calling the Async version of SaveChanges and awaiting it's completion.
6. Line 42: We're now calling hte Async version of ToList and awaiting on the result.

For a comprehensive list of available extension methods in the System.Data.Entity namespace, refer to the QueryableExtensions class. *You'll also need to add "using System.Data.Entity" to your using statements.*

```
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);
                }
            }
        }
    }
}
```

Now that the code is asyncronous, we can observe a different execution flow when we run the program:

1. **SaveChanges** begins to push the new **Blog** to the database *Once the command is sent to the database no more compute time is needed on the current managed thread. The **PerformDatabaseOperations** method returns (even though it hasn't finished executing) and program flow in the Main method continues.*
2. **Quote of the day is written to Console** *Since there is no more work to do in the Main method, the managed thread is blocked on the Wait call until the database operation completes. Once it completes, the remainder of our **PerformDatabaseOperations*** will be executed.
3. **SaveChanges** completes
4. Query for all **Blogs** is sent to the database *Again, the managed thread is free to do other work while the query*

is processed in the database. Since all other execution has completed, the thread will just halt on the Wait call though.

5. Query returns and results are written to **Console**

The takeaway

We now saw how easy it is to make use of EF's asynchronous methods. Although the advantages of async may not be very apparent with a simple console app, these same strategies can be applied in situations where long-running or network-bound activities might otherwise block the application, or cause a large number of threads to increase the memory footprint.

Code-based configuration

9/13/2018 • 4 minutes to read • Edit Online

> **NOTE**
>
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Configuration for an Entity Framework application can be specified in a config file (app.config/web.config) or through code. The latter is known as code-based configuration.

Configuration in a config file is described in a separate article. The config file takes precedence over code-based configuration. In other words, if a configuration option is set in both code and in the config file, then the setting in the config file is used.

Using DbConfiguration

Code-based configuration in EF6 and above is achieved by creating a subclass of System.Data.Entity.Config.DbConfiguration. The following guidelines should be followed when subclassing DbConfiguration:

- Create only one DbConfiguration class for your application. This class specifies app-domain wide settings.
- Place your DbConfiguration class in the same assembly as your DbContext class. (See the *Moving DbConfiguration* section if you want to change this.)
- Give your DbConfiguration class a public parameterless constructor.
- Set configuration options by calling protected DbConfiguration methods from within this constructor.

Following these guidelines allows EF to discover and use your configuration automatically by both tooling that needs to access your model and when your application is run.

Example

A class derived from DbConfiguration might look like this:

```
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"));
        }
    }
}
```

This class sets up EF to use the SQL Azure execution strategy - to automatically retry failed database operations - and to use Local DB for databases that are created by convention from Code First.

Moving DbConfiguration

There are cases where it is not possible to place your DbConfiguration class in the same assembly as your DbContext class. For example, you may have two DbContext classes each in different assemblies. There are two options for handling this.

The first option is to use the config file to specify the DbConfiguration instance to use. To do this, set the codeConfigurationType attribute of the entityFramework section. For example:

```
<entityFramework codeConfigurationType="MyNamespace.MyDbConfiguration, MyAssembly">
    ...Your EF config...
</entityFramework>
```

The value of codeConfigurationType must be the assembly and namespace qualified name of your DbConfiguration class.

The second option is to place DbConfigurationTypeAttribute on your context class. For example:

```
[DbConfigurationType(typeof(MyDbConfiguration))]
public class MyContextContext : DbContext
{
}
```

The value passed to the attribute can either be your DbConfiguration type - as shown above - or the assembly and namespace qualified type name string. For example:

```
[DbConfigurationType("MyNamespace.MyDbConfiguration, MyAssembly")]
public class MyContextContext : DbContext
{
}
```

Setting DbConfiguration explicitly

There are some situations where configuration may be needed before any DbContext type has been used. Examples of this include:

- Using DbModelBuilder to build a model without a context
- Using some other framework/utility code that utilizes a DbContext where that context is used before your application context is used

In such situations EF is unable to discover the configuration automatically and you must instead do one of the following:

- Set the DbConfiguration type in the config file, as described in the *Moving DbConfiguration* section above
- Call the static DbConfiguration.SetConfiguration method during application startup

Overriding DbConfiguration

There are some situations where you need to override the configuration set in the DbConfiguration. This is not typically done by application developers but rather by third party providers and plug-ins that cannot use a derived DbConfiguration class.

For this, EntityFramework allows an event handler to be registered that can modify existing configuration just before it is locked down. It also provides a sugar method specifically for replacing any service returned by the EF service locator. This is how it is intended to be used:

- At app startup (before EF is used) the plug-in or provider should register the event handler method for this event. (Note that this must happen before the application uses EF.)
- The event handler makes a call to ReplaceService for every service that needs to be replaced.

For example, to repalce IDbConnectionFactory and DbProviderService you would register a handler something like this:

```
DbConfiguration.Loaded += (_, a) =>
   {
       a.ReplaceService<DbProviderServices>((s, k) => new MyProviderServices(s));
       a.ReplaceService<IDbConnectionFactory>((s, k) => new MyConnectionFactory(s));
   };
```

In the code above MyProviderServices and MyConnectionFactory represent your implementations of the service.

You can also add additional dependency handlers to get the same effect.

Note that you could also wrap DbProviderFactory in this way, but doing so will only affect EF and not uses of the DbProviderFactory outside of EF. For this reason you'll probably want to continue to wrap DbProviderFactory as you have before.

You should also keep in mind the services that you run externally to your application - for example, when running migrations from the Package Manager Console. When you run migrate from the console it will attempt to find your DbConfiguration. However, whether or not it will get the wrapped service depends on where the event handler it registered. If it is registered as part of the construction of your DbConfiguration then the code should execute and the service should get wrapped. Usually this won't be the case and this means that tooling won't get the wrapped service.

Configuration File Settings

9/27/2018 • 6 minutes to read • Edit Online

Entity Framework allows a number of settings to be specified from the configuration file. In general EF follows a 'convention over configuration' principle: all the settings discussed in this post have a default behavior, you only need to worry about changing the setting when the default no longer satisfies your requirements.

A Code-Based Alternative

All of these settings can also be applied using code. Starting in EF6 we introduced code-based configuration, which provides a central way of applying configuration from code. Prior to EF6, configuration can still be applied from code but you need to use various APIs to configure different areas. The configuration file option allows these settings to be easily changed during deployment without updating your code.

The Entity Framework Configuration Section

Starting with EF4.1 you could set the database initializer for a context using the **appSettings** section of the configuration file. In EF 4.3 we introduced the custom **entityFramework** section to handle the new settings. Entity Framework will still recognize database initializers set using the old format, but we recommend moving to the new format where possible.

The **entityFramework** section was automatically added to the configuration file of your project when you installed the EntityFramework NuGet package.

```
<?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>
```

Connection Strings

This page provides more details on how Entity Framework determines the database to be used, including connection strings in the configuration file.

Connection strings go in the standard **connectionStrings** element and do not require the **entityFramework** section.

Code First based models use normal ADO.NET connection strings. For example:

```
<connectionStrings>
  <add name="BlogContext"
        providerName="System.Data.SqlClient"
        connectionString="Server=.\SQLEXPRESS;Database=Blogging;Integrated Security=True;"/>
</connectionStrings>
```

EF Designer based models use special EF connection strings. For example:

```xml
<connectionStrings>
  <add name="BlogContext"
    connectionString=
      "metadata=
        res://*/BloggingModel.csdl|
        res://*/BloggingModel.ssdl|
        res://*/BloggingModel.msl;
      provider=System.Data.SqlClient
      provider connection string=
        &quot;data source=(localdb)\mssqllocaldb;
        initial catalog=Blogging;
        integrated security=True;
        multipleactiveresultsets=True;&quot;"
     providerName="System.Data.EntityClient" />
</connectionStrings>
```

Code-Based Configuration Type (EF6 Onwards)

Starting with EF6, you can specify the DbConfiguration for EF to use for code-based configuration in your application. In most cases you don't need to specify this setting as EF will automatically discover your DbConfiguration. For details of when you may need to specify DbConfiguration in your config file see the **Moving DbConfiguration** section of Code-Based Configuration.

To set a DbConfiguration type, you specify the assembly qualified type name in the **codeConfigurationType** element.

> **NOTE**
>
> An assembly qualified name is the namespace qualified name, followed by a comma, then the assembly that the type resides in. You can optionally also specify the assembly version, culture and public key token.

```xml
<entityFramework codeConfigurationType="MyNamespace.MyConfiguration, MyAssembly">
</entityFramework>
```

EF Database Providers (EF6 Onwards)

Prior to EF6, Entity Framework-specific parts of a database provider had to be included as part of the core ADO.NET provider. Starting with EF6, the EF specific parts are now managed and registered separately.

Normally you won't need to register providers yourself. This will typically be done by the provider when you install it.

Providers are registered by including a **provider** element under the **providers** child section of the **entityFramework** section. There are two required attributes for a provider entry:

- **invariantName** identifies the core ADO.NET provider that this EF provider targets
- **type** is the assembly qualified type name of the EF provider implementation

> **NOTE**
>
> An assembly qualified name is the namespace qualified name, followed by a comma, then the assembly that the type resides in. You can optionally also specify the assembly version, culture and public key token.

As an example here is the entry created to register the default SQL Server provider when you install Entity Framework.

```
<providers>
  <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
</providers>
```

Interceptors (EF6.1 Onwards)

Starting with EF6.1 you can register interceptors in the configuration file. Interceptors allow you to run additional logic when EF performs certain operations, such as executing database queries, opening connections, etc.

Interceptors are registered by including an **interceptor** element under the **interceptors** child section of the **entityFramework** section. For example, the following configuration registers the built-in **DatabaseLogger** interceptor that will log all database operations to the Console.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework"/>
</interceptors>
```

Logging Database Operations to a File (EF6.1 Onwards)

Registering interceptors via the config file is especially useful when you want to add logging to an existing application to help debug an issue. **DatabaseLogger** supports logging to a file by supplying the file name as a constructor parameter.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Temp\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

By default this will cause the log file to be overwritten with a new file each time the app starts. To instead append to the log file if it already exists use something like:

```
<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>
```

For additional information on **DatabaseLogger** and registering interceptors, see the blog post EF 6.1: Turning on logging without recompiling.

Code First Default Connection Factory

The configuration section allows you to specify a default connection factory that Code First should use to locate a database to use for a context. The default connection factory is only used when no connection string has been added to the configuration file for a context.

When you installed the EF NuGet package a default connection factory was registered that points to either SQL Express or LocalDB, depending on which one you have installed.

To set a connection factory, you specify the assembly qualified type name in the **defaultConnectionFactory** element.

> **NOTE**
>
> An assembly qualified name is the namespace qualified name, followed by a comma, then the assembly that the type resides in. You can optionally also specify the assembly version, culture and public key token.

Here is an example of setting your own default connection factory:

```
<entityFramework>
  <defaultConnectionFactory type="MyNamespace.MyCustomFactory, MyAssembly"/>
</entityFramework>
```

The above example requires the custom factory to have a parameterless constructor. If needed, you can specify constructor parameters using the **parameters** element.

For example, the SqlCeConnectionFactory, that is included in Entity Framework, requires you to supply a provider invariant name to the constructor. The provider invariant name identifies the version of SQL Compact you want to use. The following configuration will cause contexts to use SQL Compact version 4.0 by default.

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="System.Data.SqlServerCe.4.0" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

If you don't set a default connection factory, Code First uses the SqlConnectionFactory, pointing to `.\SQLEXPRESS`. SqlConnectionFactory also has a constructor that allows you to override parts of the connection string. If you want to use a SQL Server instance other than `.\SQLEXPRESS` you can use this constructor to set the server.

The following configuration will cause Code First to use **MyDatabaseServer** for contexts that don't have an explicit connection string set.

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="Data Source=MyDatabaseServer; Integrated Security=True; MultipleActiveResultSets=True"
/>
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

By default, it's assumed that constructor arguments are of type string. You can use the type attribute to change this.

```
<parameter value="2" type="System.Int32" />
```

Database Initializers

Database initializers are configured on a per-context basis. They can be set in the configuration file using the **context** element. This element uses the assembly qualified name to identify the context being configured.

By default, Code First contexts are configured to use the CreateDatabaseIfNotExists initializer. There is a **disableDatabaseInitialization** attribute on the **context** element that can be used to disable database initialization.

For example, the following configuration disables database initialization for the Blogging.BlogContext context defined in MyAssembly.dll.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly" disableDatabaseInitialization="true" />
</contexts>
```

You can use the **databaseInitializer** element to set a custom initializer.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly" />
  </context>
</contexts>
```

Constructor parameters use the same syntax as default connection factories.

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly">
      <parameters>
        <parameter value="MyConstructorParameter" />
      </parameters>
    </databaseInitializer>
  </context>
</contexts>
```

You can configure one of the generic database initializers that are included in Entity Framework. The **type** attribute uses the .NET Framework format for generic types.

For example, if you are using Code First Migrations, you can configure the database to be migrated automatically using the `MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>` initializer.

```
<contexts>
  <context type="Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[[Blogging.BlogContext,
MyAssembly], [Blogging.Migrations.Configuration, MyAssembly]], EntityFramework" />
  </context>
</contexts>
```

Connection strings and models

9/13/2018 • 5 minutes to read • Edit Online

This topic covers how Entity Framework discovers which database connection to use, and how you can change it. Models created with Code First and the EF Designer are both covered in this topic.

Typically an Entity Framework application uses a class derived from DbContext. This derived class will call one of the constructors on the base DbContext class to control:

- How the context will connect to a database — that is, how a connection string is found/used
- Whether the context will use calculate a model using Code First or load a model created with the EF Designer
- Additional advanced options

The following fragments show some of the ways the DbContext constructors can be used.

Use Code First with connection by convention

If you have not done any other configuration in your application, then calling the parameterless constructor on DbContext will cause DbContext to run in Code First mode with a database connection created by convention. For example:

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
        // C# will call base class parameterless constructor by default
        {
        }
    }
}
```

In this example DbContext uses the namespace qualified name of your derived context class—Demo.EF.BloggingContext—as the database name and creates a connection string for this database using either SQL Express or LocalDB. If both are installed, SQL Express will be used.

Visual Studio 2010 includes SQL Express by default and Visual Studio 2012 and later includes LocalDB. During installation, the EntityFramework NuGet package checks which database server is available. The NuGet package will then update the configuration file by setting the default database server that Code First uses when creating a connection by convention. If SQL Express is running, it will be used. If SQL Express is not available then LocalDB will be registered as the default instead. No changes are made to the configuration file if it already contains a setting for the default connection factory.

Use Code First with connection by convention and specified database name

If you have not done any other configuration in your application, then calling the string constructor on DbContext with the database name you want to use will cause DbContext to run in Code First mode with a database connection created by convention to the database of that name. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingDatabase")
    {
    }
}
```

In this example DbContext uses “BloggingDatabase” as the database name and creates a connection string for this database using either SQL Express (installed with Visual Studio 2010) or LocalDB (installed with Visual Studio 2012). If both are installed, SQL Express will be used.

Use Code First with connection string in app.config/web.config file

You may choose to put a connection string in your app.config or web.config file. For example:

```
<configuration>
  <connectionStrings>
    <add name="BloggingCompactDatabase"
         providerName="System.Data.SqlServerCe.4.0"
         connectionString="Data Source=Blogging.sdf"/>
  </connectionStrings>
</configuration>
```

This is an easy way to tell DbContext to use a database server other than SQL Express or LocalDB — the example above specifies a SQL Server Compact Edition database.

If the name of the connection string matches the name of your context (either with or without namespace qualification) then it will be found by DbContext when the parameterless constructor is used. If the connection string name is different from the name of your context then you can tell DbContext to use this connection in Code First mode by passing the connection string name to the DbContext constructor. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingCompactDatabase")
    {
    }
}
```

Alternatively, you can use the form “name=<connection string name>” for the string passed to the DbContext constructor. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingCompactDatabase")
    {
    }
}
```

This form makes it explicit that you expect the connection string to be found in your config file. An exception will be thrown if a connection string with the given name is not found.

Database/Model First with connection string in app.config/web.config

file

Models created with the EF Designer are different from Code First in that your model already exists and is not generated from code when the application runs. The model typically exists as an EDMX file in your project.

The designer will add an EF connection string to your app.config or web.config file. This connection string is special in that it contains information about how to find the information in your EDMX file. For example:

```
<configuration>
  <connectionStrings>
    <add name="Northwind_Entities"
         connectionString="metadata=res://*/Northwind.csdl|
                                    res://*/Northwind.ssdl|
                                    res://*/Northwind.msl;
                           provider=System.Data.SqlClient;
                           provider connection string=
                               &quot;Data Source=.\sqlexpress;
                                     Initial Catalog=Northwind;
                                     Integrated Security=True;
                                     MultipleActiveResultSets=True&quot;"
         providerName="System.Data.EntityClient"/>
  </connectionStrings>
</configuration>
```

The EF Designer will also generate code that tells DbContext to use this connection by passing the connection string name to the DbContext constructor. For example:

```
public class NorthwindContext : DbContext
{
    public NorthwindContext()
        : base("name=Northwind_Entities")
    {
    }
}
```

DbContext knows to load the existing model (rather than using Code First to calculate it from code) because the connection string is an EF connection string containing details of the model to use.

Other DbContext constructor options

The DbContext class contains other constructors and usage patterns that enable some more advanced scenarios. Some of these are:

- You can use the DbModelBuilder class to build a Code First model without instantiating a DbContext instance. The result of this is a DbModel object. You can then pass this DbModel object to one of the DbContext constructors when you are ready to create your DbContext instance.
- You can pass a full connection string to DbContext instead of just the database or connection string name. By default this connection string is used with the System.Data.SqlClient provider; this can be changed by setting a different implementation of IConnectionFactory onto context.Database.DefaultConnectionFactory.
- You can use an existing DbConnection object by passing it to a DbContext constructor. If the connection object is an instance of EntityConnection, then the model specified in the connection will be used rather than calculating a model using Code First. If the object is an instance of some other type—for example, SqlConnection—then the context will use it for Code First mode.
- You can pass an existing ObjectContext to a DbContext constructor to create a DbContext wrapping the existing context. This can be used for existing applications that use ObjectContext but which want to take advantage of DbContext in some parts of the application.

Dependency resolution

9/13/2018 • 7 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Starting with EF6, Entity Framework contains a general-purpose mechanism for obtaining implementations of services that it requires. That is, when EF uses an instance of some interfaces or base classes it will ask for a concrete implementation of the interface or base class to use. This is achieved through use of the IDbDependencyResolver interface:

```
public interface IDbDependencyResolver
{
    object GetService(Type type, object key);
}
```

The GetService method is typically called by EF and is handled by an implementation of IDbDependencyResolver provided either by EF or by the application. When called, the type argument is the interface or base class type of the service being requested, and the key object is either null or an object providing contextual information about the requested service.

Unless otherwise stated any object returned must be thread-safe since it can be used as a singleton. In many cases the object returned is a factory in which case the factory itself must be thread-safe but the object returned from the factory does not need to be thread-safe since a new instance is requested from the factory for each use.

This article does not contain full details on how to implement IDbDependencyResolver, but instead acts as a reference for the service types (that is, the interface and base class types) for which EF calls GetService and the semantics of the key object for each of these calls.

System.Data.Entity.IDatabaseInitializer<TContext>

Version introduced: EF6.0.0

Object returned: A database initializer for the given context type

Key: Not used; will be null

Func<System.Data.Entity.Migrations.Sql.MigrationSqlGenerator>

Version introduced: EF6.0.0

Object returned: A factory to create a SQL generator that can be used for Migrations and other actions that cause a database to be created, such as database creation with database initializers.

Key: A string containing the ADO.NET provider invariant name specifying the type of database for which SQL will be generated. For example, the SQL Server SQL generator is returned for the key "System.Data.SqlClient".

> **NOTE**
>
> For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Entity.Core.Common.DbProviderServices

Version introduced: EF6.0.0

Object returned: The EF provider to use for a given provider invariant name

Key: A string containing the ADO.NET provider invariant name specifying the type of database for which a provider is needed. For example, the SQL Server provider is returned for the key "System.Data.SqlClient".

> **NOTE**
>
> For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Entity.Infrastructure.IDbConnectionFactory

Version introduced: EF6.0.0

Object returned: The connection factory that will be used when EF creates a database connection by convention. That is, when no connection or connection string is given to EF, and no connection string can be found in the app.config or web.config, then this service is used to create a connection by convention. Changing the connection factory can allow EF to use a different type of database (for example, SQL Server Compact Edition) by default.

Key: Not used; will be null

> **NOTE**
>
> For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Entity.Infrastructure.IManifestTokenService

Version introduced: EF6.0.0

Object returned: A service that can generate a provider manifest token from a connection. This service is typically used in two ways. First, it can be used to avoid Code First connecting to the database when building a model. Second, it can be used to force Code First to build a model for a specific database version -- for example, to force a model for SQL Server 2005 even if sometimes SQL Server 2008 is used.

Object lifetime: Singleton -- the same object may be used multiple times and concurrently by different threads

Key: Not used; will be null

System.Data.Entity.Infrastructure.IDbProviderFactoryService

Version introduced: EF6.0.0

Object returned: A service that can obtain a provider factory from a given connection. On .NET 4.5 the provider is publicly accessible from the connection. On .NET 4 the default implementation of this service uses some heuristics to find the matching provider. If these fail then a new implementation of this service can be registered to provide an appropriate resolution.

Key: Not used; will be null

Func<DbContext, System.Data.Entity.Infrastructure.IDbModelCacheKey>

Version introduced: EF6.0.0

Object returned: A factory that will generate a model cache key for a given context. By default, EF caches one model per DbContext type per provider. A different implementation of this service can be used to add other information, such as schema name, to the cache key.

Key: Not used; will be null

System.Data.Entity.Spatial.DbSpatialServices

Version introduced: EF6.0.0

Object returned: An EF spatial provider that adds support to the basic EF provider for geography and geometry spatial types.

Key: DbSptialServices is asked for in two ways. First, provider-specific spatial services are requested using a DbProviderInfo object (which contains invariant name and manifest token) as the key. Second, DbSpatialServices can be asked for with no key. This is used to resolve the "global spatial provider" which is used when creating stand-alone DbGeography or DbGeometry types.

> **NOTE**
>
> For more details on provider-related services in EF6 see the [EF6 provider model] section.

Func<System.Data.Entity.Infrastructure.IDbExecutionStrategy>

Version introduced: EF6.0.0

Object returned: A factory to create a service that allows a provider to implement retries or other behavior when queries and commands are executed against the database. If no implementation is provided, then EF will simply execute the commands and propagate any exceptions thrown. For SQL Server this service is used to provide a retry policy which is especially useful when running against cloud-based database servers such as SQL Azure.

Key: An ExecutionStrategyKey object that contains the provider invariant name and optionally a server name for which the execution strategy will be used.

> **NOTE**
>
> For more details on provider-related services in EF6 see the [EF6 provider model] section.

Func<DbConnection, string, System.Data.Entity.Migrations.History.HistoryContext>

Version introduced: EF6.0.0

Object returned: A factory that allows a provider to configure the mapping of the HistoryContext to the `__MigrationHistory` table used by EF Migrations. The HistoryContext is a Code First DbContext and can be configured using the normal fluent API to change things like the name of the table and the column mapping specifications.

Key: Not used; will be null

> **NOTE**
>
> For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Common.DbProviderFactory

Version introduced: EF6.0.0

Object returned: The ADO.NET provider to use for a given provider invariant name.

Key: A string containing the ADO.NET provider invariant name

> **NOTE**
>
> This service is not usually changed directly since the default implementation uses the normal ADO.NET provider registration. For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Entity.Infrastructure.IProviderInvariantName

Version introduced: EF6.0.0

Object returned: a service that is used to determine a provider invariant name for a given type of DbProviderFactory. The default implementation of this service uses the ADO.NET provider registration. This means that if the ADO.NET provider is not registered in the normal way because DbProviderFactory is being resolved by EF, then it will also be necessary to resolve this service.

Key: The DbProviderFactory instance for which an invariant name is required.

> **NOTE**
>
> For more details on provider-related services in EF6 see the EF6 provider model section.

System.Data.Entity.Core.Mapping.ViewGeneration.IViewAssemblyCache

Version introduced: EF6.0.0

Object returned: a cache of the assemblies that contain pre-generated views. A replacement is typically used to let EF know which assemblies contain pre-generated views without doing any discovery.

Key: Not used; will be null

System.Data.Entity.Infrastructure.Pluralization.IPluralizationService

Version introduced: EF6.0.0

Object returned: a service used by EF to pluralize and singularize names. By default an English pluralization service is used.

Key: Not used; will be null

System.Data.Entity.Infrastructure.Interception.IDbInterceptor

Version introduced: EF6.0.0

Objects returned: Any interceptors that should be registered when the application starts. Note that these objects are requested using the GetServices call and all interceptors returned by any dependency resolver will registered.

Key: Not used; will be null.

Func<System.Data.Entity.DbContext, Action<string>, System.Data.Entity.Infrastructure.Interception.DatabaseLogFormatter>

Version introduced: EF6.0.0

Object returned: A factory that will be used to create the database log formatter that will be used when the context.Database.Log property is set on the given context.

Key: Not used; will be null.

Func<System.Data.Entity.DbContext>

Version introduced: EF6.1.0

Object returned: A factory that will be used to create context instances for Migrations when the context does not have an accessible parameterless constructor.

Key: The Type object for the type of the derived DbContext for which a factory is needed.

Func<System.Data.Entity.Core.Metadata.Edm.IMetadataAnnotationSerializer>

Version introduced: EF6.1.0

Object returned: A factory that will be used to create serializers for serialization of strongly-typed custom annotations such that they can be serialized and desterilized into XML for use in Code First Migrations.

Key: The name of the annotation that is being serialized or deserialized.

Func<System.Data.Entity.Infrastructure.TransactionHandler>

Version introduced: EF6.1.0

Object returned: A factory that will be used to create handlers for transactions so that special handling can be applied for situations such as handling commit failures.

Key: An ExecutionStrategyKey object that contains the provider invariant name and optionally a server name for which the transaction handler will be used.

Connection management

9/13/2018 • 5 minutes to read • Edit Online

This page describes the behavior of Entity Framework with regard to passing connections to the context and the functionality of the **Database.Connection.Open()** API.

Passing Connections to the Context

Behavior for EF5 and earlier versions

There are two constructors which accept connections:

```
public DbContext(DbConnection existingConnection, bool contextOwnsConnection)
public DbContext(DbConnection existingConnection, DbCompiledModel model, bool contextOwnsConnection)
```

It is possible to use these but you have to work around a couple of limitations:

1. If you pass an open connection to either of these then the first time the framework attempts to use it an InvalidOperationException is thrown saying it cannot re-open an already open connection.
2. The contextOwnsConnection flag is interpreted to mean whether or not the underlying store connection should be disposed when the context is disposed. But, regardless of that setting, the store connection is always closed when the context is disposed. So if you have more than one DbContext with the same connection whichever context is disposed first will close the connection (similarly if you have mixed an existing ADO.NET connection with a DbContext, DbContext will always close the connection when it is disposed).

It is possible to work around the first limitation above by passing a closed connection and only executing code that would open it once all contexts have been created:

```
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();
                }
            }
        }
    }
}
```

The second limitation just means you need to refrain from disposing any of your DbContext objects until you are ready for the connection to be closed.

Behavior in EF6 and future versions

In EF6 and future versions the DbContext has the same two constructors but no longer requires that the connection passed to the constructor be closed when it is received. So this is now possible:

```
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();
            }
        }
    }
}
```

Also the contextOwnsConnection flag now controls whether or not the connection is both closed and disposed when the DbContext is disposed. So in the above example the connection is not closed when the context is disposed (line 32) as it would have been in previous versions of EF, but rather when the connection itself is disposed (line 40).

Of course it is still possible for the DbContext to take control of the connection (just set contextOwnsConnection to true or use one of the other constructors) if you so wish.

> **NOTE**
>
> There are some additional considerations when using transactions with this new model. For details see Working with Transactions.

Database.Connection.Open()

Behavior for EF5 and earlier versions

In EF5 and earlier versions there is a bug such that the **ObjectContext.Connection.State** was not updated to reflect the true state of the underlying store connection. For example, if you executed the following code you can be returned the status **Closed** even though in fact the underlying store connection is **Open**.

```
((IObjectContextAdapter)context).ObjectContext.Connection.State
```

Separately, if you open the database connection by calling Database.Connection.Open() it will be open until the next time you execute a query or call anything which requires a database connection (for example, SaveChanges()) but after that the underlying store connection will be closed. The context will then re-open and re-close the connection any time another database operation is required:

```
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();
            }
        }
    }
}
```

Behavior in EF6 and future versions

For EF6 and future versions we have taken the approach that if the calling code chooses to open the connection by calling context.Database.Connection.Open() then it has a good reason for doing so and the framework will assume that it wants control over opening and closing of the connection and will no longer close the connection automatically.

> **NOTE**
>
> This can potentially lead to connections which are open for a long time so use with care.

We also updated the code so that ObjectContext.Connection.State now keeps track of the state of the underlying connection correctly.

```
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
        }
    }
}
```

Connection resiliency and retry logic

9/13/2018 • 6 minutes to read • Edit Online

> **NOTE**
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Applications connecting to a database server have always been vulnerable to connection breaks due to back-end failures and network instability. However, in a LAN based environment working against dedicated database servers these errors are rare enough that extra logic to handle those failures is not often required. With the rise of cloud based database servers such as Windows Azure SQL Database and connections over less reliable networks it is now more common for connection breaks to occur. This could be due to defensive techniques that cloud databases use to ensure fairness of service, such as connection throttling, or to instability in the network causing intermittent timeouts and other transient errors.

Connection Resiliency refers to the ability for EF to automatically retry any commands that fail due to these connection breaks.

Execution Strategies

Connection retry is taken care of by an implementation of the IDbExecutionStrategy interface. Implementations of the IDbExecutionStrategy will be responsible for accepting an operation and, if an exception occurs, determining if a retry is appropriate and retrying if it is. There are four execution strategies that ship with EF:

1. **DefaultExecutionStrategy**: this execution strategy does not retry any operations, it is the default for databases other than sql server.
2. **DefaultSqlExecutionStrategy**: this is an internal execution strategy that is used by default. This strategy does not retry at all, however, it will wrap any exceptions that could be transient to inform users that they might want to enable connection resiliency.
3. **DbExecutionStrategy**: this class is suitable as a base class for other execution strategies, including your own custom ones. It implements an exponential retry policy, where the initial retry happens with zero delay and the delay increases exponentially until the maximum retry count is hit. This class has an abstract ShouldRetryOn method that can be implemented in derived execution strategies to control which exceptions should be retried.
4. **SqlAzureExecutionStrategy**: this execution strategy inherits from DbExecutionStrategy and will retry on exceptions that are known to be possibly transient when working with Azure SQL Database.

> **NOTE**
> Execution strategies 2 and 4 are included in the Sql Server provider that ships with EF, which is in the EntityFramework.SqlServer assembly and are designed to work with SQL Server.

Enabling an Execution Strategy

The easiest way to tell EF to use an execution strategy is with the SetExecutionStrategy method of the DbConfiguration class:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
    }
}
```

This code tells EF to use the SqlAzureExecutionStrategy when connecting to SQL Server.

Configuring the Execution Strategy

The constructor of SqlAzureExecutionStrategy can accept two parameters, MaxRetryCount and MaxDelay. MaxRetry count is the maximum number of times that the strategy will retry. The MaxDelay is a TimeSpan representing the maximum delay between retries that the execution strategy will use.

To set the maximum number of retries to 1 and the maximum delay to 30 seconds you would execue the following:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy(
            "System.Data.SqlClient",
            () => new SqlAzureExecutionStrategy(1, TimeSpan.FromSeconds(30)));
    }
}
```

The SqlAzureExecutionStrategy will retry instantly the first time a transient failure occurs, but will delay longer between each retry until either the max retry limit is exceeded or the total time hits the max delay.

The execution strategies will only retry a limited number of exceptions that are usually tansient, you will still need to handle other errors as well as catching the RetryLimitExceeded exception for the case where an error is not transient or takes too long to resolve itself.

There are some known of limitations when using a retrying execution strategy:

Streaming queries are not supported

By default, EF6 and later version will buffer query results rather than streaming them. If you want to have results streamed you can use the AsStreaming method to change a LINQ to Entities query to streaming.

```
using (var db = new BloggingContext())
{
    var query = (from b in db.Blogs
                orderby b.Url
                select b).AsStreaming();
    }
}
```

Streaming is not supported when a retrying execution strategy is registered. This limitation exists because the connection could drop part way through the results being returned. When this occurs, EF needs to re-run the entire query but has no reliable way of knowing which results have already been returned (data may have changed since the initial query was sent, results may come back in a different order, results may not have a unique identifier, etc.).

User initiated transactions are not supported

When you have configured an execution strategy that results in retries, there are some limitations around the use of transactions.

By default, EF will perform any database updates within a transaction. You don’t need to do anything to enable this, EF always does this automatically.

For example, in the following code SaveChanges is automatically performed within a transaction. If SaveChanges were to fail after inserting one of the new Site’s then the transaction would be rolled back and no changes applied to the database. The context is also left in a state that allows SaveChanges to be called again to retry applying the changes.

```
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();
}
```

When not using a retrying execution strategy you can wrap multiple operations in a single transaction. For example, the following code wraps two SaveChanges calls in a single transaction. If any part of either operation fails then none of the changes are applied.

```
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();
    }
}
```

This is not supported when using a retrying execution strategy because EF isn’t aware of any previous operations and how to retry them. For example, if the second SaveChanges failed then EF no longer has the required information to retry the first SaveChanges call.

Workaround: Suspend Execution Strategy

One possible workaround is to suspend the retrying execution strategy for the piece of code that needs to use a user initiated transaction. The easiest way to do this is to add a SuspendExecutionStrategy flag to your code based configuration class and change the execution strategy lambda to return the default (non-retying) execution strategy when the flag is set.

```
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);
            }
        }
    }
}
```

Note that we are using CallContext to store the flag value. This provides similar functionality to thread local storage but is safe to use with asynchronous code - including async query and save with Entity Framework.

We can now suspend the execution strategy for the section of code that uses a user initiated transaction.

```
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;
}
```

Workaround: Manually Call Execution Strategy

Another option is to manually use the execution strategy and give it the entire set of logic to be run, so that it can retry everything if one of the operations fails. We still need to suspend the execution strategy - using the technique shown above - so that any contexts used inside the retryable code block do not attempt to retry.

Note that any contexts should be constructed within the code block to be retried. This ensures that we are starting with a clean state for each retry.

```
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;
```

Handling transaction commit failures

9/18/2018 • 3 minutes to read • Edit Online

NOTE

EF6.1 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6.1. If you are using an earlier version, some or all of the information does not apply.

As part of 6.1 we are introducing a new connection resiliency feature for EF: the ability to detect and recover automatically when transient connection failures affect the acknowledgement of transaction commits. The full details of the scenario are best described in the blog post SQL Database Connectivity and the Idempotency Issue. In summary, the scenario is that when an exception is raised during a transaction commit there are two possible causes:

1. The transaction commit failed on the server
2. The transaction commit succeeded on the server but a connectivity issue prevented the success notification from reaching the client

When the first situation happens the application or the user can retry the operation, but when the second situation occurs retries should be avoided and the application could recover automatically. The challenge is that without the ability to detect what was the actual reason an exception was reported during commit, the application cannot choose the right course of action. The new feature in EF 6.1 allows EF to double-check with the database if the transaction succeeded and take the right course of action transparently.

Using the feature

In order to enable the feature you need include a call to SetTransactionHandler in the constructor of your **DbConfiguration**. If you are unfamiliar with **DbConfiguration**, see Code Based Configuration. This feature can be used in combination with the automatic retries we introduced in EF6, which help in the situation in which the transaction actually failed to commit on the server due to a transient failure:

```
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());
  }
}
```

How transactions are tracked

When the feature is enabled, EF will automatically add a new table to the database called **__Transactions**. A new row is inserted in this table every time a transaction is created by EF and that row is checked for existence if a transaction failure occurs during commit.

Although EF will do a best effort to prune rows from the table when they aren't needed anymore, the table can grow if the application exits prematurely and for that reason you may need to purge the table manually in some cases.

How to handle commit failures with previous Versions

Before EF 6.1 there was not mechanism to handle commit failures in the EF product. There are several ways to dealing with this situation that can be applied to previous versions of EF6:

- Option 1 - Do nothing

  The likelihood of a connection failure during transaction commit is low so it may be acceptable for your application to just fail if this condition actually occurs.

- Option 2 - Use the database to reset state

  1. Discard the current DbContext
  2. Create a new DbContext and restore the state of your application from the database
  3. Inform the user that the last operation might not have been completed successfully

- Option 3 - Manually track the transaction

  1. Add a non-tracked table to the database used to track the status of the transactions.
  2. Insert a row into the table at the beginning of each transaction.
  3. If the connection fails during the commit, check for the presence of the corresponding row in the database.
     - If the row is present, continue normally, as the transaction was committed successfully
     - If the row is absent, use an execution strategy to retry the current operation.
  4. If the commit is successful, delete the corresponding row to avoid the growth of the table.

This blog post contains sample code for accomplishing this on SQL Azure.

Databinding with WinForms

9/13/2018 • 13 minutes to read • Edit Online

This step-by-step walkthrough shows how to bind POCO types to Window Forms (WinForms) controls in a "master-detail" form. The application uses Entity Framework to populate objects with data from the database, track changes, and persist data to the database.

The model defines two types that participate in one-to-many relationship: Category (principal\master) and Product (dependent\detail). Then, the Visual Studio tools are used to bind the types defined in the model to the WinForms controls. The WinForms data-binding framework enables navigation between related objects: selecting rows in the master view causes the detail view to update with the corresponding child data.

The screen shots and code listings in this walkthrough are taken from Visual Studio 2013 but you can complete this walkthrough with Visual Studio 2012 or Visual Studio 2010.

Pre-Requisites

You need to have Visual Studio 2013, Visual Studio 2012 or Visual Studio 2010 installed to complete this walkthrough.

If you are using Visual Studio 2010, you also have to install NuGet. For more information, see Installing NuGet.

Create the Application

- Open Visual Studio
- **File -> New -> Project....**
- Select **Windows** in the left pane and **Windows FormsApplication** in the right pane
- Enter **WinFormswithEFSample** as the name
- Select **OK**

Install the Entity Framework NuGet package

- In Solution Explorer, right-click on the **WinFormswithEFSample** project
- Select **Manage NuGet Packages...**
- In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package
- Click **Install**

> **NOTE**
> In addition to the EntityFramework assembly a reference to System.ComponentModel.DataAnnotations is also added. If the project has a reference to System.Data.Entity, then it will be removed when the EntityFramework package is installed. The System.Data.Entity assembly is no longer used for Entity Framework 6 applications.

Implementing IListSource for Collections

Collection properties must implement the IListSource interface to enable two-way data binding with sorting when using Windows Forms. To do this we are going to extend ObservableCollection to add IListSource functionality.

- Add an **ObservableListSource** class to the project:
  - Right-click on the project name

- Select **Add -> New Item**
- Select **Class** and enter **ObservableListSource** for the class name
- Replace the code generated by default with the following code:

This class enables two-way data binding as well as sorting. The class derives from ObservableCollection<T> and adds an explicit implementation of IListSource. The GetList() method of IListSource is implemented to return an IBindingList implementation that stays in sync with the ObservableCollection. The IBindingList implementation generated by ToBindingList supports sorting. The ToBindingList extention method is defined in the EntityFramework assembly.

```
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());
        }
    }
}
```

Define a Model

In this walkthrough you can chose to implement a model using Code First or the EF Designer. Complete one of the two following sections.

Option 1: Define a Model using Code First

This section shows how to create a model and its associated database using Code First. Skip to the next section (**Option 2: Define a model using Database First)** if you would rather use Database First to reverse engineer your model from a database using the EF designer

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model.

- Add a new **Product** class to project
- Replace the code generated by default with the following code:

```
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; }
    }
}
```

- Add a **Category** class to the project.
- Replace the code generated by default with the following code:

```
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; } }
    }
}
```

In addition to defining entities, you need to define a class that derives from **DbContext** and exposes **DbSet<TEntity>** properties. The **DbSet** properties let the context know which types you want to include in the model. The **DbContext** and **DbSet** types are defined in the EntityFramework assembly.

An instance of the DbContext derived type manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.

- Add a new **ProductContext** class to the project.
- Replace the code generated by default with the following code:

```
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; }
    }
}
```

Compile the project.

Option 2: Define a model using Database First

This section shows how to use Database First to reverse engineer your model from a database using the EF designer. If you completed the previous section (**Option 1: Define a model using Code First)**, then skip this section and go straight to the **Lazy Loading** section.

Create an Existing Database

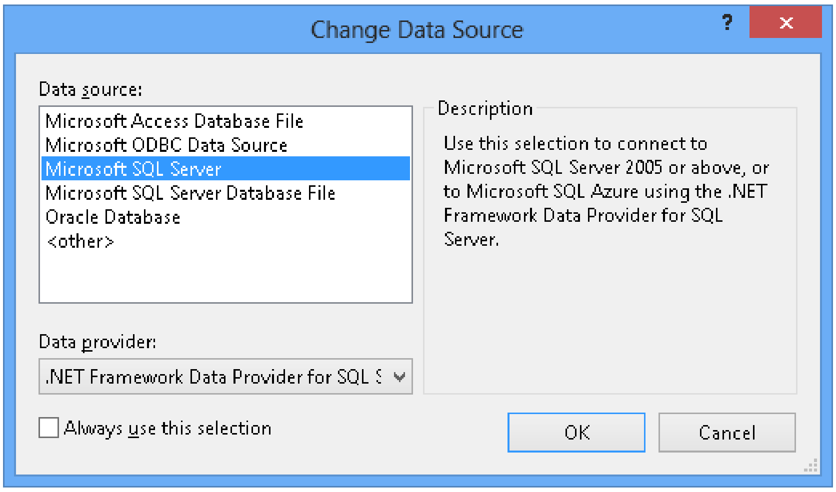
Typically when you are targeting an existing database it will already be created, but for this walkthrough we need to create a database to access.

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

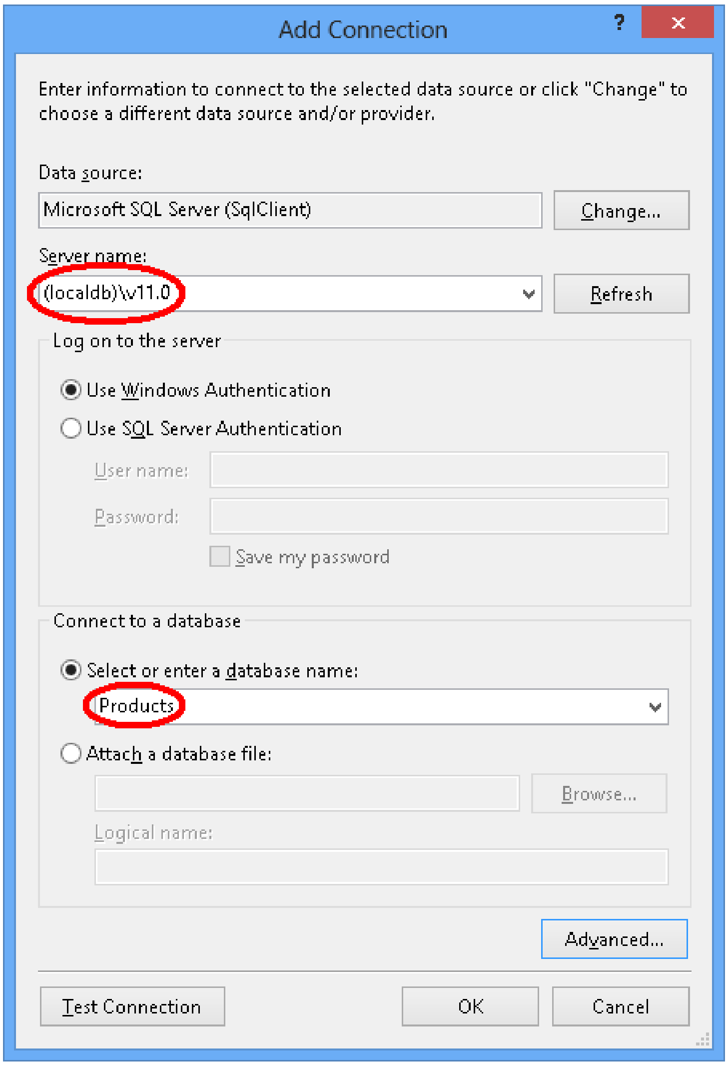
- If you are using Visual Studio 2010 you'll be creating a SQL Express database.
- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.

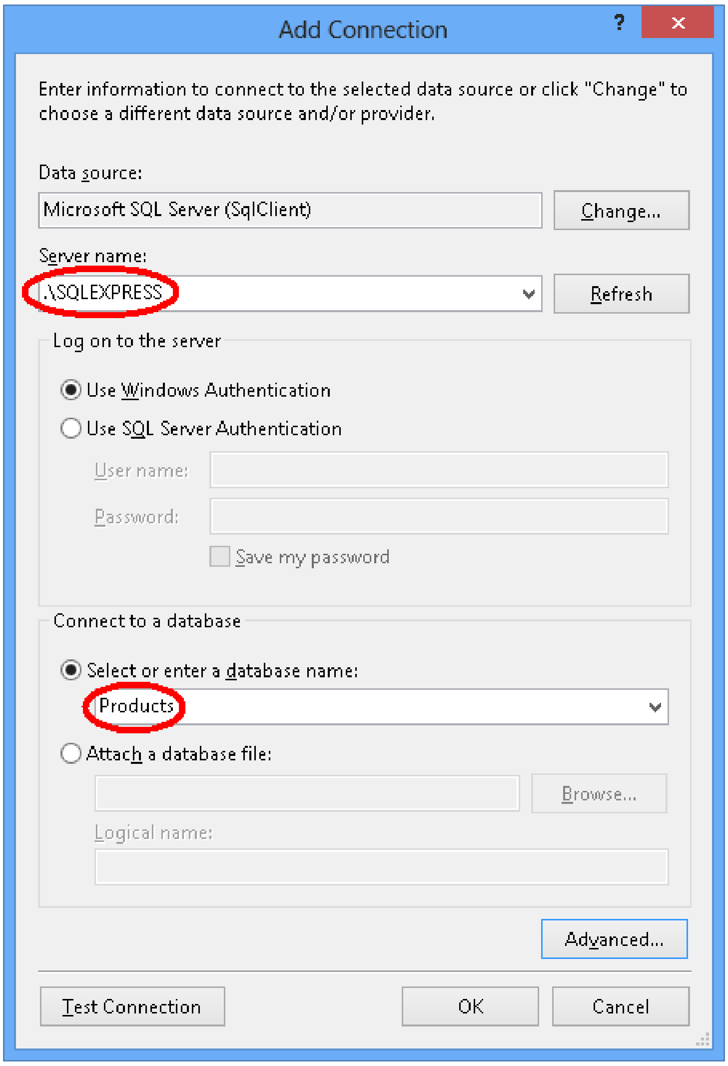
Let's go ahead and generate the database.

- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection…**
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source

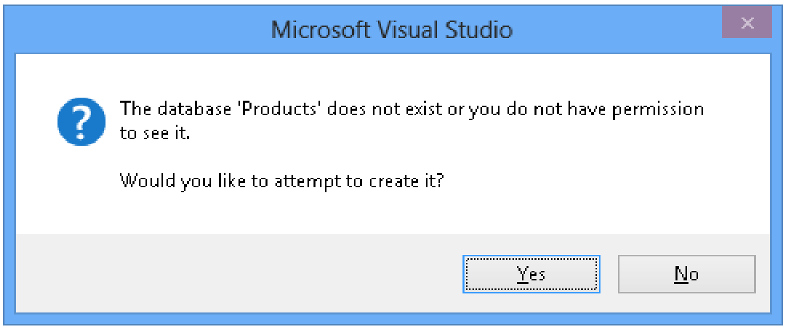


- Connect to either LocalDB or SQL Express, depending on which one you have installed, and enter **Products** as the database name


Add Connection
Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.
Data source:
Microsoft SQL Server (SqlClient)
Change...
Server name:
(localdb)\v11.0
Refresh
Log on to the server
Use Windows Authentication
Use SQL Server Authentication
User name:
Password:
Save my password
Connect to a database
Select or enter a database name:
Products
Attach a database file:
Browse...
Logical name:
Advanced...
Test Connection
OK
Cancel



- Select **OK** and you will be asked if you want to create a new database, select **Yes**



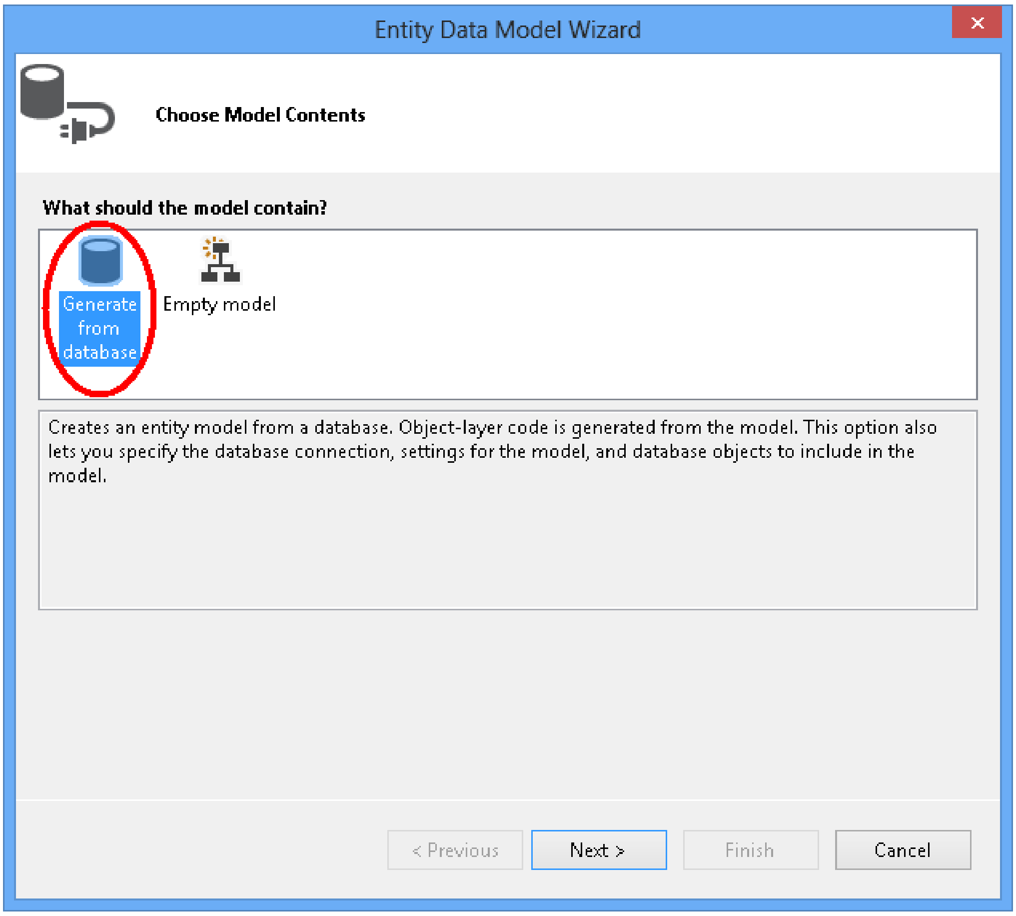
- The new database will now appear in Server Explorer, right-click on it and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **Execute**

```
    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
```

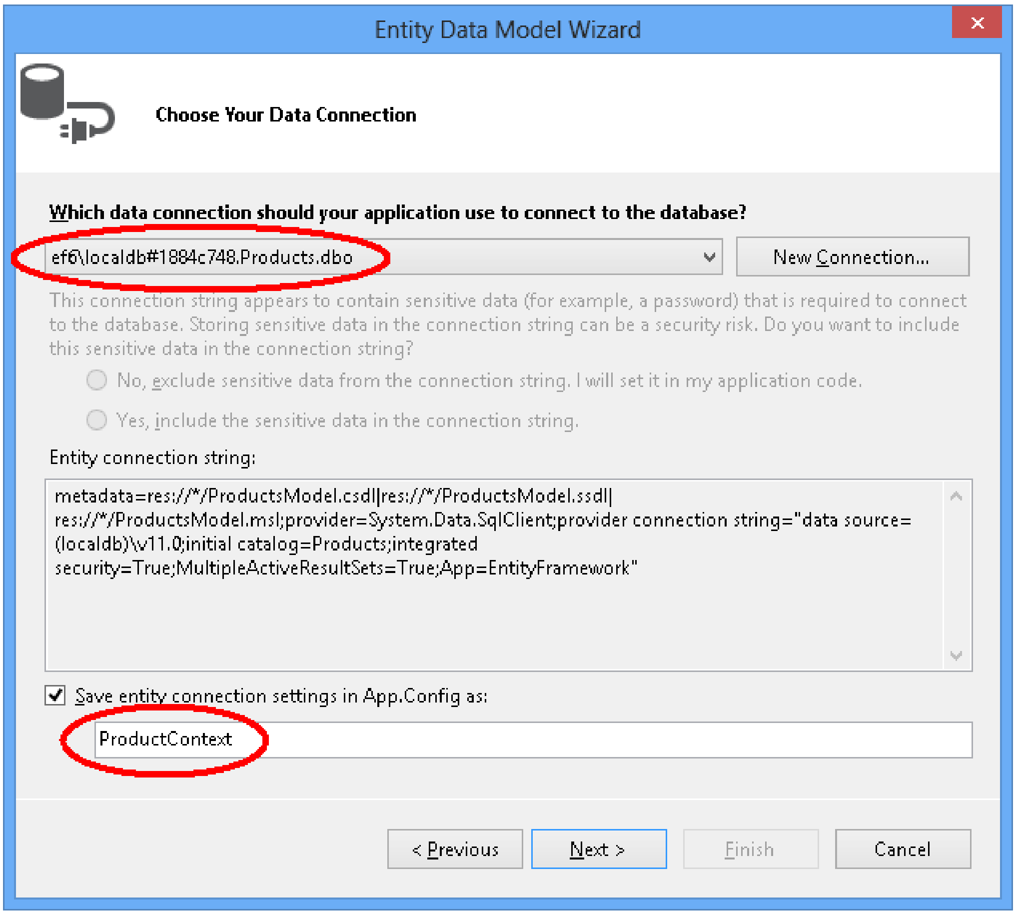
Reverse Engineer Model

We're going to make use of Entity Framework Designer, which is included as part of Visual Studio, to create our model.

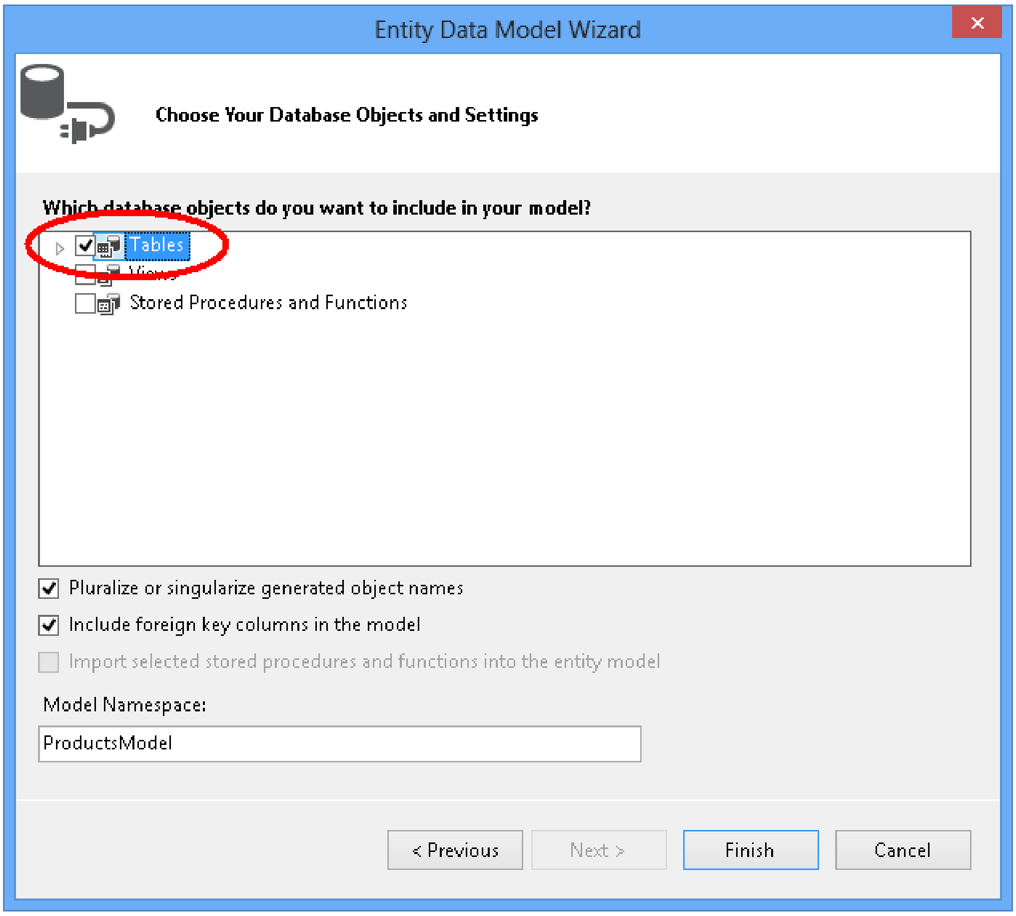
- **Project -> Add New Item…**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **ProductModel** as the name and click **OK**
- This launches the **Entity Data Model Wizard**
- Select **Generate from Database** and click **Next**



- Select the connection to the database you created in the first section, enter **ProductContext** as the name of the connection string and click **Next**



- Click the checkbox next to 'Tables' to import all tables and click 'Finish'



Once the reverse engineer process completes the new model is added to your project and opened up for you to view in the Entity Framework Designer. An App.config file has also been added to your project with the connection details for the database.

Additional Steps in Visual Studio 2010

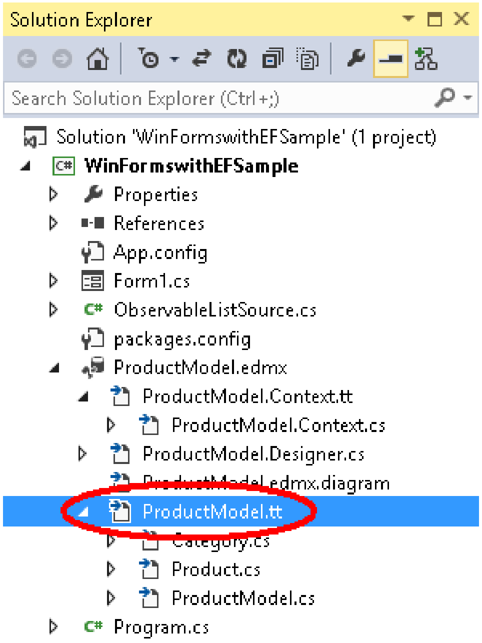
If you are working in Visual Studio 2010 then you will need to update the EF designer to use EF6 code generation.

- Right-click on an empty spot of your model in the EF Designer and select **Add Code Generation Item...**
- Select **Online Templates** from the left menu and search for **DbContext**
- Select the **EF 6.x DbContext Generator for C#,** enter **ProductsModel** as the name and click Add

Updating code generation for data binding

EF generates code from your model using T4 templates. The templates shipped with Visual Studio or downloaded from the Visual Studio gallery are intended for general purpose use. This means that the entities generated from these templates have simple ICollection<T> properties. However, when doing data binding it is desirable to have collection properties that implement IListSource. This is why we created the ObservableListSource class above and we are now going to modify the templates to make use of this class.

- Open the **Solution Explorer** and find **ProductModel.edmx** file
- Find the **ProductModel.tt** file which will be nested under the ProductModel.edmx file



- Double-click on the ProductModel.tt file to open it in the Visual Studio editor
- Find and replace the two occurrences of "**ICollection**" with "**ObservableListSource**". These are located at approximately lines 296 and 484.
- Find and replace the first occurrence of "**HashSet**" with "**ObservableListSource**". This occurrence is located at approximately line 50. **Do not** replace the second occurrence of HashSet found later in the code.
- Save the ProductModel.tt file. This should cause the code for entities to be regenerated. If the code does not regenerate automatically, then right click on ProductModel.tt and choose "Run Custom Tool".

If you now open the Category.cs file (which is nested under ProductModel.tt) then you should see that the Products collection has the type **ObservableListSource<Product>**.

Compile the project.

Lazy Loading

The **Products** property on the **Category** class and **Category** property on the **Product** class are navigation properties. In Entity Framework, navigation properties provide a way to navigate a relationship between two entity types.

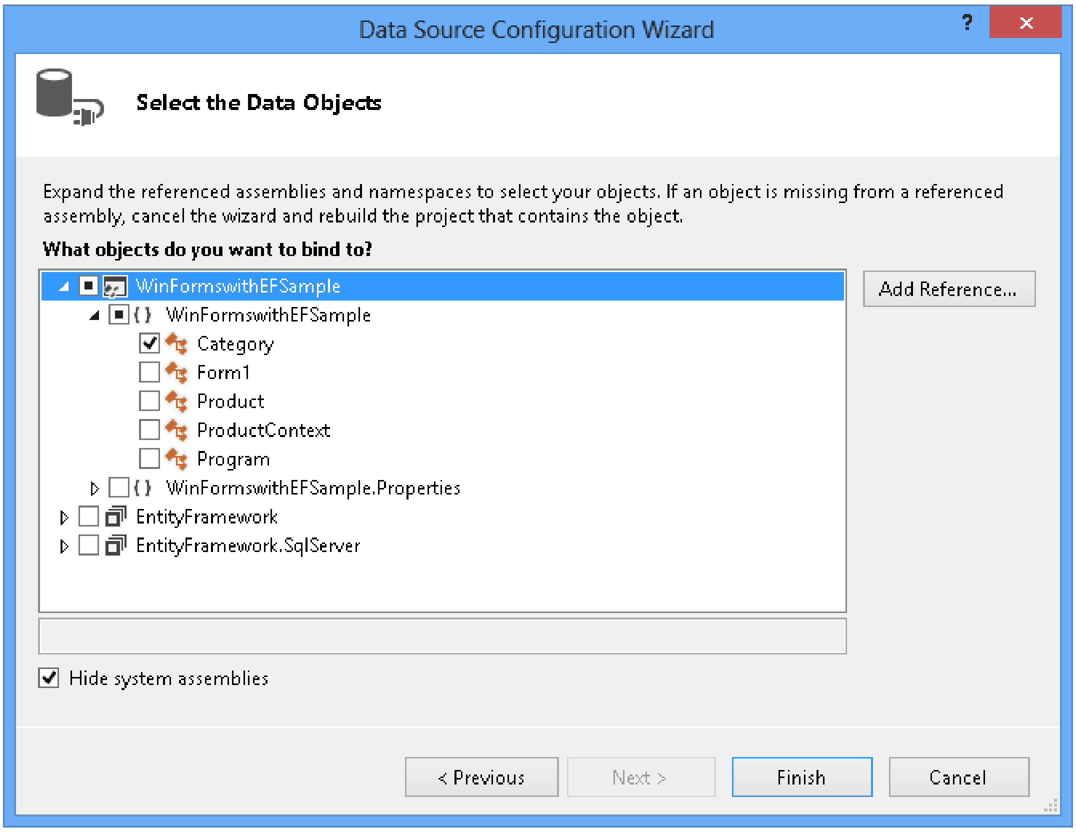
EF gives you an option of loading related entities from the database automatically the first time you access the navigation property. With this type of loading (called lazy loading), be aware that the first time you access each navigation property a separate query will be executed against the database if the contents are not already in the context.

When using POCO entity types, EF achieves lazy loading by creating instances of derived proxy types during runtime and then overriding virtual properties in your classes to add the loading hook. To get lazy loading of related objects, you must declare navigation property getters as **public** and **virtual** (**Overridable** in Visual Basic), and you class must not be **sealed** (**NotOverridable** in Visual Basic). When using Database First navigation properties are automatically made virtual to enable lazy loading. In the Code First section we chose to make the navigation properties virtual for the same reason

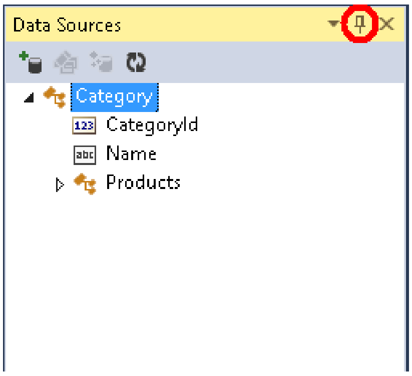
Bind Object to Controls

Add the classes that are defined in the model as data sources for this WinForms application.

- From the main menu, select **Project -> Add New Data Source ...** (in Visual Studio 2010, you need to select **Data -> Add New Data Source...**)
- In the Choose a Data Source Type window, select **Object** and click **Next**
- In the Select the Data Objects dialog, unfold the **WinFormswithEFSample** two times and select **Category** There is no need to select the Product data source, because we will get to it through the Product's property on the Category data source.



- Click **Finish.** *If the Data Sources window is not showing up, select***View -> Other Windows-> Data Sources**
- Press the pin icon, so the Data Sources window does not auto hide. You may need to hit the refresh button if the window was already visible.



- In Solution Explorer, double-click the **Form1.cs** file to open the main form in designer.

- Select the **Category** data source and drag it on the form. By default, a new DataGridView (**categoryDataGridView**) and Navigation toolbar controls are added to the designer. These controls are bound to the BindingSource (**categoryBindingSource**) and Binding Navigator (**categoryBindingNavigator**) components that are created as well.

- Edit the columns on the **categoryDataGridView**. We want to set the **CategoryId** column to read-only. The value for the **CategoryId** property is generated by the database after we save the data.

  - Right-click the DataGridView control and select Edit Columns...
  - Select the CategoryId column and set ReadOnly to True
  - Press OK

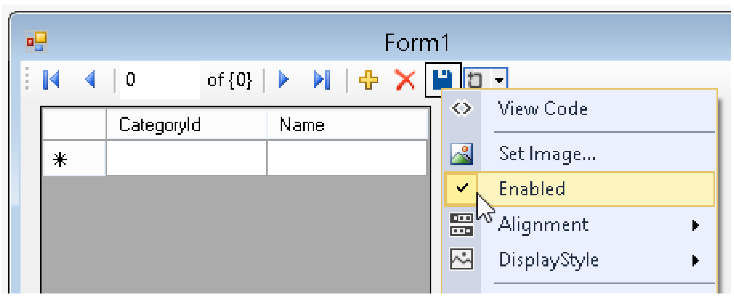
- Select Products from under the Category data source and drag it on the form. The productDataGridView and productBindingSource are added to the form.

- Edit the columns on the productDataGridView. We want to hide the CategoryId and Category columns and set ProductId to read-only. The value for the ProductId property is generated by the database after we save the data.

  - Right-click the DataGridView control and select **Edit Columns...**.
  - Select the **ProductId** column and set **ReadOnly** to **True**.
  - Select the **CategoryId** column and press the **Remove** button. Do the same with the **Category** column.
  - Press **OK**.

  So far, we associated our DataGridView controls with BindingSource components in the designer. In the next section we will add code to the code behind to set categoryBindingSource.DataSource to the collection of entities that are currently tracked by DbContext. When we dragged-and-dropped Products from under the Category, the WinForms took care of setting up the productsBindingSource.DataSource property to categoryBindingSource and productsBindingSource.DataMember property to Products. Because of this binding, only the products that belong to the currently selected Category will be displayed in the productDataGridView.

- Enable the **Save** button on the Navigation toolbar by clicking the right mouse button and selecting **Enabled**.



- Add the event handler for the save button by double-clicking on the button. This will add the event handler and bring you to the code behind for the form. The code for the **categoryBindingNavigatorSaveItem_Click** event handler will be added in the next section.

Add the Code that Handles Data Interaction

We'll now add the code to use the ProductContext to perform data access. Update the code for the main form window as shown below.

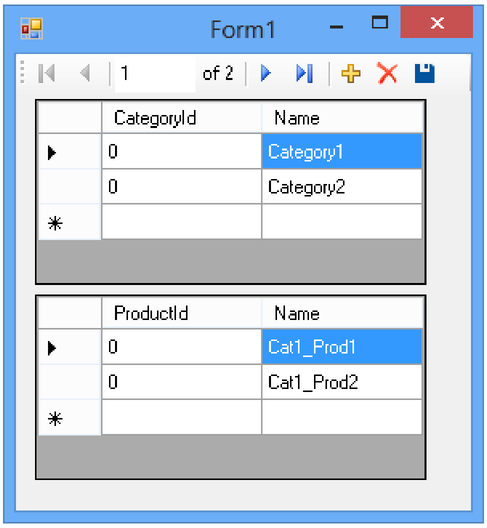
The code declares a long-running instance of ProductContext. The ProductContext object is used to query and save data to the database. The Dispose() method on the ProductContext instance is then called from the overridden OnClosing method. The code comments provide details about what the code does.

```
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
```

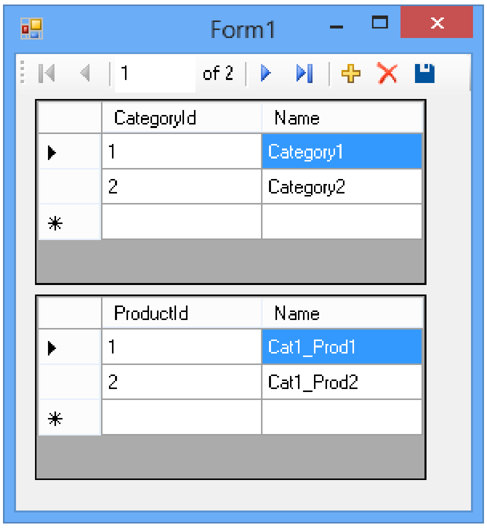
```
            // The ToList call is required because otherwise
            // the collection will be modified
            // by the Remove call while it is being enumerated.
            // In most other situations you can do LINQ to Objects directly
            // against the Local property without 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();
        }
    }
}
```

Test the Windows Forms Application

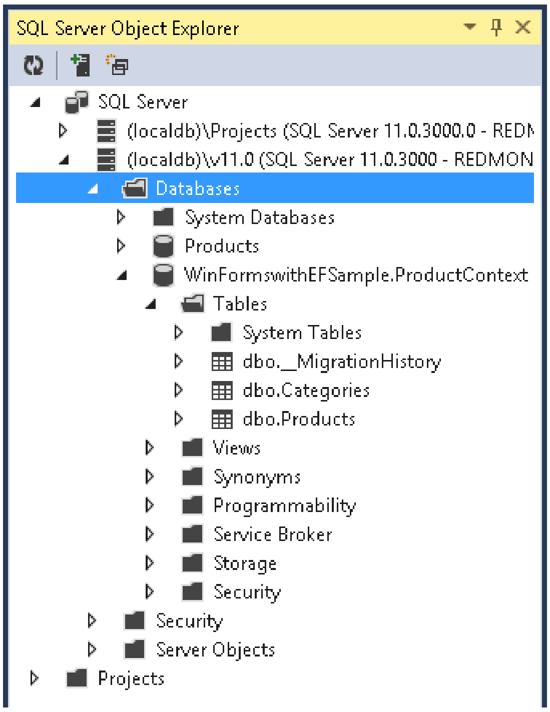
- Compile and run the application and you can test out the functionality.



- After saving the store generated keys are shown on the screen.



- If you used Code First, then you will also see that a **WinFormswithEFSample.ProductContext** database is created for you.



Databinding with WPF

10/4/2018 • 13 minutes to read • Edit Online

This step-by-step walkthrough shows how to bind POCO types to WPF controls in a “master-detail" form. The application uses the Entity Framework APIs to populate objects with data from the database, track changes, and persist data to the database.

The model defines two types that participate in one-to-many relationship: **Category** (principal\master) and **Product** (dependent\detail). Then, the Visual Studio tools are used to bind the types defined in the model to the WPF controls. The WPF data-binding framework enables navigation between related objects: selecting rows in the master view causes the detail view to update with the corresponding child data.

The screen shots and code listings in this walkthrough are taken from Visual Studio 2013 but you can complete this walkthrough with Visual Studio 2012 or Visual Studio 2010.

Use the 'Object' Option for Creating WPF Data Sources

With previous version of Entity Framework we used to recommend using the **Database** option when creating a new Data Source based on a model created with the EF Designer. This was because the designer would generate a context that derived from ObjectContext and entity classes that derived from EntityObject. Using the Database option would help you write the best code for interacting with this API surface.

The EF Designers for Visual Studio 2012 and Visual Studio 2013 generate a context that derives from DbContext together with simple POCO entity classes. With Visual Studio 2010 we recommend swapping to a code generation template that uses DbContext as described later in this walkthrough.

When using the DbContext API surface you should use the **Object** option when creating a new Data Source, as shown in this walkthrough.

If needed, you can revert to ObjectContext based code generation for models created with the EF Designer.

Pre-Requisites

You need to have Visual Studio 2013, Visual Studio 2012 or Visual Studio 2010 installed to complete this walkthrough.

If you are using Visual Studio 2010, you also have to install NuGet. For more information, see Installing NuGet.

Create the Application

- Open Visual Studio
- **File -> New -> Project….**
- Select **Windows** in the left pane and **WPFApplication** in the right pane
- Enter **WPFwithEFSample** as the name
- Select **OK**

Install the Entity Framework NuGet package

- In Solution Explorer, right-click on the **WinFormswithEFSample** project
- Select **Manage NuGet Packages…**
- In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package

- Click **Install**

> **NOTE**
>
> In addition to the EntityFramework assembly a reference to System.ComponentModel.DataAnnotations is also added. If the project has a reference to System.Data.Entity, then it will be removed when the EntityFramework package is installed. The System.Data.Entity assembly is no longer used for Entity Framework 6 applications.

Define a Model

In this walkthrough you can chose to implement a model using Code First or the EF Designer. Complete one of the two following sections.

Option 1: Define a Model using Code First

This section shows how to create a model and its associated database using Code First. Skip to the next section (**Option 2: Define a model using Database First)** if you would rather use Database First to reverse engineer your model from a database using the EF designer

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model.

- Add a new class to the **WPFwithEFSample:**
  - Right-click on the project name
  - Select **Add**, then **New Item**
  - Select **Class** and enter **Product** for the class name
- Replace the **Product** class definition with the following code:

```
    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; }
        }
    }
```

The **Products** property on the **Category** class and **Category** property on the **Product** class are navigation properties. In Entity Framework, navigation properties provide a way to navigate a relationship between two entity types.

In addition to defining entities, you need to define a class that derives from DbContext and exposes DbSet<TEntity> properties. The DbSet<TEntity> properties let the context know which types you want to include in the model.

An instance of the DbContext derived type manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.

- Add a new **ProductContext** class to the project with the following definition:

```
using System.Data.Entity;

namespace WPFwithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}
```

Compile the project.

Option 2: Define a model using Database First

This section shows how to use Database First to reverse engineer your model from a database using the EF designer. If you completed the previous section (**Option 1: Define a model using Code First)**, then skip this section and go straight to the **Lazy Loading** section.

Create an Existing Database

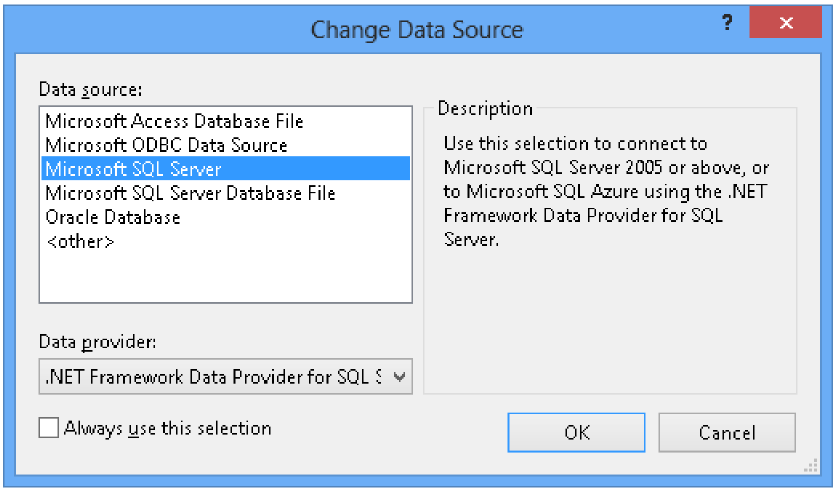
Typically when you are targeting an existing database it will already be created, but for this walkthrough we need to create a database to access.

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

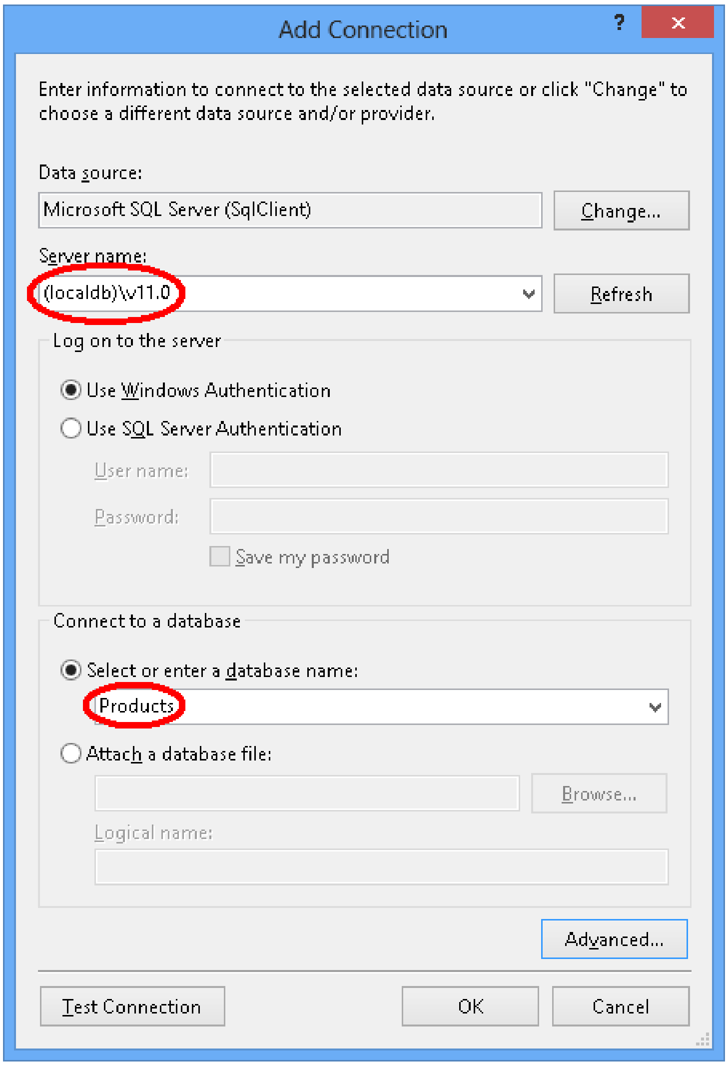
- If you are using Visual Studio 2010 you'll be creating a SQL Express database.
- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.

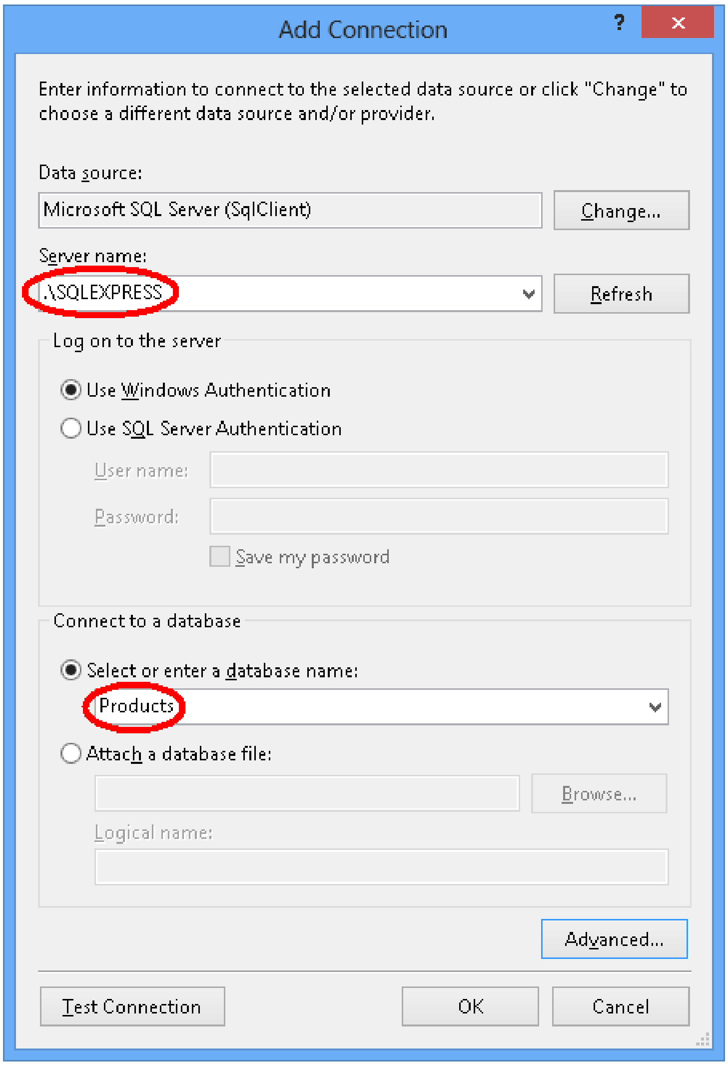
Let's go ahead and generate the database.

- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection…**
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source

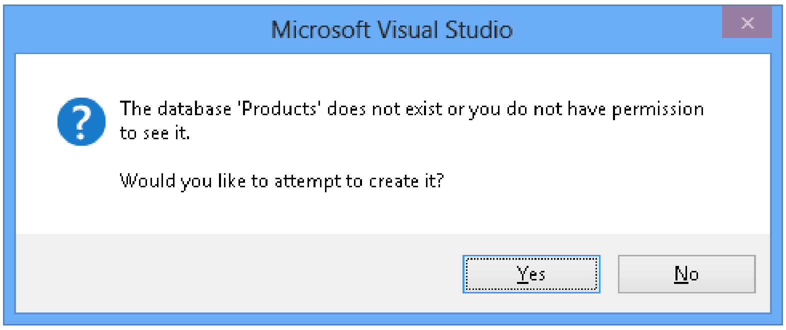


- Connect to either LocalDB or SQL Express, depending on which one you have installed, and enter **Products** as the database name





- Select **OK** and you will be asked if you want to create a new database, select **Yes**



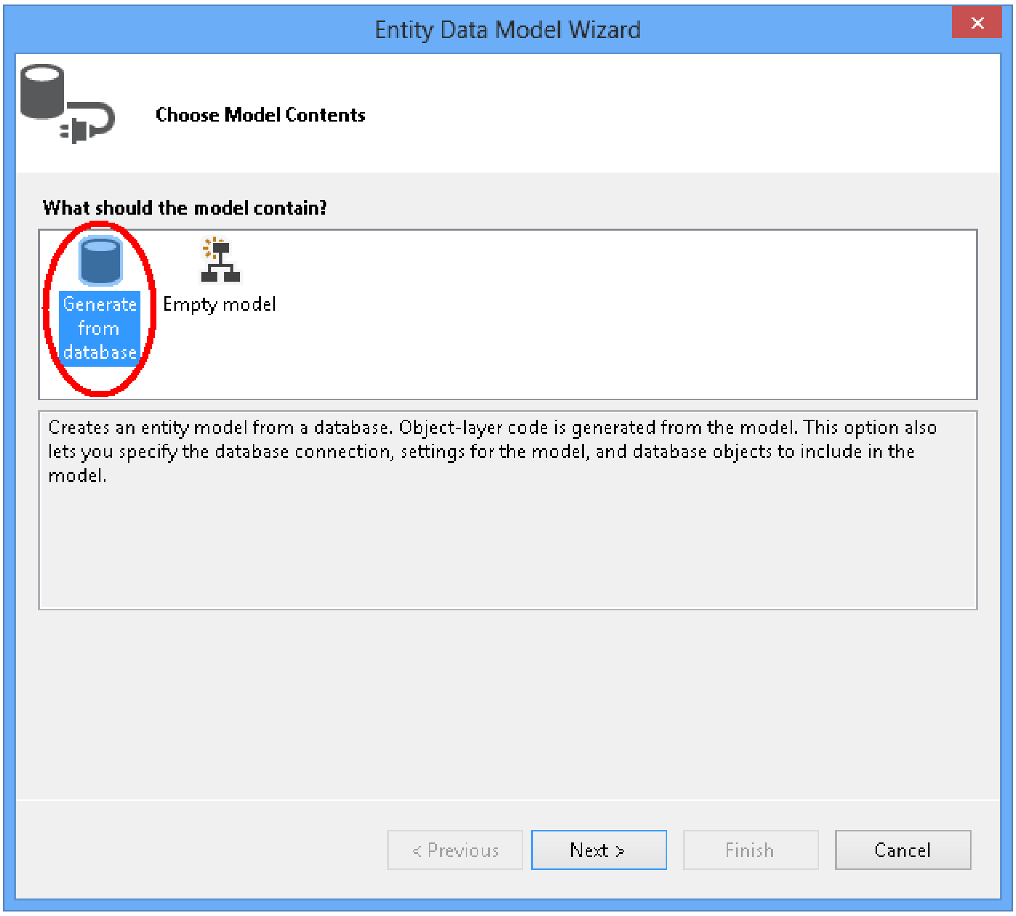
- The new database will now appear in Server Explorer, right-click on it and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **Execute**

```
    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
```

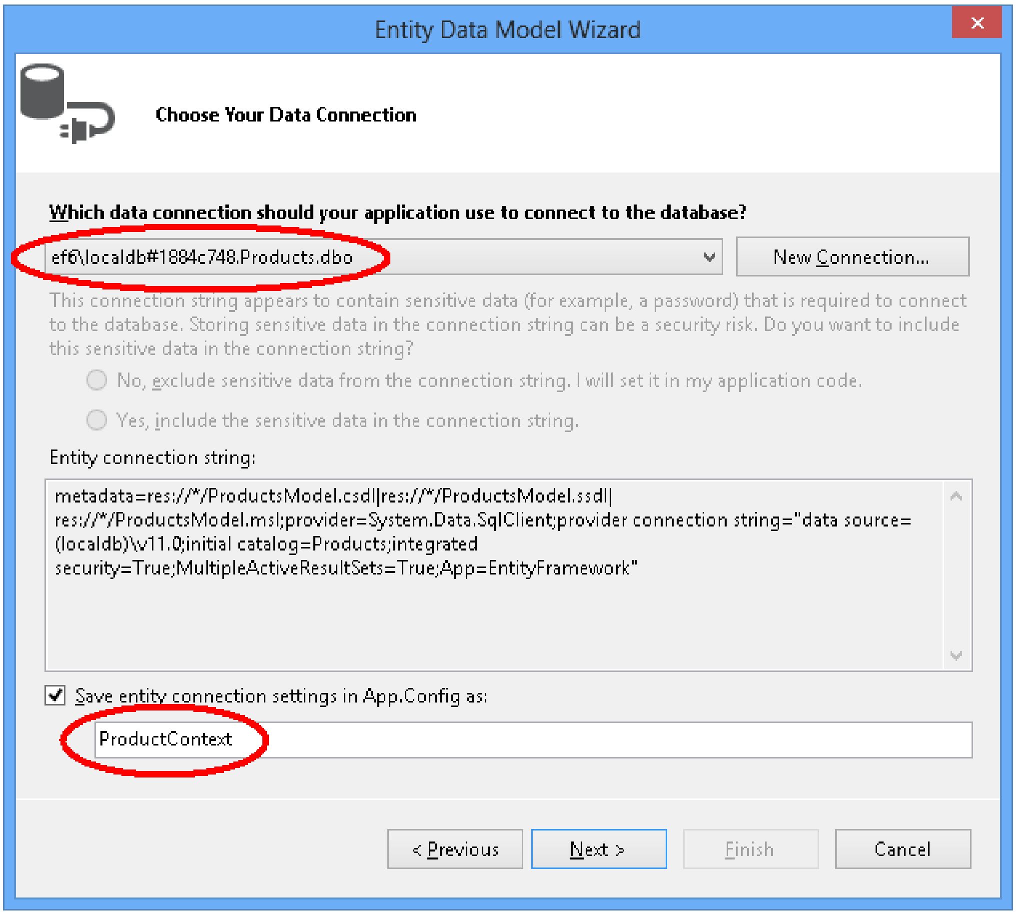
Reverse Engineer Model

We're going to make use of Entity Framework Designer, which is included as part of Visual Studio, to create our model.

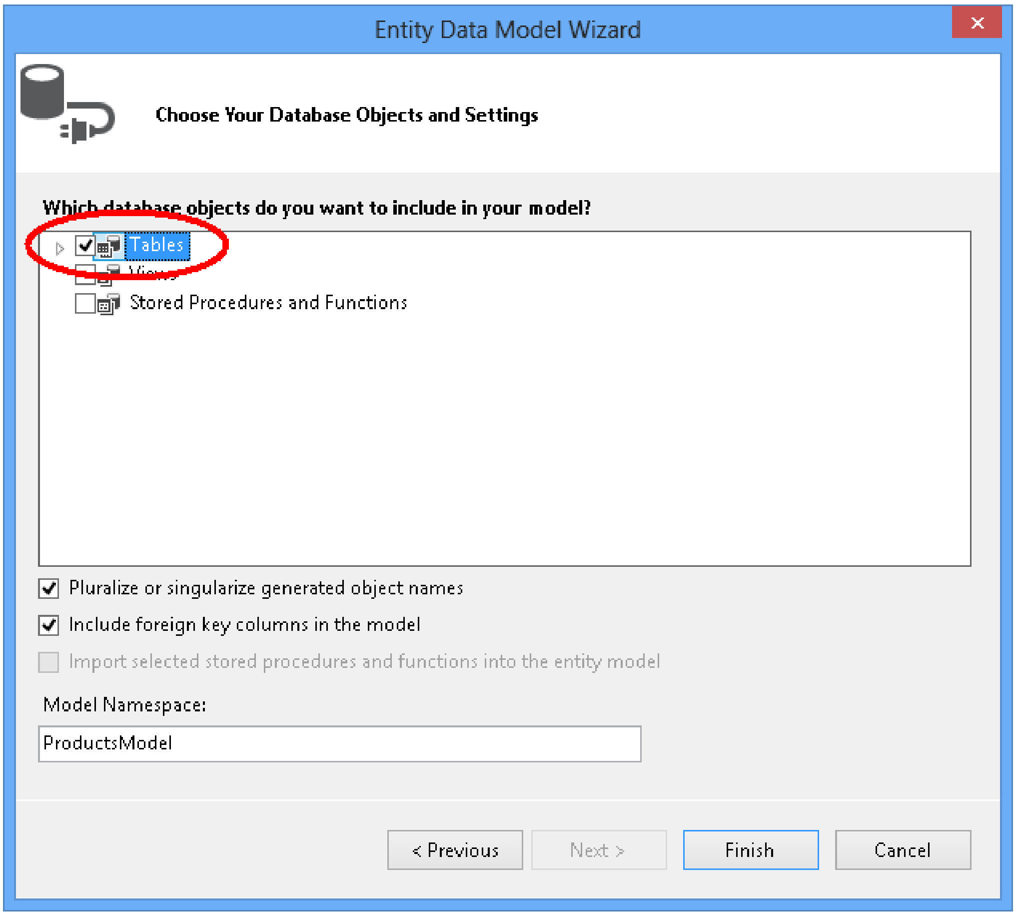
- **Project -> Add New Item...**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **ProductModel** as the name and click **OK**
- This launches the **Entity Data Model Wizard**
- Select **Generate from Database** and click **Next**



- Select the connection to the database you created in the first section, enter **ProductContext** as the name of the connection string and click **Next**



- Click the checkbox next to 'Tables' to import all tables and click 'Finish'



Once the reverse engineer process completes the new model is added to your project and opened up for you to view in the Entity Framework Designer. An App.config file has also been added to your project with the connection details for the database.

Additional Steps in Visual Studio 2010

If you are working in Visual Studio 2010 then you will need to update the EF designer to use EF6 code generation.

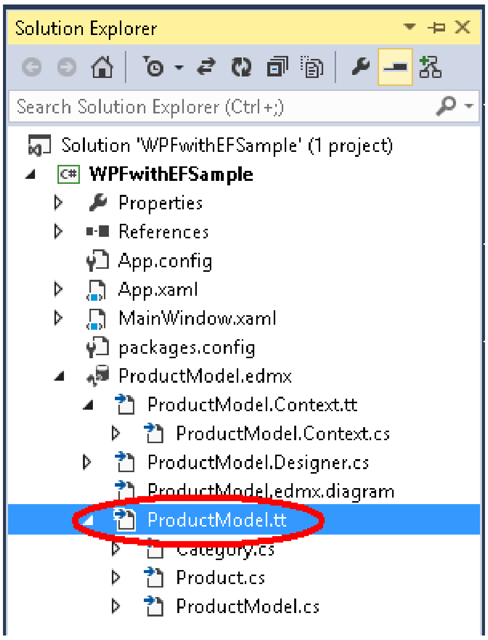
- Right-click on an empty spot of your model in the EF Designer and select **Add Code Generation Item...**
- Select **Online Templates** from the left menu and search for **DbContext**
- Select the **EF 6.x DbContext Generator for C#,** enter **ProductsModel** as the name and click Add

Updating code generation for data binding

EF generates code from your model using T4 templates. The templates shipped with Visual Studio or downloaded from the Visual Studio gallery are intended for general purpose use. This means that the entities generated from these templates have simple ICollection<T> properties. However, when doing data binding using WPF it is desirable to use **ObservableCollection** for collection properties so that WPF can keep track of changes made to the collections. To this end we will to modify the templates to use ObservableCollection.

- Open the **Solution Explorer** and find **ProductModel.edmx** file

- Find the **ProductModel.tt** file which will be nested under the ProductModel.edmx file



- Double-click on the ProductModel.tt file to open it in the Visual Studio editor

- Find and replace the two occurrences of "**ICollection**" with "**ObservableCollection**". These are located approximately at lines 296 and 484.

- Find and replace the first occurrence of "**HashSet**" with "**ObservableCollection**". This occurrence is located approximately at line 50. **Do not** replace the second occurrence of HashSet found later in the code.

- Find and replace the only occurrence of "**System.Collections.Generic**" with "**System.Collections.ObjectModel**". This is located approximately at line 424.

- Save the ProductModel.tt file. This should cause the code for entities to be regenerated. If the code does not regenerate automatically, then right click on ProductModel.tt and choose "Run Custom Tool".

If you now open the Category.cs file (which is nested under ProductModel.tt) then you should see that the Products collection has the type **ObservableCollection<Product>**.

Compile the project.

Lazy Loading

The **Products** property on the **Category** class and **Category** property on the **Product** class are navigation properties. In Entity Framework, navigation properties provide a way to navigate a relationship between two entity types.

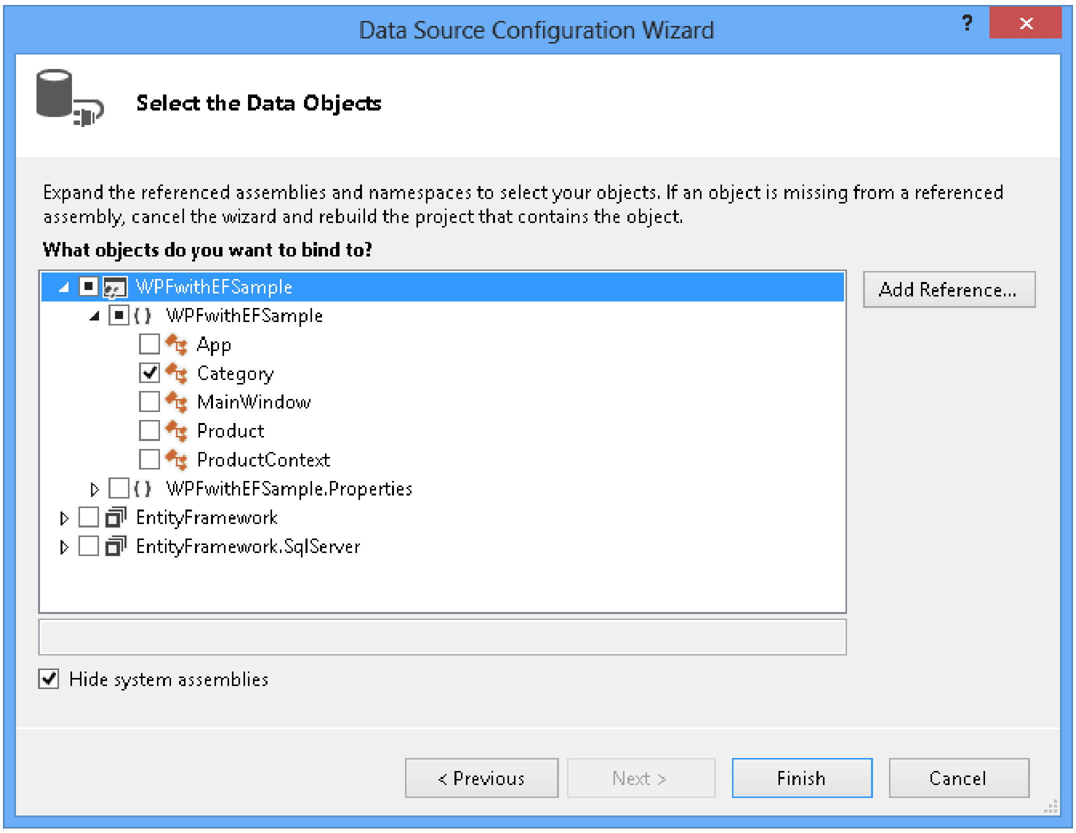
EF gives you an option of loading related entities from the database automatically the first time you access the navigation property. With this type of loading (called lazy loading), be aware that the first time you access each navigation property a separate query will be executed against the database if the contents are not already in the context.

When using POCO entity types, EF achieves lazy loading by creating instances of derived proxy types during runtime and then overriding virtual properties in your classes to add the loading hook. To get lazy loading of related objects, you must declare navigation property getters as **public** and **virtual** (**Overridable** in Visual Basic), and you class must not be **sealed** (**NotOverridable** in Visual Basic). When using Database First navigation properties are automatically made virtual to enable lazy loading. In the Code First section we chose to make the navigation properties virtual for the same reason

Bind Object to Controls

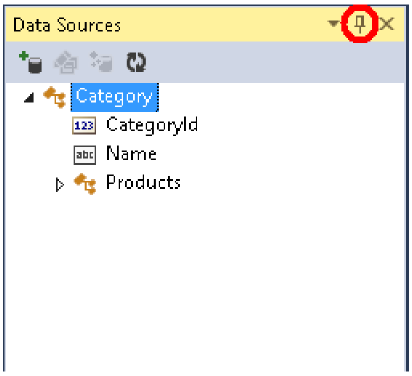
Add the classes that are defined in the model as data sources for this WPF application.

- Double-click **MainWindow.xaml** in Solution Explorer to open the main form
- From the main menu, select **Project -> Add New Data Source ...** (in Visual Studio 2010, you need to select **Data -> Add New Data Source...**)
- In the Choose a Data Source Typewindow, select **Object** and click **Next**
- In the Select the Data Objects dialog, unfold the **WPFwithEFSample** two times and select **Category** *There is no need to select the **Product** data source, because we will get to it through the **Product**'s property on the **Category** data source*



- Click **Finish.**
- The Data Sources window is opened next to the MainWindow.xaml window *If the Data Sources window is not showing up, select **View -> Other Windows-> Data Sources***

- Press the pin icon, so the Data Sources window does not auto hide. You may need to hit the refresh button if the window was already visible.



- Select the **Category** data source and drag it on the form.

The following happened when we dragged this source:

- The **categoryViewSource** resource and the **categoryDataGrid** control were added to XAML
- The DataContext property on the parent Grid element was set to "{StaticResource **categoryViewSource** }". The **categoryViewSource** resource serves as a binding source for the outer\parent Grid element. The inner Grid elements then inherit the DataContext value from the parent Grid (the categoryDataGrid's ItemsSource property is set to "{Binding}")

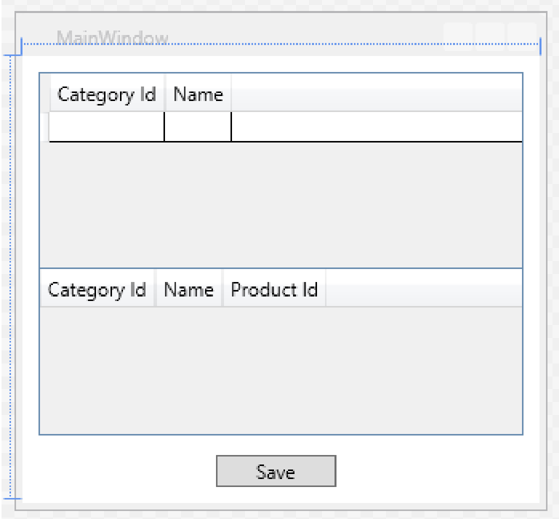
```
<Window.Resources>
    <CollectionViewSource x:Key="categoryViewSource"
                            d:DesignSource="{d:DesignInstance {x:Type local:Category}, CreateList=True}"/>
</Window.Resources>
<Grid DataContext="{StaticResource categoryViewSource}">
    <DataGrid x:Name="categoryDataGrid" AutoGenerateColumns="False" 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>
```

Adding a Details Grid

Now that we have a grid to display Categories let's add a details grid to display the associated Products.

- Select the **Products** property from under the **Category** data source and drag it on the form.
  - The **categoryProductsViewSource** resource and **productDataGrid** grid are added to XAML
  - The binding path for this resource is set to Products
  - WPF data-binding framework ensures that only Products related to the selected Category show up in **productDataGrid**
- From the Toolbox, drag **Button** on to the form. Set the **Name** property to **buttonSave** and the **Content** property to **Save**.

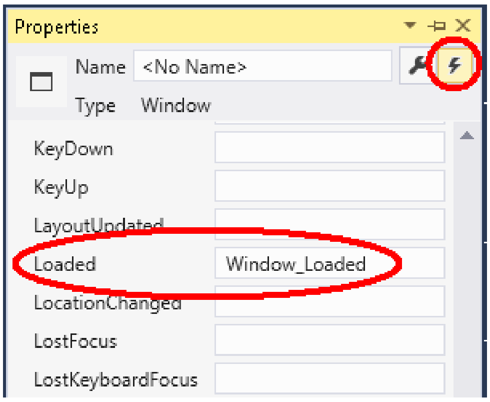
The form should look similar to this:



Add Code that Handles Data Interaction

It's time to add some event handlers to the main window.

- In the XAML window, click on the **<Window** element, this selects the main window
- In the **Properties** window choose **Events** at the top right, then double-click the text box to right of the **Loaded** label



- Also add the **Click** event for the **Save** button by double-clicking the Save button in the designer.

This brings you to the code behind for the form, we'll now edit the code to use the ProductContext to perform data access. Update the code for the MainWindow as shown below.

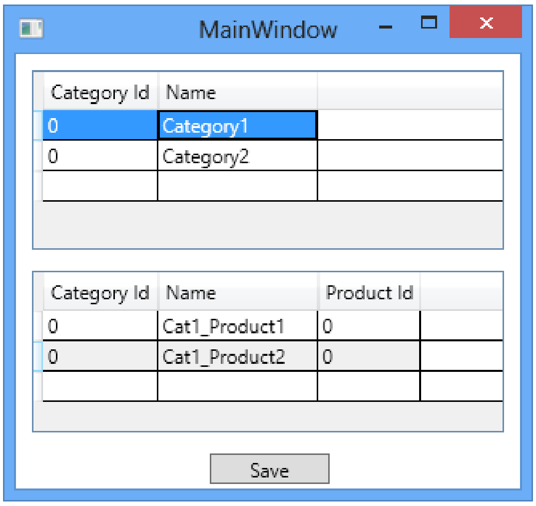
The code declares a long-running instance of **ProductContext**. The **ProductContext** object is used to query and save data to the database. The **Dispose()** on the **ProductContext** instance is then called from the overridden **OnClosing** method. The code comments provide details about what the code does.

```
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();
        }
    }

}
```

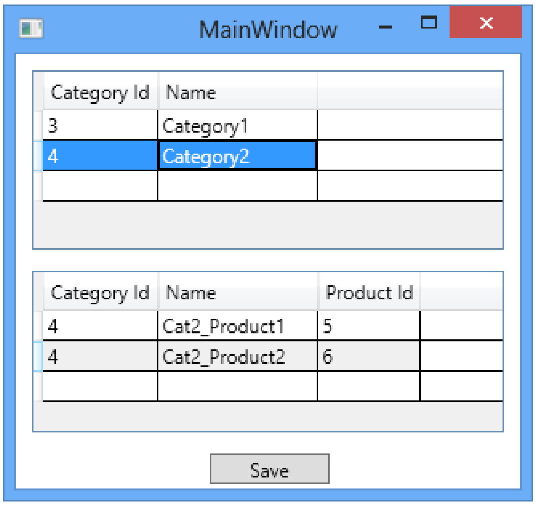
Test the WPF Application

- Compile and run the application. If you used Code First, then you will see that a **WPFwithEFSample.ProductContext** database is created for you.
- Enter a category name in the top grid and product names in the bottom grid *Do not enter anything in ID columns, because the primary key is generated by the database*



- Press the **Save** button to save the data to the database

After the call to DbContext's **SaveChanges()**, the IDs are populated with the database generated values. Because we called **Refresh()** after **SaveChanges()** the **DataGrid** controls are updated with the new values as well.



Additional Resources

To learn more about data binding to collections using WPF, see this topic in the WPF documentation.

Working with disconnected entities

10/25/2018 • 2 minutes to read • Edit Online

In an Entity Framework-based application, a context class is responsible for detecting changes applied to tracked entities. Calling the SaveChanges method persists the changes tracked by the context to the database. When working with n-tier applications, entity objects are usually modified while disconnected from the context, and you must decide how to track changes and report those changes back to the context. This topic discusses different options that are available when using Entity Framework with disconnected entities.

Web service frameworks

Web services technologies typically support patterns that can be used to persist changes on individual disconnected objects. For example, ASP.NET Web API allows you to code controller actions that can include calls to EF to persist changes made to an object on a database. In fact, the Web API tooling in Visual Studio makes it easy to scaffold a Web API controller from your Entity Framework 6 model. For more information, see using Web API with Entity Framework 6.

Historically, there have been several other Web services technologies that offered integration with Entity Framework, like WCF Data Services and RIA Services.

Low-level EF APIs

If you don't want to use an existing n-tier solution, or if you want to customize what happens inside a controller action in a Web API services, Entity Framework provides APIs that allow you to apply changes made on a disconnected tier. For more information, see Add, Attach, and entity state.

Self-Tracking Entities

Tracking changes on arbitrary graphs of entities while disconnected from the EF context is a hard problem. One of the attempts to solve it was the Self-Tracking Entities code generation template. This template generates entity classes that contain logic to track changes made on a disconnected tier as state in the entities themselves. A set of extension methods is also generated to apply those changes to a context.

This template can be used with models created using the EF Designer, but can not be used with Code First models. For more information, see Self-Tracking Entities.

> **IMPORTANT**
> We no longer recommend using the self-tracking-entities template. It will only continue to be available to support existing applications. If your application requires working with disconnected graphs of entities, consider other alternatives such as Trackable Entities, which is a technology similar to Self-Tracking-Entities that is more actively developed by the community, or writing custom code using the low-level change tracking APIs.

Self-tracking entities

9/18/2018 • 5 minutes to read • Edit Online

IMPORTANT

We no longer recommend using the self-tracking-entities template. It will only continue to be available to support existing applications. If your application requires working with disconnected graphs of entities, consider other alternatives such as Trackable Entities, which is a technology similar to Self-Tracking-Entities that is more actively developed by the community, or writing custom code using the low-level change tracking APIs.

In an Entity Framework-based application, a context is responsible for tracking changes in your objects. You then use the SaveChanges method to persist the changes to the database. When working with N-Tier applications, the entity objects are usually disconnected from the context and you must decide how to track changes and report those changes back to the context. Self-Tracking Entities (STEs) can help you track changes in any tier and then replay these changes into a context to be saved.

Use STEs only if the context is not available on a tier where the changes to the object graph are made. If the context is available, there is no need to use STEs because the context will take care of tracking changes.

This template item generates two .tt (text template) files:

- The **<model name>.tt** file generates the entity types and a helper class that contains the change-tracking logic that is used by self-tracking entities and the extension methods that allow setting state on self-tracking entities.
- The **<model name>.Context.tt** file generates a derived context and an extension class that contains **ApplyChanges** methods for the **ObjectContext** and **ObjectSet** classes. These methods examine the change-tracking information that is contained in the graph of self-tracking entities to infer the set of operations that must be performed to save the changes in the database.

Get Started

To get started, visit the Self-Tracking Entities Walkthrough page.

Functional Considerations When Working with Self-Tracking Entities

IMPORTANT

We no longer recommend using the self-tracking-entities template. It will only continue to be available to support existing applications. If your application requires working with disconnected graphs of entities, consider other alternatives such as Trackable Entities, which is a technology similar to Self-Tracking-Entities that is more actively developed by the community, or writing custom code using the low-level change tracking APIs.

Consider the following when working with self-tracking entities:

- Make sure that your client project has a reference to the assembly containing the entity types. If you add only the service reference to the client project, the client project will use the WCF proxy types and not the actual self-tracking entity types. This means that you will not get the automated notification features that manage the tracking of the entities on the client. If you intentionally do not want to include the entity types, you will have to manually set change-tracking information on the client for the changes to be sent back to the service.

- Calls to the service operation should be stateless and create a new instance of object context. We also recommend that you create object context in a **using** block.

- When you send the graph that was modified on the client to the service and then intend to continue working with the same graph on the client, you have to manually iterate through the graph and call the **AcceptChanges** method on each object to reset the change tracker.

  > If objects in your graph contain properties with database-generated values (for example, identity or concurrency values), Entity Framework will replace values of these properties with the database-generated values after the **SaveChanges** method is called. You can implement your service operation to return saved objects or a list of generated property values for the objects back to the client. The client would then need to replace the object instances or object property values with the objects or property values returned from the service operation.

- Merging graphs from multiple service requests may introduce objects with duplicate key values in the resulting graph. Entity Framework does not remove the objects with duplicate keys when you call the **ApplyChanges** method but instead throws an exception. To avoid having graphs with duplicate key values follow one of the patterns described in the following blog: Self-Tracking Entities: ApplyChanges and duplicate entities.

- When you change the relationship between objects by setting the foreign key property, the reference navigation property is set to null and not synchronized to the appropriate principal entity on the client. After the graph is attached to the object context (for example, after you call the **ApplyChanges** method), the foreign key properties and navigation properties are synchronized.

  > Not having a reference navigation property synchronized with the appropriate principal object could be an issue if you have specified cascade delete on the foreign key relationship. If you delete the principal, the delete will not be propagated to the dependent objects. If you have cascade deletes specified, use navigation properties to change relationships instead of setting the foreign key property.

- Self-tracking entities are not enabled to perform lazy loading.

- Binary serialization and serialization to ASP.NET state management objects is not supported by self-tracking entities. However, you can customize the template to add the binary serialization support. For more information, see Using Binary Serialization and ViewState with Self-Tracking Entities.

Security Considerations

The following security considerations should be taken into account when working with self-tracking entities:

- A service should not trust requests to retrieve or update data from a non-trusted client or through a non-trusted channel. A client must be authenticated: a secure channel or message envelope should be used. Clients' requests to update or retrieve data must be validated to ensure they conform to expected and legitimate changes for the given scenario.
- Avoid using sensitive information as entity keys (for example, social security numbers). This mitigates the possibility of inadvertently serializing sensitive information in the self-tracking entity graphs to a client that is not fully trusted. With independent associations, the original key of an entity that is related to the one that is being serialized might be sent to the client as well.
- To avoid propagating exception messages that contain sensitive data to the client tier, calls to **ApplyChanges** and **SaveChanges** on the server tier should be wrapped in exception-handling code.

Self-Tracking Entities Walkthrough

9/13/2018 • 12 minutes to read • Edit Online

IMPORTANT

We no longer recommend using the self-tracking-entities template. It will only continue to be available to support existing applications. If your application requires working with disconnected graphs of entities, consider other alternatives such as Trackable Entities, which is a technology similar to Self-Tracking-Entities that is more actively developed by the community, or writing custom code using the low-level change tracking APIs.

This walkthrough demonstrates the scenario in which a Windows Communication Foundation (WCF) service exposes an operation that returns an entity graph. Next, a client application manipulates that graph and submits the modifications to a service operation that validates and saves the updates to a database using Entity Framework.

Before completing this walkthrough make sure you read the Self-Tracking Entities page.

This walkthrough completes the following actions:

- Creates a database to access.
- Creates a class library that contains the model.
- Swaps to the Self-Tracking Entity Generator template.
- Moves the entity classes to a separate project.
- Creates a WCF service that exposes operations to query and save entities.
- Creates client applications (Console and WPF) that consume the service.

We'll use Database First in this walkthrough but the same techniques apply equally to Model First.

Pre-Requisites

To complete this walkthrough you will need a recent version of Visual Studio.

Create a Database

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.
- If you are using Visual Studio 2010 you'll be creating a SQL Express database.

Let's go ahead and generate the database.

- Open Visual Studio
- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection...**
- If you haven't connected to a database from Server Explorer before you'll need to select **Microsoft SQL Server** as the data source
- Connect to either LocalDB or SQL Express, depending on which one you have installed
- Enter **STESample** as the database name
- Select **OK** and you will be asked if you want to create a new database, select **Yes**

- The new database will now appear in Server Explorer
- If you are using Visual Studio 2012
  - Right-click on the database in Server Explorer and select **New Query**
  - Copy the following SQL into the new query, then right-click on the query and select **Execute**
- If you are using Visual Studio 2010
  - Select **Data -> Transact SQL Editor -> New Query Connection...**
  - Enter **.\SQLEXPRESS** as the server name and click **OK**
  - Select the **STESample** database from the drop down at the top of the query editor
  - Copy the following SQL into the new query, then right-click on the query and select **Execute 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)
```

Create the Model

First up, we need a project to put the model in.

- **File -> New -> Project...**
- Select **Visual C#** from the left pane and then **Class Library**
- Enter **STESample** as the name and click **OK**

Now we'll create a simple model in the EF Designer to access our database:

- **Project -> Add New Item...**
- Select **Data** from the left pane and then **ADO.NET Entity Data Model**
- Enter **BloggingModel** as the name and click **OK**
- Select **Generate from database** and click **Next**
- Enter the connection information for the database that you created in the previous section
- Enter **BloggingContext** as the name for the connection string and click **Next**
- Check the box next to **Tables** and click **Finish**

Swap to STE Code Generation

Now we need to disable the default code generation and swap to Self-Tracking Entities.

If you are using Visual Studio 2012

- Expand **BloggingModel.edmx** in **Solution Explorer** and delete the **BloggingModel.tt** and **BloggingModel.Context.tt** *This will disable the default code generation*
- Right-click an empty area on the EF Designer surface and select **Add Code Generation Item...**
- Select **Online** from the left pane and search for **STE Generator**
- Select the **STE Generator for C#** template, enter **STETemplate** as the name and click **Add**
- The **STETemplate.tt** and **STETemplate.Context.tt** files are added nested under the BloggingModel.edmx file

If you are using Visual Studio 2010

- Right-click an empty area on the EF Designer surface and select **Add Code Generation Item...**
- Select **Code** from the left pane and then **ADO.NET Self-Tracking Entity Generator**
- Enter **STETemplate** as the name and click **Add**
- The **STETemplate.tt** and **STETemplate.Context.tt** files are added directly to your project

Move Entity Types into Separate Project

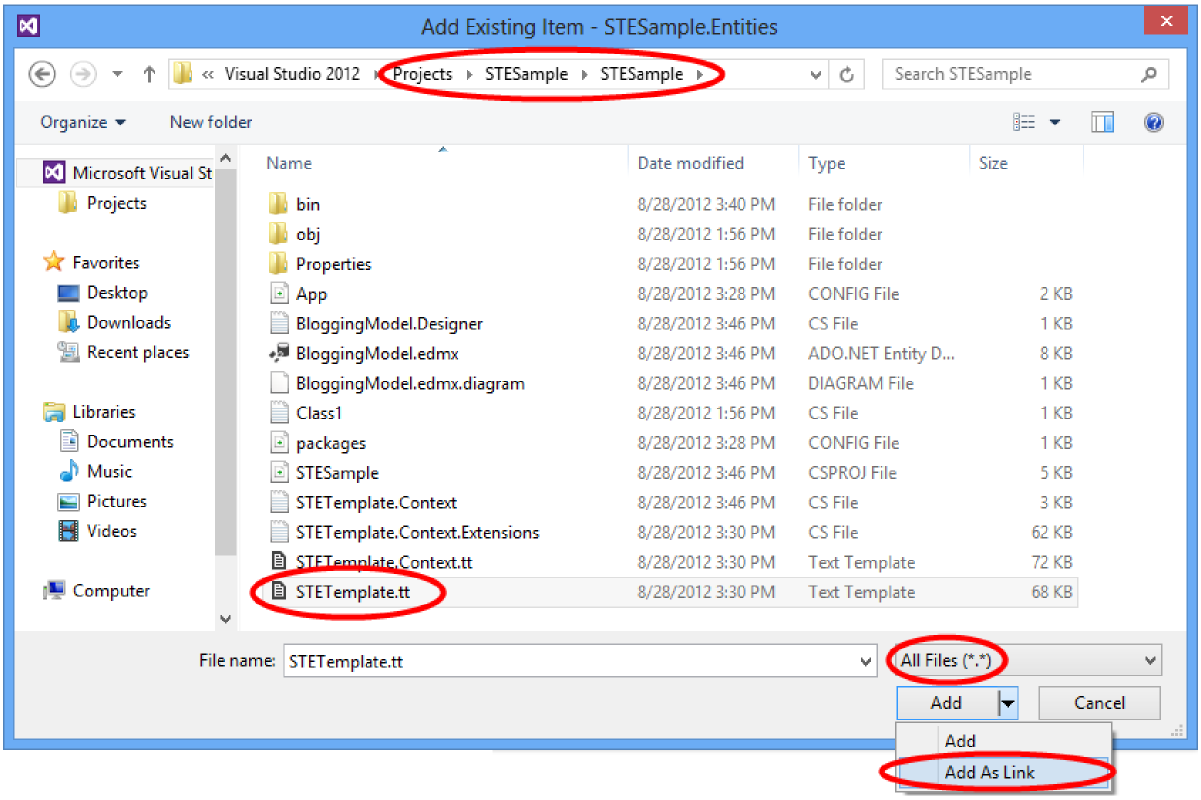
To use Self-Tracking Entities our client application needs access to the entity classes generated from our model. Because we don't want to expose the whole model to the client application we're going to move the entity classes into a separate project.

The first step is to stop generating entity classes in the existing project:

- Right-click on **STETemplate.tt** in **Solution Explorer** and select **Properties**
- In the **Properties** window clear **TextTemplatingFileGenerator** from the **CustomTool** property
- Expand **STETemplate.tt** in **Solution Explorer** and delete all files nested under it

Next, we are going to add a new project and generate the entity classes in it

- **File -> Add -> Project...**
- Select **Visual C#** from the left pane and then **Class Library**
- Enter **STESample.Entities** as the name and click **OK**
- **Project -> Add Existing Item...**
- Navigate to the **STESample** project folder
- Select to view **All Files (*.*)**
- Select the **STETemplate.tt** file
- Click on the drop down arrow next to the **Add** button and select **Add As Link**



We're also going to make sure the entity classes get generated in the same namespace as the context. This just reduces the number of using statements we need to add throughout our application.

- Right-click on the linked **STETemplate.tt** in **Solution Explorer** and select **Properties**
- In the **Properties** window set **Custom Tool Namespace** to **STESample**

The code generated by the STE template will need a reference to **System.Runtime.Serialization** in order to compile. This library is needed for the WCF **DataContract** and **DataMember** attributes that are used on the serializable entity types.

- Right click on the **STESample.Entities** project in **Solution Explorer** and select **Add Reference...**
  - In Visual Studio 2012 - check the box next to **System.Runtime.Serialization** and click **OK**
  - In Visual Studio 2010 - select **System.Runtime.Serialization** and click **OK**

Finally, the project with our context in it will need a reference to the entity types.

- Right click on the **STESample** project in **Solution Explorer** and select **Add Reference...**
  - In Visual Studio 2012 - select **Solution** from the left pane, check the box next to **STESample.Entities** and click **OK**
  - In Visual Studio 2010 - select the **Projects** tab, select **STESample.Entities** and click **OK**

> **NOTE**
>
> Another option for moving the entity types to a separate project is to move the template file, rather than linking it from its default location. If you do this, you will need to update the **inputFile** variable in the template to provide the relative path to the edmx file (in this example that would be **..\BloggingModel.edmx**).

Create a WCF Service

Now it's time to add a WCF Service to expose our data, we'll start by creating the project.

- **File -> Add -> Project...**
- Select **Visual C#** from the left pane and then **WCF Service Application**

- Enter **STESample.Service** as the name and click **OK**
- Add a reference to the **System.Data.Entity** assembly
- Add a reference to the **STESample** and **STESample.Entities** projects

We need to copy the EF connection string to this project so that it is found at runtime.

- Open the **App.Config** file for the **STESample **project and copy the **connectionStrings** element
- Paste the **connectionStrings** element as a child element of the **configuration** element of the **Web.Config** file in the **STESample.Service** project

Now it's time to implement the actual service.

- Open **IService1.cs** and replace the contents with the following code

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

namespace STESample.Service
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        List<Blog> GetBlogs();

        [OperationContract]
        void UpdateBlog(Blog blog);
    }
}
```

- Open **Service1.svc** and replace the contents with the following code

```
    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.");
                    }
                }
            }
        }
    }
```

Consume the Service from a Console Application

Let's create a console application that uses our service.

- **File -> New -> Project...**
- Select **Visual C#** from the left pane and then **Console Application**
- Enter **STESample.ConsoleTest** as the name and click **OK**
- Add a reference to the **STESample.Entities** project

We need a service reference to our WCF service

- Right-click the **STESample.ConsoleTest** project in **Solution Explorer** and select **Add Service Reference...**
- Click **Discover**

- Enter **BloggingService** as the namespace and click **OK**

Now we can write some code to consume the service.

- Open **Program.cs** and replace the contents with the following code.

```
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"
```

```
                    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);
            }
        }
    }
}
```

You can now run the application to see it in action.

- Right-click the **STESample.ConsoleTest** project in **Solution Explorer** and select **Debug -> Start new instance**

You'll see the following output when the application executes.

```
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...
```

Consume the Service from a WPF Application

Let's create a WPF application that uses our service.

- **File -> New -> Project...**
- Select **Visual C#** from the left pane and then **WPF Application**
- Enter **STESample.WPFTest** as the name and click **OK**
- Add a reference to the **STESample.Entities** project

We need a service reference to our WCF service

- Right-click the **STESample.WPFTest** project in **Solution Explorer** and select **Add Service Reference...**
- Click **Discover**
- Enter **BloggingService** as the namespace and click **OK**

Now we can write some code to consume the service.

- Open **MainWindow.xaml** and replace the contents with the following code.

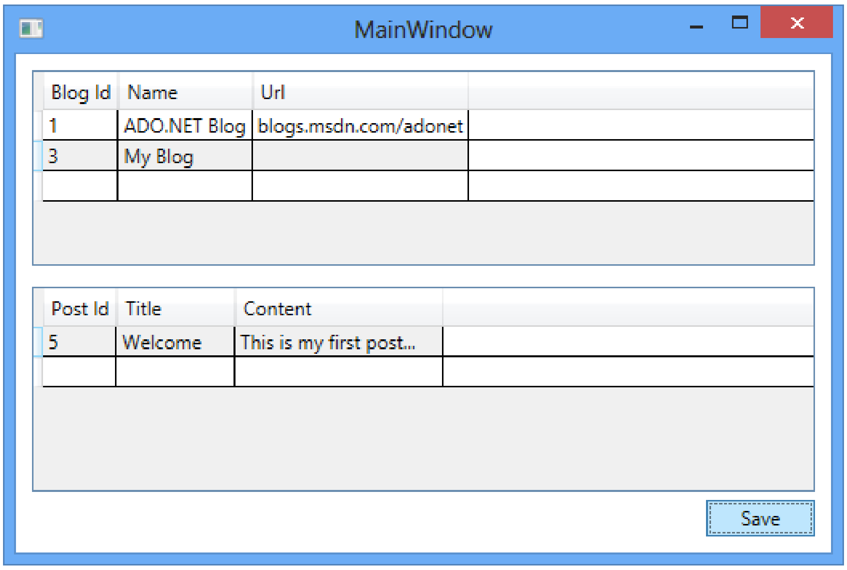
```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:STESample="clr-namespace:STESample;assembly=STESample.Entities"
    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>
```

- Open the code behind for MainWindow (**MainWindow.xaml.cs**) and replace the contents with the following code

```
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();
            }
        }
    }
}
```

You can now run the application to see it in action.

- Right-click the **STESample.WPFTest** project in **Solution Explorer** and select **Debug -> Start new instance**
- You can manipulate the data using the screen and save it via the service using the **Save** button


MainWindow
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...
Save

Logging and intercepting database operations

9/13/2018 • 13 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Starting with Entity Framework 6, anytime Entity Framework sends a command to the database this command can be intercepted by application code. This is most commonly used for logging SQL, but can also be used to modify or abort the command.

Specifically, EF includes:

- A Log property for the context similar to DataContext.Log in LINQ to SQL
- A mechanism to customize the content and formatting of the output sent to the log
- Low-level building blocks for interception giving greater control/flexibility

Context Log property

The DbContext.Database.Log property can be set to a delegate for any method that takes a string. Most commonly it is used with any TextWriter by setting it to the “Write” method of that TextWriter. All SQL generated by the current context will be logged to that writer. For example, the following code will log SQL to the console:

```
using (var context = new BlogContext())
{
    context.Database.Log = Console.Write;

    // Your code here...
}
```

Notice that context.Database.Log is set to Console.Write. This is all that is needed to log SQL to the console.

Let’s add some simple query/insert/update code so that we can see some output:

```
using (var context = new BlogContext())
{
    context.Database.Log = Console.Write;

    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();
}
```

This will generate the following output:

```
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
```

(Note that this is the output assuming any database initialization has already happened. If database initialization had not already happened then there would be a lot more output showing all the work Migrations does under the covers to check for or create a new database.)

What gets logged?

When the Log property is set all of the following will be logged:

- SQL for all different kinds of commands. For example:
  - Queries, including normal LINQ queries, eSQL queries, and raw queries from methods such as SqlQuery
  - Inserts, updates, and deletes generated as part of SaveChanges
  - Relationship loading queries such as those generated by lazy loading
- Parameters
- Whether or not the command is being executed asynchronously
- A timestamp indicating when the command started executing
- Whether or not the command completed successfully, failed by throwing an exception, or, for async, was canceled
- Some indication of the result value
- The approximate amount of time it took to execute the command. Note that this is the time from sending the command to getting the result object back. It does not include time to read the results.

Looking at the example output above, each of the four commands logged are:

- The query resulting from the call to context.Blogs.First
  - Notice that the ToString method of getting the SQL would not have worked for this query since “First” does not provide an IQueryable on which ToString could be called
- The query resulting from the lazy-loading of blog.Posts
  - Notice the parameter details for the key value for which lazy loading is happening
  - Only properties of the parameter that are set to non-default values are logged. For example, the Size property is only shown if it is non-zero.
- Two commands resulting from SaveChangesAsync; one for the update to change a post title, the other for an insert to add a new post
  - Notice the parameter details for the FK and Title properties
  - Notice that these commands are being executed asynchronously

Logging to different places

As shown above logging to the console is super easy. It’s also easy to log to memory, file, etc. by using different kinds of TextWriter.

If you are familiar with LINQ to SQL you might notice that in LINQ to SQL the Log property is set to the actual TextWriter object (for example, Console.Out) while in EF the Log property is set to a method that accepts a string (for example, Console.Write or Console.Out.Write). The reason for this is to decouple EF from TextWriter by accepting any delegate that can act as a sink for strings. For example, imagine that you already have some logging framework and it defines a logging method like so:

```
public class MyLogger
{
    public void Log(string component, string message)
    {
        Console.WriteLine("Component: {0} Message: {1} ", component, message);
    }
}
```

This could be hooked up to the EF Log property like this:

```
var logger = new MyLogger();
context.Database.Log = s => logger.Log("EFApp", s);
```

Result logging

The default logger logs command text (SQL), parameters, and the “Executing” line with a timestamp before the command is sent to the database. A “completed” line containing elapsed time is logged following execution of the command.

Note that for async commands the “completed” line is not logged until the async task actually completes, fails, or is canceled.

The “completed” line contains different information depending on the type of command and whether or not execution was successful.

Successful execution

For commands that complete successfully the output is “Completed in x ms with result: ” followed by some indication of what the result was. For commands that return a data reader the result indication is the type of DbDataReader returned. For commands that return an integer value such as the update command shown above the result shown is that integer.

Failed execution

For commands that fail by throwing an exception, the output contains the message from the exception. For example, using SqlQuery to query against a table that does exist will result in log output something like this:

```
SELECT * from ThisTableIsMissing
-- Executing at 5/13/2013 10:19:05 AM
-- Failed in 1 ms with error: Invalid object name 'ThisTableIsMissing'.
```

Canceled execution

For async commands where the task is canceled the result could be failure with an exception, since this is what the underlying ADO.NET provider often does when an attempt is made to cancel. If this doesn't happen and the task is canceled cleanly then the output will look something like this:

```
update Blogs set Title = 'No' where Id = -1
-- Executing asynchronously at 5/13/2013 10:21:10 AM
-- Canceled in 1 ms
```

Changing log content and formatting

Under the covers the Database.Log property makes use of a DatabaseLogFormatter object. This object effectively binds an IDbCommandInterceptor implementation (see below) to a delegate that accepts strings and a DbContext. This means that methods on DatabaseLogFormatter are called before and after the execution of commands by EF. These DatabaseLogFormatter methods gather and format log output and send it to the delegate.

Customizing DatabaseLogFormatter

Changing what is logged and how it is formatted can be achieved by creating a new class that derives from DatabaseLogFormatter and overrides methods as appropriate. The most common methods to override are:

- LogCommand – Override this to change how commands are logged before they are executed. By default LogCommand calls LogParameter for each parameter; you may choose to do the same in your override or handle parameters differently instead.
- LogResult – Override this to change how the outcome from executing a command is logged.
- LogParameter – Override this to change the formatting and content of parameter logging.

For example, suppose we wanted to log just a single line before each command is sent to the database. This can be done with two overrides:

- Override LogCommand to format and write the single line of SQL
- Override LogResult to do nothing.

The code would look something like this:

```
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)
    {
    }
}
```

To log output simply call the Write method which will send output to the configured write delegate.

(Note that this code does simplistic removal of line breaks just as an example. It will likely not work well for viewing complex SQL.)

Setting the DatabaseLogFormatter

Once a new DatabaseLogFormatter class has been created it needs to be registered with EF. This is done using code-based configuration. In a nutshell this means creating a new class that derives from DbConfiguration in the same assembly as your DbContext class and then calling SetDatabaseLogFormatter in the constructor of this new class. For example:

```
public class MyDbConfiguration : DbConfiguration
{
    public MyDbConfiguration()
    {
        SetDatabaseLogFormatter(
            (context, writeAction) => new OneLineFormatter(context, writeAction));
    }
}
```

Using the new DatabaseLogFormatter

This new DatabaseLogFormatter will now be used anytime Database.Log is set. So, running the code from part 1 will now result in the following output:

```
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()'
```

Interception building blocks

So far we have looked at how to use DbContext.Database.Log to log the SQL generated by EF. But this code is actually a relatively thin façade over some low-level building blocks for more general interception.

Interception interfaces

The interception code is built around the concept of interception interfaces. These interfaces inherit from IDbInterceptor and define methods that are called when EF performs some action. The intent is to have one interface per type of object being intercepted. For example, the IDbCommandInterceptor interface defines methods that are called before EF makes a call to ExecuteNonQuery, ExecuteScalar, ExecuteReader, and related methods. Likewise, the interface defines methods that are called when each of these operations completes. The DatabaseLogFormatter class that we looked at above implements this interface to log commands.

The interception context

Looking at the methods defined on any of the interceptor interfaces it is apparent that every call is given an object of type DbInterceptionContext or some type derived from this such as DbCommandInterceptionContext<>. This object contains contextual information about the action that EF is taking. For example, if the action is being taken on behalf of a DbContext, then the DbContext is included in the DbInterceptionContext. Similarly, for commands that are being executed asynchronously, the IsAsync flag is set on DbCommandInterceptionContext.

Result handling

The DbCommandInterceptionContext<> class contains a properties called Result, OriginalResult, Exception, and OriginalException. These properties are set to null/zero for calls to the interception methods that are called before the operation is executed — that is, for the ...Executing methods. If the operation is executed and succeeds, then Result and OriginalResult are set to the result of the operation. These values can then be observed in the interception methods that are called after the operation has executed — that is, on the ...Executed methods. Likewise, if the operation throws, then the Exception and OriginalException properties will be set.

Suppressing execution

If an interceptor sets the Result property before the command has executed (in one of the ...Executing methods) then EF will not attempt to actually execute the command, but will instead just use the result set. In other words, the interceptor can suppress execution of the command but have EF continue as if the command had been executed.

An example of how this might be used is the command batching that has traditionally been done with a wrapping provider. The interceptor would store the command for later execution as a batch but would “pretend” to EF that the command had executed as normal. Note that it requires more than this to implement batching, but this is an example of how changing the interception result might be used.

Execution can also be suppressed by setting the Exception property in one of the ...Executing methods. This causes EF to continue as if execution of the operation had failed by throwing the given exception. This may, of course, cause the application to crash, but it may also be a transient exception or some other exception that is handled by EF. For example, this could be used in test environments to test the behavior of an application when command execution fails.

Changing the result after execution

If an interceptor sets the Result property after the command has executed (in one of the ...Executed methods) then EF will use the changed result instead of the result that was actually returned from the operation. Similarly, if an interceptor sets the Exception property after the command has executed, then EF will throw the set exception as if the operation had thrown the exception.

An interceptor can also set the Exception property to null to indicate that no exception should be thrown. This can be useful if execution of the operation failed but the interceptor wishes EF to continue as if the operation had succeeded. This usually also involves setting the Result so that EF has some result value to work with as it continues.

OriginalResult and OriginalException

After EF has executed an operation it will set either the Result and OriginalResult properties if execution did not fail or the Exception and OriginalException properties if execution failed with an exception.

The OriginalResult and OriginalException properties are read-only and are only set by EF after actually executing an operation. These properties cannot be set by interceptors. This means that any interceptor can distinguish between an exception or result that has been set by some other interceptor as opposed to the real exception or result that occurred when the operation was executed.

Registering interceptors

Once a class that implements one or more of the interception interfaces has been created it can be registered with EF using the DbInterception class. For example:

```
DbInterception.Add(new NLogCommandInterceptor());
```

Interceptors can also be registered at the app-domain level using the DbConfiguration code-based configuration mechanism.

Example: Logging to NLog

Let's put all this together into an example that using IDbCommandInterceptor and NLog to:

- Log a warning for any command that is executed non-asynchronously
- Log an error for any command that throws when executed

Here's the class that does the logging, which should be registered as shown above:

```
public class NLogCommandInterceptor : IDbCommandInterceptor
{
    private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

    public void NonQueryExecuting(
        DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void NonQueryExecuted(
        DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ReaderExecuting(
        DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ReaderExecuted(
        DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ScalarExecuting(
        DbCommand command, DbCommandInterceptionContext<object> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ScalarExecuted(
        DbCommand command, DbCommandInterceptionContext<object> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    private void LogIfNonAsync<TResult>(
        DbCommand command, DbCommandInterceptionContext<TResult> interceptionContext)
    {
        if (!interceptionContext.IsAsync)
        {
            Logger.Warn("Non-async command used: {0}", command.CommandText);
        }
    }

    private void LogIfError<TResult>(
        DbCommand command, DbCommandInterceptionContext<TResult> interceptionContext)
    {
        if (interceptionContext.Exception != null)
        {
            Logger.Error("Command {0} failed with exception {1}",
                command.CommandText, interceptionContext.Exception);
        }
    }
}
```

Notice how this code uses the interception context to discover when a command is being executed non-asynchronously and to discover when there was an error executing a command.

Performance considerations for EF 4, 5, and 6

10/25/2018 • 69 minutes to read • Edit Online

By David Obando, Eric Dettinger and others

Published: April 2012

Last updated: May 2014

1. Introduction

Object-Relational Mapping frameworks are a convenient way to provide an abstraction for data access in an object-oriented application. For .NET applications, Microsoft's recommended O/RM is Entity Framework. With any abstraction though, performance can become a concern.

This whitepaper was written to show the performance considerations when developing applications using Entity Framework, to give developers an idea of the Entity Framework internal algorithms that can affect performance, and to provide tips for investigation and improving performance in their applications that use Entity Framework. There are a number of good topics on performance already available on the web, and we've also tried pointing to these resources where possible.

Performance is a tricky topic. This whitepaper is intended as a resource to help you make performance related decisions for your applications that use Entity Framework. We have included some test metrics to demonstrate performance, but these metrics aren't intended as absolute indicators of the performance you will see in your application.

For practical purposes, this document assumes Entity Framework 4 is run under .NET 4.0 and Entity Framework 5 and 6 are run under .NET 4.5. Many of the performance improvements made for Entity Framework 5 reside within the core components that ship with .NET 4.5.

Entity Framework 6 is an out of band release and does not depend on the Entity Framework components that ship with .NET. Entity Framework 6 work on both .NET 4.0 and .NET 4.5, and can offer a big performance benefit to those who haven’t upgraded from .NET 4.0 but want the latest Entity Framework bits in their application. When this document mentions Entity Framework 6, it refers to the latest version available at the time of this writing: version 6.1.0.

2. Cold vs. Warm Query Execution

The very first time any query is made against a given model, the Entity Framework does a lot of work behind the scenes to load and validate the model. We frequently refer to this first query as a "cold" query. Further queries against an already loaded model are known as "warm" queries, and are much faster.

Let’s take a high-level view of where time is spent when executing a query using Entity Framework, and see where things are improving in Entity Framework 6.

First Query Execution – cold query

CODE USER WRITES	ACTION	EF4 PERFORMANCE IMPACT	EF5 PERFORMANCE IMPACT	EF6 PERFORMANCE IMPACT
`using(var db = new MyContext())` `{`	Context creation	Medium	Medium	Low
`var q1 =` `from c in db.Customers` `where c.Id == id1` `select c;`	Query expression creation	Low	Low	Low
`var c1 = q1.First();`	LINQ query execution	- Metadata loading: High but cached - View generation: Potentially very high but cached - Parameter evaluation: Medium - Query translation: Medium - Materializer generation: Medium but cached - Database query execution: Potentially high + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium - Identity lookup: Medium	- Metadata loading: High but cached - View generation: Potentially very high but cached - Parameter evaluation: Low - Query translation: Medium but cached - Materializer generation: Medium but cached - Database query execution: Potentially high (Better queries in some situations) + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium - Identity lookup: Medium	- Metadata loading: High but cached - View generation: Medium but cached - Parameter evaluation: Low - Query translation: Medium but cached - Materializer generation: Medium but cached - Database query execution: Potentially high (Better queries in some situations) + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium (Faster than EF5) - Identity lookup: Medium
`}`	Connection.Close	Low	Low	Low

Second Query Execution – warm query

CODE USER WRITES	ACTION	EF4 PERFORMANCE IMPACT	EF5 PERFORMANCE IMPACT	EF6 PERFORMANCE IMPACT
`using(var db = new MyContext())` `{`	Context creation	Medium	Medium	Low
`var q1 =` `from c in db.Customers` `where c.Id == id1` `select c;`	Query expression creation	Low	Low	Low

CODE USER WRITES	ACTION	EF4 PERFORMANCE IMPACT	EF5 PERFORMANCE IMPACT	EF6 PERFORMANCE IMPACT
`var c1 = q1.First();`	LINQ query execution	- Metadata ~~loading~~ lookup: ~~High but cached~~ Low - View ~~generation~~ lookup: ~~Potentially very high but cached~~ Low - Parameter evaluation: Medium - Query ~~translation~~ lookup: Medium - Materializer ~~generation~~ lookup: ~~Medium but cached~~ Low - Database query execution: Potentially high + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium - Identity lookup: Medium	- Metadata ~~loading~~ lookup: ~~High but cached~~ Low - View ~~generation~~ lookup: ~~Potentially very high but cached~~ Low - Parameter evaluation: Low - Query ~~translation~~ lookup: ~~Medium but cached~~ Low - Materializer ~~generation~~ lookup: ~~Medium but cached~~ Low - Database query execution: Potentially high (Better queries in some situations) + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium - Identity lookup: Medium	- Metadata ~~loading~~ lookup: ~~High but cached~~ Low - View ~~generation~~ lookup: ~~Medium but cached~~ Low - Parameter evaluation: Low - Query ~~translation~~ lookup: ~~Medium but cached~~ Low - Materializer ~~generation~~ lookup: ~~Medium but cached~~ Low - Database query execution: Potentially high (Better queries in some situations) + Connection.Open + Command.ExecuteReader + DataReader.Read Object materialization: Medium (Faster than EF5) - Identity lookup: Medium
`}`	Connection.Close	Low	Low	Low

There are several ways to reduce the performance cost of both cold and warm queries, and we'll take a look at these in the following section. Specifically, we'll look at reducing the cost of model loading in cold queries by using pre-generated views, which should help alleviate performance pains experienced during view generation. For warm queries, we'll cover query plan caching, no tracking queries, and different query execution options.

2.1 What is View Generation?

In order to understand what view generation is, we must first understand what “Mapping Views” are. Mapping Views are executable representations of the transformations specified in the mapping for each entity set and association. Internally, these mapping views take the shape of CQTs (canonical query trees). There are two types of mapping views:

- Query views: these represent the transformation necessary to go from the database schema to the conceptual model.
- Update views: these represent the transformation necessary to go from the conceptual model to the database schema.

Keep in mind that the conceptual model might differ from the database schema in various ways. For example, one single table might be used to store the data for two different entity types. Inheritance and non-trivial mappings play a role in the complexity of the mapping views.

The process of computing these views based on the specification of the mapping is what we call view generation. View generation can either take place dynamically when a model is loaded, or at build time, by using "pre-generated views"; the latter are serialized in the form of Entity SQL statements to a C# or VB file.

When views are generated, they are also validated. From a performance standpoint, the vast majority of the cost of view generation is actually the validation of the views which ensures that the connections between the entities make sense and have the correct cardinality for all the supported operations.

When a query over an entity set is executed, the query is combined with the corresponding query view, and the result of this composition is run through the plan compiler to create the representation of the query that the backing store can understand. For SQL Server, the final result of this compilation will be a T-SQL SELECT statement. The first time an update over an entity set is performed, the update view is run through a similar process to transform it into DML statements for the target database.

2.2 Factors that affect View Generation performance

The performance of view generation step not only depends on the size of your model but also on how interconnected the model is. If two Entities are connected via an inheritance chain or an Association, they are said to be connected. Similarly if two tables are connected via a foreign key, they are connected. As the number of connected Entities and tables in your schemas increase, the view generation cost increases.

The algorithm that we use to generate and validate views is exponential in the worst case, though we do use some optimizations to improve this. The biggest factors that seem to negatively affect performance are:

- Model size, referring to the number of entities and the amount of associations between these entities.
- Model complexity, specifically inheritance involving a large number of types.
- Using Independent Associations, instead of Foreign Key Associations.

For small, simple models the cost may be small enough to not bother using pre-generated views. As model size and complexity increase, there are several options available to reduce the cost of view generation and validation.

2.3 Using Pre-Generated Views to decrease model load time

For detailed information on how to use pre-generated views on Entity Framework 6 visit Pre-Generated Mapping Views

2.3.1 Pre-Generated views using the Entity Framework Power Tools Community Edition

You can use the Entity Framework 6 Power Tools Community Edition to generate views of EDMX and Code First models by right-clicking the model class file and using the Entity Framework menu to select “Generate Views”. The Entity Framework Power Tools Community Edition work only on DbContext-derived contexts.

2.3.2 How to use Pre-generated views with a model created by EDMGen

EDMGen is a utility that ships with .NET and works with Entity Framework 4 and 5, but not with Entity Framework 6. EDMGen allows you to generate a model file, the object layer and the views from the command line. One of the outputs will be a Views file in your language of choice, VB or C#. This is a code file containing Entity SQL snippets for each entity set. To enable pre-generated views, you simply include the file in your project.

If you manually make edits to the schema files for the model, you will need to re-generate the views file. You can do this by running EDMGen with the **/mode:ViewGeneration** flag.

2.3.3 How to use Pre-Generated Views with an EDMX file

You can also use EDMGen to generate views for an EDMX file - the previously referenced MSDN topic describes how to add a pre-build event to do this - but this is complicated and there are some cases where it isn't possible. It's generally easier to use a T4 template to generate the views when your model is in an edmx file.

The ADO.NET team blog has a post that describes how to use a T4 template for view generation (<http://blogs.msdn.com/b/adonet/archive/2008/06/20/how-to-use-a-t4-template-for-view-generation.aspx>). This post includes a template that can be downloaded and added to your project. The template was written for the first version of Entity Framework, so they aren’t guaranteed to work with the latest versions of Entity Framework. However, you can download a more up-to-date set of view generation templates for Entity Framework 4 and 5from the Visual Studio Gallery:

- VB.NET: <http://visualstudiogallery.msdn.microsoft.com/118b44f2-1b91-4de2-a584-7a680418941d>

- C#: <http://visualstudiogallery.msdn.microsoft.com/ae7730ce-ddab-470f-8456-1b313cd2c44d>

If you’re using Entity Framework 6 you can get the view generation T4 templates from the Visual Studio Gallery at <http://visualstudiogallery.msdn.microsoft.com/18a7db90-6705-4d19-9dd1-0a6c23d0751f>.

2.4 Reducing the cost of view generation

Using pre-generated views moves the cost of view generation from model loading (run time) to design time. While this improves startup performance at runtime, you will still experience the pain of view generation while you are developing. There are several additional tricks that can help reduce the cost of view generation, both at compile time and run time.

2.4.1 Using Foreign Key Associations to reduce view generation cost

We have seen a number of cases where switching the associations in the model from Independent Associations to Foreign Key Associations dramatically improved the time spent in view generation.

To demonstrate this improvement, we generated two versions of the Navision model by using EDMGen. *Note: seeappendix Cfor a description of the Navision model.* The Navision model is interesting for this exercise due to its very large amount of entities and relationships between them.

One version of this very large model was generated with Foreign Keys Associations and the other was generated with Independent Associations. We then timed how long it took to generate the views for each model. Entity Framework5 test used the GenerateViews() method from class EntityViewGenerator to generate the views, while the Entity Framework 6 test used the GenerateViews() method from class StorageMappingItemCollection. This due to code restructuring that occurred in the Entity Framework 6 codebase.

Using Entity Framework 5, view generation for the model with Foreign Keys took 65 minutes in a lab machine. It's unknown how long it would have taken to generate the views for the model that used independent associations. We left the test running for over a month before the machine was rebooted in our lab to install monthly updates.

Using Entity Framework 6, view generation for the model with Foreign Keys took 28 seconds in the same lab machine. View generation for the model that uses Independent Associations took 58 seconds. The improvements done to Entity Framework 6 on its view generation code mean that many projects won’t need pre-generated views to obtain faster startup times.

It’s important to remark that pre-generating views in Entity Framework 4 and 5 can be done with EDMGen or the Entity Framework Power Tools. For Entity Framework 6 view generation can be done via the Entity Framework Power Tools or programmatically as described in Pre-Generated Mapping Views.

2.4.1.1 How to use Foreign Keys instead of Independent Associations

When using EDMGen or the Entity Designer in Visual Studio, you get FKs by default, and it only takes a single checkbox or command line flag to switch between FKs and IAs.

If you have a large Code First model, using Independent Associations will have the same effect on view generation. You can avoid this impact by including Foreign Key properties on the classes for your dependent objects, though some developers will consider this to be polluting their object model. You can find more information on this subject in <http://blog.oneunicorn.com/2011/12/11/whats-the-deal-with-mapping-foreign-keys-using-the-entity-framework/>.

WHEN USING	DO THIS
Entity Designer	After adding an association between two entities, make sure you have a referential constraint. Referential constraints tell Entity Framework to use Foreign Keys instead of Independent Associations. For additional details visit <http://blogs.msdn.com/b/efdesign/archive/2009/03/16/foreign-keys-in-the-entity-framework.aspx>.

WHEN USING	DO THIS
EDMGen	When using EDMGen to generate your files from the database, your Foreign Keys will be respected and added to the model as such. For more information on the different options exposed by EDMGen visit http://msdn.microsoft.com/library/bb387165.aspx.
Code First	See the "Relationship Convention" section of the Code First Conventions topic for information on how to include foreign key properties on dependent objects when using Code First.

2.4.2 Moving your model to a separate assembly

When your model is included directly in your application's project and you generate views through a pre-build event or a T4 template, view generation and validation will take place whenever the project is rebuilt, even if the model wasn't changed. If you move the model to a separate assembly and reference it from your application's project, you can make other changes to your application without needing to rebuild the project containing the model.

Note: when moving your model to separate assemblies remember to copy the connection strings for the model into the application configuration file of the client project.

2.4.3 Disable validation of an edmx-based model

EDMX models are validated at compile time, even if the model is unchanged. If your model has already been validated, you can suppress validation at compile time by setting the "Validate on Build" property to false in the properties window. When you change your mapping or model, you can temporarily re-enable validation to verify your changes.

Note that performance improvements were made to the Entity Framework Designer for Entity Framework 6, and the cost of the "Validate on Build" is much lower than in previous versions of the designer.

3 Caching in the Entity Framework

Entity Framework has the following forms of caching built-in:

1. Object caching – the ObjectStateManager built into an ObjectContext instance keeps track in memory of the objects that have been retrieved using that instance. This is also known as first-level cache.
2. Query Plan Caching - reusing the generated store command when a query is executed more than once.
3. Metadata caching - sharing the metadata for a model across different connections to the same model.

Besides the caches that EF provides out of the box, a special kind of ADO.NET data provider known as a wrapping provider can also be used to extend Entity Framework with a cache for the results retrieved from the database, also known as second-level caching.

3.1 Object Caching

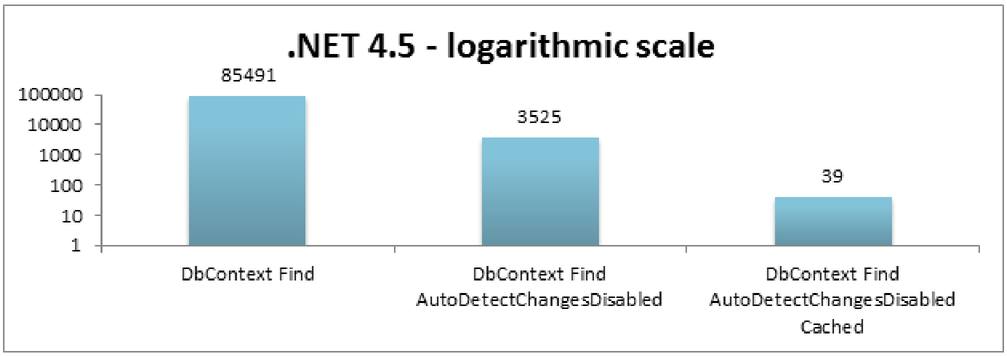
By default when an entity is returned in the results of a query, just before EF materializes it, the ObjectContext will check if an entity with the same key has already been loaded into its ObjectStateManager. If an entity with the same keys is already present EF will include it in the results of the query. Although EF will still issue the query against the database, this behavior can bypass much of the cost of materializing the entity multiple times.

3.1.1 Getting entities from the object cache using DbContext Find

Unlike a regular query, the Find method in DbSet (APIs included for the first time in EF 4.1) will perform a search in memory before even issuing the query against the database. It's important to note that two different ObjectContext instances will have two different ObjectStateManager instances, meaning that they have separate object caches.

Find uses the primary key value to attempt to find an entity tracked by the context. If the entity is not in the context then a query will be executed and evaluated against the database, and null is returned if the entity is not found in the context or in the database. Note that Find also returns entities that have been added to the context but have not yet been saved to the database.

There is a performance consideration to be taken when using Find. Invocations to this method by default will trigger a validation of the object cache in order to detect changes that are still pending commit to the database. This process can be very expensive if there are a very large number of objects in the object cache or in a large object graph being added to the object cache, but it can also be disabled. In certain cases, you may perceive over an order of magnitude of difference in calling the Find method when you disable auto detect changes. Yet a second order of magnitude is perceived when the object actually is in the cache versus when the object has to be retrieved from the database. Here is an example graph with measurements taken using some of our microbenchmarks, expressed in milliseconds, with a load of 5000 entities:



Example of Find with auto-detect changes disabled:

```
context.Configuration.AutoDetectChangesEnabled = false;
var product = context.Products.Find(productId);
context.Configuration.AutoDetectChangesEnabled = true;
...
```

What you have to consider when using the Find method is:

1. If the object is not in the cache the benefits of Find are negated, but the syntax is still simpler than a query by key.
2. If auto detect changes is enabled the cost of the Find method may increase by one order of magnitude, or even more depending on the complexity of your model and the amount of entities in your object cache.

Also, keep in mind that Find only returns the entity you are looking for and it does not automatically loads its associated entities if they are not already in the object cache. If you need to retrieve associated entities, you can use a query by key with eager loading. For more information see **8.1 Lazy Loading vs. Eager Loading**.

3.1.2 Performance issues when the object cache has many entities

The object cache helps to increase the overall responsiveness of Entity Framework. However, when the object cache has a very large amount of entities loaded it may affect certain operations such as Add, Remove, Find, Entry, SaveChanges and more. In particular, operations that trigger a call to DetectChanges will be negatively affected by very large object caches. DetectChanges synchronizes the object graph with the object state manager and its performance will determined directly by the size of the object graph. For more information about DetectChanges, see Tracking Changes in POCO Entities.

When using Entity Framework 6, developers are able to call AddRange and RemoveRange directly on a DbSet, instead of iterating on a collection and calling Add once per instance. The advantage of using the range methods is that the cost of DetectChanges is only paid once for the entire set of entities as opposed to once per each added entity.

3.2 Query Plan Caching

The first time a query is executed, it goes through the internal plan compiler to translate the conceptual query into the store command (for example, the T-SQL which is executed when run against SQL Server). If query plan caching is enabled, the next time the query is executed the store command is retrieved directly from the query plan cache for execution, bypassing the plan compiler.

The query plan cache is shared across ObjectContext instances within the same AppDomain. You don't need to hold onto an ObjectContext instance to benefit from query plan caching.

3.2.1 Some notes about Query Plan Caching

- The query plan cache is shared for all query types: Entity SQL, LINQ to Entities, and CompiledQuery objects.
- By default, query plan caching is enabled for Entity SQL queries, whether executed through an EntityCommand or through an ObjectQuery. It is also enabled by default for LINQ to Entities queries in Entity Framework on .NET 4.5, and in Entity Framework 6
    - Query plan caching can be disabled by setting the EnablePlanCaching property (on EntityCommand or ObjectQuery) to false. For example:

```
                    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;
```

- For parameterized queries, changing the parameter's value will still hit the cached query. But changing a parameter's facets (for example, size, precision, or scale) will hit a different entry in the cache.
- When using Entity SQL, the query string is part of the key. Changing the query at all will result in different cache entries, even if the queries are functionally equivalent. This includes changes to casing or whitespace.
- When using LINQ, the query is processed to generate a part of the key. Changing the LINQ expression will therefore generate a different key.
- Other technical limitations may apply; see Autocompiled Queries for more details.

3.2.2 Cache eviction algorithm

Understanding how the internal algorithm works will help you figure out when to enable or disable query plan caching. The cleanup algorithm is as follows:

1. Once the cache contains a set number of entries (800), we start a timer that periodically (once-per-minute) sweeps the cache.
2. During cache sweeps, entries are removed from the cache on a LFRU (Least frequently – recently used) basis. This algorithm takes both hit count and age into account when deciding which entries are ejected.
3. At the end of each cache sweep, the cache again contains 800 entries.

All cache entries are treated equally when determining which entries to evict. This means the store command for a CompiledQuery has the same chance of eviction as the store command for an Entity SQL query.

Note that the cache eviction timer is kicked in when there are 800 entities in the cache, but the cache is only swept 60 seconds after this timer is started. That means that for up to 60 seconds your cache may grow to be quite large.

3.2.3 Test Metrics demonstrating query plan caching performance

To demonstrate the effect of query plan caching on your application's performance, we performed a test where we executed a number of Entity SQL queries against the Navision model. See the appendix for a description of the Navision model and the types of queries which were executed. In this test, we first iterate through the list of queries and execute each one once to add them to the cache (if caching is enabled). This step is untimed. Next, we sleep the main thread for over 60 seconds to allow cache sweeping to take place; finally, we iterate through the list a 2nd time to execute the cached queries. Additionally, he SQL Server plan cache is flushed before each set of queries is executed so that the times we obtain accurately reflect the benefit given by the query plan cache.

3.2.3.1 Test Results

TEST	EF5 NO CACHE	EF5 CACHED	EF6 NO CACHE	EF6 CACHED
Enumerating all 18723 queries	124	125.4	124.3	125.3
Avoiding sweep (just the first 800 queries, regardless of complexity)	41.7	5.5	40.5	5.4
Just the AggregatingSubtotals queries (178 total - which avoids sweep)	39.5	4.5	38.1	4.6

All times in seconds.

Moral - when executing lots of distinct queries (for example, dynamically created queries), caching doesn't help and the resulting flushing of the cache can keep the queries that would benefit the most from plan caching from actually using it.

The AggregatingSubtotals queries are the most complex of the queries we tested with. As expected, the more complex the query is, the more benefit you will see from query plan caching.

Because a CompiledQuery is really a LINQ query with its plan cached, the comparison of a CompiledQuery versus the equivalent Entity SQL query should have similar results. In fact, if an app has lots of dynamic Entity SQL queries, filling the cache with queries will also effectively cause CompiledQueries to “decompile” when they are flushed from the cache. In this scenario, performance may be improved by disabling caching on the dynamic queries to prioritize the CompiledQueries. Better yet, of course, would be to rewrite the app to use parameterized queries instead of dynamic queries.

3.3 Using CompiledQuery to improve performance with LINQ queries

Our tests indicate that using CompiledQuery can bring a benefit of 7% over autocompiled LINQ queries; this means that you’ll spend 7% less time executing code from the Entity Framework stack; it does not mean your application will be 7% faster. Generally speaking, the cost of writing and maintaining CompiledQuery objects in EF 5.0 may not be worth the trouble when compared to the benefits. Your mileage may vary, so exercise this option if your project requires the extra push. Note that CompiledQueries are only compatible with ObjectContext-derived models, and not compatible with DbContext-derived models.

For more information on creating and invoking a CompiledQuery, see Compiled Queries (LINQ to Entities).

There are two considerations you have to take when using a CompiledQuery, namely the requirement to use static instances and the problems they have with composability. Here follows an in-depth explanation of these two considerations.

3.3.1 Use static CompiledQuery instances

Since compiling a LINQ query is a time-consuming process, we don’t want to do it every time we need to fetch data from the database. CompiledQuery instances allow you to compile once and run multiple times, but you have to be careful and procure to re-use the same CompiledQuery instance every time instead of compiling it over and over again. The use of static members to store the CompiledQuery instances becomes necessary; otherwise you won’t see any benefit.

For example, suppose your page has the following method body to handle displaying the products for the selected category:

```
// 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;
```

In this case, you will create a new CompiledQuery instance on the fly every time the method is called. Instead of seeing performance benefits by retrieving the store command from the query plan cache, the CompiledQuery will go through the plan compiler every time a new instance is created. In fact, you will be polluting your query plan cache with a new CompiledQuery entry every time the method is called.

Instead, you want to create a static instance of the compiled query, so you are invoking the same compiled query every time the method is called. One way to so this is by adding the CompiledQuery instance as a member of your object context. You can then make things a little cleaner by accessing the CompiledQuery through a helper method:

```
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();
    }
```

This helper method would be invoked as follows:

```
this.productsGrid.DataSource = context.GetProductsForCategory(selectedCategory);
```

3.3.2 Composing over a CompiledQuery

The ability to compose over any LINQ query is extremely useful; to do this, you simply invoke a method after the IQueryable such as *Skip()* or *Count()*. However, doing so essentially returns a new IQueryable object. While there's nothing to stop you technically from composing over a CompiledQuery, doing so will cause the generation of a new IQueryable object that requires passing through the plan compiler again.

Some components will make use of composed IQueryable objects to enable advanced functionality. For example, ASP.NET's GridView can be data-bound to an IQueryable object via the SelectMethod property. The GridView will then compose over this IQueryable object to allow sorting and paging over the data model. As you can see, using a CompiledQuery for the GridView would not hit the compiled query but would generate a new autocompiled query.

The Customer Advisory Team discusses this in their "Potential Performance Issues with Compiled LINQ Query Re-

Compiles" blog post: http://blogs.msdn.com/b/appfabriccat/archive/2010/08/06/potential-performance-issues-with-compiled-linq-query-re-compiles.aspx.

One place where you may run into this is when adding progressive filters to a query. For example, suppose you had a Customers page with several drop-down lists for optional filters (for example, Country and OrdersCount). You can compose these filters over the IQueryable results of a CompiledQuery, but doing so will result in the new query going through the plan compiler every time you execute it.

```
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();
}
```

To avoid this re-compilation, you can rewrite the CompiledQuery to take the possible filters into account:

```
    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)
        );
```

Which would be invoked in the UI like:

```
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();
}
```

A tradeoff here is the generated store command will always have the filters with the null checks, but these should be fairly simple for the database server to optimize:

```
...
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 Metadata caching

The Entity Framework also supports Metadata caching. This is essentially caching of type information and type-to-database mapping information across different connections to the same model. The Metadata cache is unique per AppDomain.

3.4.1 Metadata Caching algorithm

1. Metadata information for a model is stored in an ItemCollection for each EntityConnection.
   - As a side note, there are different ItemCollection objects for different parts of the model. For example, StoreItemCollections contains the information about the database model; ObjectItemCollection contains information about the data model; EdmItemCollection contains information about the conceptual model.
2. If two connections use the same connection string, they will share the same ItemCollection instance.
3. Functionally equivalent but textually different connection strings may result in different metadata caches. We do tokenize connection strings, so simply changing the order of the tokens should result in shared metadata. But two connection strings that seem functionally the same may not be evaluated as identical after tokenization.
4. The ItemCollection is periodically checked for use. If it is determined that a workspace has not been accessed recently, it will be marked for cleanup on the next cache sweep.
5. Merely creating an EntityConnection will cause a metadata cache to be created (though the item collections in it will not be initialized until the connection is opened). This workspace will remain in-memory until the caching algorithm determines it is not “in use”.

The Customer Advisory Team has written a blog post that describes holding a reference to an ItemCollection in order to avoid "deprecation" when using large models: <http://blogs.msdn.com/b/appfabriccat/archive/2010/10/22/metadataworkspace-reference-in-wcf-services.aspx>.

3.4.2 The relationship between Metadata Caching and Query Plan Caching

The query plan cache instance lives in the MetadataWorkspace's ItemCollection of store types. This means that cached store commands will be used for queries against any context instantiated using a given MetadataWorkspace. It also means that if you have two connections strings that are slightly different and don't match after tokenizing, you will have different query plan cache instances.

3.5 Results caching

With results caching (also known as "second-level caching"), you keep the results of queries in a local cache. When issuing a query, you first see if the results are available locally before you query against the store. While results caching isn't directly supported by Entity Framework, it's possible to add a second level cache by using a wrapping provider. An example wrapping provider with a second-level cache is Alachisoft's Entity Framework Second Level Cache based on NCache.

This implementation of second-level caching is an injected functionality that takes place after the LINQ expression has been evaluated (and funcletized) and the query execution plan is computed or retrieved from the first-level cache. The second-level cache will then store only the raw database results, so the materialization pipeline still executes afterwards.

3.5.1 Additional references for results caching with the wrapping provider

- Julie Lerman has written a "Second-Level Caching in Entity Framework and Windows Azure" MSDN article that includes how to update the sample wrapping provider to use Windows Server AppFabric caching: https://msdn.microsoft.com/magazine/hh394143.aspx

- If you are working with Entity Framework 5, the team blog has a post that describes how to get things running with the caching provider for Entity Framework 5: <http://blogs.msdn.com/b/adonet/archive/2010/09/13/ef-caching-with-jarek-kowalski-s-provider.aspx>. It also includes a T4 template to help automate adding the 2nd-level caching to your project.

4 Autocompiled Queries

When a query is issued against a database using Entity Framework, it must go through a series of steps before actually materializing the results; one such step is Query Compilation. Entity SQL queries were known to have good performance as they are automatically cached, so the second or third time you execute the same query it can skip the plan compiler and use the cached plan instead.

Entity Framework 5 introduced automatic caching for LINQ to Entities queries as well. In past editions of Entity Framework creating a CompiledQuery to speed your performance was a common practice, as this would make your LINQ to Entities query cacheable. Since caching is now done automatically without the use of a CompiledQuery, we call this feature "autocompiled queries". For more information about the query plan cache and its mechanics, see Query Plan Caching.

Entity Framework detects when a query requires to be recompiled, and does so when the query is invoked even if it had been compiled before. Common conditions that cause the query to be recompiled are:

- Changing the MergeOption associated to your query. The cached query will not be used, instead the plan compiler will run again and the newly created plan gets cached.
- Changing the value of ContextOptions.UseCSharpNullComparisonBehavior. You get the same effect as changing the MergeOption.

Other conditions can prevent your query from using the cache. Common examples are:

- Using IEnumerable<T>.Contains<>(T value).
- Using functions that produce queries with constants.
- Using the properties of a non-mapped object.
- Linking your query to another query that requires to be recompiled.

4.1 Using IEnumerable<T>.Contains<T>(T value)

Entity Framework does not cache queries that invoke IEnumerable<T>.Contains<T>(T value) against an in-memory collection, since the values of the collection are considered volatile. The following example query will not be cached, so it will always be processed by the plan compiler:

```
int[] ids = new int[10000];
...
using (var context = new MyContext())
{
    var query = context.MyEntities
                    .Where(entity => ids.Contains(entity.Id));

    var results = query.ToList();
    ...
}
```

Note that the size of the IEnumerable against which Contains is executed determines how fast or how slow your query is compiled. Performance can suffer significantly when using large collections such as the one shown in the example above.

Entity Framework 6 contains optimizations to the way IEnumerable<T>.Contains<T>(T value) works when queries are executed. The SQL code that is generated is much faster to produce and more readable, and in most cases it also executes faster in the server.

4.2 Using functions that produce queries with constants

The Skip(), Take(), Contains() and DefautIfEmpty() LINQ operators do not produce SQL queries with parameters but instead put the values passed to them as constants. Because of this, queries that might otherwise be identical end up polluting the query plan cache, both on the EF stack and on the database server, and do not get reutilized unless the same constants are used in a subsequent query execution. For example:

```
var id = 10;
...
using (var context = new MyContext())
{
    var query = context.MyEntities.Select(entity => entity.Id).Contains(id);

    var results = query.ToList();
    ...
}
```

In this example, each time this query is executed with a different value for id the query will be compiled into a new plan.

In particular pay attention to the use of Skip and Take when doing paging. In EF6 these methods have a lambda overload that effectively makes the cached query plan reusable because EF can capture variables passed to these methods and translate them to SQLparameters. This also helps keep the cache cleaner since otherwise each query with a different constant for Skip and Take would get its own query plan cache entry.

Consider the following code, which is suboptimal but is only meant to exemplify this class of queries:

```
var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i < count; ++i)
{
    var currentCustomer = customers.Skip(i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}
```

A faster version of this same code would involve calling Skip with a lambda:

```
var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i \< count; ++i)
{
    var currentCustomer = customers.Skip(() => i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}
```

The second snippet may run up to 11% faster because the same query plan is used every time the query is run, which saves CPU time and avoids polluting the query cache. Furthermore, because the parameter to Skip is in a closure the code might as well look like this now:

```
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 Using the properties of a non-mapped object

When a query uses the properties of a non-mapped object type as a parameter then the query will not get cached. For example:

```
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();
    ...
}
```

In this example, assume that class NonMappedType is not part of the Entity model. This query can easily be changed to not use a non-mapped type and instead use a local variable as the parameter to the query:

```
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();
    ...
}
```

In this case, the query will be able to get cached and will benefit from the query plan cache.

4.4 Linking to queries that require recompiling

Following the same example as above, if you have a second query that relies on a query that needs to be recompiled, your entire second query will also be recompiled. Here's an example to illustrate this scenario:

```
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();
    ...
}
```

The example is generic, but it illustrates how linking to firstQuery is causing secondQuery to be unable to get cached. If firstQuery had not been a query that requires recompiling, then secondQuery would have been cached.

5 NoTracking Queries

5.1 Disabling change tracking to reduce state management overhead

If you are in a read-only scenario and want to avoid the overhead of loading the objects into the ObjectStateManager, you can issue "No Tracking" queries. Change tracking can be disabled at the query level.

Note though that by disabling change tracking you are effectively turning off the object cache. When you query for an entity, we can't skip materialization by pulling the previously-materialized query results from the ObjectStateManager. If you are repeatedly querying for the same entities on the same context, you might actually see a performance benefit from enabling change tracking.

When querying using ObjectContext, ObjectQuery and ObjectSet instances will remember a MergeOption once it is set, and queries that are composed on them will inherit the effective MergeOption of the parent query. When using DbContext, tracking can be disabled by calling the AsNoTracking() modifier on the DbSet.

5.1.1 Disabling change tracking for a query when using DbContext

You can switch the mode of a query to NoTracking by chaining a call to the AsNoTracking() method in the query. Unlike ObjectQuery, the DbSet and DbQuery classes in the DbContext API don’t have a mutable property for the MergeOption.

```
var productsForCategory = from p in context.Products.AsNoTracking()
                          where p.Category.CategoryName == selectedCategory
                          select p;
```

5.1.2 Disabling change tracking at the query level using ObjectContext

```
var productsForCategory = from p in context.Products
                          where p.Category.CategoryName == selectedCategory
                          select p;

((ObjectQuery)productsForCategory).MergeOption = MergeOption.NoTracking;
```

5.1.3 Disabling change tracking for an entire entity set using ObjectContext

```
context.Products.MergeOption = MergeOption.NoTracking;

var productsForCategory = from p in context.Products
                          where p.Category.CategoryName == selectedCategory
                          select p;
```

5.2 Test Metrics demonstrating the performance benefit of NoTracking queries

In this test we look at the cost of filling the ObjectStateManager by comparing Tracking to NoTracking queries for the Navision model. See the appendix for a description of the Navision model and the types of queries which were executed. In this test, we iterate through the list of queries and execute each one once. We ran two variations of the test, once with NoTracking queries and once with the default merge option of "AppendOnly". We ran each variation 3 times and take the mean value of the runs. Between the tests we clear the query cache on the SQL Server and shrink the tempdb by running the following commands:

1. DBCC DROPCLEANBUFFERS
2. DBCC FREEPROCCACHE
3. DBCC SHRINKDATABASE (tempdb, 0)

Test Results, median over 3 runs:

	NO TRACKING – WORKING SET	NO TRACKING – TIME	APPEND ONLY – WORKING SET	APPEND ONLY – TIME
Entity Framework 5	460361728	1163536 ms	596545536	1273042 ms
Entity Framework 6	647127040	190228 ms	832798720	195521 ms

Entity Framework 5 will have a smaller memory footprint at the end of the run than Entity Framework 6 does. The additional memory consumed by Entity Framework 6 is the result of additional memory structures and code that enable new features and better performance.

There's also a clear difference in memory footprint when using the ObjectStateManager. Entity Framework 5 increased its footprint by 30% when keeping track of all the entities we materialized from the database. Entity Framework 6 increased its footprint by 28% when doing so.

In terms of time, Entity Framework 6 outperforms Entity Framework 5 in this test by a large margin. Entity Framework 6 completed the test in roughly 16% of the time consumed by Entity Framework 5. Additionally, Entity Framework 5 takes 9% more time to complete when the ObjectStateManager is being used. In comparison, Entity Framework 6 is using 3% more time when using the ObjectStateManager.

6 Query Execution Options

Entity Framework offers several different ways to query. We'll take a look at the following options, compare the pros and cons of each, and examine their performance characteristics:

- LINQ to Entities.
- No Tracking LINQ to Entities.
- Entity SQL over an ObjectQuery.
- Entity SQL over an EntityCommand.
- ExecuteStoreQuery.
- SqlQuery.
- CompiledQuery.

6.1 LINQ to Entities queries

```
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
```

Pros

- Suitable for CUD operations.
- Fully materialized objects.
- Simplest to write with syntax built into the programming language.
- Good performance.

Cons

- Certain technical restrictions, such as:
  - Patterns using DefaultIfEmpty for OUTER JOIN queries result in more complex queries than simple OUTER JOIN statements in Entity SQL.
  - You still can't use LIKE with general pattern matching.

6.2 No Tracking LINQ to Entities queries

When the context derives ObjectContext:

```
context.Products.MergeOption = MergeOption.NoTracking;
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
```

When the context derives DbContext:

```
var q = context.Products.AsNoTracking()
                        .Where(p => p.Category.CategoryName == "Beverages");
```

Pros

- Improved performance over regular LINQ queries.
- Fully materialized objects.
- Simplest to write with syntax built into the programming language.

Cons

- Not suitable for CUD operations.
- Certain technical restrictions, such as:
  - Patterns using DefaultIfEmpty for OUTER JOIN queries result in more complex queries than simple OUTER JOIN statements in Entity SQL.
  - You still can't use LIKE with general pattern matching.

Note that queries that project scalar properties are not tracked even if the NoTracking is not specified. For example:

```
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages").Select(p => new { p.ProductName
});
```

This particular query doesn't explicitly specify being NoTracking, but since it's not materializing a type that's known to the object state manager then the materialized result is not tracked.

6.3 Entity SQL over an ObjectQuery

```
ObjectQuery<Product> products = context.Products.Where("it.Category.CategoryName = 'Beverages'");
```

Pros

- Suitable for CUD operations.
- Fully materialized objects.
- Supports query plan caching.

Cons

- Involves textual query strings which are more prone to user error than query constructs built into the language.

6.4 Entity SQL over an Entity Command

```
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
    }
}
```

Pros

- Supports query plan caching in .NET 4.0 (plan caching is supported by all other query types in .NET 4.5).

Cons

- Involves textual query strings which are more prone to user error than query constructs built into the language.
- Not suitable for CUD operations.
- Results are not automatically materialized, and must be read from the data reader.

6.5 SqlQuery and ExecuteStoreQuery

SqlQuery on Database:

```
// use this to obtain entities and not track them
var q1 = context.Database.SqlQuery<Product>("select * from products");
```

SqlQuery on DbSet:

```
// use this to obtain entities and have them tracked
var q2 = context.Products.SqlQuery("select * from products");
```

ExecyteStoreQuery:

```
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')"
);
```

Pros

- Generally fastest performance since plan compiler is bypassed.
- Fully materialized objects.
- Suitable for CUD operations when used from the DbSet.

Cons

- Query is textual and error prone.
- Query is tied to a specific backend by using store semantics instead of conceptual semantics.
- When inheritance is present, handcrafted query needs to account for mapping conditions for the type requested.

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");
```

Pros

- Provides up to a 7% performance improvement over regular LINQ queries.
- Fully materialized objects.
- Suitable for CUD operations.

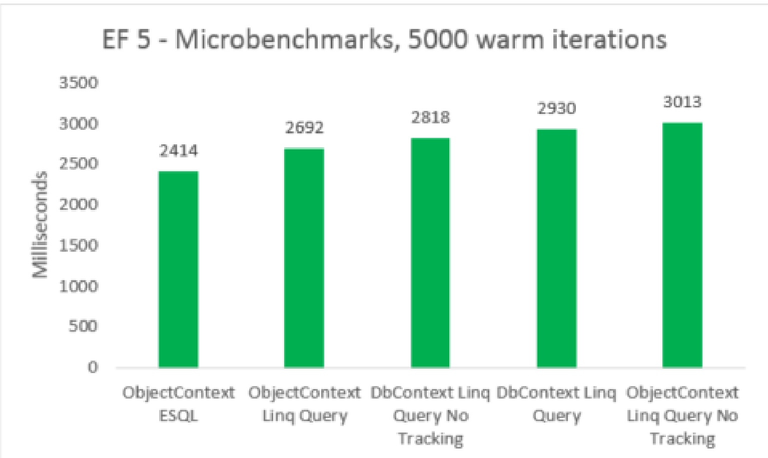
Cons

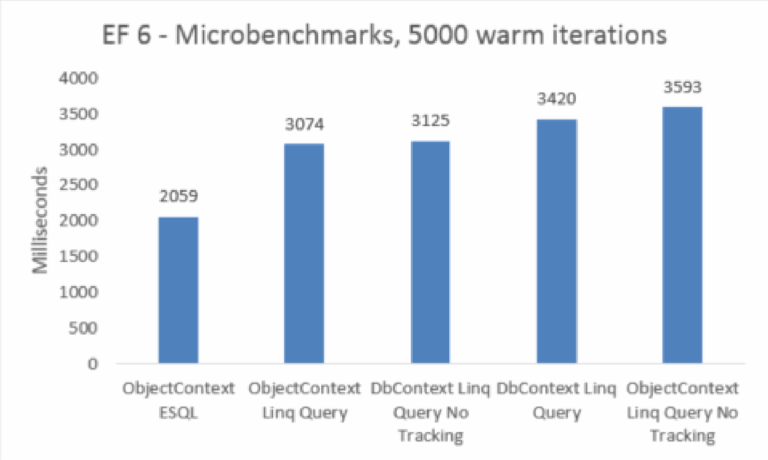
- Increased complexity and programming overhead.
- The performance improvement is lost when composing on top of a compiled query.
- Some LINQ queries can't be written as a CompiledQuery - for example, projections of anonymous types.

6.7 Performance Comparison of different query options

Simple microbenchmarks where the context creation was not timed were put to the test. We measured querying 5000 times for a set of non-cached entities in a controlled environment. These numbers are to be taken with a warning: they do not reflect actual numbers produced by an application, but instead they are a very accurate measurement of how much of a performance difference there is when different querying options are compared apples-to-apples, excluding the cost of creating a new context.

EF	TEST	TIME (MS)	MEMORY
EF5	ObjectContext ESQL	2414	38801408
EF5	ObjectContext Linq Query	2692	38277120
EF5	DbContext Linq Query No Tracking	2818	41840640
EF5	DbContext Linq Query	2930	41771008
EF5	ObjectContext Linq Query No Tracking	3013	38412288
EF6	ObjectContext ESQL	2059	46039040
EF6	ObjectContext Linq Query	3074	45248512
EF6	DbContext Linq Query No Tracking	3125	47575040
EF6	DbContext Linq Query	3420	47652864
EF6	ObjectContext Linq Query No Tracking	3593	45260800



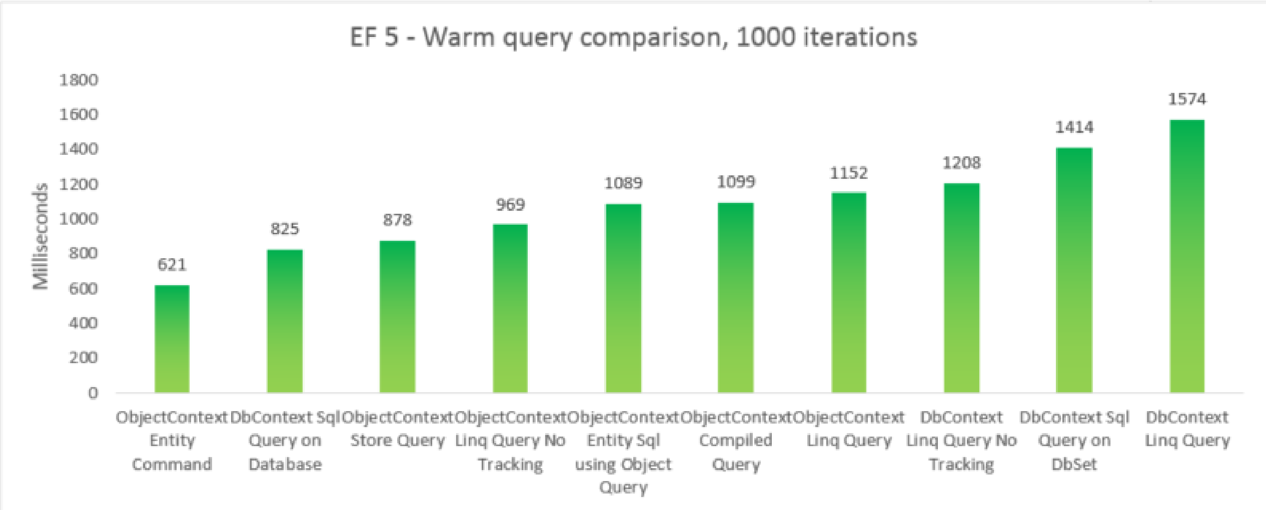


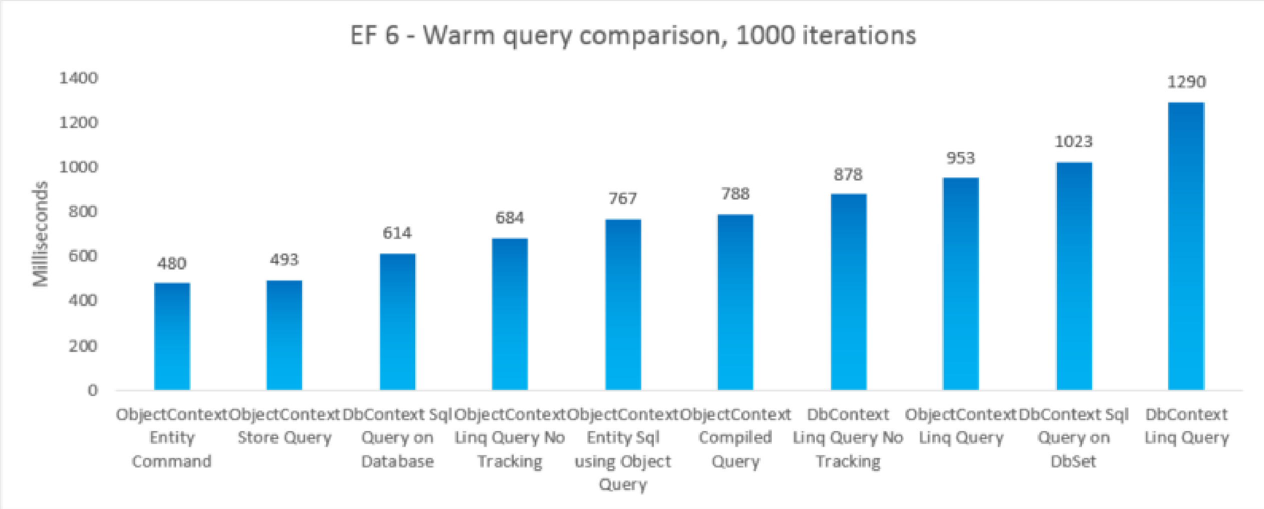
Microbenchmarks are very sensitive to small changes in the code. In this case, the difference between the costs of Entity Framework 5 and Entity Framework 6 are due to the addition of interception and transactional improvements. These microbenchmarks numbers, however, are an amplified vision into a very small fragment of what Entity Framework does. Real-world scenarios of warm queries should not see a performance regression when upgrading from Entity Framework 5 to Entity Framework 6.

To compare the real-world performance of the different query options, we created 5 separate test variations where we use a different query option to select all products whose category name is "Beverages". Each iteration includes the cost of creating the context, and the cost of materializing all returned entities. 10 iterations are run untimed before taking the sum of 1000 timed iterations. The results shown are the median run taken from 5 runs of each test. For more information, see Appendix B which includes the code for the test.

EF	TEST	TIME (MS)	MEMORY
EF5	ObjectContext Entity Command	621	39350272
EF5	DbContext Sql Query on Database	825	37519360
EF5	ObjectContext Store Query	878	39460864
EF5	ObjectContext Linq Query No Tracking	969	38293504
EF5	ObjectContext Entity Sql using Object Query	1089	38981632
EF5	ObjectContext Compiled Query	1099	38682624
EF5	ObjectContext Linq Query	1152	38178816
EF5	DbContext Linq Query No Tracking	1208	41803776
EF5	DbContext Sql Query on DbSet	1414	37982208
EF5	DbContext Linq Query	1574	41738240

EF	TEST	TIME (MS)	MEMORY
EF6	ObjectContext Entity Command	480	47247360
EF6	ObjectContext Store Query	493	46739456
EF6	DbContext Sql Query on Database	614	41607168
EF6	ObjectContext Linq Query No Tracking	684	46333952
EF6	ObjectContext Entity Sql using Object Query	767	48865280
EF6	ObjectContext Compiled Query	788	48467968
EF6	DbContext Linq Query No Tracking	878	47554560
EF6	ObjectContext Linq Query	953	47632384
EF6	DbContext Sql Query on DbSet	1023	41992192
EF6	DbContext Linq Query	1290	47529984





NOTE

For completeness, we included a variation where we execute an Entity SQL query on an EntityCommand. However, because results are not materialized for such queries, the comparison isn't necessarily apples-to-apples. The test includes a close approximation to materializing to try making the comparison fairer.

In this end-to-end case, Entity Framework 6 outperforms Entity Framework 5 due to performance improvements made on several parts of the stack, including a much lighter DbContext initialization and faster MetadataCollection<T> lookups.

7 Design time performance considerations

7.1 Inheritance Strategies

Another performance consideration when using Entity Framework is the inheritance strategy you use. Entity Framework supports 3 basic types of inheritance and their combinations:

- Table per Hierarchy (TPH) – where each inheritance set maps to a table with a discriminator column to indicate which particular type in the hierarchy is being represented in the row.
- Table per Type (TPT) – where each type has its own table in the database; the child tables only define the columns that the parent table doesn’t contain.
- Table per Class (TPC) – where each type has its own full table in the database; the child tables define all their fields, including those defined in parent types.

If your model uses TPT inheritance, the queries which are generated will be more complex than those that are generated with the other inheritance strategies, which may result on longer execution times on the store. It will generally take longer to generate queries over a TPT model, and to materialize the resulting objects.

See the "Performance Considerations when using TPT (Table per Type) Inheritance in the Entity Framework" MSDN blog post: <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 Avoiding TPT in Model First or Code First applications

When you create a model over an existing database that has a TPT schema, you don't have many options. But when creating an application using Model First or Code First, you should avoid TPT inheritance for performance concerns.

When you use Model First in the Entity Designer Wizard, you will get TPT for any inheritance in your model. If you want to switch to a TPH inheritance strategy with Model First, you can use the "Entity Designer Database Generation Power Pack" available from the Visual Studio Gallery (<http://visualstudiogallery.msdn.microsoft.com/df3541c3-d833-4b65-b942-989e7ec74c87/>).

When using Code First to configure the mapping of a model with inheritance, EF will use TPH by default, therefore all entities in the inheritance hierarchy will be mapped to the same table. See the "Mapping with the Fluent API" section of the "Code First in Entity Framework4.1" article in MSDN Magazine (http://msdn.microsoft.com/magazine/hh126815.aspx) for more details.

7.2 Upgrading from EF4 to improve model generation time

A SQL Server-specific improvement to the algorithm that generates the store-layer (SSDL) of the model is available in Entity Framework 5 and 6, and as an update to Entity Framework 4 when Visual Studio 2010 SP1 is installed. The following test results demonstrate the improvement when generating a very big model, in this case the Navision model. See Appendix C for more details about it.

The model contains 1005 entity sets and 4227 association sets.

CONFIGURATION	BREAKDOWN OF TIME CONSUMED
Visual Studio 2010, Entity Framework 4	SSDL Generation: 2 hr 27 min Mapping Generation: 1 second CSDL Generation: 1 second ObjectLayer Generation: 1 second View Generation: 2 h 14 min
Visual Studio 2010 SP1, Entity Framework 4	SSDL Generation: 1 second Mapping Generation: 1 second CSDL Generation: 1 second ObjectLayer Generation: 1 second View Generation: 1 hr 53 min
Visual Studio 2013, Entity Framework 5	SSDL Generation: 1 second Mapping Generation: 1 second CSDL Generation: 1 second ObjectLayer Generation: 1 second View Generation: 65 minutes
Visual Studio 2013, Entity Framework 6	SSDL Generation: 1 second Mapping Generation: 1 second CSDL Generation: 1 second ObjectLayer Generation: 1 second View Generation: 28 seconds.

It's worth noting that when generating the SSDL, the load is almost entirely spent on the SQL Server, while the client development machine is waiting idle for results to come back from the server. DBAs should particularly appreciate this improvement. It's also worth noting that essentially the entire cost of model generation takes place in View Generation now.

7.3 Splitting Large Models with Database First and Model First

As model size increases, the designer surface becomes cluttered and difficult to use. We typically consider a model with more than 300 entities to be too large to effectively use the designer. The following blog post describes several options for splitting large models: <http://blogs.msdn.com/b/adonet/archive/2008/11/25/working-with-large-models-in-entity-framework-part-2.aspx>.

The post was written for the first version of Entity Framework, but the steps still apply.

7.4 Performance considerations with the Entity Data Source Control

We've seen cases in multi-threaded performance and stress tests where the performance of a web application using the EntityDataSource Control deteriorates significantly. The underlying cause is that the EntityDataSource repeatedly calls MetadataWorkspace.LoadFromAssembly on the assemblies referenced by the Web application to discover the types to be used as entities.

The solution is to set the ContextTypeName of the EntityDataSource to the type name of your derived ObjectContext class. This turns off the mechanism that scans all referenced assemblies for entity types.

Setting the ContextTypeName field also prevents a functional problem where the EntityDataSource in .NET 4.0 throws a ReflectionTypeLoadException when it can't load a type from an assembly via reflection. This issue has been fixed in .NET 4.5.

7.5 POCO entities and change tracking proxies

Entity Framework enables you to use custom data classes together with your data model without making any modifications to the data classes themselves. This means that you can use "plain-old" CLR objects (POCO), such as existing domain objects, with your data model. These POCO data classes (also known as persistence-ignorant objects), which are mapped to entities that are defined in a data model, support most of the same query, insert, update, and delete behaviors as entity types that are generated by the Entity Data Model tools.

Entity Framework can also create proxy classes derived from your POCO types, which are used when you want to enable features such as lazy loading and automatic change tracking on POCO entities. Your POCO classes must meet certain requirements to allow Entity Framework to use proxies, as described here: http://msdn.microsoft.com/library/dd468057.aspx.

Chance tracking proxies will notify the object state manager each time any of the properties of your entities has its value changed, so Entity Framework knows the actual state of your entities all the time. This is done by adding notification events to the body of the setter methods of your properties, and having the object state manager processing such events. Note that creating a proxy entity will typically be more expensive than creating a non-proxy POCO entity due to the added set of events created by Entity Framework.

When a POCO entity does not have a change tracking proxy, changes are found by comparing the contents of your entities against a copy of a previous saved state. This deep comparison will become a lengthy process when you have many entities in your context, or when your entities have a very large amount of properties, even if none of them changed since the last comparison took place.

In summary: you'll pay a performance hit when creating the change tracking proxy, but change tracking will help you speed up the change detection process when your entities have many properties or when you have many entities in your model. For entities with a small number of properties where the amount of entities doesn't grow too much, having change tracking proxies may not be of much benefit.

8 Loading Related Entities

8.1 Lazy Loading vs. Eager Loading

Entity Framework offers several different ways to load the entities that are related to your target entity. For example, when you query for Products, there are different ways that the related Orders will be loaded into the Object State Manager. From a performance standpoint, the biggest question to consider when loading related entities will be whether to use Lazy Loading or Eager Loading.

When using Eager Loading, the related entities are loaded along with your target entity set. You use an Include statement in your query to indicate which related entities you want to bring in.

When using Lazy Loading, your initial query only brings in the target entity set. But whenever you access a navigation property, another query is issued against the store to load the related entity.

Once an entity has been loaded, any further queries for the entity will load it directly from the Object State Manager, whether you are using lazy loading or eager loading.

8.2 How to choose between Lazy Loading and Eager Loading

The important thing is that you understand the difference between Lazy Loading and Eager Loading so that you can make the correct choice for your application. This will help you evaluate the tradeoff between multiple requests against the database versus a single request that may contain a large payload. It may be appropriate to use eager loading in some parts of your application and lazy loading in other parts.

As an example of what's happening under the hood, suppose you want to query for the customers who live in the UK and their order count.

Using Eager Loading

```
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);
}
```

Using Lazy Loading

```
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);
}
```

When using eager loading, you'll issue a single query that returns all customers and all orders. The store command looks like:

```
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
```

When using lazy loading, you'll issue the following query initially:

```
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]
```

And each time you access the Orders navigation property of a customer another query like the following is issued against the store:

```
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'
```

For more information, see the Loading Related Objects.

8.2.1 Lazy Loading versus Eager Loading cheat sheet

There’s no such thing as a one-size-fits-all to choosing eager loading versus lazy loading. Try first to understand the differences between both strategies so you can do a well informed decision; also, consider if your code fits to any of the following scenarios:

SCENARIO	OUR SUGGESTION
Do you need to access many navigation properties from the fetched entities?	**No** - Both options will probably do. However, if the payload your query is bringing is not too big, you may experience performance benefits by using Eager loading as it’ll require less network round trips to materialize your objects. **Yes** - If you need to access many navigation properties from the entities, you’d do that by using multiple include statements in your query with Eager loading. The more entities you include, the bigger the payload your query will return. Once you include three or more entities into your query, consider switching to Lazy loading.

SCENARIO	OUR SUGGESTION
Do you know exactly what data will be needed at run time?	**No** - Lazy loading will be better for you. Otherwise, you may end up querying for data that you will not need. **Yes** - Eager loading is probably your best bet; it will help loading entire sets faster. If your query requires fetching a very large amount of data, and this becomes too slow, then try Lazy loading instead.
Is your code executing far from your database? (increased network latency)	**No** - When the network latency is not an issue, using Lazy loading may simplify your code. Remember that the topology of your application may change, so don't take database proximity for granted. **Yes** - When the network is a problem, only you can decide what fits better for your scenario. Typically Eager loading will be better because it requires fewer round trips.

8.2.2 Performance concerns with multiple Includes

When we hear performance questions that involve server response time problems, the source of the issue is frequently queries with multiple Include statements. While including related entities in a query is powerful, it's important to understand what's happening under the covers.

It takes a relatively long time for a query with multiple Include statements in it to go through our internal plan compiler to produce the store command. The majority of this time is spent trying to optimize the resulting query. The generated store command will contain an Outer Join or Union for each Include, depending on your mapping. Queries like this will bring in large connected graphs from your database in a single payload, which will acerbate any bandwidth issues, especially when there is a lot of redundancy in the payload (for example, when multiple levels of Include are used to traverse associations in the one-to-many direction).

You can check for cases where your queries are returning excessively large payloads by accessing the underlying TSQL for the query by using ToTraceString and executing the store command in SQL Server Management Studio to see the payload size. In such cases you can try to reduce the number of Include statements in your query to just bring in the data you need. Or you may be able to break your query into a smaller sequence of subqueries, for example:

Before breaking the query:

```
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)
    {
        ...
    }
}
```

After breaking the query:

```
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)
    {
        ...
    }
}
```

This will work only on tracked queries, as we are making use of the ability the context has to perform identity resolution and association fixup automatically.

As with lazy loading, the tradeoff will be more queries for smaller payloads. You can also use projections of individual properties to explicitly select only the data you need from each entity, but you will not be loading entities in this case, and updates will not be supported.

8.2.3 Workaround to get lazy loading of properties

Entity Framework currently doesn't support lazy loading of scalar or complex properties. However, in cases where you have a table that includes a large object such as a BLOB, you can use table splitting to separate the large properties into a separate entity. For example, suppose you have a Product table that includes a varbinary photo column. If you don't frequently need to access this property in your queries, you can use table splitting to bring in only the parts of the entity that you normally need. The entity representing the product photo will only be loaded when you explicitly need it.

A good resource that shows how to enable table splitting is Gil Fink's "Table Splitting in Entity Framework" blog post: <http://blogs.microsoft.co.il/blogs/gilf/archive/2009/10/13/table-splitting-in-entity-framework.aspx>.

9 Other considerations

9.1 Server Garbage Collection

Some users might experience resource contention that limits the parallelism they are expecting when the Garbage Collector is not properly configured. Whenever EF is used in a multithreaded scenario, or in any application that resembles a server-side system, make sure to enable Server Garbage Collection. This is done via a simple setting in your application config file:

```
<?xmlversion="1.0" encoding="utf-8" ?>
<configuration>
        <runtime>
               <gcServer enabled="true" />
        </runtime>
</configuration>
```

This should decrease your thread contention and increase your throughput by up to 30% in CPU saturated scenarios. In general terms, you should always test how your application behaves using the classic Garbage Collection (which is better tuned for UI and client side scenarios) as well as the Server Garbage Collection.

9.2 AutoDetectChanges

As mentioned earlier, Entity Framework might show performance issues when the object cache has many entities.

Certain operations, such as Add, Remove, Find, Entry and SaveChanges, trigger calls to DetectChanges which might consume a large amount of CPU based on how large the object cache has become. The reason for this is that the object cache and the object state manager try to stay as synchronized as possible on each operation performed to a context so that the produced data is guaranteed to be correct under a wide array of scenarios.

It is generally a good practice to leave Entity Framework's automatic change detection enabled for the entire life of your application. If your scenario is being negatively affected by high CPU usage and your profiles indicate that the culprit is the call to DetectChanges, consider temporarily turning off AutoDetectChanges in the sensitive portion of your code:

```
try
{
    context.Configuration.AutoDetectChangesEnabled = false;
    var product = context.Products.Find(productId);
    ...
}
finally
{
    context.Configuration.AutoDetectChangesEnabled = true;
}
```

Before turning off AutoDetectChanges, it's good to understand that this might cause Entity Framework to lose its ability to track certain information about the changes that are taking place on the entities. If handled incorrectly, this might cause data inconsistency on your application. For more information on turning off AutoDetectChanges, read <http://blog.oneunicorn.com/2012/03/12/secrets-of-detectchanges-part-3-switching-off-automatic-detectchanges/>.

9.3 Context per request

Entity Framework's contexts are meant to be used as short-lived instances in order to provide the most optimal performance experience. Contexts are expected to be short lived and discarded, and as such have been implemented to be very lightweight and reutilize metadata whenever possible. In web scenarios it's important to keep this in mind and not have a context for more than the duration of a single request. Similarly, in non-web scenarios, context should be discarded based on your understanding of the different levels of caching in the Entity Framework. Generally speaking, one should avoid having a context instance throughout the life of the application, as well as contexts per thread and static contexts.

9.4 Database null semantics

Entity Framework by default will generate SQL code that has C# null comparison semantics. Consider the following example query:

```
            int? categoryId = 7;
            int? supplierId = 8;
            decimal? unitPrice = 0;
            short? unitsInStock = 100;
            short? unitsOnOrder = 20;
            short? reorderLevel = null;

            var q = from p incontext.Products
                    wherep.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();
```

In this example, we're comparing a number of nullable variables against nullable properties on the entity, such as SupplierID and UnitPrice. The generated SQL for this query will ask if the parameter value is the same as the column value, or if both the parameter and the column values are null. This will hide the way the database server handles nulls and will provide a consistent C# null experience across different database vendors. On the other hand, the generated code is a bit convoluted and may not perform well when the amount of comparisons in the where statement of the query grows to a large number.

One way to deal with this situation is by using database null semantics. Note that this might potentially behave differently to the C# null semantics since now Entity Framework will generate simpler SQL that exposes the way the database engine handles null values. Database null semantics can be activated per-context with one single configuration line against the context configuration:

```
                context.Configuration.UseDatabaseNullSemantics = true;
```

Small to medium sized queries will not display a perceptible performance improvement when using database null semantics, but the difference will become more noticeable on queries with a large number of potential null comparisons.

In the example query above, the performance difference was less than 2% in a microbenchmark running in a controlled environment.

9.5 Async

Entity Framework 6 introduced support of async operations when running on .NET 4.5 or later. For the most part, applications that have IO related contention will benefit the most from using asynchronous query and save operations. If your application does not suffer from IO contention, the use of async will, in the best cases, run synchronously and return the result in the same amount of time as a synchronous call, or in the worst case, simply defer execution to an asynchronous task and add extra time to the completion of your scenario.

For information on how asynchronous programming work that will help you deciding if async will improve the performance of your application visit http://msdn.microsoft.com/library/hh191443.aspx. For more information on the use of async operations on Entity Framework, see Async Query and Save.

9.6 NGEN

Entity Framework 6 does not come in the default installation of .NET framework. As such, the Entity Framework assemblies are not NGEN'd by default which means that all the Entity Framework code is subject to the same JIT'ing costs as any other MSIL assembly. This might degrade the F5 experience while developing and also the cold startup of your application in the production environments. In order to reduce the CPU and memory costs of JIT'ing it is advisable to NGEN the Entity Framework images as appropriate. For more information on how to improve the startup performance of Entity Framework 6 with NGEN, see Improving Startup Performance with NGen.

9.7 Code First versus EDMX

Entity Framework reasons about the impedance mismatch problem between object oriented programming and relational databases by having an in-memory representation of the conceptual model (the objects), the storage schema (the database) and a mapping between the two. This metadata is called an Entity Data Model, or EDM for short. From this EDM, Entity Framework will derive the views to roundtrip data from the objects in memory to the database and back.

When Entity Framework is used with an EDMX file that formally specifies the conceptual model, the storage schema, and the mapping, then the model loading stage only has to validate that the EDM is correct (for example, make sure that no mappings are missing), then generate the views, then validate the views and have this metadata ready for use. Only then can a query be executed or new data be saved to the data store.

The Code First approach is, at its heart, a sophisticated Entity Data Model generator. The Entity Framework has to produce an EDM from the provided code; it does so by analyzing the classes involved in the model, applying conventions and configuring the model via the Fluent API. After the EDM is built, the Entity Framework essentially behaves the same way as it would had an EDMX file been present in the project. Thus, building the model from Code First adds extra complexity that translates into a slower startup time for the Entity Framework when compared to having an EDMX. The cost is completely dependent on the size and complexity of the model that's being built.

When choosing to use EDMX versus Code First, it's important to know that the flexibility introduced by Code First increases the cost of building the model for the first time. If your application can withstand the cost of this first-time load then typically Code First will be the preferred way to go.

10 Investigating Performance

10.1 Using the Visual Studio Profiler

If you are having performance issues with the Entity Framework, you can use a profiler like the one built into Visual Studio to see where your application is spending its time. This is the tool we used to generate the pie charts in the "Exploring the Performance of the ADO.NET Entity Framework - Part 1" blog post (<http://blogs.msdn.com/b/adonet/archive/2008/02/04/exploring-the-performance-of-the-ado-net-entity-framework-part-1.aspx>) that show where Entity Framework spends its time during cold and warm queries.

The "Profiling Entity Framework using the Visual Studio 2010 Profiler" blog post written by the Data and Modeling Customer Advisory Team shows a real-world example of how they used the profiler to investigate a performance problem. <http://blogs.msdn.com/b/dmcat/archive/2010/04/30/profiling-entity-framework-using-the-visual-studio-2010-profiler.aspx>. This post was written for a windows application. If you need to profile a web application the Windows Performance Recorder (WPR) and Windows Performance Analyzer (WPA) tools may work better than working from Visual Studio. WPR and WPA are part of the Windows Performance Toolkit which is included with the Windows Assessment and Deployment Kit (http://www.microsoft.com/download/details.aspx?id=39982).

10.2 Application/Database profiling

Tools like the profiler built into Visual Studio tell you where your application is spending time. Another type of profiler is available that performs dynamic analysis of your running application, either in production or pre-production depending on needs, and looks for common pitfalls and anti-patterns of database access.

Two commercially available profilers are the Entity Framework Profiler (<http://efprof.com>) and ORMProfiler (<http://ormprofiler.com>).

If your application is an MVC application using Code First, you can use StackExchange's MiniProfiler. Scott Hanselman describes this tool in his blog at: <http://www.hanselman.com/blog/NuGetPackageOfTheWeek9ASPNETMiniProfilerFromStackExchangeRocksYourWorld.aspx>.

For more information on profiling your application's database activity, see Julie Lerman's MSDN Magazine article titled Profiling Database Activity in the Entity Framework.

10.3 Database logger

If you are using Entity Framework 6 also consider using the built-in logging functionality. The Database property of the context can be instructed to log its activity via a simple one-line configuration:

```
using (var context = newQueryComparison.DbC.NorthwindEntities())
{
    context.Database.Log = Console.WriteLine;
    var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
    q.ToList();
}
```

In this example the database activity will be logged to the console, but the Log property can be configured to call any Action<string> delegate.

If you want to enable database logging without recompiling, and you are using Entity Framework 6.1 or later, you can do so by adding an interceptor in the web.config or app.config file of your application.

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Path\To\My\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

For more information on how to add logging without recompiling go to <http://blog.oneunicorn.com/2014/02/09/ef-6-1-turning-on-logging-without-recompiling/>.

11 Appendix

11.1 A. Test Environment

This environment uses a 2-machine setup with the database on a separate machine from the client application. Machines are in the same rack, so network latency is relatively low, but more realistic than a single-machine environment.

11.1.1 App Server

11.1.1.1 Software Environment

- Entity Framework 4 Software Environment
  - OS Name: Windows Server 2008 R2 Enterprise SP1.
  - Visual Studio 2010 – Ultimate.
  - Visual Studio 2010 SP1 (only for some comparisons).
- Entity Framework 5 and 6 Software Environment
  - OS Name: Windows 8.1 Enterprise
  - Visual Studio 2013 – Ultimate.

11.1.1.2 Hardware Environment

- Dual Processor: Intel(R) Xeon(R) CPU L5520 W3530 @ 2.27GHz, 2261 Mhz8 GHz, 4 Core(s), 84 Logical Processor(s).
- 2412 GB RamRAM.
- 136 GB SCSI250GB SATA 7200 rpm 3GB/s drive split into 4 partitions.

11.1.2 DB server

11.1.2.1 Software Environment

- OS Name: Windows Server 2008 R28.1 Enterprise SP1.
- SQL Server 2008 R22012.

11.1.2.2 Hardware Environment

- Single Processor: Intel(R) Xeon(R) CPU L5520 @ 2.27GHz, 2261 MhzES-1620 0 @ 3.60GHz, 4 Core(s), 8 Logical Processor(s).
- 824 GB RamRAM.
- 465 GB ATA500GB SATA 7200 rpm 6GB/s drive split into 4 partitions.

11.2 B. Query performance comparison tests

The Northwind model was used to execute these tests. It was generated from the database using the Entity Framework designer. Then, the following code was used to compare the performance of the query execution options:

```
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 entityConnectionStr =
@"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=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(entityConnectionStr))
            {
                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();
            }
        }
    }
}
```

11.3 C. Navision Model

The Navision database is a large database used to demo Microsoft Dynamics – NAV. The generated conceptual model contains 1005 entity sets and 4227 association sets. The model used in the test is “flat” – no inheritance has been added to it.

11.3.1 Queries used for Navision tests

The queries list used with the Navision model contains 3 categories of Entity SQL queries:

11.3.1.1 Lookup

A simple lookup query with no aggregations

- Count: 16232
- Example:

```
  <Query complexity="Lookup">
    <CommandText>Select value distinct top(4) e.Idle_Time From NavisionFKContext.Session as e</CommandText>
  </Query>
```

11.3.1.2 SingleAggregating

A normal BI query with multiple aggregations, but no subtotals (single query)

- Count: 2313
- Example:

```
  <Query complexity="SingleAggregating">
    <CommandText>NavisionFK.MDF_SessionLogin_Time_Max()</CommandText>
  </Query>
```

Where MDF_SessionLogin_Time_Max() is defined in the model as:

```
  <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

A BI query with aggregations and subtotals (via union all)

- Count: 178
- Example:

```
  <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>
```

Improving startup performance with NGen

9/13/2018 • 4 minutes to read • Edit Online

> **NOTE**
>
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

The .NET Framework supports the generation of native images for managed applications and libraries as a way to help applications start faster and also in some cases use less memory. Native images are created by translating managed code assemblies into files containing native machine instructions before the application is executed, relieving the .NET JIT (Just-In-Time) compiler from having to generate the native instructions at application runtime.

Prior to version 6, the EF runtime's core libraries were part of the .NET Framework and native images were generated automatically for them. Starting with version 6 the whole EF runtime has been combined into the EntityFramework NuGet package. Native images have to now be generated using the NGen.exe command line tool to obtain similar results.

Empirical observations show that native images of the EF runtime assemblies can cut between 1 and 3 seconds of application startup time.

How to use NGen.exe

The most basic function of the NGen.exe tool is to "install" (that is, to create and persist to disk) native images for an assembly and all its direct dependencies. Here is how you can achieve that:

1. Open a Command Prompt window as an administrator
2. Change the current working directory to the location of the assemblies you want to generate native images for:

```
cd <*Assemblies location*>
```

3. Depending on your operating system and the application's configuration you might need to generate native images for 32 bit architecture, 64 bit architecture or for both.

   For 32 bit run:

```
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\ngen install <Assembly name>
```

```
For 64 bit run:
```

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\ngen install <Assembly name>
```

> **TIP**
>
> Generating native images for the wrong architecture is a very common mistake. When in doubt you can simply generate native images for all the architectures that apply to the operating system installed in the machine.

NGen.exe also supports other functions such as uninstalling and displaying the installed native images, queuing the generation of multiple images, etc. For more details of usage read the NGen.exe documentation.

When to use NGen.exe

When it comes to decide which assemblies to generate native images for in an application based on EF version 6 or greater, you should consider the following options:

- **The main EF runtime assembly, EntityFramework.dll**: A typical EF based application executes a significant amount of code from this assembly on startup or on its first access to the database. Consequently, creating native images of this assembly will produce the biggest gains in startup performance.
- **Any EF provider assembly used by your application**: Startup time can also benefit slightly from generating native images of these. For example, if the application uses the EF provider for SQL Server you will want to generate a native image for EntityFramework.SqlServer.dll.
- **Your application's assemblies and other dependencies**: The NGen.exe documentation covers general criteria for choosing which assemblies to generate native images for and the impact of native images on security, advanced options such as "hard binding", scenarios such as using native images in debugging and profiling scenarios, etc.

> **TIP**
>
> Make sure you carefully measure the impact of using native images on both the startup performance and the overall performance of your application and compare them against actual requirements. While native images will generally help improve startup up performance and in some cases reduce memory usage, not all scenarios will benefit equally. For instance, on steady state execution (that is, once all the methods being used by the application have been invoked at least once) code generated by the JIT compiler can in fact yield slightly better performance than native images.

Using NGen.exe in a development machine

During development the .NET JIT compiler will offer the best overall tradeoff for code that is changing frequently. Generating native images for compiled dependencies such as the EF runtime assemblies can help accelerate development and testing by cutting a few seconds out at the beginning of each execution.

A good place to find the EF runtime assemblies is the NuGet package location for the solution. For example, for an application using EF 6.0.2 with SQL Server and targeting .NET 4.5 or greater you can type the following in a Command Prompt window (remember to open it as an administrator):

```
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**
>
> This takes advantage of the fact that installing the native images for EF provider for SQL Server will also by default install the native images for the main EF runtime assembly. This works because NGen.exe can detect that EntityFramework.dll is a direct dependency of the EntityFramework.SqlServer.dll assembly located in the same directory.

Creating native images during setup

The WiX Toolkit supports queuing the generation of native images for managed assemblies during setup, as explained in this how-to guide. Another alternative is to create a custom setup task that execute the NGen.exe command.

Verifying that native images are being used for EF

You can verify that a specific application is using a native assembly by looking for loaded assemblies that have the extension ".ni.dll" or ".ni.exe". For example, a native image for the EF's main runtime assembly will be called EntityFramework.ni.dll. An easy way to inspect the loaded .NET assemblies of a process is to use Process Explorer.

Other things to be aware of

Creating a native image of an assembly should not be confused with registering the assembly in the GAC (Global Assembly Cache). NGen.exe allows creating images of assemblies that are not in the GAC, and in fact, multiple applications that use a specific version of EF can share the same native image. While Windows 8 can automatically create native images for assemblies placed in the GAC, the EF runtime is optimized to be deployed alongside your application and we do not recommend registering it in the GAC as this has a negative impact on assembly resolution and servicing your applications among other aspects.

Pre-generated mapping views

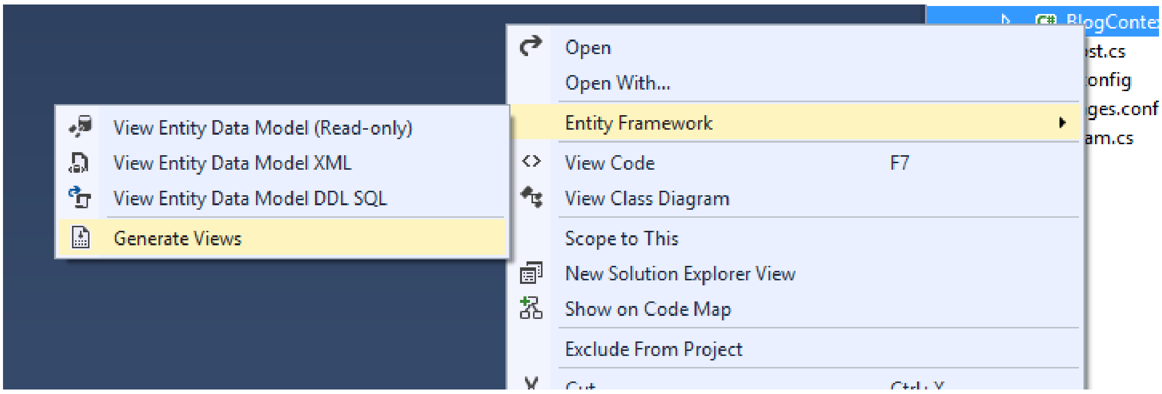
9/30/2018 • 4 minutes to read • Edit Online

Before the Entity Framework can execute a query or save changes to the data source, it must generate a set of mapping views to access the database. These mapping views are a set of Entity SQL statement that represent the database in an abstract way, and are part of the metadata which is cached per application domain. If you create multiple instances of the same context in the same application domain, they will reuse mapping views from the cached metadata rather than regenerating them. Because mapping view generation is a significant part of the overall cost of executing the first query, the Entity Framework enables you to pre-generate mapping views and include them in the compiled project. For more information, see Performance Considerations (Entity Framework).

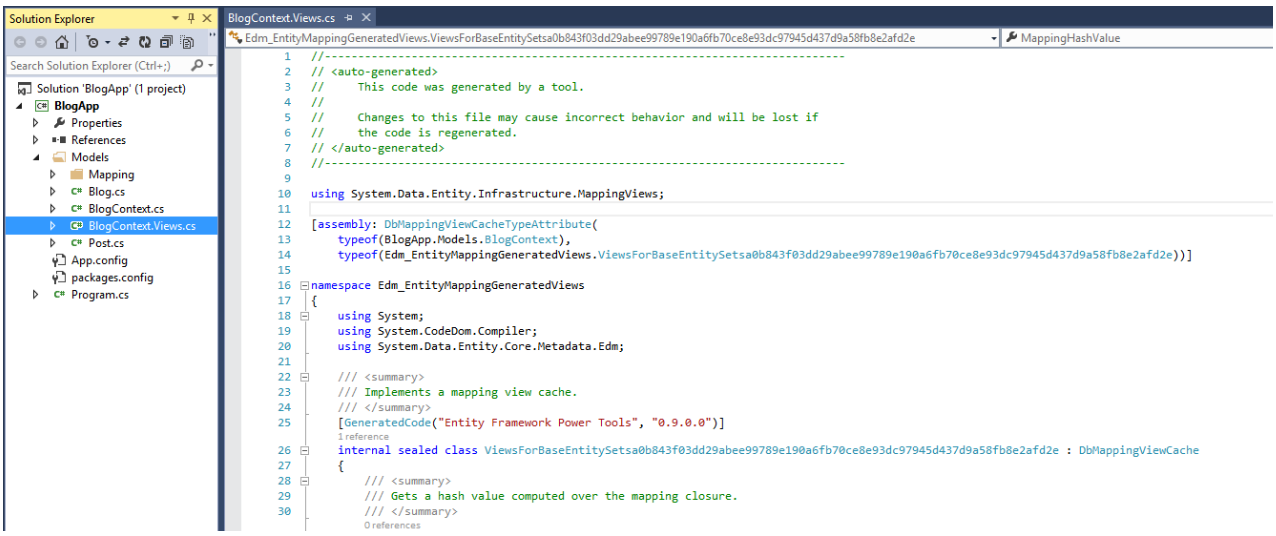
Generating Mapping Views with the EF Power Tools Community Edition

The easiest way to pre-generate views is to use the EF Power Tools Community Edition. Once you have the Power Tools installed you will have a menu option to Generate Views, as below.

- For **Code First** models right-click on the code file that contains your DbContext class.
- For **EF Designer** models right-click on your EDMX file.



Once the process is finished you will have a class similar to the following generated



Now when you run your application EF will use this class to load views as required. If your model changes and you do not re-generate this class then EF will throw an exception.

Generating Mapping Views from Code - EF6 Onwards

The other way to generate views is to use the APIs that EF provides. When using this method you have the freedom to serialize the views however you like, but you also need to load the views yourself.

> **NOTE**
>
> **EF6 Onwards Only** - The APIs shown in this section were introduced in Entity Framework 6. If you are using an earlier version this information does not apply.

Generating Views

The APIs to generate views are on the System.Data.Entity.Core.Mapping.StorageMappingItemCollection class. You can retrieve a StorageMappingCollection for a Context by using the MetadataWorkspace of an ObjectContext. If you are using the newer DbContext API then you can access this by using the IObjectContextAdapter like below, in this code we have an instance of your derived DbContext called dbContext:

```
var objectContext = ((IObjectContextAdapter) dbContext).ObjectContext;
var  mappingCollection = (StorageMappingItemCollection)objectContext.MetadataWorkspace
                                                                    .GetItemCollection(DataSpace.CSSpace);
```

Once you have the StorageMappingItemCollection then you can get access to the GenerateViews and ComputeMappingHashValue methods.

```
public Dictionary\<EntitySetBase, DbMappingView> GenerateViews(IList<EdmSchemaError> errors)
public string ComputeMappingHashValue()
```

The first method creates a dictionary with an entry for each view in the container mapping. The second method computes a hash value for the single container mapping and is used at runtime to validate that the model has not changed since the views were pre-generated. Overrides of the two methods are provided for complex scenarios involving multiple container mappings.

When generating views you will call the GenerateViews method and then write out the resulting EntitySetBase and DbMappingView. You will also need to store the hash generated by the ComputeMappingHashValue method.

Loading Views

In order to load the views generated by the GenerateViews method, you can provide EF with a class that inherits from the DbMappingViewCache abstract class. DbMappingViewCache specifies two methods that you must implement:

```
public abstract string MappingHashValue { get; }
public abstract DbMappingView GetView(EntitySetBase extent);
```

The MappingHashValue property must return the hash generated by the ComputeMappingHashValue method. When EF is going to ask for views it will first generate and compare the hash value of the model with the hash returned by this property. If they do not match then EF will throw an EntityCommandCompilationException exception.

The GetView method will accept an EntitySetBase and you need to return a DbMappingVIew containing the EntitySql that was generated for that was associated with the given EntitySetBase in the dictionary generated by the GenerateViews method. If EF asks for a view that you do not have then GetView should return null.

The following is an extract from the DbMappingViewCache that is generated with the Power Tools as described above, in it we see one way to store and retrieve the EntitySql required.

```
    public override string MappingHashValue
    {
        get { return "a0b843f03dd29abee99789e190a6fb70ce8e93dc97945d437d9a58fb8e2afd2e"; }
    }

    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");
    }
```

To have EF use your DbMappingViewCache you add use the DbMappingViewCacheTypeAttribute, specifying the context that it was created for. In the code below we associate the BlogContext with the MyMappingViewCache class.

```
    [assembly: DbMappingViewCacheType(typeof(BlogContext), typeof(MyMappingViewCache))]
```

For more complex scenarios, mapping view cache instances can be provided by specifying a mapping view cache factory. This can be done by implementing the abstract class System.Data.Entity.Infrastructure.MappingViews.DbMappingViewCacheFactory. The instance of the mapping view cache factory that is used can be retrieved or set using the StorageMappingItemCollection.MappingViewCacheFactoryproperty.

Entity Framework 6 Providers

10/25/2018 • 3 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

The Entity Framework is now being developed under an open-source license and EF6 and above will not be shipped as part of the .NET Framework. This has many advantages but also requires that EF providers be rebuilt against the EF6 assemblies. This means that EF providers for EF5 and below will not work with EF6 until they have been rebuilt.

Which providers are available for EF6?

Providers we are aware of that have been rebuilt for EF6 include:

- **Microsoft SQL Server provider**
  - Built from the Entity Framework open source code base
  - Shipped as part of the EntityFramework NuGet package
- **Microsoft SQL Server Compact Edition provider**
  - Built from the Entity Framework open source code base
  - Shipped in the EntityFramework.SqlServerCompact NuGet package
- **Devart dotConnect Data Providers**
  - There are third-party providers from Devart for a variety of databases including Oracle, MySQL, PostgreSQL, SQLite, Salesforce, DB2, and SQL Server
- **CData Software providers**
  - There are third-party providers from CData Software for a variety of data stores including Salesforce, Azure Table Storage, MySql, and many more
- **Firebird provider**
  - Available as a NuGet Package
- **Visual Fox Pro provider**
  - Available as a NuGet package
- **MySQL**
  - MySQL Connector/Net
- **PostgreSQL**
  - Npgsql is available as a NuGet package
- **Oracle**
  - ODP.NET is available as a NuGet package

Note that inclusion in this list does not indicate the level of functionality or support for a given provider, only that a build for EF6 has been made available.

Registering EF providers

Starting with Entity Framework 6 EF providers can be registered using either code-based configuration or in the application's config file.

Config file registration

Registration of the EF provider in app.config or web.config has the following format:

```
<entityFramework>
   <providers>
     <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
   </providers>
</entityFramework>
```

Note that often if the EF provider is installed from NuGet, then the NuGet package will automatically add this registration to the config file. If you install the NuGet package into a project that is not the startup project for your app, then you may need to copy the registration into the config file for your startup project.

The “invariantName” in this registration is the same invariant name used to identify an ADO.NET provider. This can be found as the “invariant” attribute in a DbProviderFactories registration and as the “providerName” attribute in a connection string registration. The invariant name to use should also be included in documentation for the provider. Examples of invariant names are “System.Data.SqlClient” for SQL Server and “System.Data.SqlServerCe.4.0” for SQL Server Compact.

The “type” in this registration is the assembly-qualified name of the provider type that derives from “System.Data.Entity.Core.Common.DbProviderServices”. For example, the string to use for SQL Compact is “System.Data.Entity.SqlServerCompact.SqlCeProviderServices, EntityFramework.SqlServerCompact”. The type to use here should be included in documentation for the provider.

Code-based registration

Starting with Entity Framework 6 application-wide configuration for EF can be specified in code. For full details see *Entity Framework Code-Based Configuration*. The normal way to register an EF provider using code-based configuration is to create a new class that derives from System.Data.Entity.DbConfiguration and place it in the same assembly as your DbContext class. Your DbConfiguration class should then register the provider in its constructor. For example, to register the SQL Compact provider the DbConfiguration class looks like this:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices(
            SqlCeProviderServices.ProviderInvariantName,
            SqlCeProviderServices.Instance);
    }
}
```

In this code “SqlCeProviderServices.ProviderInvariantName” is a convenience for the SQL Server Compact provider invariant name string (“System.Data.SqlServerCe.4.0”) and SqlCeProviderServices.Instance returns the singleton instance of the SQL Compact EF provider.

What if the provider I need isn’t available?

If the provider is available for previous versions of EF, then we encourage you to contact the owner of the provider and ask them to create an EF6 version. You should include a reference to the documentation for the EF6 provider model.

Can I write a provider myself?

It is certainly possible to create an EF provider yourself although it should not be considered a trivial undertaking. The the link above about the EF6 provider model is a good place to start. You may also find it useful to use the code for the SQL Server and SQL CE provider included in the EF open source codebase as a starting point or for reference.

Note that starting with EF6 the EF provider is less tightly coupled to the underlying ADO.NET provider. This makes it easier to write an EF provider without needing to write or wrap the ADO.NET classes.

The Entity Framework 6 provider model

10/25/2018 • 15 minutes to read • Edit Online

The Entity Framework provider model allows Entity Framework to be used with different types of database server. For example, one provider can be plugged in to allow EF to be used against Microsoft SQL Server, while another provider can be plugged into to allow EF to be used against Microsoft SQL Server Compact Edition. The providers for EF6 that we are aware of can be found on the Entity Framework providers page.

Certain changes were required to the way EF interacts with providers to allow EF to be released under an open source license. These changes require rebuilding of EF providers against the EF6 assemblies together with new mechanisms for registration of the provider.

Rebuilding

With EF6 the core code that was previously part of the .NET Framework is now being shipped as out-of-band (OOB) assemblies. Details on how to build applications against EF6 can be found on the Updating applications for EF6 page. Providers will also need to be rebuilt using these instructions.

Provider types overview

An EF provider is really a collection of provider-specific services defined by CLR types that these services extend from (for a base class) or implement (for an interface). Two of these services are fundamental and necessary for EF to function at all. Others are optional and only need to be implemented if specific functionality is required and/or the default implementations of these services does not work for the specific database server being targeted.

Fundamental provider types

DbProviderFactory

EF depends on having a type derived from System.Data.Common.DbProviderFactory for performing all low-level database access. DbProviderFactory is not actually part of EF but is instead a class in the .NET Framework that serves an entry point for ADO.NET providers that can be used by EF, other O/RMs or directly by an application to obtain instances of connections, commands, parameters and other ADO.NET abstractions in a provider agnostic way. More information about DbProviderFactory an be found in the MSDN documentation for ADO.NET.

DbProviderServices

EF depends on having a type derived from DbProviderServices for providing additional functionality needed by EF on top of the functionality already provided by the ADO.NET provider. In older versions of EF the DbProviderServices class was part of the .NET Framework and was found in the System.Data.Common namespace. Starting with EF6 this class is now part of EntityFramework.dll and is in the System.Data.Entity.Core.Common namespace.

More details about the fundamental functionality of a DbProviderServices implementation can be found on MSDN. However, note that as of the time of writing this information is not updated for EF6 although most of the concepts are still valid. The SQL Server and SQL Server Compact implementations of DbProviderServices are also checked into to the open-source codebase and can serve as useful references for other implementations.

In older versions of EF the DbProviderServices implementation to use was obtained directly from an ADO.NET provider. This was done by casting DbProviderFactory to IServiceProvider and calling the GetService method.

This tightly coupled the EF provider to the DbProviderFactory. This coupling blocked EF from being moved out of the .NET Framework and therefore for EF6 this tight coupling has been removed and an implementation of DbProviderServices is now registered directly in the application's configuration file or in code-based configuration as described in more detail the *Registering DbProviderServices* section below.

Additional services

In addition to the fundamental services described above there are also many other services used by EF which are either always or sometimes provider-specific. Default provider-specific implementations of these services can be supplied by a DbProviderServices implementation. Applications can also override the implementations of these services, or provide implementations when a DbProviderServices type does not provide a default. This is described in more detail in the *Resolving additional services* section below.

The additional service types that a provider may be of interest to a provider are listed below. More details about each of these service types can be found in the API documentation.

IDbExecutionStrategy

This is an optional service that allows a provider to implement retries or other behavior when queries and commands are executed against the database. If no implementation is provided, then EF will simply execute the commands and propagate any exceptions thrown. For SQL Server this service is used to provide a retry policy which is especially useful when running against cloud-based database servers such as SQL Azure.

IDbConnectionFactory

This is an optional service that allows a provider to create DbConnection objects by convention when given only a database name. Note that while this service can be resolved by a DbProviderServices implementation it has been present since EF 4.1 and can also be explicitly set in either the config file or in code. The provider will only get a chance to resolve this service if it registered as the default provider (see *The default provider* below) and if a default connection factory has not been set elsewhere.

DbSpatialServices

This is an optional services that allows a provider to add support for geography and geometry spatial types. An implementation of this service must be supplied in order for an application to use EF with spatial types. DbSptialServices is asked for in two ways. First, provider-specific spatial services are requested using a DbProviderInfo object (which contains invariant name and manifest token) as key. Second, DbSpatialServices can be asked for with no key. This is used to resolve the "global spatial provider" that is used when creating stand-alone DbGeography or DbGeometry types.

MigrationSqlGenerator

This is an optional service that allows EF Migrations to be used for the generation of SQL used in creating and modifying database schemas by Code First. An implementation is required in order to support Migrations. If an implementation is provided then it will also be used when databases are created using database initializers or the Database.Create method.

Func<DbConnection, string, HistoryContextFactory>

This is an optional service that allows a provider to configure the mapping of the HistoryContext to the `__MigrationHistory` table used by EF Migrations. The HistoryContext is a Code First DbContext and can be configured using the normal fluent API to change things like the name of the table and the column mapping specifications. The default implementation of this service returned by EF for all providers may work for a given database server if all the default table and column mappings are supported by that provider. In such a case the provider does not need to supply an implementation of this service.

IDbProviderFactoryResolver

This is an optional service for obtaining the correct DbProviderFactory from a given DbConnection object. The default implementation of this service returned by EF for all providers is intended to work for all providers.

However, when running on .NET 4, the DbProviderFactory is not publicly accessible from one if its DbConnections. Therefore, EF uses some heuristics to search the registered providers to find a match. It is possible that for some providers these heuristics will fail and in such situations the provider should supply a new implementation.

Registering DbProviderServices

The DbProviderServices implementation to use can be registered either in the application's configuration file (app.config or web.config) or using code-based configuration. In either case the registration uses the provider's "invariant name" as a key. This allows multiple providers to be registered and used in a single application. The invariant name used for EF registrations is the same as the invariant name used for ADO.NET provider registration and connection strings. For example, for SQL Server the invariant name "System.Data.SqlClient" is used.

Config file registration

The DbProviderServices type to use is registered as a provider element in the providers list of the entityFramework section of the application's config file. For example:

```
<entityFramework>
  <providers>
    <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
  </providers>
</entityFramework>
```

The *type* string must be the assembly-qualified type name of the DbProviderServices implementation to use.

Code-based registration

Starting with EF6 providers can also be registered using code. This allows an EF provider to be used without any change to the application's configuration file. To use code-based configuration an application should create a DbConfiguration class as described in the code-based configuration documentation. The constructor of the DbConfiguration class should then call SetProviderServices to register the EF provider. For example:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices("My.New.Provider", new MyProviderServices());
    }
}
```

Resolving additional services

As mentioned above in the *Provider types overview* section, a DbProviderServices class can also be used to resolve additional services. This is possible because DbProviderServices implements IDbDependencyResolver and each registered DbProviderServices type is added as a "default resolver". The IDbDpendencyResolver mechanism is described in more detail in Dependency Resolution. However, it is not necessary to understand all the concepts in this specification to resolve additional services in a provider.

The most common way for a provider to resolve additional services is to call DbProviderServices.AddDependencyResolver for each service in the constructor of the DbProviderServices class. For example, SqlProviderServices (the EF provider for SQL Server) has code similar to this for initialization:

```
private SqlProviderServices()
{
    AddDependencyResolver(new SingletonDependencyResolver<IDbConnectionFactory>(
        new SqlConnectionFactory()));

    AddDependencyResolver(new ExecutionStrategyResolver<DefaultSqlExecutionStrategy>(
        "System.data.SqlClient", null, () => new DefaultSqlExecutionStrategy()));

    AddDependencyResolver(new SingletonDependencyResolver<Func<MigrationSqlGenerator>>(
        () => new SqlServerMigrationSqlGenerator(), "System.data.SqlClient"));

    AddDependencyResolver(new SingletonDependencyResolver<DbSpatialServices>(
        SqlSpatialServices.Instance,
        k =>
        {
            var asSpatialKey = k as DbProviderInfo;
            return asSpatialKey == null
                || asSpatialKey.ProviderInvariantName == ProviderInvariantName;
        }));
}
```

This constructor uses the following helper classes:

- SingletonDependencyResolver: provides a simple way to resolve Singleton services—that is, services for which the same instance is returned each time that GetService is called. Transient services are often registered as a singleton factory that will be used to create transient instances on demand.
- ExecutionStrategyResolver: a resolver specific to returning IExecutionStrategy implementations.

Instead of using DbProviderServices.AddDependencyResolver it is also possible to override DbProviderServices.GetService and resolve additional services directly. This method will be called when EF needs a service defined by a certain type and, in some cases, for a given key. The method should return the service if it can, or return null to opt-out of returning the service and instead allow another class to resolve it. For example, to resolve the default connection factory the code in GetService might look something like this:

```
public override object GetService(Type type, object key)
{
    if (type == typeof(IDbConnectionFactory))
    {
        return new SqlConnectionFactory();
    }
    return null;
}
```

Registration order

When multiple DbProviderServices implementations are registered in an application’s config file they will be added as secondary resolvers in the order that they are listed. Since resolvers are always added to the top of the secondary resolver chain this means that the provider at the end of the list will get a chance to resolve dependencies before the others. (This can seem a little counter-intuitive at first, but it makes sense if you imagine taking each provider out of the list and stacking it on top of the existing providers.)

This ordering usually doesn’t matter because most provider services are provider-specific and keyed by provider invariant name. However, for services that are not keyed by provider invariant name or some other provider-specific key the service will be resolved based on this ordering. For example, if it is not explicitly set differently somewhere else, then the default connection factory will come from the topmost provider in the chain.

Additional config file registrations

It is possible to explicitly register some of the additional provider services described above directly in an application’s config file. When this is done the registration in the config file will be used instead of anything returned by the GetService method of the DbProviderServices implementation.

Registering the default connection factory

Starting with EF5 the EntityFramework NuGet package automatically registered either the SQL Express connection factory or the LocalDb connection factory in the config file.

For example:

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework" >
</entityFramework>
```

The *type* is the assembly-qualified type name for the default connection factory, which must implement IDbConnectionFactory.

It is recommended that a provider NuGet package set the default connection factory in this way when installed. See *NuGet Packages for providers* below.

Additional EF6 provider changes

Spatial provider changes

Providers that support spatial types must now implement some additional methods on classes deriving from DbSpatialDataReader:

- `public abstract bool IsGeographyColumn(int ordinal)`
- `public abstract bool IsGeometryColumn(int ordinal)`

There are also new asynchronous versions of existing methods that are recommended to be overridden as the default implementations delegate to the synchronous methods and therefore do not execute asynchronously:

- `public virtual Task<DbGeography> GetGeographyAsync(int ordinal, CancellationToken cancellationToken)`
- `public virtual Task<DbGeometry> GetGeometryAsync(int ordinal, CancellationToken cancellationToken)`

Native support for Enumerable.Contains

EF6 introduces a new expression type, DbInExpression, which was added to address performance issues around use of Enumerable.Contains in LINQ queries. The DbProviderManifest class has a new virtual method, SupportsInExpression, which is called by EF to determine if a provider handles the new expression type. For compatibility with existing provider implementations the method returns false. To benefit from this improvement, an EF6 provider can add code to handle DbInExpression and override SupportsInExpression to return true. An instance of DbInExpression can be created by calling the DbExpressionBuilder.In method. A DbInExpression instance is composed of a DbExpression, usually representing a table column, and a list of DbConstantExpression to test for a match.

NuGet packages for providers

One way to make an EF6 provider available is to release it as a NuGet package. Using a NuGet package has the following advantages:

- It is easy to use NuGet to add the provider registration to the application’s config file
- Additional changes can be made to the config file to set the default connection factory so that connections made by convention will use the registered provider
- NuGet handles adding binding redirects so that the EF6 provider should continue to work even after a new EF package is released

An example of this is the EntityFramework.SqlServerCompact package which is included in the open source codebase. This package provides a good template for creating EF provider NuGet packages.

PowerShell commands

When the EntityFramework NuGet package is installed it registers a PowerShell module that contains two commands that are very useful for provider packages:

- Add-EFProvider adds a new entity for the provider in the target project's configuration file and makes sure it is at the end of the list of registered providers.
- Add-EFDefaultConnectionFactory either adds or updates the defaultConnectionFactory registration in the target project's configuration file.

Both these commands take care of adding an entityFramework section to the config file and adding a providers collection if necessary.

It is intended that these commands be called from the install.ps1 NuGet script. For example, install.ps1 for the SQL Compact provider looks similar to this:

```
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>
```

More information about these commands can be obtained by using get-help in the Package Manager Console window.

Wrapping providers

A wrapping provider is an EF and/or ADO.NET provider that wraps an existing provider to extend it with other functionality such as profiling or tracing capabilities. Wrapping providers can be registered in the normal way, but it is often more convenient to setup the wrapping provider at runtime by intercepting the resolution of provider-related services. The static event OnLockingConfiguration on the DbConfiguration class can be used to do this.

OnLockingConfiguration is called after EF has determined where all EF configuration for the app domain will be obtained from but before it is locked for use. At app startup (before EF is used) the app should register an event handler for this event. (We are considering adding support for registering this handler in the config file but this is not yet supported.) The event handler should then make a call to ReplaceService for every service that needs to be wrapped.

For example, to wrap IDbConnectionFactory and DbProviderService, a handler something like this should be registered:

```
DbConfiguration.OnLockingConfiguration +=
    (_, a) =>
    {
        a.ReplaceService<DbProviderServices>(
            (s, k) => new MyWrappedProviderServices(s));

        a.ReplaceService<IDbConnectionFactory>(
            (s, k) => new MyWrappedConnectionFactory(s));
    };
```

The service that has been resolved and should now be wrapped together with the key that was used to resolve the service are passed to the handler. The handler can then wrap this service and replace the returned service with the wrapped version.

Resolving a DbProviderFactory with EF

DbProviderFactory is one of the fundamental provider types needed by EF as described in the *Provider types overview* section above. As already mentioned, It is not an EF type and registration is usually not part of EF configuration, but is instead the normal ADO.NET provider registration in the machine.config file and/or application's config file.

Despite this EF still uses its normal dependency resolution mechanism when looking for a DbProviderFactory to use. The default resolver uses the normal ADO.NET registration in the config files and so this is usually transparent. But because of the normal dependency resolution mechanism is used it means that an IDbDependencyResolver can be used to resolve a DbProviderFactory even when normal ADO.NET registration has not been done.

Resolving DbProviderFactory in this way has several implications:

- An application using code-based configuration can add calls in their DbConfiguration class to register the appropriate DbProviderFactory. This is especially useful for applications that do not want to (or cannot) make use of any file-based configuration at all.
- The service can be wrapped or replaced using ReplaceService as described in the *Wrapping providers* section above
- Theoretically, a DbProviderServices implementation could resolve a DbProviderFactory.

The important point to note about doing any of these things is that they will only affect the lookup of DbProviderFactory by EF. Other non-EF code may still expect the ADO.NET provider to be registered in the normal way and may fail if the registration is not found. For this reason it is normally better for a DbProviderFactory to be registered in the normal ADO.NET way.

Related services

If EF is used to resolve a DbProviderFactory, then it should also resolve the IProviderInvariantName and IDbProviderFactoryResolver services.

IProviderInvariantName is a service that is used to determine a provider invariant name for a given type of DbProviderFactory. The default implementation of this service uses the ADO.NET provider registration. This means that if the ADO.NET provider is not registered in the normal way because DbProviderFactory is being resolved by EF, then it will also be necessary to resolve this service. Note that a resolver for this service is automatically added when using the DbConfiguration.SetProviderFactory method.

As described in the *Provider types overview* section above, the IDbProviderFactoryResolver is used to obtain the correct DbProviderFactory from a given DbConnection object. The default implementation of this service when running on .NET 4 uses the ADO.NET provider registration. This means that if the ADO.NET provider is not registered in the normal way because DbProviderFactory is being resolved by EF, then it will also be necessary to resolve this service.

Provider Support for Spatial Types

10/25/2018 • 2 minutes to read • Edit Online

Entity Framework supports working with spatial data through the DbGeography or DbGeometry classes. These classes rely on database-specific functionality offered by the Entity Framework provider. Not all providers support spatial data and those that do may have additional prerequisites such as the installation of spatial type assemblies. More information about provider support for spatial types is provided below.

Additional information on how to use spatial types in an application can be found in two walkthroughs, one for Code First, the other for Database First or Model First:

- Spatial Data Types in Code First
- Spatial Data Types in EF Designer

EF releases that support spatial types

Support for spatial types was introduced in EF5. However, in EF5 spatial types are only supported when the application targets and runs on .NET 4.5.

Starting with EF6 spatial types are supported for applications targeting both .NET 4 and .NET 4.5.

EF providers that support spatial types

EF5

The Entity Framework providers for EF5 that we are aware of that support spatial types are:

- Microsoft SQL Server provider
  - This provider is shipped as part of EF5.
  - This provider depends on some additional low-level libraries that may need to be installed—see below for details.
- Devart dotConnect for Oracle
  - This is a third-party provider from Devart.

If you know of an EF5 provider that supports spatial types then please get in contact and we will be happy to add it to this list.

EF6

The Entity Framework providers for EF6 that we are aware of that support spatial types are:

- Microsoft SQL Server provider
  - This provider is shipped as part of EF6.
  - This provider depends on some additional low-level libraries that may need to be installed—see below for details.
- Devart dotConnect for Oracle
  - This is a third-party provider from Devart.

If you know of an EF6 provider that supports spatial types then please get in contact and we will be happy to add it to this list.

Prerequisites for spatial types with Microsoft SQL Server

SQL Server spatial support depends on the low-level, SQL Server-specific types SqlGeography and SqlGeometry. These types live in Microsoft.SqlServer.Types.dll assembly, and this assembly is not shipped as part of EF or as part of the .NET Framework.

When Visual Studio is installed it will often also install a version of SQL Server, and this will include installation of the Microsoft.SqlServer.Types.dll.

If SQL Server is not installed on the machine where you want to use spatial types, or if spatial types were excluded from the SQL Server installation, then you will need to install them manually. The types can be installed using `SQLSysClrTypes.msi`, which is part of Microsoft SQL Server Feature Pack. Spatial types are SQL Server version-specific, so we recommend search for "SQL Server Feature Pack" in the Microsoft Download Center, then select and download the option that corresponds to the version of SQL Server you will use.

Working with proxies

9/13/2018 • 2 minutes to read • Edit Online

When creating instances of POCO entity types, Entity Framework often creates instances of a dynamically generated derived type that acts as a proxy for the entity. This proxy overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed. For example, this mechanism is used to support lazy loading of relationships. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Disabling proxy creation

Sometimes it is useful to prevent Entity Framework from creating proxy instances. For example, serializing non-proxy instances is considerably easier than serializing proxy instances. Proxy creation can be turned off by clearing the ProxyCreationEnabled flag. One place you could do this is in the constructor of your context. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.ProxyCreationEnabled = false;
    }

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

Note that the EF will not create proxies for types where there is nothing for the proxy to do. This means that you can also avoid proxies by having types that are sealed and/or have no virtual properties.

Explicitly creating an instance of a proxy

A proxy instance will not be created if you create an instance of an entity using the new operator. This may not be a problem, but if you need to create a proxy instance (for example, so that lazy loading or proxy change tracking will work) then you can do so using the Create method of DbSet. For example:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Create();
}
```

The generic version of Create can be used if you want to create an instance of a derived entity type. For example:

```
using (var context = new BloggingContext())
{
    var admin = context.Users.Create<Administrator>();
}
```

Note that the Create method does not add or attach the created entity to the context.

Note that the Create method will just create an instance of the entity type itself if creating a proxy type for the entity would have no value because it would not do anything. For example, if the entity type is sealed and/or has no virtual properties then Create will just create an instance of the entity type.

Getting the actual entity type from a proxy type

Proxy types have names that look something like this:

System.Data.Entity.DynamicProxies.Blog_5E43C6C196972BF0754973E48C9C941092D86818CD94005E9A759B70BF6E48E6

You can find the entity type for this proxy type using the GetObjectType method from ObjectContext. For example:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    var entityType = ObjectContext.GetObjectType(blog.GetType());
}
```

Note that if the type passed to GetObjectType is an instance of an entity type that is not a proxy type then the type of entity is still returned. This means you can always use this method to get the actual entity type without any other checking to see if the type is a proxy type or not.

Testing with a mocking framework

10/6/2018 • 8 minutes to read • Edit Online

> **NOTE**
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

When writing tests for your application it is often desirable to avoid hitting the database. Entity Framework allows you to achieve this by creating a context – with behavior defined by your tests – that makes use of in-memory data.

Options for creating test doubles

There are two different approaches that can be used to create an in-memory version of your context.

- **Create your own test doubles** – This approach involves writing your own in-memory implementation of your context and DbSets. This gives you a lot of control over how the classes behave but can involve writing and owning a reasonable amount of code.
- **Use a mocking framework to create test doubles** – Using a mocking framework (such as Moq) you can have the in-memory implementations of you context and sets created dynamically at runtime for you.

This article will deal with using a mocking framework. For creating your own test doubles see Testing with Your Own Test Doubles.

To demonstrate using EF with a mocking framework we are going to use Moq. The easiest way to get Moq is to install the Moq package from NuGet.

Testing with pre-EF6 versions

The scenario shown in this article is dependent on some changes we made to DbSet in EF6. For testing with EF5 and earlier version see Testing with a Fake Context.

Limitations of EF in-memory test doubles

In-memory test doubles can be a good way to provide unit test level coverage of bits of your application that use EF. However, when doing this you are using LINQ to Objects to execute queries against in-memory data. This can result in different behavior than using EF's LINQ provider (LINQ to Entities) to translate queries into SQL that is run against your database.

One example of such a difference is loading related data. If you create a series of Blogs that each have related Posts, then when using in-memory data the related Posts will always be loaded for each Blog. However, when running against a database the data will only be loaded if you use the Include method.

For this reason, it is recommended to always include some level of end-to-end testing (in addition to your unit tests) to ensure your application works correctly against a database.

Following along with this article

This article gives complete code listings that you can copy into Visual Studio to follow along if you wish. It's easiest to create a **Unit Test Project** and you will need to target **.NET Framework 4.5** to complete the sections that use async.

The EF model

The service we're going to test makes use of an EF model made up of the BloggingContext and the Blog and Post classes. This code may have been generated by the EF Designer or be a Code First model.

```
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; }
    }
}
```

Virtual DbSet properties with EF Designer

Note that the DbSet properties on the context are marked as virtual. This will allow the mocking framework to derive from our context and overriding these properties with a mocked implementation.

If you are using Code First then you can edit your classes directly. If you are using the EF Designer then you'll need to edit the T4 template that generates your context. Open up the <model_name>.Context.tt file that is nested under you edmx file, find the following fragment of code and add in the virtual keyword as shown.

```
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));
}
```

Service to be tested

To demonstrate testing with in-memory test doubles we are going to be writing a couple of tests for a BlogService. The service is capable of creating new blogs (AddBlog) and returning all Blogs ordered by name (GetAllBlogs). In addition to GetAllBlogs, we've also provided a method that will asynchronously get all blogs ordered by name (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();
        }
    }
}
```

Testing non-query scenarios

That's all we need to do to start testing non-query methods. The following test uses Moq to create a context. It then creates a DbSet<Blog> and wires it up to be returned from the context's Blogs property. Next, the context is used to create a new BlogService which is then used to create a new blog – using the AddBlog method. Finally, the test verifies that the service added a new Blog and called SaveChanges on the context.

```
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());
        }
    }
}
```

Testing query scenarios

In order to be able to execute queries against our DbSet test double we need to setup an implementation of IQueryable. The first step is to create some in-memory data – we're using a List<Blog>. Next, we create a context and DBSet<Blog> then wire up the IQueryable implementation for the DbSet – they're just delegating to the LINQ to Objects provider that works with List<T>.

We can then create a BlogService based on our test doubles and ensure that the data we get back from GetAllBlogs is ordered by name.

```
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);
        }
    }
}
```

Testing with async queries

Entity Framework 6 introduced a set of extension methods that can be used to asynchronously execute a query. Examples of these methods include ToListAsync, FirstAsync, ForEachAsync, etc.

Because Entity Framework queries make use of LINQ, the extension methods are defined on IQueryable and IEnumerable. However, because they are only designed to be used with Entity Framework you may receive the following error if you try to use them on a LINQ query that isn't an Entity Framework query:

> The source IQueryable doesn't implement IDbAsyncEnumerable{0}. Only sources that implement IDbAsyncEnumerable can be used for Entity Framework asynchronous operations. For more details see http://go.microsoft.com/fwlink/?LinkId=287068.

Whilst the async methods are only supported when running against an EF query, you may want to use them in your unit test when running against an in-memory test double of a DbSet.

In order to use the async methods we need to create an in-memory DbAsyncQueryProvider to process the async query. Whilst it would be possible to setup a query provider using Moq, it is much easier to create a test double implementation in code. The code for this implementation is as follows:

```
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; }
        }
    }
}
```

Now that we have an async query provider we can write a unit test for our new GetAllBlogsAsync method.

```
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);
        }
    }
}
```

Testing with your own test doubles

9/13/2018 • 9 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

When writing tests for your application it is often desirable to avoid hitting the database. Entity Framework allows you to achieve this by creating a context – with behavior defined by your tests – that makes use of in-memory data.

Options for creating test doubles

There are two different approaches that can be used to create an in-memory version of your context.

- **Create your own test doubles** – This approach involves writing your own in-memory implementation of your context and DbSets. This gives you a lot of control over how the classes behave but can involve writing and owning a reasonable amount of code.
- **Use a mocking framework to create test doubles** – Using a mocking framework (such as Moq) you can have the in-memory implementations of you context and sets created dynamically at runtime for you.

This article will deal with creating your own test double. For information on using a mocking framework see Testing with a Mocking Framework.

Testing with pre-EF6 versions

The code shown in this article is compatible with EF6. For testing with EF5 and earlier version see Testing with a Fake Context.

Limitations of EF in-memory test doubles

In-memory test doubles can be a good way to provide unit test level coverage of bits of your application that use EF. However, when doing this you are using LINQ to Objects to execute queries against in-memory data. This can result in different behavior than using EF's LINQ provider (LINQ to Entities) to translate queries into SQL that is run against your database.

One example of such a difference is loading related data. If you create a series of Blogs that each have related Posts, then when using in-memory data the related Posts will always be loaded for each Blog. However, when running against a database the data will only be loaded if you use the Include method.

For this reason, it is recommended to always include some level of end-to-end testing (in addition to your unit tests) to ensure your application works correctly against a database.

Following along with this article

This article gives complete code listings that you can copy into Visual Studio to follow along if you wish. It's easiest to create a **Unit Test Project** and you will need to target **.NET Framework 4.5** to complete the sections that use async.

Creating a context interface

We're going to look at testing a service that makes use of an EF model. In order to be able to replace our EF context with an in-memory version for testing, we'll define an interface that our EF context (and it's in-memory double) will imeplement.

The service we are going to test will query and modify data using the DbSet properties of our context and also call SaveChanges to push changes to the database. So we're including these members on the interface.

```
using System.Data.Entity;

namespace TestingDemo
{
    public interface IBloggingContext
    {
        DbSet<Blog> Blogs { get; }
        DbSet<Post> Posts { get; }
        int SaveChanges();
    }
}
```

The EF model

The service we're going to test makes use of an EF model made up of the BloggingContext and the Blog and Post classes. This code may have been generated by the EF Designer or be a Code First model.

```
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; }
    }
}
```

Implementing the context interface with the EF Designer

Note that our context implements the IBloggingContext interface.

If you are using Code First then you can edit your context directly to implement the interface. If you are using the

EF Designer then you'll need to edit the T4 template that generates your context. Open up the <model_name>.Context.tt file that is nested under you edmx file, find the following fragment of code and add in the interface as shown.

```
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> : DbContext, IBloggingContext
```

Service to be tested

To demonstrate testing with in-memory test doubles we are going to be writing a couple of tests for a BlogService. The service is capable of creating new blogs (AddBlog) and returning all Blogs ordered by name (GetAllBlogs). In addition to GetAllBlogs, we've also provided a method that will asynchronously get all blogs ordered by name (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();
        }
    }
}
```

Creating the in-memory test doubles

Now that we have the real EF model and the service that can use it, it's time to create the in-memory test double that we can use for testing. We've created a TestContext test double for our context. In test doubles we get to choose the behavior we want in order to support the tests we are going to run. In this example we're just capturing the number of times SaveChanges is called, but you can include whatever logic is needed to verify the scenario you are testing.

We've also created a TestDbSet that provides an in-memory implementation of DbSet. We've provided a complete implemention for all the methods on DbSet (except for Find), but you only need to implement the members that your test scenario will use.

TestDbSet makes use of some other infrastructure classes that we've included to ensure that async queries can be processed.

```
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>, IQueryable, 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; }
        }
    }
}
```

Implementing Find

The Find method is difficult to implement in a generic fashion. If you need to test code that makes use of the Find method it is easiest to create a test DbSet for each of the entity types that need to support find. You can then write logic to find that particular type of entity, as shown below.

```
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);
        }
    }
}
```

Writing some tests

That's all we need to do to start testing. The following test creates a TestContext and then a service based on this context. The service is then used to create a new blog – using the AddBlog method. Finally, the test verifies that the service added a new Blog to the context's Blogs property and called SaveChanges on the context.

This is just an example of the types of things you can test with an in-memory test double and you can adjust the logic of the test doubles and the verification to meet your requirements.

```
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);
        }
    }
}
```

Here is another example of a test - this time one that performs a query. The test starts by creating a test context with some data in its Blog property - note that the data is not in alphabetical order. We can then create a BlogService based on our test context and ensure that the data we get back from GetAllBlogs is ordered by name.

```
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);
        }
    }
}
```

Finally, we'll write one more test that uses our async method to ensure that the async infrastructure we included in TestDbSet is working.

```
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);
        }
    }
}
```

Testability and Entity Framework 4.0

9/18/2018 • 43 minutes to read • Edit Online

Scott Allen

Published: May 2010

Introduction

This white paper describes and demonstrates how to write testable code with the ADO.NET Entity Framework 4.0 and Visual Studio 2010. This paper does not try to focus on a specific testing methodology, like test-driven design (TDD) or behavior-driven design (BDD). Instead this paper will focus on how to write code that uses the ADO.NET Entity Framework yet remains easy to isolate and test in an automated fashion. We'll look at common design patterns that facilitate testing in data access scenarios and see how to apply those patterns when using the framework. We'll also look at specific features of the framework to see how those features can work in testable code.

What Is Testable Code?

The ability to verify a piece of software using automated unit tests offers many desirable benefits. Everyone knows that good tests will reduce the number of software defects in an application and increase the application's quality - but having unit tests in place goes far beyond just finding bugs.

A good unit test suite allows a development team to save time and remain in control of the software they create. A team can make changes to existing code, refactor, redesign, and restructure software to meet new requirements, and add new components into an application all while knowing the test suite can verify the application's behavior. Unit tests are part of a quick feedback cycle to facilitate change and preserve the maintainability of software as complexity increases.

Unit testing comes with a price, however. A team has to invest the time to create and maintain unit tests. The amount of effort required to create these tests is directly related to the **testability** of the underlying software. How easy is the software to test? A team designing software with testability in mind will create effective tests faster than the team working with un-testable software.

Microsoft designed the ADO.NET Entity Framework 4.0 (EF4) with testability in mind. This doesn't mean developers will be writing unit tests against framework code itself. Instead, the testability goals for EF4 make it easy to create testable code that builds on top of the framework. Before we look at specific examples, it's worthwhile to understand the qualities of testable code.

The Qualities of Testable Code

Code that is easy to test will always exhibit at least two traits. First, testable code is easy to **observe**. Given some set of inputs, it should be easy to observe the output of the code. For example, testing the following method is easy because the method directly returns the result of a calculation.

```
public int Add(int x, int y) {
    return x + y;
}
```

Testing a method is difficult if the method writes the computed value into a network socket, a database table, or a file like the following code. The test has to perform additional work to retrieve the value.

```
public void AddAndSaveToFile(int x, int y) {
     var results = string.Format("The answer is {0}", x + y);
     File.WriteAllText("results.txt", results);
}
```

Secondly, testable code is easy to **isolate**. Let's use the following pseudo-code as a bad example of testable code.

```
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;
}
```

The method is easy to observe – we can pass in an insurance policy and verify the return value matches an expected result. However, to test the method we'll need to have a database installed with the correct schema, and configure the SMTP server in case the method tries to send an email.

The unit test only wants to verify the calculation logic inside the method, but the test might fail because the email server is offline, or because the database server moved. Both of these failures are unrelated to the behavior the test wants to verify. The behavior is difficult to isolate.

Software developers who strive to write testable code often strive to maintain a separation of concerns in the code they write. The above method should focus on the business calculations and delegate the database and email implementation details to other components. Robert C. Martin calls this the Single Responsibility Principle. An object should encapsulate a single, narrow responsibility, like calculating the value of a policy. All other database and notification work should be the responsibility of some other object. Code written in this fashion is easier to isolate because it is focused on a single task.

In .NET we have the abstractions we need to follow the Single Responsibility Principle and achieve isolation. We can use interface definitions and force the code to use the interface abstraction instead of a concrete type. Later in this paper we'll see how a method like the bad example presented above can work with interfaces that *look* like they will talk to the database. At test time, however, we can substitute a dummy implementation that doesn't talk to the database but instead holds data in memory. This dummy implementation will isolate the code from unrelated problems in the data access code or database configuration.

There are additional benefits to isolation. The business calculation in the last method should only take a few milliseconds to execute, but the test itself might run for several seconds as the code hops around the network and talks to various servers. Unit tests should run fast to facilitate small changes. Unit tests should also be repeatable and not fail because a component unrelated to the test has a problem. Writing code that is easy to observe and to isolate means developers will have an easier time writing tests for the code, spend less time waiting for tests to execute, and more importantly, spend less time tracking down bugs that do not exist.

Hopefully you can appreciate the benefits of testing and understand the qualities that testable code exhibits. We are about to address how to write code that works with EF4 to save data into a database while remaining observable and easy to isolate, but first we'll narrow our focus to discuss testable designs for data access.

Design Patterns for Data Persistence

Both of the bad examples presented earlier had too many responsibilities. The first bad example had to perform a calculation *and* write to a file. The second bad example had to read data from a database *and* perform a business calculation *and* send email. By designing smaller methods that separate concerns and delegate responsibility to other components you'll make great strides towards writing testable code. The goal is to build functionality by composing actions from small and focused abstractions.

When it comes to data persistence the small and focused abstractions we are looking for are so common they've been documented as design patterns. Martin Fowler's book Patterns of Enterprise Application Architecture was the first work to describe these patterns in print. We'll provide a brief description of these patterns in the following sections before we show how these ADO.NET Entity Framework implements and works with these patterns.

The Repository Pattern

Fowler says a repository "mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects". The goal of the repository pattern is to isolate code from the minutiae of data access, and as we saw earlier isolation is a required trait for testability.

The key to the isolation is how the repository exposes objects using a collection-like interface. The logic you write to use the repository has no idea how the repository will materialize the objects you request. The repository might talk to a database, or it might just return objects from an in-memory collection. All your code needs to know is that the repository appears to maintain the collection, and you can retrieve, add, and delete objects from the collection.

In existing .NET applications a concrete repository often inherits from a generic interface like the following:

```
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);
}
```

We'll make a few changes to the interface definition when we provide an implementation for EF4, but the basic concept remains the same. Code can use a concrete repository implementing this interface to retrieve an entity by its primary key value, to retrieve a collection of entities based on the evaluation of a predicate, or simply retrieve all available entities. The code can also add and remove entities through the repository interface.

Given an IRepository of Employee objects, code can perform the following operations.

```
var employeesNamedScott =
    repository
        .FindBy(e => e.Name == "Scott")
        .OrderBy(e => e.HireDate);
var firstEmployee = repository.FindById(1);
var newEmployee = new Employee() {/*... */};
repository.Add(newEmployee);
```

Since the code is using an interface (IRepository of Employee), we can provide the code with different implementations of the interface. One implementation might be an implementation backed by EF4 and persisting objects into a Microsoft SQL Server database. A different implementation (one we use during testing) might be backed by an in-memory List of Employee objects. The interface will help to achieve isolation in the code.

Notice the IRepository<T> interface does not expose a Save operation. How do we update existing objects? You might come across IRepository definitions that do include the Save operation, and implementations of these repositories will need to immediately persist an object into the database. However, in many applications we don't want to persist objects individually. Instead, we want to bring objects to life, perhaps from different repositories, modify those objects as part of a business activity, and then persist all the objects as part of a single, atomic operation. Fortunately, there is a pattern to allow this type of behavior.

The Unit of Work Pattern

Fowler says a unit of work will “maintain a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems”. It is the responsibility of the unit of work to track changes to the objects we bring to life from a repository and persist any changes we’ve made to the objects when we tell the unit of work to commit the changes. It’s also the responsibility of the unit of work to take the new objects we’ve added to all repositories and insert the objects into a database, and also mange deletion.

If you’ve ever done any work with ADO.NET DataSets then you’ll already be familiar with the unit of work pattern. ADO.NET DataSets had the ability to track our updates, deletions, and insertion of DataRow objects and could (with the help of a TableAdapter) reconcile all our changes to a database. However, DataSet objects model a disconnected subset of the underlying database. The unit of work pattern exhibits the same behavior, but works with business objects and domain objects that are isolated from data access code and unaware of the database.

An abstraction to model the unit of work in .NET code might look like the following:

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<Order> Orders { get; }
    IRepository<Customer> Customers { get; }
    void Commit();
}
```

By exposing repository references from the unit of work we can ensure a single unit of work object has the ability to track all entities materialized during a business transaction. The implementation of the Commit method for a real unit of work is where all the magic happens to reconcile in-memory changes with the database.

Given an IUnitOfWork reference, code can make changes to business objects retrieved from one or more repositories and save all the changes using the atomic Commit operation.

```
var firstEmployee = unitofWork.Employees.FindById(1);
var firstCustomer = unitofWork.Customers.FindById(1);
firstEmployee.Name = "Alex";
firstCustomer.Name = "Christopher";
unitofWork.Commit();
```

The Lazy Load Pattern

Fowler uses the name lazy load to describe “an object that doesn’t contain all of the data you need but knows how to get it”. Transparent lazy loading is an important feature to have when writing testable business code and working with a relational database. As an example, consider the following code.

```
var employee = repository.FindById(id);
// ... and later ...
foreach(var timeCard in employee.TimeCards) {
    // .. manipulate the timeCard
}
```

How is the TimeCards collection populated? There are two possible answers. One answer is that the employee repository, when asked to fetch an employee, issues a query to retrieve both the employee along with the employee’s associated time card information. In relational databases this generally requires a query with a JOIN clause and may result in retrieving more information than an application needs. What if the application never needs to touch the TimeCards property?

A second answer is to load the TimeCards property “on demand”. This lazy loading is implicit and transparent to the business logic because the code does not invoke special APIs to retrieve time card information. The code assumes the time card information is present when needed. There is some magic involved with lazy loading that generally involves runtime interception of method invocations. The intercepting code is responsible for talking to the database and retrieving time card information while leaving the business logic free to be business logic. This lazy load magic allows the business code to isolate itself from data retrieval operations and results in more testable code.

The drawback to a lazy load is that when an application *does* need the time card information the code will execute an additional query. This isn't a concern for many applications, but for performance sensitive applications or applications looping through a number of employee objects and executing a query to retrieve time cards during each iteration of the loop (a problem often referred to as the N+1 query problem), lazy loading is a drag. In these scenarios an application might want to eagerly load time card information in the most efficient manner possible.

Fortunately, we'll see how EF4 supports both implicit lazy loads and efficient eager loads as we move into the next section and implement these patterns.

Implementing Patterns with the Entity Framework

The good news is that all of the design patterns we described in the last section are straightforward to implement with EF4. To demonstrate we are going to use a simple ASP.NET MVC application to edit and display Employees and their associated time card information. We'll start by using the following "plain old CLR objects" (POCOs).

```
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; }
}
```

These class definitions will change slightly as we explore different approaches and features of EF4, but the intent is to keep these classes as persistence ignorant (PI) as possible. A PI object doesn't know *how*, or even *if*, the state it holds lives inside a database. PI and POCOs go hand in hand with testable software. Objects using a POCO approach are less constrained, more flexible, and easier to test because they can operate without a database present.

With the POCOs in place we can create an Entity Data Model (EDM) in Visual Studio (see figure 1). We will not use the EDM to generate code for our entities. Instead, we want to use the entities we lovingly craft by hand. We will only use the EDM to generate our database schema and provide the metadata EF4 needs to map objects into the database.

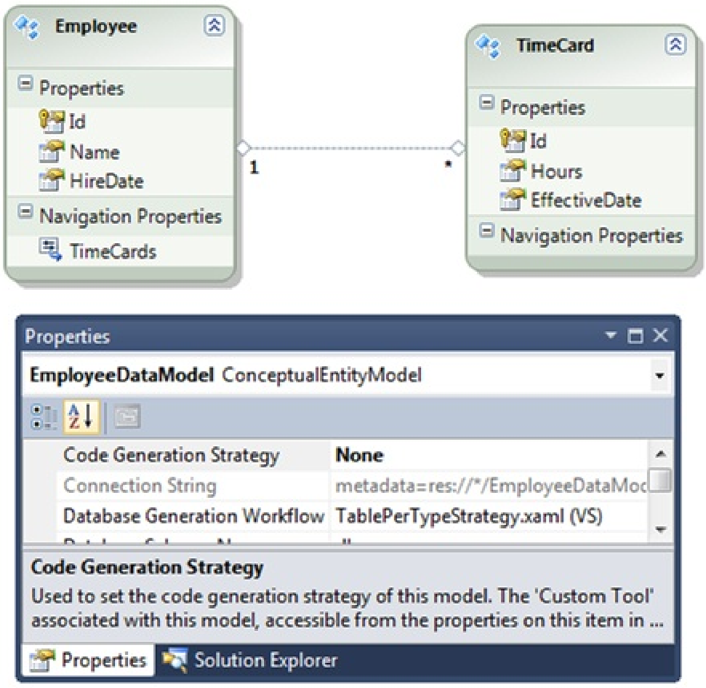


Figure 1

Note: if you want to develop the EDM model first, it is possible to generate clean, POCO code from the EDM. You can do this with a Visual Studio 2010 extension provided by the Data Programmability team. To download the extension, launch the Extension Manager from the Tools menu in Visual Studio and search the online gallery of templates for “POCO” (See figure 2). There are several POCO templates available for EF. For more information on using the template, see “ Walkthrough: POCO Template for the Entity Framework”.

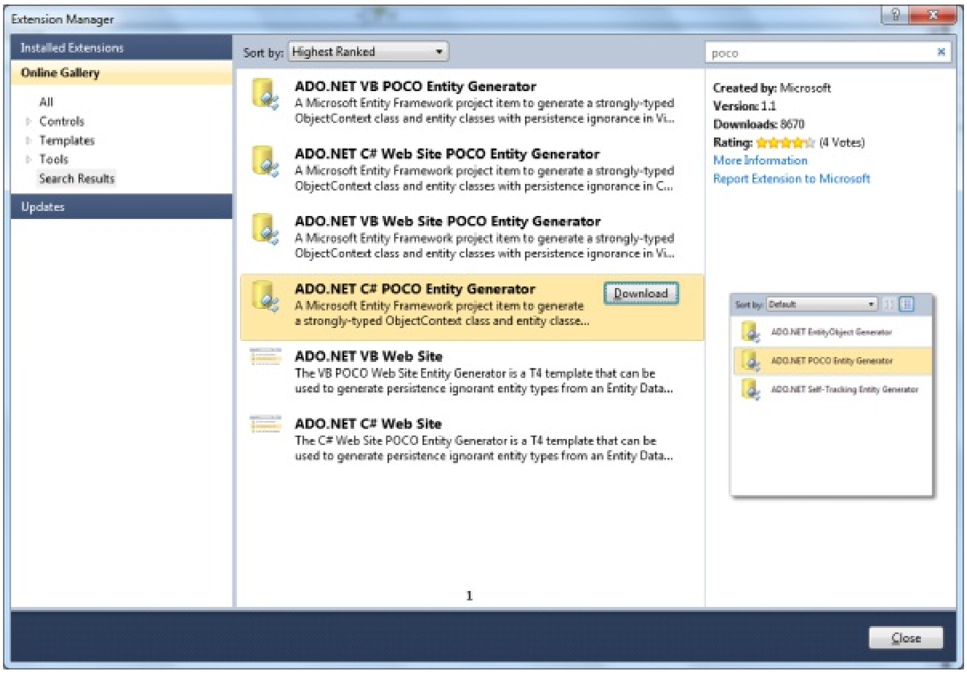


Figure 2

From this POCO starting point we will explore two different approaches to testable code. The first approach I call the EF approach because it leverages abstractions from the Entity Framework API to implement units of work and repositories. In the second approach we will create our own custom repository abstractions and then see the advantages and disadvantages of each approach. We’ll start by exploring the EF approach.

An EF Centric Implementation

Consider the following controller action from an ASP.NET MVC project. The action retrieves an Employee object and returns a result to display a detailed view of the employee.

```
public ViewResult Details(int id) {
    var employee = _unitOfWork.Employees
                              .Single(e => e.Id == id);
    return View(employee);
}
```

Is the code testable? There are at least two tests we'd need to verify the action's behavior. First, we'd like to verify the action returns the correct view – an easy test. We'd also want to write a test to verify the action retrieves the correct employee, and we'd like to do this without executing code to query the database. Remember we want to isolate the code under test. Isolation will ensure the test doesn't fail because of a bug in the data access code or database configuration. If the test fails, we will know we have a bug in the controller logic, and not in some lower level system component.

To achieve isolation we'll need some abstractions like the interfaces we presented earlier for repositories and units of work. Remember the repository pattern is designed to mediate between domain objects and the data mapping layer. In this scenario EF4 *is* the data mapping layer, and already provides a repository-like abstraction named IObjectSet<T> (from the System.Data.Objects namespace). The interface definition looks like the following.

```
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);
}
```

IObjectSet<T> meets the requirements for a repository because it resembles a collection of objects (via IEnumerable<T>) and provides methods to add and remove objects from the simulated collection. The Attach and Detach methods expose additional capabilities of the EF4 API. To use IObjectSet<T> as the interface for repositories we need a unit of work abstraction to bind repositories together.

```
public interface IUnitOfWork {
    IObjectSet<Employee> Employees { get; }
    IObjectSet<TimeCard> TimeCards { get; }
    void Commit();
}
```

One concrete implementation of this interface will talk to SQL Server and is easy to create using the ObjectContext class from EF4. The ObjectContext class is the real unit of work in the 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";
}
```

Bringing an IObjectSet<T> to life is as easy as invoking the CreateObjectSet method of the ObjectContext object. Behind the scenes the framework will use the metadata we provided in the EDM to produce a concrete ObjectSet<T>. We'll stick with returning the IObjectSet<T> interface because it will help preserve testability in client code.

This concrete implementation is useful in production, but we need to focus on how we'll use our IUnitOfWork abstraction to facilitate testing.

The Test Doubles

To isolate the controller action we'll need the ability to switch between the real unit of work (backed by an ObjectContext) and a test double or "fake" unit of work (performing in-memory operations). The common approach to perform this type of switching is to not let the MVC controller instantiate a unit of work, but instead pass the unit of work into the controller as a constructor parameter.

```
class EmployeeController : Controller {
  publicEmployeeController(IUnitOfWork unitOfWork)  {
      _unitOfWork = unitOfWork;
  }
  ...
}
```

The above code is an example of dependency injection. We don't allow the controller to create it's dependency (the unit of work) but inject the dependency into the controller. In an MVC project it is common to use a custom controller factory in combination with an inversion of control (IoC) container to automate dependency injection. These topics are beyond the scope of this article, but you can read more by following the references at the end of this article.

A fake unit of work implementation that we can use for testing might look like the following.

```
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;
    }
}
```

Notice the fake unit of work exposes a Commited property. It's sometimes useful to add features to a fake class that facilitate testing. In this case it is easy to observe if code commits a unit of work by checking the Commited property.

We'll also need a fake IObjectSet<T> to hold Employee and TimeCard objects in memory. We can provide a single implementation using generics.

```
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();
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }

    readonly HashSet<T> _set;
    readonly IQueryable<T> _queryableSet;
}
```

This test double delegates most of its work to an underlying HashSet<T> object. Note that IObjectSet<T> requires a generic constraint enforcing T as a class (a reference type), and also forces us to implement IQueryable<T>. It is easy to make an in-memory collection appear as an IQueryable<T> using the standard LINQ operator AsQueryable.

The Tests

Traditional unit tests will use a single test class to hold all of the tests for all of the actions in a single MVC controller. We can write these tests, or any type of unit test, using the in memory fakes we've built. However, for this article we will avoid the monolithic test class approach and instead group our tests to focus on a specific piece of functionality. For example, "create new employee" might be the functionality we want to test, so we will use a single test class to verify the single controller action responsible for creating a new employee.

There is some common setup code we need for all these fine grained test classes. For example, we always need to create our in-memory repositories and fake unit of work. We also need an instance of the employee controller with the fake unit of work injected. We'll share this common setup code across test classes by using a base class.

```
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;
}
```

The “object mother” we use in the base class is one common pattern for creating test data. An object mother contains factory methods to instantiate test entities for use across multiple test fixtures.

```
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
}
```

We can use the EmployeeControllerTestBase as the base class for a number of test fixtures (see figure 3). Each test fixture will test a specific controller action. For example, one test fixture will focus on testing the Create action used during an HTTP GET request (to display the view for creating an employee), and a different fixture will focus on the Create action used in an HTTP POST request (to take information submitted by the user to create an employee). Each derived class is only responsible for the setup needed in its specific context, and to provide the assertions needed to verify the outcomes for its specific test context.

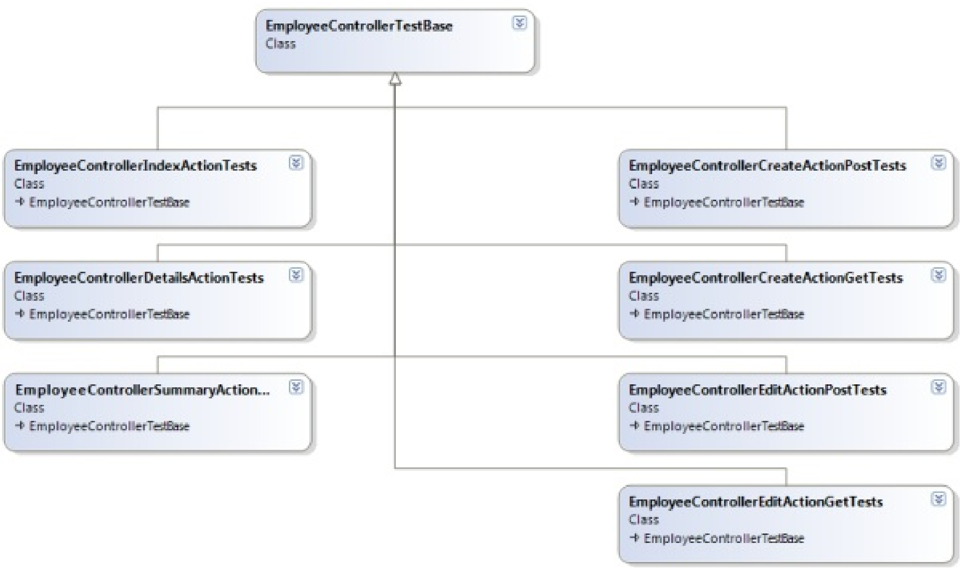


Figure 3

The naming convention and test style presented here isn't required for testable code – it's just one approach. Figure 4 shows the tests running in the Jet Brains Resharper test runner plugin for Visual Studio 2010.

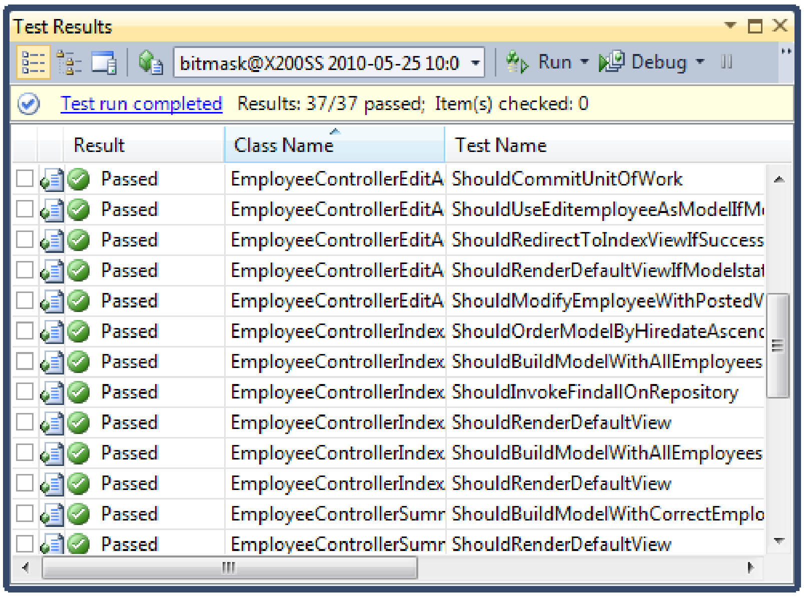


Figure 4

With a base class to handle the shared setup code, the unit tests for each controller action are small and easy to write. The tests will execute quickly (since we are performing in-memory operations), and shouldn't fail because of unrelated infrastructure or environmental concerns (because we've isolated the unit under test).

```
[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)
    };
}
```

In these tests, the base class does most of the setup work. Remember the base class constructor creates the in-memory repository, a fake unit of work, and an instance of the EmployeeController class. The test class derives from this base class and focuses on the specifics of testing the Create method. In this case the specifics boil down to the "arrange, act, and assert" steps you'll see in any unit testing procedure:

- Create a newEmployee object to simulate incoming data.
- Invoke the Create action of the EmployeeController and pass in the newEmployee.
- Verify the Create action produces the expected results (the employee appears in the repository).

What we've built allows us to test any of the EmployeeController actions. For example, when we write tests for the Index action of the Employee controller we can inherit from the test base class to establish the same base setup for our tests. Again the base class will create the in-memory repository, the fake unit of work, and an instance of the

EmployeeController. The tests for the Index action only need to focus on invoking the Index action and testing the qualities of the model the action returns.

```
[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)));
    }
    // ...
}
```

The tests we are creating with in-memory fakes are oriented towards testing the *state* of the software. For example, when testing the Create action we want to inspect the state of the repository after the create action executes – does the repository hold the new employee?

```
[TestMethod]
public void ShouldAddNewEmployeeToRepository() {
    _controller.Create(_newEmployee);
    Assert.IsTrue(_repository.Contains(_newEmployee));
}
```

Later we'll look at interaction based testing. Interaction based testing will ask if the code under test invoked the proper methods on our objects and passed the correct parameters. For now we'll move on the cover another design pattern – the lazy load.

Eager Loading and Lazy Loading

At some point in the ASP.NET MVC web application we might wish to display an employee's information and include the employee's associated time cards. For example, we might have a time card summary display that shows the employee's name and the total number of time cards in the system. There are several approaches we can take to implement this feature.

Projection

One easy approach to create the summary is to construct a model dedicated to the information we want to display in the view. In this scenario the model might look like the following.

```
public class EmployeeSummaryViewModel {
    public string Name { get; set; }
    public int TotalTimeCards { get; set; }
}
```

Note that the EmployeeSummaryViewModel is not an entity – in other words it is not something we want to persist in the database. We are only going to use this class to shuffle data into the view in a strongly typed manner. The view model is like a data transfer object (DTO) because it contains no behavior (no methods) – only properties.

The properties will hold the data we need to move. It is easy to instantiate this view model using LINQ's standard projection operator – the Select operator.

```
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);
}
```

There are two notable features to the above code. First – the code is easy to test because it is still easy to observe and isolate. The Select operator works just as well against our in-memory fakes as it does against the real unit of work.

```
[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);
    }
    // ...
}
```

The second notable feature is how the code allows EF4 to generate a single, efficient query to assemble employee and time card information together. We've loaded employee information and time card information into the same object without using any special APIs. The code merely expressed the information it requires using standard LINQ operators that work against in-memory data sources as well as remote data sources. EF4 was able to translate the expression trees generated by the LINQ query and C# compiler into a single and efficient T-SQL query.

```
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]
```

There are other times when we don't want to work with a view model or DTO object, but with real entities. When we know we need an employee *and* the employee's time cards, we can eagerly load the related data in an unobtrusive and efficient manner.

Explicit Eager Loading

When we want to eagerly load related entity information we need some mechanism for business logic (or in this scenario, controller action logic) to express its desire to the repository. The EF4 ObjectQuery<T> class defines an Include method to specify the related objects to retrieve during a query. Remember the EF4 ObjectContext exposes entities via the concrete ObjectSet<T> class which inherits from ObjectQuery<T>. If we were using ObjectSet<T> references in our controller action we could write the following code to specify an eager load of time card information for each employee.

```
_employees.Include("TimeCards")
          .Where(e => e.HireDate.Year > 2009);
```

However, since we are trying to keep our code testable we are not exposing ObjectSet<T> from outside the real unit of work class. Instead, we rely on the IObjectSet<T> interface which is easier to fake, but IObjectSet<T> does not define an Include method. The beauty of LINQ is that we can create our own Include operator.

```
public static class QueryableExtensions {
    public static IQueryable<T> Include<T>
            (this IQueryable<T> sequence, string path) {
        var objectQuery = sequence as ObjectQuery<T>;
        if(objectQuery != null)
        {
            return objectQuery.Include(path);
        }
        return sequence;
    }
}
```

Notice this Include operator is defined as an extension method for IQueryable<T> instead of IObjectSet<T>. This gives us the ability to use the method with a wider range of possible types, including IQueryable<T>, IObjectSet<T>, ObjectQuery<T>, and ObjectSet<T>. In the event the underlying sequence is not a genuine EF4 ObjectQuery<T>, then there is no harm done and the Include operator is a no-op. If the underlying sequence *is* an ObjectQuery<T> (or derived from ObjectQuery<T>), then EF4 will see our requirement for additional data and formulate the proper SQL query.

With this new operator in place we can explicitly request an eager load of time card information from the repository.

```
public ViewResult Index() {
    var model = _unitOfWork.Employees
                           .Include("TimeCards")
                           .OrderBy(e => e.HireDate);
    return View(model);
}
```

When run against a real ObjectContext, the code produces the following single query. The query gathers enough information from the database in one trip to materialize the employee objects and fully populate their TimeCards property.

```sql
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
```

The great news is the code inside the action method remains fully testable. We don't need to provide any additional features for our fakes to support the Include operator. The bad news is we had to use the Include operator inside of the code we wanted to keep persistence ignorant. This is a prime example of the type of tradeoffs you'll need to evaluate when building testable code. There are times when you need to let persistence concerns leak outside the repository abstraction to meet performance goals.

The alternative to eager loading is lazy loading. Lazy loading means we do *not* need our business code to explicitly announce the requirement for associated data. Instead, we use our entities in the application and if additional data is needed Entity Framework will load the data on demand.

Lazy Loading

It's easy to imagine a scenario where we don't know what data a piece of business logic will need. We might know the logic needs an employee object, but we may branch into different execution paths where some of those paths require time card information from the employee, and some do not. Scenarios like this are perfect for implicit lazy loading because data magically appears on an as-needed basis.

Lazy loading, also known as deferred loading, does place some requirements on our entity objects. POCOs with true persistence ignorance would not face any requirements from the persistence layer, but true persistence ignorance is practically impossible to achieve. Instead we measure persistence ignorance in relative degrees. It would be unfortunate if we needed to inherit from a persistence oriented base class or use a specialized collection to achieve lazy loading in POCOs. Fortunately, EF4 has a less intrusive solution.

Virtually Undetectable

When using POCO objects, EF4 can dynamically generate runtime proxies for entities. These proxies invisibly wrap the materialized POCOs and provide additional services by intercepting each property get and set operation to perform additional work. One such service is the lazy loading feature we are looking for. Another service is an efficient change tracking mechanism which can record when the program changes the property values of an entity. The list of changes is used by the ObjectContext during the SaveChanges method to persist any modified entities using UPDATE commands.

For these proxies to work, however, they need a way to hook into property get and set operations on an entity, and the proxies achieve this goal by overriding virtual members. Thus, if we want to have implicit lazy loading and efficient change tracking we need to go back to our POCO class definitions and mark properties as virtual.

```
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; }
}
```

We can still say the Employee entity is mostly persistence ignorant. The only requirement is to use virtual members and this does not impact the testability of the code. We don't need to derive from any special base class, or even use a special collection dedicated to lazy loading. As the code demonstrates, any class implementing ICollection<T> is available to hold related entities.

There is also one minor change we need to make inside our unit of work. Lazy loading is *off* by default when working directly with an ObjectContext object. There is a property we can set on the ContextOptions property to enable deferred loading, and we can set this property inside our real unit of work if we want to enable lazy loading everywhere.

```
public class SqlUnitOfWork : IUnitOfWork {
     public SqlUnitOfWork() {
         // ...
         _context = new ObjectContext(connectionString);
         _context.ContextOptions.LazyLoadingEnabled = true;
     }
     // ...
 }
```

With implicit lazy loading enabled, application code can use an employee and the employee's associated time cards while remaining blissfully unaware of the work required for EF to load the extra data.

```
var employee = _unitOfWork.Employees
                          .Single(e => e.Id == id);
foreach (var card in employee.TimeCards) {
    // ...
}
```

Lazy loading makes the application code easier to write, and with the proxy magic the code remains completely testable. In-memory fakes of the unit of work can simply preload fake entities with associated data when needed during a test.

At this point we'll turn our attention from building repositories using IObjectSet<T> and look at abstractions to hide all signs of the persistence framework.

Custom Repositories

When we first presented the unit of work design pattern in this article we provided some sample code for what the unit of work might look like. Let's re-present this original idea using the employee and employee time card scenario we've been working with.

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<TimeCard> TimeCards { get;  }
    void Commit();
}
```

The primary difference between this unit of work and the unit of work we created in the last section is how this unit of work does not use any abstractions from the EF4 framework (there is no IObjectSet<T>). IObjectSet<T> works well as a repository interface, but the API it exposes might not perfectly align with our application's needs. In this upcoming approach we will represent repositories using a custom IRepository<T> abstraction.

Many developers who follow test-driven design, behavior-driven design, and domain driven methodologies design prefer the IRepository<T> approach for several reasons. First, the IRepository<T> interface represents an "anti-corruption" layer. As described by Eric Evans in his Domain Driven Design book an anti-corruption layer keeps your domain code away from infrastructure APIs, like a persistence API. Secondly, developers can build methods into the repository that meet the exact needs of an application (as discovered while writing tests). For example, we might frequently need to locate a single entity using an ID value, so we can add a FindById method to the repository interface. Our IRepository<T> definition will look like the following.

```
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);
}
```

Notice we'll drop back to using an IQueryable<T> interface to expose entity collections. IQueryable<T> allows LINQ expression trees to flow into the EF4 provider and give the provider a holistic view of the query. A second option would be to return IEnumerable<T>, which means the EF4 LINQ provider will only see the expressions built inside of the repository. Any grouping, ordering, and projection done outside of the repository will not be composed into the SQL command sent to the database, which can hurt performance. On the other hand, a repository returning only IEnumerable<T> results will never surprise you with a new SQL command. Both approaches will work, and both approaches remain testable.

It's straightforward to provide a single implementation of the IRepository<T> interface using generics and the 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;
}
```

The IRepository<T> approach gives us some additional control over our queries because a client has to invoke a method to get to an entity. Inside the method we could provide additional checks and LINQ operators to enforce application constraints. Notice the interface has two constraints on the generic type parameter. The first constraint is the class cons taint required by ObjectSet<T>, and the second constraint forces our entities to implement IEntity – an abstraction created for the application. The IEntity interface forces entities to have a readable Id property, and we can then use this property in the FindById method. IEntity is defined with the following code.

```
public interface IEntity {
    int Id { get; }
}
```

IEntity could be considered a small violation of persistence ignorance since our entities are required to implement this interface. Remember persistence ignorance is about tradeoffs, and for many the FindById functionality will outweigh the constraint imposed by the interface. The interface has no impact on testability.

Instantiating a live IRepository<T> requires an EF4 ObjectContext, so a concrete unit of work implementation should manage the instantiation.

```
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";
}
```

Using the Custom Repository

Using our custom repository is not significantly different from using the repository based on IObjectSet<T>. Instead of applying LINQ operators directly to a property, we'll first need to invoke one the repository's methods to grab an IQueryable<T> reference.

```
public ViewResult Index() {
    var model = _repository.FindAll()
                           .Include("TimeCards")
                           .OrderBy(e => e.HireDate);
    return View(model);
}
```

Notice the custom Include operator we implemented previously will work without change. The repository's FindById method removes duplicated logic from actions trying to retrieve a single entity.

```
public ViewResult Details(int id) {
    var model = _repository.FindById(id);
    return View(model);
}
```

There is no significant difference in the testability of the two approaches we've examined. We could provide fake implementations of IRepository<T> by building concrete classes backed by HashSet<Employee> - just like what we did in the last section. However, some developers prefer to use mock objects and mock object frameworks instead of building fakes. We'll look at using mocks to test our implementation and discuss the differences between mocks and fakes in the next section.

Testing with Mocks

There are different approaches to building what Martin Fowler calls a "test double". A test double (like a movie stunt double) is an object you build to "stand in" for real, production objects during tests. The in-memory repositories we created are test doubles for the repositories that talk to SQL Server. We've seen how to use these test-doubles during the unit tests to isolate code and keep tests running fast.

The test doubles we've built have real, working implementations. Behind the scenes each one stores a concrete collection of objects, and they will add and remove objects from this collection as we manipulate the repository during a test. Some developers like to build their test doubles this way – with real code and working implementations. These test doubles are what we call *fakes*. They have working implementations, but they aren't real enough for production use. The fake repository doesn't actually write to the database. The fake SMTP server doesn't actually send an email message over the network.

Mocks versus Fakes

There is another type of test double known as a *mock*. While fakes have working implementations, mocks come with no implementation. With the help of a mock object framework we construct these mock objects at run time and use them as test doubles. In this section we'll be using the open source mocking framework Moq. Here is a simple example of using Moq to dynamically create a test double for an employee repository.

```
Mock<IRepository<Employee>> mock =
    new Mock<IRepository<Employee>>();
IRepository<Employee> repository = mock.Object;
repository.Add(new Employee());
var employee = repository.FindById(1);
```

We ask Moq for an IRepository<Employee> implementation and it builds one dynamically. We can get to the object implementing IRepository<Employee> by accessing the Object property of the Mock<T> object. It is this inner object we can pass into our controllers, and they won't know if this is a test double or the real repository. We can invoke methods on the object just like we would invoke methods on an object with a real implementation.

You must be wondering what the mock repository will do when we invoke the Add method. Since there is no implementation behind the mock object, Add does nothing. There is no concrete collection behind the scenes like we had with the fakes we wrote, so the employee is discarded. What about the return value of FindById? In this case the mock object does the only thing it can do, which is return a default value. Since we are returning a reference type (an Employee), the return value is a null value.

Mocks might sound worthless; however, there are two more features of mocks we haven't talked about. First, the Moq framework records all the calls made on the mock object. Later in the code we can ask Moq if anyone invoked the Add method, or if anyone invoked the FindById method. We'll see later how we can use this "black box" recording feature in tests.

The second great feature is how we can use Moq to program a mock object with *expectations*. An expectation tells the mock object how to respond to any given interaction. For example, we can program an expectation into our mock and tell it to return an employee object when someone invokes FindById. The Moq framework uses a Setup API and lambda expressions to program these expectations.

```
[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);
}
```

In this sample we ask Moq to dynamically build a repository, and then we program the repository with an expectation. The expectation tells the mock object to return a new employee object with an Id value of 5 when someone invokes the FindById method passing a value of 5. This test passes, and we didn't need to build a full implementation to fake IRepository<T>.

Let's revisit the tests we wrote earlier and rework them to use mocks instead of fakes. Just like before, we'll use a base class to setup the common pieces of infrastructure we need for all of the controller's tests.

```
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;
}
```

The setup code remains mostly the same. Instead of using fakes, we'll use Moq to construct mock objects. The base class arranges for the mock unit of work to return a mock repository when code invokes the Employees property. The rest of the mock setup will take place inside the test fixtures dedicated to each specific scenario. For example, the test fixture for the Index action will setup the mock repository to return a list of employees when the action invokes the FindAll method of the mock repository.

```
[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
}
```

Except for the expectations, our tests look similar to the tests we had before. However, with the recording ability of a mock framework we can approach testing from a different angle. We'll look at this new perspective in the next section.

State versus Interaction Testing

There are different techniques you can use to test software with mock objects. One approach is to use state based testing, which is what we have done in this paper so far. State based testing makes assertions about the state of the software. In the last test we invoked an action method on the controller and made an assertion about the model it should build. Here are some other examples of testing state:

- Verify the repository contains the new employee object after Create executes.
- Verify the model holds a list of all employees after Index executes.
- Verify the repository does not contain a given employee after Delete executes.

Another approach you'll see with mock objects is to verify *interactions*. While state based testing makes assertions about the state of objects, interaction based testing makes assertions about how objects interact. For example:

- Verify the controller invokes the repository's Add method when Create executes.
- Verify the controller invokes the repository's FindAll method when Index executes.
- Verify the controller invokes the unit of work's Commit method to save changes when Edit executes.

Interaction testing often requires less test data, because we aren't poking inside of collections and verifying counts. For example, if we know the Details action invokes a repository's FindById method with the correct value - then the action is probably behaving correctly. We can verify this behavior without setting up any test data to return from FindById.

```
[TestClass]
public class EmployeeControllerDetailsActionTests
           : EmployeeControllerTestBase {
     // ...
    [TestMethod]
    public void ShouldInvokeRepositoryToFindEmployee() {
        var result = _controller.Details(_detailsId);
        _repository.Verify(r => r.FindById(_detailsId));
    }
    int _detailsId = 1;
}
```

The only setup required in the above test fixture is the setup provided by the base class. When we invoke the controller action, Moq will record the interactions with the mock repository. Using the Verify API of Moq, we can ask Moq if the controller invoked FindById with the proper ID value. If the controller did not invoke the method, or invoked the method with an unexpected parameter value, the Verify method will throw an exception and the test will fail.

Here is another example to verify the Create action invokes Commit on the current unit of work.

```
[TestMethod]
public void ShouldCommitUnitOfWork() {
    _controller.Create(_newEmployee);
    _unitOfWork.Verify(u => u.Commit());
}
```

One danger with interaction testing is the tendency to over specify interactions. The ability of the mock object to record and verify every interaction with the mock object doesn't mean the test should try to verify every interaction. Some interactions are implementation details and you should only verify the interactions *required* to satisfy the current test.

The choice between mocks or fakes largely depends on the system you are testing and your personal (or team) preferences. Mock objects can drastically reduce the amount of code you need to implement test doubles, but not everyone is comfortable programming expectations and verifying interactions.

Conclusions

In this paper we've demonstrated several approaches to creating testable code while using the ADO.NET Entity Framework for data persistence. We can leverage built in abstractions like IObjectSet<T>, or create our own abstractions like IRepository<T>. In both cases, the POCO support in the ADO.NET Entity Framework 4.0 allows the consumers of these abstractions to remain persistent ignorant and highly testable. Additional EF4 features like implicit lazy loading allows business and application service code to work without worrying about the details of a relational data store. Finally, the abstractions we create are easy to mock or fake inside of unit tests, and we can use these test doubles to achieve fast running, highly isolated, and reliable tests.

Additional Resources

- Robert C. Martin, " The Single Responsibility Principle"
- Martin Fowler, Catalog of Patterns from *Patterns of Enterprise Application Architecture*
- Griffin Caprio, " Dependency Injection"
- Data Programmability Blog, " Walkthrough: Test Driven Development with the Entity Framework 4.0".
- Data Programmability Blog, " Using Repository and Unit of Work patterns with Entity Framework 4.0"
- Dave Astels, " BDD Intro"
- Aaron Jensen, " Introducing Machine Specifications"
- Eric Lee, " BDD with MSTest"
- Eric Evans, " Domain Driven Design"
- Martin Fowler, " Mocks Aren't Stubs"
- Martin Fowler, " Test Double"
- Jeremy Miller, " State versus Interaction Testing"
- Moq

Biography

Scott Allen is a member of the technical staff at Pluralsight and the founder of OdeToCode.com. In 15 years of commercial software development, Scott has worked on solutions for everything from 8-bit embedded devices to highly scalable ASP.NET web applications. You can reach Scott on his blog at OdeToCode, or on Twitter at http://twitter.com/OdeToCode.

Creating a Model

9/18/2018 • 2 minutes to read • Edit Online

An EF model stores the details about how application classes and properties map to database tables and columns. There are two main ways to create an EF model:

- **Using Code First**: The developer writes code to specify the model. EF generates the models and mappings at runtime based on entity classes and additional model configuration provided by the developer.
- **Using the EF Designer**: The developer draws boxes and lines to specify the model using the EF Designer. The resulting model is stored as XML in a file with the EDMX extension. The application's domain objects are typically generated automatically from the conceptual model.

EF workflows

Both of these approaches can be used to target an existing database or create a new database, resulting in 4 different workflows. Find out about which one is best for you:

	I JUST WANT TO WRITE CODE...	I WANT TO USE A DESIGNER...
I am creating a new database	Use **Code First** to define your model in code and then generate a database.	Use **Model First** to define your model using boxes and lines and then generate a database.
I need to access an existing database	Use **Code First** to create a code based model that maps to an existing database.	Use **Database First** to create a boxes and lines model that maps to an existing database.

Watch the video: What EF workflow should I use?

This short video explains the differences, and how to find the one that is right for you.

Presented By: Rowan Miller


WMV | MP4 | WMV (ZIP)

If after watching the video you still don't feel comfortable deciding if you want to use the EF Designer or Code First, learn both!

A look under the hood

Regardless of whether you use Code First or the EF Designer, an EF model always has several components:

- The application's domain objects or entity types themselves. This is often referred to as the object layer
- A conceptual model consisting of domain-specific entity types and relationships, described using the Entity Data Model. This layer is often referred to with the letter "C", for *conceptual*.
- A storage model representing tables, columns and relationships as defined in the database. This layer is often referred to with the later "S", for *storage*.
- A mapping between the conceptual model and the database schema. This mapping is often referred to as

"C-S" mapping.

EF's mapping engine leverages the "C-S" mapping to transform operations against entities - such as create, read, update, and delete - into equivalent operations against tables in the database.

The mapping between the conceptual model and the application's objects is often referred to as "O-C" mapping. Compared to the "C-S" mapping, "O-C" mapping is implicit and one-to-one: entities, properties and relationships defined in the conceptual model are required to match the shapes and types of the .NET objects. From EF4 and beyond, the objects layer can be composed of simple objects with properties without any dependencies on EF. These are usually referred to as Plain-Old CLR Objects (POCO) and mapping of types and properties is performed base on name matching conventions. Previously, in EF 3.5 there were specific restrictions for the object layer, like entities having to derive from the EntityObject class and having to carry EF attributes to implement the "O-C" mapping.

Code First to a New Database

9/18/2018 • 10 minutes to read • Edit Online

This video and step-by-step walkthrough provide an introduction to Code First development targeting a new database. This scenario includes targeting a database that doesn't exist and Code First will create, or an empty database that Code First will add new tables to. Code First allows you to define your model using C# or VB.Net classes. Additional configuration can optionally be performed using attributes on your classes and properties or by using a fluent API.

Watch the video

This video provides an introduction to Code First development targeting a new database. This scenario includes targeting a database that doesn't exist and Code First will create, or an empty database that Code First will add new tables to. Code First allows you to define your model using C# or VB.Net classes. Additional configuration can optionally be performed using attributes on your classes and properties or by using a fluent API.

Presented By: Rowan Miller

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have at least Visual Studio 2010 or Visual Studio 2012 installed to complete this walkthrough.

If you are using Visual Studio 2010, you will also need to have NuGet installed.

1. Create the Application

To keep things simple we're going to build a basic console application that uses Code First to perform data access.

- Open Visual Studio
- **File -> New -> Project…**
- Select **Windows** from the left menu and **Console Application**
- Enter **CodeFirstNewDatabaseSample** as the name
- Select **OK**

2. Create the Model

Let's define a very simple model using classes. We're just defining them in the Program.cs file but in a real world application you would split your classes out into separate files and potentially a separate project.

Below the Program class definition in Program.cs add the following two classes.

```
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; }
}
```

You'll notice that we're making the two navigation properties (Blog.Posts and Post.Blog) virtual. This enables the Lazy Loading feature of Entity Framework. Lazy Loading means that the contents of these properties will be automatically loaded from the database when you try to access them.

3. Create a Context

Now it's time to define a derived context, which represents a session with the database, allowing us to query and save data. We define a context that derives from System.Data.Entity.DbContext and exposes a typed DbSet<TEntity> for each class in our model.

We're now starting to use types from the Entity Framework so we need to add the EntityFramework NuGet package.

- **Project –> Manage NuGet Packages...** Note: If you don't have the **Manage NuGet Packages...** option you should install the latest version of NuGet
- Select the **Online** tab
- Select the **EntityFramework** package
- Click **Install**

Add a using statement for System.Data.Entity at the top of Program.cs.

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

Below the Post class in Program.cs add the following derived context.

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

Here is a complete listing of what Program.cs should now contain.

```
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; }
    }
}
```

That is all the code we need to start storing and retrieving data. Obviously there is quite a bit going on behind the scenes and we'll take a look at that in a moment but first let's see it in action.

4. Reading & Writing Data

Implement the Main method in Program.cs as shown below. This code creates a new instance of our context and then uses it to insert a new Blog. Then it uses a LINQ query to retrieve all Blogs from the database ordered alphabetically by 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();
        }
    }
}
```

You can now run the application and test it out.

```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

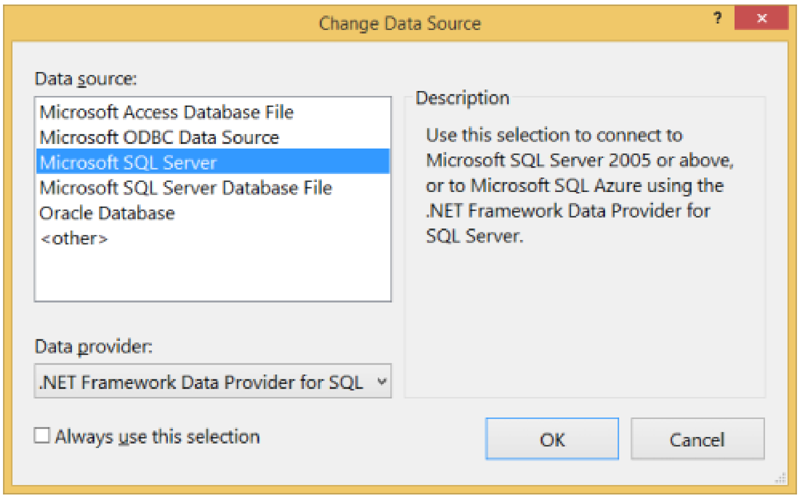
Where's My Data?

By convention DbContext has created a database for you.

- If a local SQL Express instance is available (installed by default with Visual Studio 2010) then Code First has created the database on that instance
- If SQL Express isn't available then Code First will try and use LocalDB (installed by default with Visual Studio 2012)
- The database is named after the fully qualified name of the derived context, in our case that is **CodeFirstNewDatabaseSample.BloggingContext**

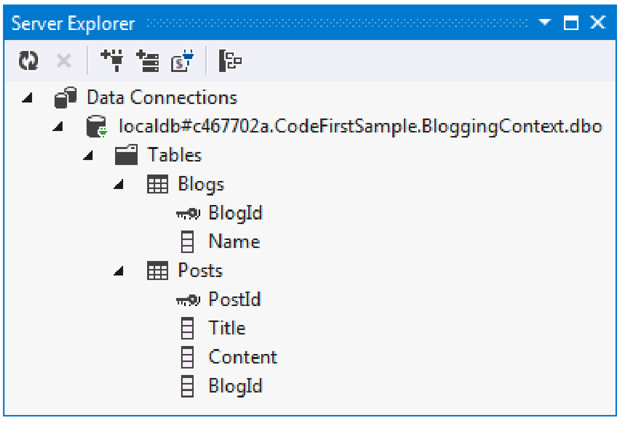
These are just the default conventions and there are various ways to change the database that Code First uses, more information is available in the **How DbContext Discovers the Model and Database Connection** topic. You can connect to this database using Server Explorer in Visual Studio

- **View -> Server Explorer**
- Right click on **Data Connections** and select **Add Connection…**
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source



- Connect to either LocalDB or SQL Express, depending on which one you have installed

We can now inspect the schema that Code First created.



DbContext worked out what classes to include in the model by looking at the DbSet properties that we defined. It then uses the default set of Code First conventions to determine table and column names, determine data types, find primary keys, etc. Later in this walkthrough we'll look at how you can override these conventions.

5. Dealing with Model Changes

Now it's time to make some changes to our model, when we make these changes we also need to update the database schema. To do this we are going to use a feature called Code First Migrations, or Migrations for short.

Migrations allows us to have an ordered set of steps that describe how to upgrade (and downgrade) our database schema. Each of these steps, known as a migration, contains some code that describes the changes to be applied.

The first step is to enable Code First Migrations for our BloggingContext.

- **Tools -> Library Package Manager -> Package Manager Console**
- Run the **Enable-Migrations** command in Package Manager Console
- A new Migrations folder has been added to our project that contains two items:
  - **Configuration.cs** – This file contains the settings that Migrations will use for migrating BloggingContext. We don't need to change anything for this walkthrough, but here is where you can specify seed data, register providers for other databases, changes the namespace that migrations are generated in etc.
  - **<timestamp>_InitialCreate.cs** – This is your first migration, it represents the changes that have already been applied to the database to take it from being an empty database to one that includes the Blogs and Posts tables. Although we let Code First automatically create these tables for us, now that we

have opted in to Migrations they have been converted into a Migration. Code First has also recorded in our local database that this Migration has already been applied. The timestamp on the filename is used for ordering purposes.

Now let's make a change to our model, add a Url property to the Blog class:

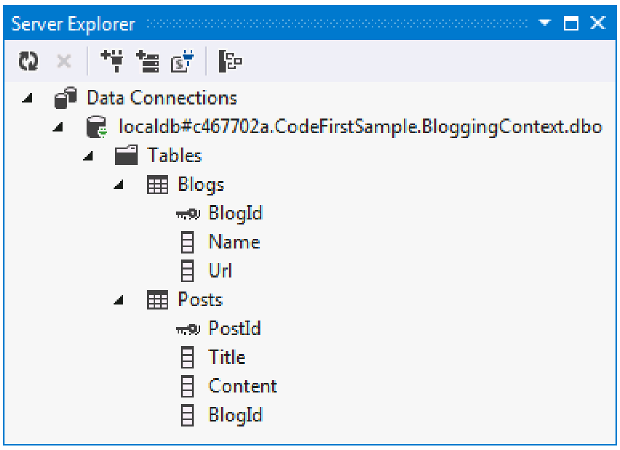
```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

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

- Run the **Add-Migration AddUrl** command in Package Manager Console. The Add-Migration command checks for changes since your last migration and scaffolds a new migration with any changes that are found. We can give migrations a name; in this case we are calling the migration 'AddUrl'. The scaffolded code is saying that we need to add a Url column, that can hold string data, to the dbo.Blogs table. If needed, we could edit the scaffolded code but that's not required in this case.

```
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");
        }
    }
}
```

- Run the **Update-Database** command in Package Manager Console. This command will apply any pending migrations to the database. Our InitialCreate migration has already been applied so migrations will just apply our new AddUrl migration. Tip: You can use the **–Verbose** switch when calling Update-Database to see the SQL that is being executed against the database.

The new Url column is now added to the Blogs table in the database:



6. Data Annotations

So far we've just let EF discover the model using its default conventions, but there are going to be times when our classes don't follow the conventions and we need to be able to perform further configuration. There are two options for this; we'll look at Data Annotations in this section and then the fluent API in the next section.

- Let's add a User class to our model

```
public class User
{
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- We also need to add a set to our derived context

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

- If we tried to add a migration we'd get an error saying *"EntityType 'User' has no key defined. Define the key for this EntityType."* because EF has no way of knowing that Username should be the primary key for User.
- We're going to use Data Annotations now so we need to add a using statement at the top of Program.cs

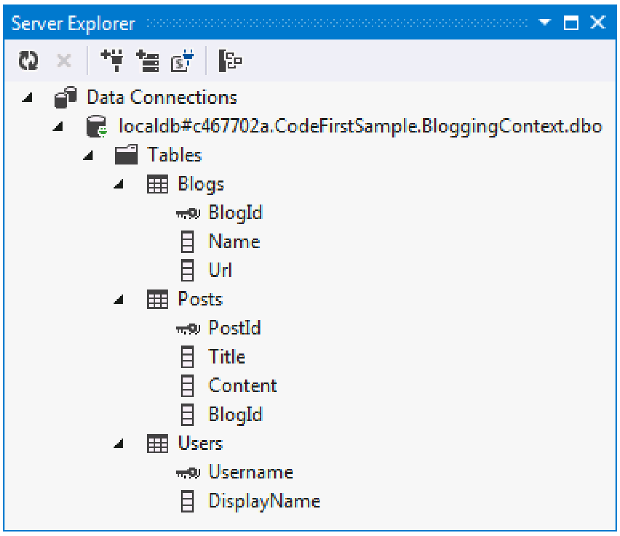
```
using System.ComponentModel.DataAnnotations;
```

- Now annotate the Username property to identify that it is the primary key

```
public class User
{
    [Key]
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- Use the **Add-Migration AddUser** command to scaffold a migration to apply these changes to the database
- Run the **Update-Database** command to apply the new migration to the database

The new table is now added to the database:



The full list of annotations supported by EF is:

- KeyAttribute
- StringLengthAttribute
- MaxLengthAttribute
- ConcurrencyCheckAttribute
- RequiredAttribute
- TimestampAttribute
- ComplexTypeAttribute
- ColumnAttribute
- TableAttribute
- InversePropertyAttribute
- ForeignKeyAttribute
- DatabaseGeneratedAttribute
- NotMappedAttribute

7. Fluent API

In the previous section we looked at using Data Annotations to supplement or override what was detected by convention. The other way to configure the model is via the Code First fluent API.

Most model configuration can be done using simple data annotations. The fluent API is a more advanced way of specifying model configuration that covers everything that data annotations can do in addition to some more advanced configuration not possible with data annotations. Data annotations and the fluent API can be used together.

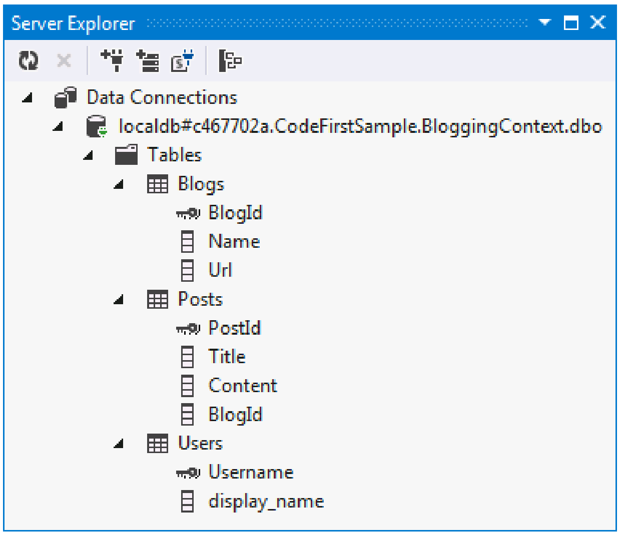
To access the fluent API you override the OnModelCreating method in DbContext. Let’s say we wanted to rename the column that User.DisplayName is stored in to display_name.

- Override the OnModelCreating method on BloggingContext with the following code

```
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");
    }
}
```

- Use the **Add-Migration ChangeDisplayName** command to scaffold a migration to apply these changes to the database.
- Run the **Update-Database** command to apply the new migration to the database.

The DisplayName column is now renamed to display_name:



Summary

In this walkthrough we looked at Code First development using a new database. We defined a model using classes then used that model to create a database and store and retrieve data. Once the database was created we used Code First Migrations to change the schema as our model evolved. We also saw how to configure a model using Data Annotations and the Fluent API.

Code First to an Existing Database

9/13/2018 • 5 minutes to read • Edit Online

This video and step-by-step walkthrough provide an introduction to Code First development targeting an existing database. Code First allows you to define your model using C# or VB.Net classes. Optionally additional configuration can be performed using attributes on your classes and properties or by using a fluent API.

Watch the video

This video is now available on Channel 9.

Pre-Requisites

You will need to have **Visual Studio 2012** or **Visual Studio 2013** installed to complete this walkthrough.

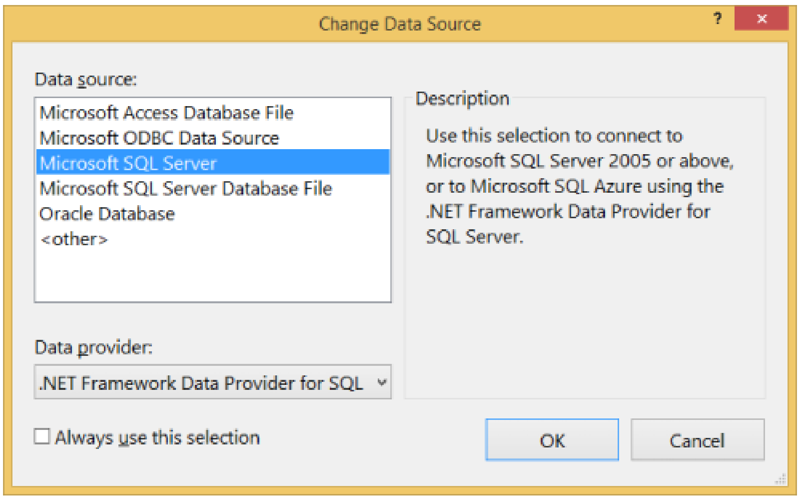
You will also need version **6.1** (or later) of the **Entity Framework Tools for Visual Studio** installed. See Get Entity Framework for information on installing the latest version of the Entity Framework Tools.

1. Create an Existing Database

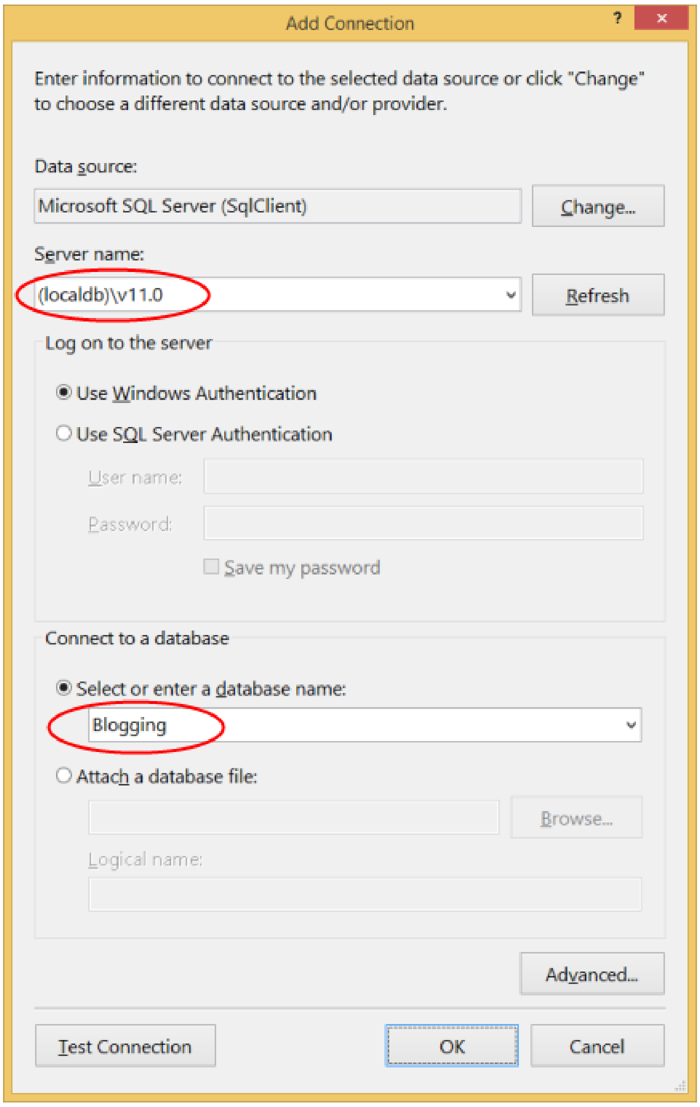
Typically when you are targeting an existing database it will already be created, but for this walkthrough we need to create a database to access.

Let's go ahead and generate the database.

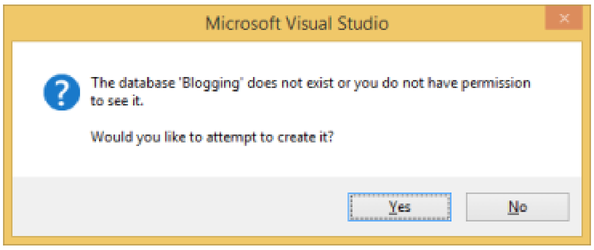
- Open Visual Studio
- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection...**
- If you haven't connected to a database from **Server Explorer** before you'll need to select **Microsoft SQL Server** as the data source



- Connect to your LocalDB instance, and enter **Blogging** as the database name



- Select **OK** and you will be asked if you want to create a new database, select **Yes**



- The new database will now appear in Server Explorer, right-click on it and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **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. Create the Application

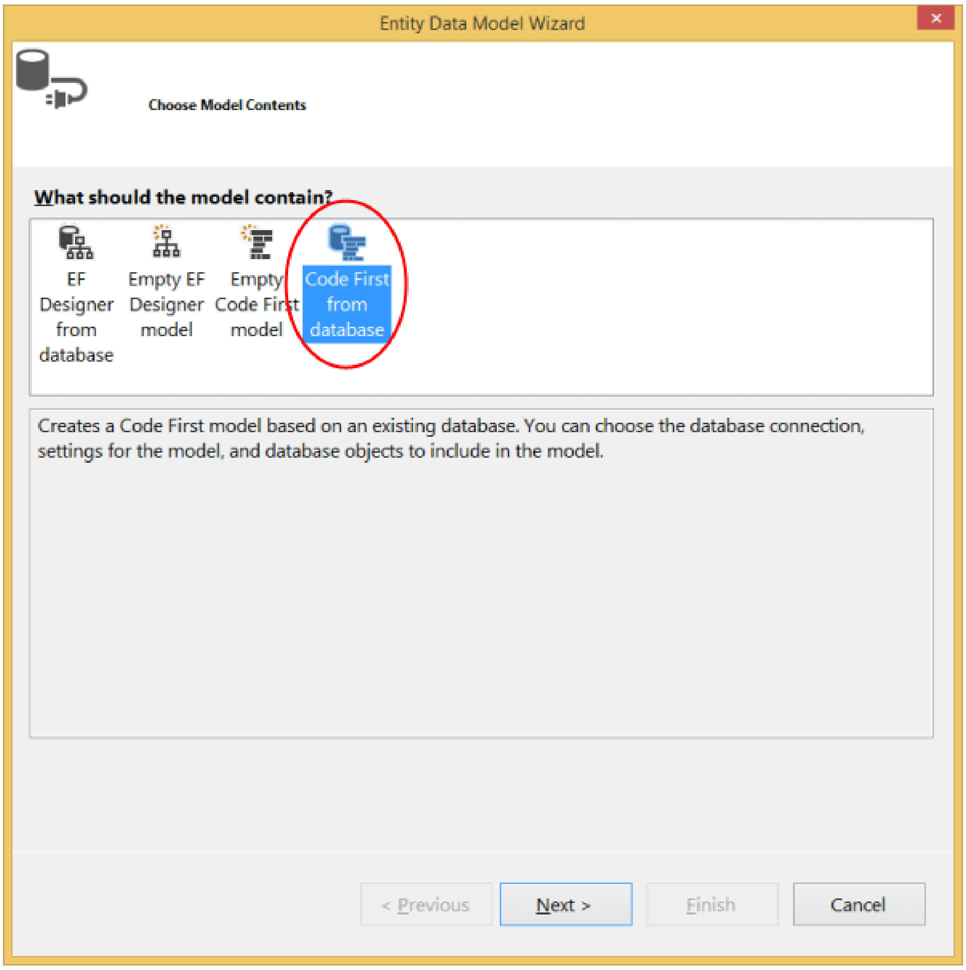
To keep things simple we're going to build a basic console application that uses Code First to perform data access:

- Open Visual Studio
- **File -> New -> Project…**
- Select **Windows** from the left menu and **Console Application**
- Enter **CodeFirstExistingDatabaseSample** as the name
- Select **OK**

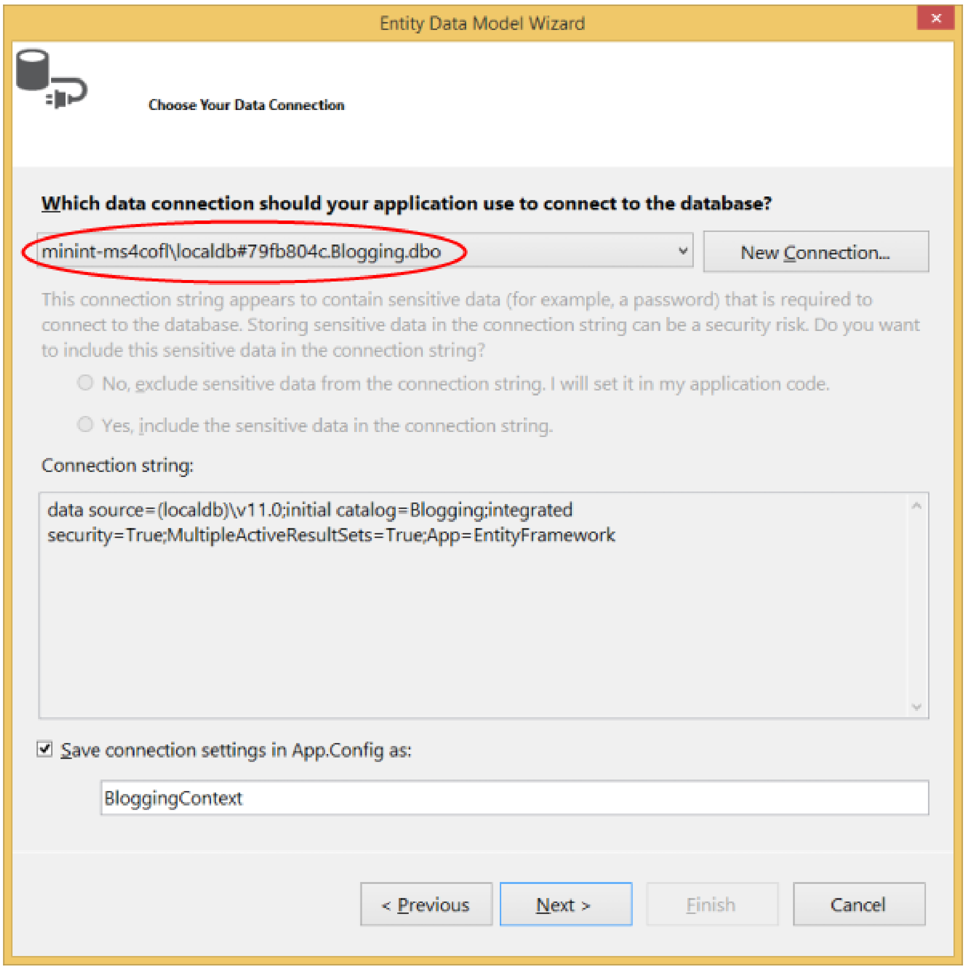
3. Reverse Engineer Model

We're going to make use of the Entity Framework Tools for Visual Studio to help us generate some initial code to map to the database. These tools are just generating code that you could also type by hand if you prefer.

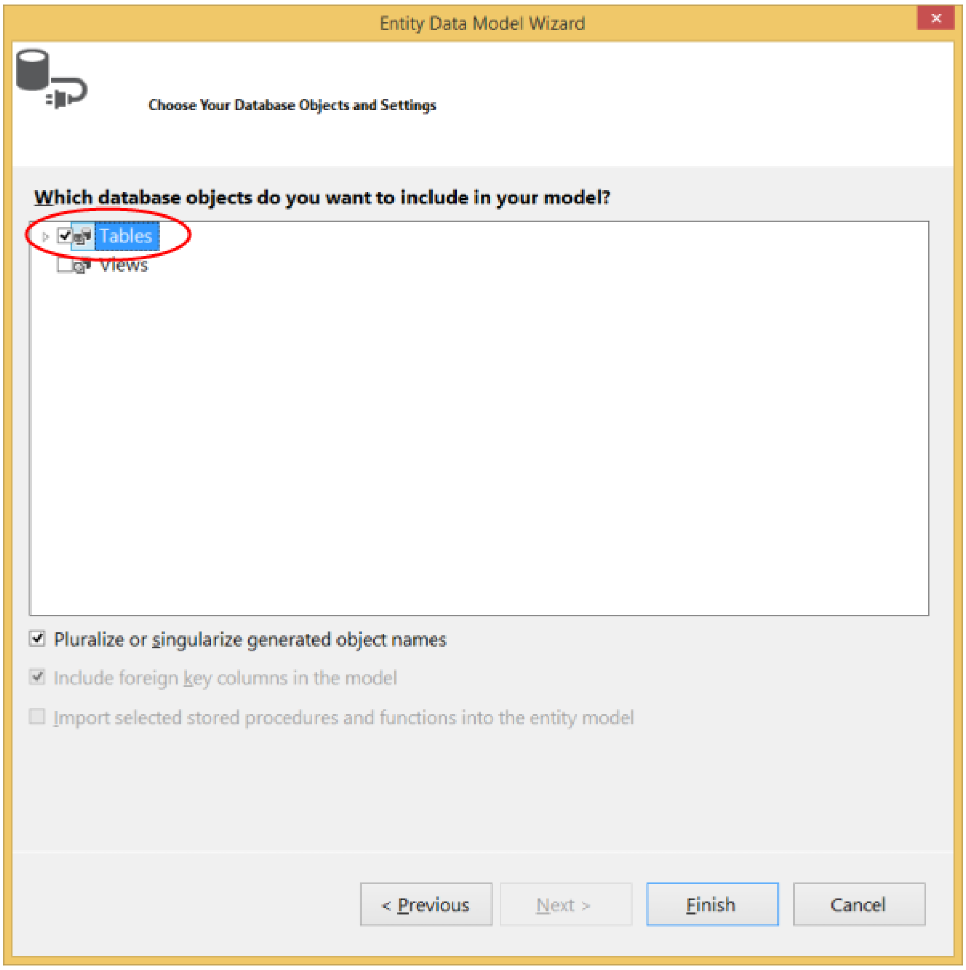
- **Project -> Add New Item…**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **BloggingContext** as the name and click **OK**
- This launches the **Entity Data Model Wizard**
- Select **Code First from Database** and click **Next**



- Select the connection to the database you created in the first section and click **Next**



- Click the checkbox next to **Tables** to import all tables and click **Finish**



Once the reverse engineer process completes a number of items will have been added to the project, let's take a look at what's been added.

Configuration file

An App.config file has been added to the project, this file contains the connection string to the existing database.

```
<connectionStrings>
  <add
    name="BloggingContext"
    connectionString="data source=(localdb)\mssqllocaldb;initial catalog=Blogging;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

You'll notice some other settings in the configuration file too, these are default EF settings that tell Code First where to create databases. Since we are mapping to an existing database these setting will be ignored in our application.

Derived Context

A **BloggingContext** class has been added to the project. The context represents a session with the database, allowing us to query and save data. The context exposes a **DbSet<TEntity>** for each type in our model. You'll also notice that the default constructor calls a base constructor using the **name=** syntax. This tells Code First that the connection string to use for this context should be loaded from the configuration file.

```
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)
        {
        }
    }
```

*You should always use the **name=** syntax when you are using a connection string in the config file. This ensures that if the connection string is not present then Entity Framework will throw rather than creating a new database by convention.*

Model classes

Finally, a **Blog** and **Post** class have also been added to the project. These are the domain classes that make up our model. You'll see Data Annotations applied to the classes to specify configuration where the Code First conventions would not align with the structure of the existing database. For example, you'll see the **StringLength** annotation on **Blog.Name** and **Blog.Url** since they have a maximum length of **200** in the database (the Code First default is to use the maximun length supported by the database provider - **nvarchar(max)** in 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. Reading & Writing Data

Now that we have a model it's time to use it to access some data. Implement the **Main** method in **Program.cs** as shown below. This code creates a new instance of our context and then uses it to insert a new **Blog**. Then it uses a LINQ query to retrieve all **Blogs** from the database ordered alphabetically by **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();
        }
    }
}
```

You can now run the application and test it out.

```
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...
```

What if My Database Changes?

The Code First to Database wizard is designed to generate a starting point set of classes that you can then tweak and modify. If your database schema changes you can either manually edit the classes or perform another reverse engineer to overwrite the classes.

Using Code First Migrations to an Existing Database

If you want to use Code First Migrations with an existing database, see Code First Migrations to an existing database.

Summary

In this walkthrough we looked at Code First development using an existing database. We used the Entity Framework Tools for Visual Studio to reverse engineer a set of classes that mapped to the database and could be used to store and retrieve data.

Code First Data Annotations

11/3/2018 • 16 minutes to read • Edit Online

NOTE

EF4.1 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 4.1. If you are using an earlier version, some or all of this information does not apply.

The content on this page is adapted from an article originally written by Julie Lerman (<http://thedatafarm.com>).

Entity Framework Code First allows you to use your own domain classes to represent the model that EF relies on to perform querying, change tracking, and updating functions. Code First leverages a programming pattern referred to as 'convention over configuration.' Code First will assume that your classes follow the conventions of Entity Framework, and in that case, will automatically work out how to perform it's job. However, if your classes do not follow those conventions, you have the ability to add configurations to your classes to provide EF with the requisite information.

Code First gives you two ways to add these configurations to your classes. One is using simple attributes called DataAnnotations, and the second is using Code First's Fluent API, which provides you with a way to describe configurations imperatively, in code.

This article will focus on using DataAnnotations (in the System.ComponentModel.DataAnnotations namespace) to configure your classes – highlighting the most commonly needed configurations. DataAnnotations are also understood by a number of .NET applications, such as ASP.NET MVC which allows these applications to leverage the same annotations for client-side validations.

The model

I'll demonstrate Code First DataAnnotations with a simple pair of classes: Blog and 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; }
}
```

As they are, the Blog and Post classes conveniently follow code first convention and require no tweaks to enable EF compatability. However, you can also use the annotations to provide more information to EF about the classes and the database to which they map.

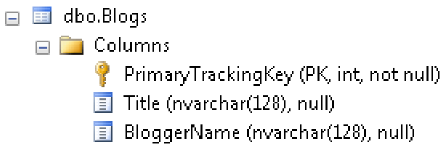
Key

Entity Framework relies on every entity having a key value that is used for entity tracking. One convention of Code First is implicit key properties; Code First will look for a property named “Id”, or a combination of class name and “Id”, such as “BlogId”. This property will map to a primary key column in the database.

The Blog and Post classes both follow this convention. What if they didn’t? What if Blog used the name *PrimaryTrackingKey* instead, or even *foo*? If code first does not find a property that matches this convention it will throw an exception because of Entity Framework’s requirement that you must have a key property. You can use the key annotation to specify which property is to be used as the 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; }
}
```

If you are using code first’s database generation feature, the Blog table will have a primary key column named PrimaryTrackingKey, which is also defined as Identity by default.



Composite keys

Entity Framework supports composite keys - primary keys that consist of more than one property. For example, you could have a Passport class whose primary key is a combination of PassportNumber and IssuingCountry.

```
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; }
}
```

Attempting to use the above class in your EF model would result in an `InvalidOperationException`:

Unable to determine composite primary key ordering for type 'Passport'. Use the ColumnAttribute or the HasKey method to specify an order for composite primary keys.

In order to use composite keys, Entity Framework requires you to define an order for the key properties. You can do this by using the Column annotation to specify an order.

> **NOTE**
>
> The order value is relative (rather than index based) so any values can be used. For example, 100 and 200 would be acceptable in place of 1 and 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; }
}
```

If you have entities with composite foreign keys, then you must specify the same column ordering that you used for the corresponding primary key properties.

Only the relative ordering within the foreign key properties needs to be the same, the exact values assigned to **Order** do not need to match. For example, in the following class, 3 and 4 could be used in place of 1 and 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; }
}
```

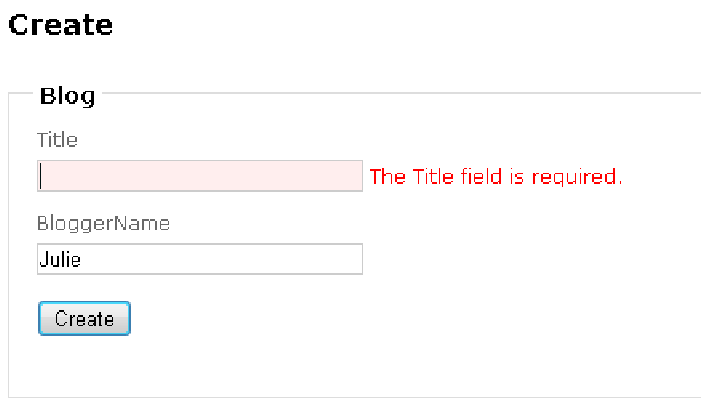
Required

The Required annotation tells EF that a particular property is required.

Adding Required to the Title property will force EF (and MVC) to ensure that the property has data in it.

```
[Required]
public string Title { get; set; }
```

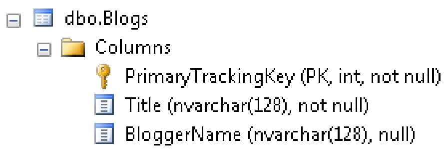
With no additional no code or markup changes in the application, an MVC application will perform client side validation, even dynamically building a message using the property and annotation names.



The Required attribute will also affect the generated database by making the mapped property non-nullable. Notice that the Title field has changed to “not null”.

NOTE

In some cases it may not be possible for the column in the database to be non-nullable even though the property is required. For example, when using a TPH inheritance strategy data for multiple types is stored in a single table. If a derived type includes a required property the column cannot be made non-nullable since not all types in the hierarchy will have this property.



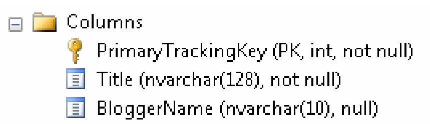
MaxLength and MinLength

The MaxLength and MinLength attributes allow you to specify additional property validations, just as you did with Required.

Here is the BloggerName with length requirements. The example also demonstrates how to combine attributes.

```
[MaxLength(10),MinLength(5)]
public string BloggerName { get; set; }
```

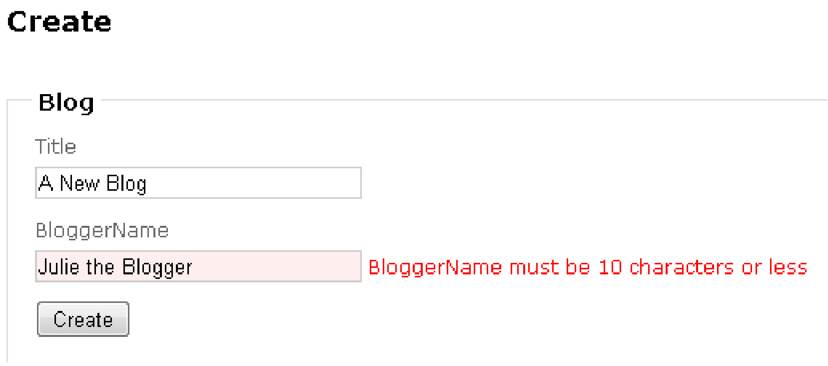
The MaxLength annotation will impact the database by setting the property’s length to 10.



MVC client-side annotation and EF 4.1 server-side annotation will both honor this validation, again dynamically building an error message: “The field BloggerName must be a string or array type with a maximum length of '10'.” That message is a little long. Many annotations let you specify an error message with the ErrorMessage attribute.

```
[MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]
public string BloggerName { get; set; }
```

You can also specify ErrorMessage in the Required annotation.



NotMapped

Code first convention dictates that every property that is of a supported data type is represented in the database. But this isn't always the case in your applications. For example you might have a property in the Blog class that creates a code based on the Title and BloggerName fields. That property can be created dynamically and does not need to be stored. You can mark any properties that do not map to the database with the NotMapped annotation such as this BlogCode property.

```
[NotMapped]
public string BlogCode
{
    get
    {
        return Title.Substring(0, 1) + ":" + BloggerName.Substring(0, 1);
    }
}
```

ComplexType

It's not uncommon to describe your domain entities across a set of classes and then layer those classes to describe a complete entity. For example, you may add a class called BlogDetails to your model.

```
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

Notice that BlogDetails does not have any type of key property. In domain driven design, BlogDetails is referred to as a value object. Entity Framework refers to value objects as complex types. Complex types cannot be tracked on their own.

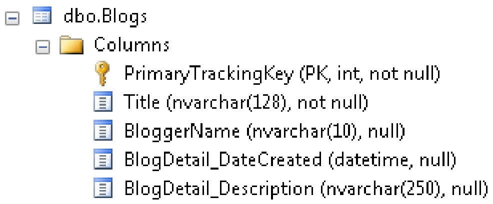
However as a property in the Blog class, BlogDetails it will be tracked as part of a Blog object. In order for code first to recognize this, you must mark the BlogDetails class as a ComplexType.

```
[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

Now you can add a property in the Blog class to represent the BlogDetails for that blog.

```
    public BlogDetails BlogDetail { get; set; }
```

In the database, the Blog table will contain all of the properties of the blog including the properties contained in its BlogDetail property. By default, each one is preceded with the name of the complex type, BlogDetail.



Another interesting note is that although the DateCreated property was defined as a non-nullable DateTime in the class, the relevant database field is nullable. You must use the Required annotation if you wish to affect the database schema.

ConcurrencyCheck

The ConcurrencyCheck annotation allows you to flag one or more properties to be used for concurrency checking in the database when a user edits or deletes an entity. If you've been working with the EF Designer, this aligns with setting a property's ConcurrencyMode to Fixed.

Let’s see how ConcurrencyCheck works by adding it to the BloggerName property.

```
[ConcurrencyCheck, MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]
public string BloggerName { get; set; }
```

When SaveChanges is called, because of the ConcurrencyCheck annotation on the BloggerName field, the original value of that property will be used in the update. The command will attempt to locate the correct row by filtering not only on the key value but also on the original value of BloggerName. Here are the critical parts of the UPDATE command sent to the database, where you can see the command will update the row that has a PrimaryTrackingKey is 1 and a BloggerName of “Julie” which was the original value when that blog was retrieved from the database.

```
where (([PrimaryTrackingKey] = @4) and ([BloggerName] = @5))
@4=1,@5=N'Julie'
```

If someone has changed the blogger name for that blog in the meantime, this update will fail and you’ll get a DbUpdateConcurrencyException that you'll need to handle.

TimeStamp

It's more common to use rowversion or timestamp fields for concurrency checking. But rather than using the ConcurrencyCheck annotation, you can use the more specific TimeStamp annotation as long as the type of the property is byte array. Code first will treat Timestamp properties the same as ConcurrencyCheck properties, but it will also ensure that the database field that code first generates is non-nullable. You can only have one timestamp property in a given class.

Adding the following property to the Blog class:

```
[Timestamp]
public Byte[] TimeStamp { get; set; }
```

results in code first creating a non-nullable timestamp column in the database table.

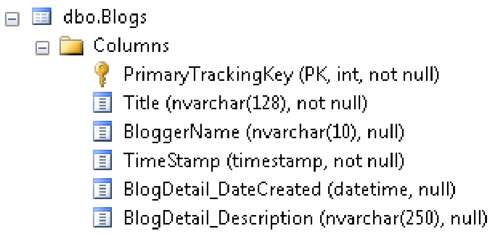


Table and Column

If you are letting Code First create the database, you may want to change the name of the tables and columns it is creating. You can also use Code First with an existing database. But it's not always the case that the names of the classes and properties in your domain match the names of the tables and columns in your database.

My class is named Blog and by convention, code first presumes this will map to a table named Blogs. If that's not the case you can specify the name of the table with the Table attribute. Here for example, the annotation is specifying that the table name is InternalBlogs.

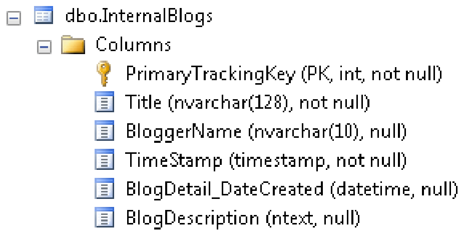
```
[Table("InternalBlogs")]
public class Blog
```

The Column annotation is a more adept in specifying the attributes of a mapped column. You can stipulate a name, data type or even the order in which a column appears in the table. Here is an example of the Column attribute.

```
[Column("BlogDescription", TypeName="ntext")]
public String Description {get;set;}
```

Don't confuse Column's TypeName attribute with the DataType DataAnnotation. DataType is an annotation used for the UI and is ignored by Code First.

Here is the table after it's been regenerated. The table name has changed to InternalBlogs and Description column from the complex type is now BlogDescription. Because the name was specified in the annotation, code first will not use the convention of starting the column name with the name of the complex type.



DatabaseGenerated

An important database features is the ability to have computed properties. If you're mapping your Code First classes to tables that contain computed columns, you don't want Entity Framework to try to update those columns. But you do want EF to return those values from the database after you've inserted or updated data. You can use the DatabaseGenerated annotation to flag those properties in your class along with the Computed enum. Other enums are None and Identity.

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public DateTime DateCreated { get; set; }
```

You can use database generated on byte or timestamp columns when code first is generating the database, otherwise you should only use this when pointing to existing databases because code first won't be able to determine the formula for the computed column.

You read above that by default, a key property that is an integer will become an identity key in the database. That would be the same as setting DatabaseGenerated to DatabaseGeneratedOption.Identity. If you do not want it to be an identity key, you can set the value to DatabaseGeneratedOption.None.

Index

> **NOTE**
>
> **EF6.1 Onwards Only** - The Index attribute was introduced in Entity Framework 6.1. If you are using an earlier version the information in this section does not apply.

You can create an index on one or more columns using the **IndexAttribute**. Adding the attribute to one or more properties will cause EF to create the corresponding index in the database when it creates the database, or scaffold the corresponding **CreateIndex** calls if you are using Code First Migrations.

For example, the following code will result in an index being created on the **Rating** column of the **Posts** table in the database.

```
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; }
}
```

By default, the index will be named **IX_<property name>** (IX_Rating in the above example). You can also specify a name for the index though. The following example specifies that the index should be named **PostRatingIndex**.

```
[Index("PostRatingIndex")]
public int Rating { get; set; }
```

By default, indexes are non-unique, but you can use the **IsUnique** named parameter to specify that an index should be unique. The following example introduces a unique index on a **User**'s login name.

```
public class User
{
    public int UserId { get; set; }

    [Index(IsUnique = true)]
    [StringLength(200)]
    public string Username { get; set; }

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

Multiple-Column Indexes

Indexes that span multiple columns are specified by using the same name in multiple Index annotations for a given table. When you create multi-column indexes, you need to specify an order for the columns in the index. For example, the following code creates a multi-column index on **Rating** and **BlogId** called **IX_BlogAndRating**. **BlogId** is the first column in the index and **Rating** is the second.

```
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; }
}
```

Relationship Attributes: InverseProperty and ForeignKey

NOTE

This page provides information about setting up relationships in your Code First model using Data Annotations. For general information about relationships in EF and how to access and manipulate data using relationships, see Relationships & Navigation Properties.*

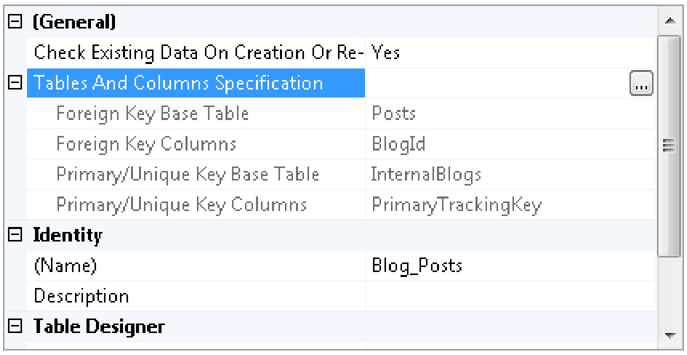
Code first convention will take care of the most common relationships in your model, but there are some cases where it needs help.

Changing the name of the key property in the Blog class created a problem with its relationship to Post.

When generating the database, code first sees the BlogId property in the Post class and recognizes it, by the convention that it matches a class name plus "Id", as a foreign key to the Blog class. But there is no BlogId property in the blog class. The solution for this is to create a navigation property in the Post and use the Foreign DataAnnotation to help code first understand how to build the relationship between the two classes —using the Post.BlogId property — as well as how to specify constraints in the database.

```
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; }
}
```

The constraint in the database shows a relationship between InternalBlogs.PrimaryTrackingKey and Posts.BlogId.



The InverseProperty is used when you have multiple relationships between classes.

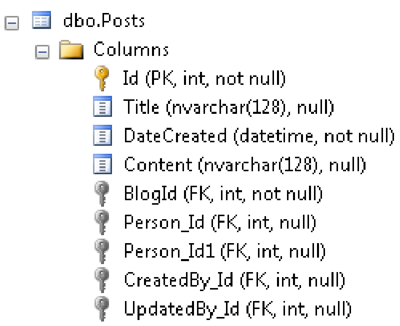
In the Post class, you may want to keep track of who wrote a blog post as well as who edited it. Here are two new navigation properties for the Post class.

```
public Person CreatedBy { get; set; }
public Person UpdatedBy { get; set; }
```

You'll also need to add in the Person class referenced by these properties. The Person class has navigation properties back to the Post, one for all of the posts written by the person and one for all of the posts updated by that person.

```
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; }
}
```

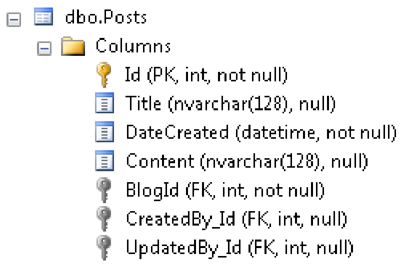
Code first is not able to match up the properties in the two classes on its own. The database table for Posts should have one foreign key for the CreatedBy person and one for the UpdatedBy person but code first will create four will foreign key properties: Person_Id, Person_Id1, CreatedBy_Id and UpdatedBy_Id.



To fix these problems, you can use the InverseProperty annotation to specify the alignment of the properties.

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

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

Because the PostsWritten property in Person knows that this refers to the Post type, it will build the relationship to Post.CreatedBy. Similarly, PostsUpdated will be connected to Post.UpdatedBy. And code first will not create the extra foreign keys.



Summary

DataAnnotations not only let you describe client and server side validation in your code first classes, but they also allow you to enhance and even correct the assumptions that code first will make about your classes based on its conventions. With DataAnnotations you can not only drive database schema generation, but you can also map your code first classes to a pre-existing database.

While they are very flexible, keep in mind that DataAnnotations provide only the most commonly needed configuration changes you can make on your code first classes. To configure your classes for some of the edge cases, you should look to the alternate configuration mechanism, Code First's Fluent API .

Defining DbSets

9/13/2018 • 2 minutes to read • Edit Online

When developing with the Code First workflow you define a derived DbContext that represents your session with the database and exposes a DbSet for each type in your model. This topic covers the various ways you can define the DbSet properties.

DbContext with DbSet properties

The common case shown in Code First examples is to have a DbContext with public automatic DbSet properties for the entity types of your model. For example:

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

When used in Code First mode, this will configure Blogs and Posts as entity types, as well as configuring other types reachable from these. In addition DbContext will automatically call the setter for each of these properties to set an instance of the appropriate DbSet.

DbContext with IDbSet properties

There are situations, such as when creating mocks or fakes, where it is more useful to declare your set properties using an interface. In such cases the IDbSet interface can be used in place of DbSet. For example:

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

This context works in exactly the same way as the context that uses the DbSet class for its set properties.

DbContext with read-only set properties

If you do not wish to expose public setters for your DbSet or IDbSet properties you can instead create read-only properties and create the set instances yourself. For example:

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs
    {
        get { return Set<Blog>(); }
    }

    public DbSet<Post> Posts
    {
        get { return Set<Post>(); }
    }
}
```

Note that DbContext caches the instance of DbSet returned from the Set method so that each of these properties will return the same instance every time it is called.

Discovery of entity types for Code First works in the same way here as it does for properties with public getters and setters.

Enum Support - Code First

9/18/2018 • 4 minutes to read • Edit Online

> **NOTE**
>
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

This video and step-by-step walkthrough shows how to use enum types with Entity Framework Code First. It also demonstrates how to use enums in a LINQ query.

This walkthrough will use Code First to create a new database, but you can also use Code First to map to an existing database.

Enum support was introduced in Entity Framework 5. To use the new features like enums, spatial data types, and table-valued functions, you must target .NET Framework 4.5. Visual Studio 2012 targets .NET 4.5 by default.

In Entity Framework, an enumeration can have the following underlying types: **Byte**, **Int16**, **Int32**, **Int64** , or **SByte**.

Watch the video

This video shows how to use enum types with Entity Framework Code First. It also demonstrates how to use enums in a LINQ query.

Presented By: Julia Kornich

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have Visual Studio 2012, Ultimate, Premium, Professional, or Web Express edition installed to complete this walkthrough.

Set up the Project

1. Open Visual Studio 2012
2. On the **File** menu, point to **New**, and then click **Project**
3. In the left pane, click **Visual C#**, and then select the **Console** template
4. Enter **EnumCodeFirst** as the name of the project and click **OK**

Define a New Model using Code First

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model. The code below defines the Department class.

The code also defines the DepartmentNames enumeration. By default, the enumeration is of **int** type. The Name property on the Department class is of the DepartmentNames type.

Open the Program.cs file and paste the following class definitions.

```
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; }
}
```

Define the DbContext Derived Type

In addition to defining entities, you need to define a class that derives from DbContext and exposes DbSet<TEntity> properties. The DbSet<TEntity> properties let the context know which types you want to include in the model.

An instance of the DbContext derived type manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.

The DbContext and DbSet types are defined in the EntityFramework assembly. We will add a reference to this DLL by using the EntityFramework NuGet package.

1. In Solution Explorer, right-click on the project name.
2. Select **Manage NuGet Packages…**
3. In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package.
4. Click **Install**

Note, that in addition to the EntityFramework assembly, references to System.ComponentModel.DataAnnotations and System.Data.Entity assemblies are added as well.

At the top of the Program.cs file, add the following using statement:

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

In the Program.cs add the context definition.

```
public partial class EnumTestContext : DbContext
{
    public DbSet<Department> Departments { get; set; }
}
```

Persist and Retrieve Data

Open the Program.cs file where the Main method is defined. Add the following code into the Main function. The code adds a new Department object to the context. It then saves the data. The code also executes a LINQ query that returns a Department where the name is 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);
}
```

Compile and run the application. The program produces the following output:

```
DepartmentID: 1 Name: English
```

View the Generated Database

When you run the application the first time, the Entity Framework creates a database for you. Because we have Visual Studio 2012 installed, the database will be created on the LocalDB instance. By default, the Entity Framework names the database after the fully qualified name of the derived context (for this example that is **EnumCodeFirst.EnumTestContext**). The subsequent times the existing database will be used.

Note, that if you make any changes to your model after the database has been created, you should use Code First Migrations to update the database schema. See Code First to a New Database for an example of using Migrations.

To view the database and data, do the following:

1. In the Visual Studio 2012 main menu, select **View** -> **SQL Server Object Explorer**.
2. If LocalDB is not in the list of servers, click the right mouse button on **SQL Server** and select **Add SQL Server** Use the default **Windows Authentication** to connect to the LocalDB instance
3. Expand the LocalDB node
4. Unfold the **Databases** folder to see the new database and browse to the **Department** table Note, that Code First does not create a table that maps to the enumeration type
5. To view data, right-click on the table and select **View Data**

Summary

In this walkthrough we looked at how to use enum types with Entity Framework Code First.

Spatial - Code First

9/18/2018 • 4 minutes to read • Edit Online

> **NOTE**
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

The video and step-by-step walkthrough shows how to map spatial types with Entity Framework Code First. It also demonstrates how to use a LINQ query to find a distance between two locations.

This walkthrough will use Code First to create a new database, but you can also use Code First to an existing database.

Spatial type support was introduced in Entity Framework 5. Note that to use the new features like spatial type, enums, and Table-valued functions, you must target .NET Framework 4.5. Visual Studio 2012 targets .NET 4.5 by default.

To use spatial data types you must also use an Entity Framework provider that has spatial support. See provider support for spatial types for more information.

There are two main spatial data types: geography and geometry. The geography data type stores ellipsoidal data (for example, GPS latitude and longitude coordinates). The geometry data type represents Euclidean (flat) coordinate system.

Watch the video

This video shows how to map spatial types with Entity Framework Code First. It also demonstrates how to use a LINQ query to find a distance between two locations.

Presented By: Julia Kornich

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have Visual Studio 2012, Ultimate, Premium, Professional, or Web Express edition installed to complete this walkthrough.

Set up the Project

1. Open Visual Studio 2012
2. On the **File** menu, point to **New**, and then click **Project**
3. In the left pane, click **Visual C#**, and then select the **Console** template
4. Enter **SpatialCodeFirst** as the name of the project and click **OK**

Define a New Model using Code First

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model. The code below defines the University class.

The University has the Location property of the DbGeography type. To use the DbGeography type, you must add a reference to the System.Data.Entity assembly and also add the System.Data.Spatial using statement.

Open the Program.cs file and paste the following using statements at the top of the file:

```
using System.Data.Spatial;
```

Add the following University class definition to the Program.cs file.

```
public class University
{
    public int UniversityID { get; set; }
    public string Name { get; set; }
    public DbGeography Location { get; set; }
}
```

Define the DbContext Derived Type

In addition to defining entities, you need to define a class that derives from DbContext and exposes DbSet<TEntity> properties. The DbSet<TEntity> properties let the context know which types you want to include in the model.

An instance of the DbContext derived type manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.

The DbContext and DbSet types are defined in the EntityFramework assembly. We will add a reference to this DLL by using the EntityFramework NuGet package.

1. In Solution Explorer, right-click on the project name.
2. Select **Manage NuGet Packages…**
3. In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package.
4. Click **Install**

Note, that in addition to the EntityFramework assembly, a reference to the System.ComponentModel.DataAnnotations assembly is also added.

At the top of the Program.cs file, add the following using statement:

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

In the Program.cs add the context definition.

```
public partial class UniversityContext : DbContext
{
    public DbSet<University> Universities { get; set; }
}
```

Persist and Retrieve Data

Open the Program.cs file where the Main method is defined. Add the following code into the Main function.

The code adds two new University objects to the context. Spatial properties are initialized by using the DbGeography.FromText method. The geography point represented as WellKnownText is passed to the method. The code then saves the data. Then, the LINQ query that that returns a University object where its location is closest to the specified location, is constructed and executed.

```
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);
}
```

Compile and run the application. The program produces the following output:

```
The closest University to you is: School of Fine Art.
```

View the Generated Database

When you run the application the first time, the Entity Framework creates a database for you. Because we have Visual Studio 2012 installed, the database will be created on the LocalDB instance. By default, the Entity Framework names the database after the fully qualified name of the derived context (in this example that is **SpatialCodeFirst.UniversityContext**). The subsequent times the existing database will be used.

Note, that if you make any changes to your model after the database has been created, you should use Code First Migrations to update the database schema. See Code First to a New Database for an example of using Migrations.

To view the database and data, do the following:

1. In the Visual Studio 2012 main menu, select **View** -> **SQL Server Object Explorer**.
2. If LocalDB is not in the list of servers, click the right mouse button on **SQL Server** and select **Add SQL Server** Use the default **Windows Authentication** to connect to the the LocalDB instance
3. Expand the LocalDB node
4. Unfold the **Databases** folder to see the new database and browse to the **Universities** table
5. To view data, right-click on the table and select **View Data**

Summary

In this walkthrough we looked at how to use spatial types with Entity Framework Code First.

Code First Conventions

9/13/2018 • 6 minutes to read • Edit Online

Code First enables you to describe a model by using C# or Visual Basic .NET classes. The basic shape of the model is detected by using conventions. Conventions are sets of rules that are used to automatically configure a conceptual model based on class definitions when working with Code First. The conventions are defined in the System.Data.Entity.ModelConfiguration.Conventions namespace.

You can further configure your model by using data annotations or the fluent API. Precedence is given to configuration through the fluent API followed by data annotations and then conventions. For more information see Data Annotations, Fluent API - Relationships, Fluent API - Types & Properties and Fluent API with VB.NET.

A detailed list of Code First conventions is available in the API Documentation. This topic provides an overview of the conventions used by Code First.

Type Discovery

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model. In addition to defining the classes, you also need to let **DbContext** know which types you want to include in the model. To do this, you define a context class that derives from **DbContext** and exposes **DbSet** properties for the types that you want to be part of the model. Code First will include these types and also will pull in any referenced types, even if the referenced types are defined in a different assembly.

If your types participate in an inheritance hierarchy, it is enough to define a **DbSet** property for the base class, and the derived types will be automatically included, if they are in the same assembly as the base class.

In the following example, there is only one **DbSet** property defined on the **SchoolEntities** class (**Departments**). Code First uses this property to discover and pull in any referenced types.

```
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; }
}
```

If you want to exclude a type from the model, use the **NotMapped** attribute or the **DbModelBuilder.Ignore** fluent API.

```
modelBuilder.Ignore<Department>();
```

Primary Key Convention

Code First infers that a property is a primary key if a property on a class is named “ID” (not case sensitive), or the class name followed by "ID". If the type of the primary key property is numeric or GUID it will be configured as an identity column.

```
public class Department
{
    // Primary key
    public int DepartmentID { get; set; }

    . . .

}
```

Relationship Convention

In Entity Framework, navigation properties provide a way to navigate a relationship between two entity types. Every object can have a navigation property for every relationship in which it participates. Navigation properties allow you to navigate and manage relationships in both directions, returning either a reference object (if the multiplicity is either one or zero-or-one) or a collection (if the multiplicity is many). Code First infers relationships based on the navigation properties defined on your types.

In addition to navigation properties, we recommend that you include foreign key properties on the types that represent dependent objects. Any property with the same data type as the principal primary key property and with a name that follows one of the following formats represents a foreign key for the relationship: '<navigation property name><principal primary key property name>', '<principal class name><primary key property name>', or '<principal primary key property name>'. If multiple matches are found then precedence is given in the order listed above. Foreign key detection is not case sensitive. When a foreign key property is detected, Code First infers the multiplicity of the relationship based on the nullability of the foreign key. If the property is nullable then the relationship is registered as optional; otherwise the relationship is registered as required.

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 multiplicity and cascade delete behavior detected by convention can be overridden by using the fluent API.

In the following example the navigation properties and a foreign key are used to define the relationship between the Department and Course classes.

```
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

If you have multiple relationships between the same types (for example, suppose you define the **Person** and **Book** classes, where the **Person** class contains the **ReviewedBooks** and **AuthoredBooks** navigation properties and the **Book** class contains the **Author** and **Reviewer** navigation properties) you need to manually configure the relationships by using Data Annotations or the fluent API. For more information, see Data Annotations - Relationships and Fluent API - Relationships.

Complex Types Convention

When Code First discovers a class definition where a primary key cannot be inferred, and no primary key is registered through data annotations or the fluent API, then the type is automatically registered as a complex type. Complex type detection also requires that the type does not have properties that reference entity types and is not referenced from a collection property on another type. Given the following class definitions Code First would infer that **Details** is a complex type because it has no primary key.

```
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; }
}
```

Connection String Convention

To learn about the conventions that DbContext uses to discover the connection to use see Connections and Models.

Removing Conventions

You can remove any of the conventions defined in the System.Data.Entity.ModelConfiguration.Conventions namespace. The following example removes **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>();
    }
}
```

Custom Conventions

Custom conventions are supported in EF6 onwards. For more information see Custom Code First Conventions.

Custom Code First Conventions

9/13/2018 • 11 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

When using Code First your model is calculated from your classes using a set of conventions. The default Code First Conventions determine things like which property becomes the primary key of an entity, the name of the table an entity maps to, and what precision and scale a decimal column has by default.

Sometimes these default conventions are not ideal for your model, and you have to work around them by configuring many individual entities using Data Annotations or the Fluent API. Custom Code First Conventions let you define your own conventions that provide configuration defaults for your model. In this walkthrough, we will explore the different types of custom conventions and how to create each of them.

Model-Based Conventions

This page covers the DbModelBuilder API for custom conventions. This API should be sufficient for authoring most custom conventions. However, there is also the ability to author model-based conventions - conventions that manipulate the final model once it is created - to handle advanced scenarios. For more information, see Model-Based Conventions.

Our Model

Let's start by defining a simple model that we can use with our conventions. Add the following classes to your project.

```
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; }
}
```

Introducing Custom Conventions

Let's write a convention that configures any property named Key to be the primary key for its entity type.

Conventions are enabled on the model builder, which can be accessed by overriding OnModelCreating in the context. Update the ProductContext class as follows:

```
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());
    }
}
```

Now, any property in our model named Key will be configured as the primary key of whatever entity its part of.

We could also make our conventions more specific by filtering on the type of property that we are going to configure:

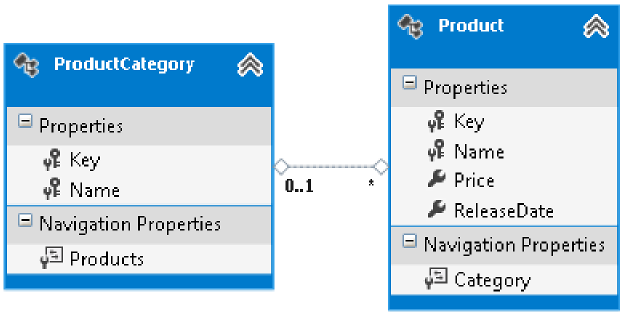
```
modelBuilder.Properties<int>()
            .Where(p => p.Name == "Key")
            .Configure(p => p.IsKey());
```

This will configure all properties called Key to be the primary key of their entity, but only if they are an integer.

An interesting feature of the IsKey method is that it is additive. Which means that if you call IsKey on multiple properties and they will all become part of a composite key. The one caveat for this is that when you specify multiple properties for a key you must also specify an order for those properties. You can do this by calling the HasColumnOrder method like below:

```
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));
```

This code will configure the types in our model to have a composite key consisting of the int Key column and the string Name column. If we view the model in the designer it would look like this:



Another example of property conventions is to configure all DateTime properties in my model to map to the datetime2 type in SQL Server instead of datetime. You can achieve this with the following:

```
modelBuilder.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
```

Convention Classes

Another way of defining conventions is to use a Convention Class to encapsulate your convention. When using a Convention Class then you create a type that inherits from the Convention class in the System.Data.Entity.ModelConfiguration.Conventions namespace.

We can create a Convention Class with the datetime2 convention that we showed earlier by doing the following:

```
public class DateTime2Convention : Convention
{
    public DateTime2Convention()
    {
        this.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
    }
}
```

To tell EF to use this convention you add it to the Conventions collection in OnModelCreating, which if you've been following along with the walkthrough will look like this:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Properties<int>()
                .Where(p => p.Name.EndsWith("Key"))
                .Configure(p => p.IsKey());

    modelBuilder.Conventions.Add(new DateTime2Convention());
}
```

As you can see we add an instance of our convention to the conventions collection. Inheriting from Convention provides a convenient way of grouping and sharing conventions across teams or projects. You could, for example, have a class library with a common set of conventions that all of your organizations projects use.

Custom Attributes

Another great use of conventions is to enable new attributes to be used when configuring a model. To illustrate this, let's create an attribute that we can use to mark String properties as non-Unicode.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class NonUnicode : Attribute
{
}
```

Now, let's create a convention to apply this attribute to our model:

```
modelBuilder.Properties()
            .Where(x => x.GetCustomAttributes(false).OfType<NonUnicode>().Any())
            .Configure(c => c.IsUnicode(false));
```

With this convention we can add the NonUnicode attribute to any of our string properties, which means the column in the database will be stored as varchar instead of nvarchar.

One thing to note about this convention is that if you put the NonUnicode attribute on anything other than a string property then it will throw an exception. It does this because you cannot configure IsUnicode on any type other than a string. If this happens, then you can make your convention more specific, so that it filters out anything that isn't a string.

While the above convention works for defining custom attributes there is another API that can be much easier to use, especially when you want to use properties from the attribute class.

For this example we are going to update our attribute and change it to an IsUnicode attribute, so it looks like this:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
internal class IsUnicode : Attribute
{
    public bool Unicode { get; set; }

    public IsUnicode(bool isUnicode)
    {
        Unicode = isUnicode;
    }
}
```

Once we have this, we can set a bool on our attribute to tell the convention whether or not a property should be Unicode. We could do this in the convention we have already by accessing the ClrProperty of the configuration class like this:

```
modelBuilder.Properties()
            .Where(x => x.GetCustomAttributes(false).OfType<IsUnicode>().Any())
            .Configure(c => c.IsUnicode(c.ClrPropertyInfo.GetCustomAttribute<IsUnicode>().Unicode));
```

This is easy enough, but there is a more succinct way of achieving this by using the Having method of the conventions API. The Having method has a parameter of type Func<PropertyInfo, T> which accepts the PropertyInfo the same as the Where method, but is expected to return an object. If the returned object is null then the property will not be configured, which means you can filter out properties with it just like Where, but it is different in that it will also capture the returned object and pass it to the Configure method. This works like the following:

```
modelBuilder.Properties()
            .Having(x =>x.GetCustomAttributes(false).OfType<IsUnicode>().FirstOrDefault())
            .Configure((config, att) => config.IsUnicode(att.Unicode));
```

Custom attributes are not the only reason to use the Having method, it is useful anywhere that you need to reason about something that you are filtering on when configuring your types or properties.

Configuring Types

So far all of our conventions have been for properties, but there is another area of the conventions API for configuring the types in your model. The experience is similar to the conventions we have seen so far, but the options inside configure will be at the entity instead of property level.

One of the things that Type level conventions can be really useful for is changing the table naming convention, either to map to an existing schema that differs from the EF default or to create a new database with a different naming convention. To do this we first need a method that can accept the TypeInfo for a type in our model and return what the table name for that type should be:

```
private string GetTableName(Type type)
{
    var result = Regex.Replace(type.Name, ".[A-Z]", m => m.Value[0] + "_" + m.Value[1]);

    return result.ToLower();
}
```

This method takes a type and returns a string that uses lower case with underscores instead of CamelCase. In our model this means that the ProductCategory class will be mapped to a table called product_category instead of ProductCategories.

Once we have that method we can call it in a convention like this:

```
modelBuilder.Types()
            .Configure(c => c.ToTable(GetTableName(c.ClrType)));
```

This convention configures every type in our model to map to the table name that is returned from our GetTableName method. This convention is the equivalent to calling the ToTable method for each entity in the model using the Fluent API.

One thing to note about this is that when you call ToTable EF will take the string that you provide as the exact table name, without any of the pluralization that it would normally do when determining table names. This is why the table name from our convention is product_category instead of product_categories. We can resolve that in our convention by making a call to the pluralization service ourselves.

In the following code we will use the Dependency Resolution feature added in EF6 to retrieve the pluralization service that EF would have used and pluralize our table name.

```
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**
>
> The generic version of GetService is an extension method in the System.Data.Entity.Infrastructure.DependencyResolution namespace, you will need to add a using statement to your context in order to use it.

ToTable and Inheritance

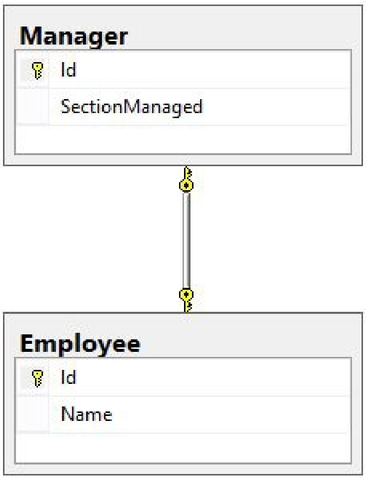
Another important aspect of ToTable is that if you explicitly map a type to a given table, then you can alter the mapping strategy that EF will use. If you call ToTable for every type in an inheritance hierarchy, passing the type name as the name of the table like we did above, then you will change the default Table-Per-Hierarchy (TPH) mapping strategy to Table-Per-Type (TPT). The best way to describe this is whith a concrete example:

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Manager : Employee
{
    public string SectionManaged { get; set; }
}
```

By default both employee and manager are mapped to the same table (Employees) in the database. The table will contain both employees and managers with a discriminator column that will tell you what type of instance is stored in each row. This is TPH mapping as there is a single table for the hierarchy. However, if you call ToTable on both classe then each type will instead be mapped to its own table, also known as TPT since each type has its own table.

```
modelBuilder.Types()
            .Configure(c=>c.ToTable(c.ClrType.Name));
```

The code above will map to a table structure that looks like the following:



You can avoid this, and maintain the default TPH mapping, in a couple ways:

1. Call ToTable with the same table name for each type in the hierarchy.
2. Call ToTable only on the base class of the hierarchy, in our example that would be employee.

Execution Order

Conventions operate in a last wins manner, the same as the Fluent API. What this means is that if you write two conventions that configure the same option of the same property, then the last one to execute wins. As an example, in the code below the max length of all strings is set to 500 but we then configure all properties called Name in the model to have a max length of 250.

```
modelBuilder.Properties<string>()
            .Configure(c => c.HasMaxLength(500));

modelBuilder.Properties<string>()
            .Where(x => x.Name == "Name")
            .Configure(c => c.HasMaxLength(250));
```

Because the convention to set max length to 250 is after the one that sets all strings to 500, all the properties called Name in our model will have a MaxLength of 250 while any other strings, such as descriptions, would be 500. Using conventions in this way means that you can provide a general convention for types or properties in your model and then overide them for subsets that are different.

The Fluent API and Data Annotations can also be used to override a convention in specific cases. In our example above if we had used the Fluent API to set the max length of a property then we could have put it before or after the convention, because the more specific Fluent API will win over the more general Configuration Convention.

Built-in Conventions

Because custom conventions could be affected by the default Code First conventions, it can be useful to add conventions to run before or after another convention. To do this you can use the AddBefore and AddAfter methods of the Conventions collection on your derived DbContext. The following code would add the convention class we created earlier so that it will run before the built in key discovery convention.

```
modelBuilder.Conventions.AddBefore<IdKeyDiscoveryConvention>(new DateTime2Convention());
```

This is going to be of the most use when adding conventions that need to run before or after the built in conventions, a list of the built in conventions can be found here: System.Data.Entity.ModelConfiguration.Conventions Namespace.

You can also remove conventions that you do not want applied to your model. To remove a convention, use the Remove method. Here is an example of removing the PluralizingTableNameConvention.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
}
```

Model-Based Conventions

9/18/2018 • 5 minutes to read • Edit Online

> **NOTE**
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Model based conventions are an advanced method of convention based model configuration. For most scenarios the Custom Code First Convention API on DbModelBuilder should be used. An understanding of the DbModelBuilder API for conventions is recommended before using model based conventions.

Model based conventions allow the creation of conventions that affect properties and tables which are not configurable through standard conventions. Examples of these are discriminator columns in table per hierarchy models and Independent Association columns.

Creating a Convention

The first step in creating a model based convention is choosing when in the pipeline the convention needs to be applied to the model. There are two types of model conventions, Conceptual (C-Space) and Store (S-Space). A C-Space convention is applied to the model that the application builds, whereas an S-Space convention is applied to the version of the model that represents the database and controls things such as how automatically-generated columns are named.

A model convention is a class that extends from either IConceptualModelConvention or IStoreModelConvention. These interfaces both accept a generic type that can be of type MetadataItem which is used to filter the data type that the convention applies to.

Adding a Convention

Model conventions are added in the same way as regular conventions classes. In the **OnModelCreating** method, add the convention to the list of conventions for a model.

```
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>();
    }
}
```

A convention can also be added in relation to another convention using the Conventions.AddBefore<> or Conventions.AddAfter<> methods. For more information about the conventions that Entity Framework applies see the notes section.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.AddAfter<IdKeyDiscoveryConvention>(new MyModelBasedConvention());
}
```

Example: Discriminator Model Convention

Renaming columns generated by EF is an example of something that you can't do with the other conventions APIs. This is a situation where using model conventions is your only option.

An example of how to use a model based convention to configure generated columns is customizing the way discriminator columns are named. Below is an example of a simple model based convention that renames every column in the model named "Discriminator" to "EntityType". This includes columns that the developer simply named "Discriminator". Since the "Discriminator" column is a generated column this needs to run in S-Space.

```
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";
        }
    }
}
```

Example: General IA Renaming Convention

Another more complicated example of model based conventions in action is to configure the way that Independent Associations (IAs) are named. This is a situation where Model conventions are applicable because IAs are generated by EF and aren't present in the model that the DbModelBuilder API can access.

When EF generates an IA, it creates a column named EntityType_KeyName. For example, for an association named Customer with a key column named CustomerId it would generate a column named Customer_CustomerId. The following convention strips the '_' character out of the column name that is generated for the 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);
            }
        }
    }
}
```

Extending Existing Conventions

If you need to write a convention that is similar to one of the conventions that Entity Framework already applies to your model you can always extend that convention to avoid having to rewrite it from scratch. An example of this is to replace the existing Id matching convention with a custom one. An added benefit to overriding the key convention is that the overridden method will get called only if there is no key already detected or explicitly configured. A list of conventions that used by Entity Framework is available here: 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;
    }
}
```

We then need to add our new convention before the existing key convention. After we add the CustomKeyDiscoveryConvention, we can remove the IdKeyDiscoveryConvention. If we didn't remove the existing IdKeyDiscoveryConvention this convention would still take precedence over the Id discovery convention since it is run first, but in the case where no "key" property is found, the "id" convention will run. We see this behavior because each convention sees the model as updated by the previous convention (rather than operating on it independently and all being combined together) so that if for example, a previous convention updated a column name to match something of interest to your custom convention (when before that the name was not of interest) then it will apply to that column.

```
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>();
    }
}
```

Notes

A list of conventions that are currently applied by Entity Framework is available in the MSDN documentation here: http://msdn.microsoft.com/library/system.data.entity.modelconfiguration.conventions.aspx. This list is pulled directly from our source code. The source code for Entity Framework 6 is available on GitHub and many of the conventions used by Entity Framework are good starting points for custom model based conventions.

Fluent API - Relationships

9/13/2018 • 6 minutes to read • Edit Online

NOTE

This page provides information about setting up relationships in your Code First model using the fluent API. For general information about relationships in EF and how to access and manipulate data using relationships, see Relationships & Navigation Properties.

When working with Code First, you define your model by defining your domain CLR classes. By default, Entity Framework uses the Code First conventions to map your classes to the database schema. If you use the Code First naming conventions, in most cases you can rely on Code First to set up relationships between your tables based on the foreign keys and navigation properties that you define on the classes. If you do not follow the conventions when defining your classes, or if you want to change the way the conventions work, you can use the fluent API or data annotations to configure your classes so Code First can map the relationships between your tables.

Introduction

When configuring a relationship with the fluent API, you start with the EntityTypeConfiguration instance and then use the HasRequired, HasOptional, or HasMany method to specify the type of relationship this entity participates in. The HasRequired and HasOptional methods take a lambda expression that represents a reference navigation property. The HasMany method takes a lambda expression that represents a collection navigation property. You can then configure an inverse navigation property by using the WithRequired, WithOptional, and WithMany methods. These methods have overloads that do not take arguments and can be used to specify cardinality with unidirectional navigations.

You can then configure foreign key properties by using the HasForeignKey method. This method takes a lambda expression that represents the property to be used as the foreign key.

Configuring a Required-to-Optional Relationship (One-to–Zero-or-One)

The following example configures a one-to-zero-or-one relationship. The OfficeAssignment has the InstructorID property that is a primary key and a foreign key, because the name of the property does not follow the convention the HasKey method is used to configure the primary key.

```
// 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);
```

Configuring a Relationship Where Both Ends Are Required (One-to-One)

In most cases Entity Framework can infer which type is the dependent and which is the principal in a relationship.

However, when both ends of the relationship are required or both sides are optional Entity Framework cannot identify the dependent and principal. When both ends of the relationship are required, use WithRequiredPrincipal or WithRequiredDependent after the HasRequired method. When both ends of the relationship are optional, use WithOptionalPrincipal or WithOptionalDependent after the HasOptional method.

```
// 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);
```

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<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

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<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
    {
        m.ToTable("CourseInstructor");
        m.MapLeftKey("CourseID");
        m.MapRightKey("InstructorID");
    });
```

Configuring a Relationship with One Navigation Property

A one-directional (also called unidirectional) relationship is when a navigation property is defined on only one of the relationship ends and not on both. By convention, Code First always interprets a unidirectional relationship as one-to-many. For example, if you want a one-to-one relationship between Instructor and OfficeAssignment, where you have a navigation property on only the Instructor type, you need to use the fluent API to configure this relationship.

```
// Configure the primary Key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal();
```

Enabling Cascade Delete

You can configure cascade delete on a relationship by using the WillCascadeOnDelete method. 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.

You can remove these cascade delete conventions by using:

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>()

The following code configures the relationship to be required and then disables cascade delete.

```
modelBuilder.Entity<Course>()
    .HasRequired(t => t.Department)
    .WithMany(t => t.Courses)
    .HasForeignKey(d => d.DepartmentID)
    .WillCascadeOnDelete(false);
```

Configuring a Composite Foreign Key

If the primary key on the Department type consisted of DepartmentID and Name properties, you would configure the primary key for the Department and the foreign key on the Course types as follows:

```
// Composite primary key
modelBuilder.Entity<Department>()
.HasKey(d => new { d.DepartmentID, d.Name });

// Composite foreign key
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

Renaming a Foreign Key That Is Not Defined in the Model

If you choose not to define a foreign key on the CLR type, but want to specify what name it should have in the database, do the following:

```
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Configuring a Foreign Key Name That Does Not Follow the Code First Convention

If the foreign key property on the Course class was called SomeDepartmentID instead of DepartmentID, you would need to do the following to specify that you want SomeDepartmentID to be the foreign key:

```
modelBuilder.Entity<Course>()
         .HasRequired(c => c.Department)
         .WithMany(d => d.Courses)
         .HasForeignKey(c => c.SomeDepartmentID);
```

Model Used in Samples

The following Code First model is used for the samples on this page.

```
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 - Configuring and Mapping Properties and Types

11/27/2018 • 11 minutes to read • Edit Online

When working with Entity Framework Code First the default behavior is to map your POCO classes to tables using a set of conventions baked into EF. Sometimes, however, you cannot or do not want to follow those conventions and need to map entities to something other than what the conventions dictate.

There are two main ways you can configure EF to use something other than conventions, namely annotations or EFs fluent API. The annotations only cover a subset of the fluent API functionality, so there are mapping scenarios that cannot be achieved using annotations. This article is designed to demonstrate how to use the fluent API to configure properties.

The code first fluent API is most commonly accessed by overriding the OnModelCreating method on your derived DbContext. The following samples are designed to show how to do various tasks with the fluent api and allow you to copy the code out and customize it to suit your model, if you wish to see the model that they can be used with as-is then it is provided at the end of this article.

Model-Wide Settings

Default Schema (EF6 onwards)

Starting with EF6 you can use the HasDefaultSchema method on DbModelBuilder to specify the database schema to use for all tables, stored procedures, etc. This default setting will be overridden for any objects that you explicitly configure a different schema for.

```
modelBuilder.HasDefaultSchema("sales");
```

Custom Conventions (EF6 onwards)

Starting with EF6 you can create your own conventions to supplement the ones included in Code First. For more details, see Custom Code First Conventions.

Property Mapping

The Property method is used to configure attributes for each property belonging to an entity or complex type. The Property method is used to obtain a configuration object for a given property. The options on the configuration object are specific to the type being configured; IsUnicode is available only on string properties for example.

Configuring a Primary Key

The Entity Framework convention for primary keys is:

1. Your class defines a property whose name is "ID" or "Id"
2. or a class name followed by "ID" or "Id"

To explicitly set a property to be a primary key, you can use the HasKey method. In the following example, the HasKey method is used to configure the InstructorID primary key on the OfficeAssignment type.

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

Configuring a Composite Primary Key

The following example configures the DepartmentID and Name properties to be the composite primary key of the Department type.

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

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<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

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. If Code First creates a database from this model it will also set the maximum length of the Name column to 50 characters.

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

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. If Code First creates a database from this model then the column used to store this property will usually be non-nullable.

> **NOTE**
>
> In some cases it may not be possible for the column in the database to be non-nullable even though the property is required. For example, when using a TPH inheritance strategy data for multiple types is stored in a single table. If a derived type includes a required property the column cannot be made non-nullable since not all types in the hierarchy will have this property.

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

Configuring an Index on one or more properties

> **NOTE**
>
> **EF6.1 Onwards Only** - The Index attribute was introduced in Entity Framework 6.1. If you are using an earlier version the information in this section does not apply.

Creating indexes isn't natively supported by the Fluent API, but you can make use of the support for **IndexAttribute** via the Fluent API. Index attributes are processed by including a model annotation on the model that is then turned into an Index in the database later in the pipeline. You can manually add these same annotations using the Fluent API.

The easiest way to do this is to create an instance of **IndexAttribute** that contains all the settings for the new index. You can then create an instance of **IndexAnnotation** which is an EF specific type that will convert the **IndexAttribute** settings into a model annotation that can be stored on the EF model. These can then be passed to the **HasColumnAnnotation** method on the Fluent API, specifying the name **Index** for the annotation.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute()));
```

For a complete list of the settings available in **IndexAttribute**, see the *Index* section of Code First Data Annotations. This includes customizing the index name, creating unique indexes, and creating multi-column indexes.

You can specify multiple index annotations on a single property by passing an array of **IndexAttribute** to the constructor of **IndexAnnotation**.

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation(
        "Index",
        new IndexAnnotation(new[]
            {
                new IndexAttribute("Index1"),
                new IndexAttribute("Index2") { IsUnique = true }
            }));
```

Specifying Not to Map a CLR Property to a Column in the Database

The following example shows how to specify that a property on a CLR type is not mapped to a column in the database.

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

Mapping a CLR Property to a Specific Column in the Database

The following example maps the Name CLR property to the DepartmentName database column.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

Renaming a Foreign Key That Is Not Defined in the Model

If you choose not to define a foreign key on a CLR type, but want to specify what name it should have in the database, do the following:

```
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

Configuring whether a String Property Supports Unicode Content

By default strings are Unicode (nvarchar in SQL Server). You can use the IsUnicode method to specify that a string should be of varchar type.

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .IsUnicode(false);
```

Configuring the Data Type of a Database Column

The HasColumnType method enables mapping to different representations of the same basic type. Using this method does not enable you to perform any conversion of the data at run time. Note that IsUnicode is the preferred way of setting columns to varchar, as it is database agnostic.

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

Configuring Properties on a Complex Type

There are two ways to configure scalar properties on a complex type.

You can call Property on ComplexTypeConfiguration.

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

You can also use the dot notation to access a property of a complex type.

```
modelBuilder.Entity<OnsiteCourse>()
    .Property(t => t.Details.Location)
    .HasMaxLength(20);
```

Configuring a Property to Be Used as an Optimistic Concurrency Token

To specify that a property in an entity represents a concurrency token, you can use either the ConcurrencyCheck attribute or the IsConcurrencyToken method.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

You can also use the IsRowVersion method to configure the property to be a row version in the database. Setting the property to be a row version automatically configures it to be an optimistic concurrency token.

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsRowVersion();
```

Type Mapping

Specifying That a Class Is a Complex Type

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<Details>();
```

Specifying Not to Map a CLR Entity Type to a Table in the Database

The following example shows how to exclude a CLR type from being mapped to a table in the database.

```
modelBuilder.Ignore<OnlineCourse>();
```

Mapping an Entity Type to a Specific Table in the Database

All properties of Department will be mapped to columns in a table called t_ Department.

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

You can also specify the schema name like this:

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

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<Course>()
    .Map<Course>(m => m.Requires("Type").HasValue("Course"))
    .Map<OnsiteCourse>(m => m.Requires("Type").HasValue("OnsiteCourse"));
```

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<Course>().ToTable("Course");
modelBuilder.Entity<OnsiteCourse>().ToTable("OnsiteCourse");
```

Mapping the Table-Per-Concrete Class (TPC) Inheritance

In the TPC mapping scenario, all non-abstract types in the hierarchy are mapped to individual tables. The tables that map to the derived classes have no relationship to the table that maps to the base class in the database. All properties of a class, including inherited properties, are mapped to columns of the corresponding table.

Call the MapInheritedProperties method to configure each derived type. MapInheritedProperties remaps all properties that were inherited from the base class to new columns in the table for the derived class.

NOTE

Note that because the tables participating in TPC inheritance hierarchy do not share a primary key there will be duplicate entity keys when inserting in tables that are mapped to subclasses if you have database generated values with the same identity seed. To solve this problem you can either specify a different initial seed value for each table or switch off identity on the primary key property. Identity is the default value for integer key properties when working with 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");
});
```

Mapping Properties of an Entity Type to Multiple Tables in the Database (Entity Splitting)

Entity splitting allows the properties of an entity type to be spread across multiple tables. In the following example, the Department entity is split into two tables: Department and DepartmentDetails. Entity splitting uses multiple calls to the Map method to map a subset of properties to a specific table.

```
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");
    });
```

Mapping Multiple Entity Types to One Table in the Database (Table Splitting)

The following example maps two entity types that share a primary key to one table.

```
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

Mapping an Entity Type to Insert/Update/Delete Stored Procedures (EF6 onwards)

Starting with EF6 you can map an entity to use stored procedures for insert update and delete. For more details see, Code First Insert/Update/Delete Stored Procedures.

Model Used in Samples

The following Code First model is used for the samples on this page.

```
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 with VB.NET

9/18/2018 • 9 minutes to read • Edit Online

Code First allows you to define your model using C# or VB.NET classes. Additional configuration can optionally be performed using attributes on your classes and properties or by using a fluent API. This walkthrough shows how to perform fluent API configuration using VB.NET.

This page assumes you have a basic understanding of Code First. Check out the following walkthroughs for more information on Code First:

- Code First to a New Database
- Code First to an Existing Database

Pre-Requisites

You will need to have at least Visual Studio 2010 or Visual Studio 2012 installed to complete this walkthrough.

If you are using Visual Studio 2010, you will also need to have NuGet installed

Create the Application

To keep things simple we're going to build a basic console application that uses Code First to perform data access.

- Open Visual Studio
- **File -> New -> Project…**
- Select **Windows** from the left menu and **Console Application**
- Enter **CodeFirstVBSample** as the name
- Select **OK**

Define the Model

In this step you will define VB.NET POCO entity types that represent the conceptual model. The classes do not need to derive from any base classes or implement any interfaces.

- Add a new class to the project, enter **SchoolModel** for the class name
- Replace the contents of the new class with the following code

```
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
```

```
    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
```

```
    Navigation property
        Public Overridable Property Instructor() As Instructor
    End Class
```

Define a Derived Context

We're about to start to using types from the Entity Framework so we need to add the EntityFramework NuGet package.

- **Project –> **Manage NuGet Packages…**

> **NOTE**
>
> If you don't have the **Manage NuGet Packages…** option you should install the latest version of NuGet

- Select the **Online** tab
- Select the **EntityFramework** package
- Click **Install**

Now it's time to define a derived context, which represents a session with the database, allowing us to query and save data. We define a context that derives from System.Data.Entity.DbContext and exposes a typed DbSet<TEntity> for each class in our model.

- Add a new class to the project, enter **SchoolContext** for the class name
- Replace the contents of the new class with the following code

```
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
```

Configuring with the Fluent API

This section demonstrates how to use the fluent APIs to configure types to tables mapping, properties to columns mapping, and relationships between tables\type in your model. The fluent API is exposed through the **DbModelBuilder** type and is most commonly accessed by overriding the **OnModelCreating** method on **DbContext**.

- Copy the following code and add it to the **OnModelCreating** method defined on the **SchoolContext** class The comments explain what each mapping does

```
' 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)
```

Using the Model

Let's perform some data access using the **SchoolContext** to see out model in action.

- Open the Module1.vb file where the Main function is defined
- Copy and paste the following Module1 definition

```vb
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
```

You can now run the application and test it out.

```
Enter a name for a new Department: Computing
All Departments in the database:
Computing
Press any key to exit...
```

Code First Insert, Update, and Delete Stored Procedures

9/13/2018 • 8 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

By default, Code First will configure all entities to perform insert, update and delete commands using direct table access. Starting in EF6 you can configure your Code First model to use stored procedures for some or all entities in your model.

Basic Entity Mapping

You can opt into using stored procedures for insert, update and delete using the Fluent API.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures();
```

Doing this will cause Code First to use some conventions to build the expected shape of the stored procedures in the database.

- Three stored procedures named **<type_name>_Insert**, **<type_name>_Update** and **<type_name>_Delete** (for example, Blog_Insert, Blog_Update and Blog_Delete).
- Parameter names correspond to the property names.

  NOTE

  If you use HasColumnName() or the Column attribute to rename the column for a given property then this name is used for parameters instead of the property name.

- **The insert stored procedure** will have a parameter for every property, except for those marked as store generated (identity or computed). The stored procedure should return a result set with a column for each store generated property.
- **The update stored procedure** will have a parameter for every property, except for those marked with a store generated pattern of 'Computed'. Some concurrency tokens require a parameter for the original value, see the *Concurrency Tokens* section below for details. The stored procedure should return a result set with a column for each computed property.
- **The delete stored procedure** should have a parameter for the key value of the entity (or multiple parameters if the entity has a composite key). Additionally, the delete procedure should also have parameters for any independent association foreign keys on the target table (relationships that do not have corresponding foreign key properties declared in the entity). Some concurrency tokens require a parameter for the original value, see the *Concurrency Tokens* section below for details.

Using the following class as an example:

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

The default stored procedures would be:

```
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
```

Overriding the Defaults

You can override part or all of what was configured by default.

You can change the name of one or more stored procedures. This example renames the update stored procedure only.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures(s =>
    s.Update(u => u.HasName("modify_blog")));
```

This example renames all three stored procedures.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures(s =>
    s.Update(u => u.HasName("modify_blog"))
     .Delete(d => d.HasName("delete_blog"))
     .Insert(i => i.HasName("insert_blog")));
```

In these examples the calls are chained together, but you can also use lambda block syntax.

```
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"));
    });
```

This example renames the parameter for the BlogId property on the update stored procedure.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures(s =>
    s.Update(u => u.Parameter(b => b.BlogId, "blog_id")));
```

These calls are all chainable and composable. Here is an example that renames all three stored procedures and their parameters.

```
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")));
```

You can also change the name of the columns in the result set that contains database generated values.

```
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
```

Relationships Without a Foreign Key in the Class (Independent Associations)

When a foreign key property is included in the class definition, the corresponding parameter can be renamed in the same way as any other property. When a relationship exists without a foreign key property in the class, the default parameter name is **<navigation_property_name>_<primary_key_name>**.

For example, the following class definitions would result in a Blog_BlogId parameter being expected in the stored procedures to insert and update Posts.

```
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; }
}
```

Overriding the Defaults

You can change parameters for foreign keys that are not included in the class by supplying the path to the primary key property to the Parameter method.

```
modelBuilder
  .Entity<Post>()
  .MapToStoredProcedures(s =>
    s.Insert(i => i.Parameter(p => p.Blog.BlogId, "blog_id")));
```

If you don’t have a navigation property on the dependent entity (i.e no Post.Blog property) then you can use the Association method to identify the other end of the relationship and then configure the parameters that correspond to each of the key property(s).

```
modelBuilder
  .Entity<Post>()
  .MapToStoredProcedures(s =>
    s.Insert(i => i.Navigation<Blog>(
      b => b.Posts,
      c => c.Parameter(b => b.BlogId, "blog_id"))));
```

Concurrency Tokens

Update and delete stored procedures may also need to deal with concurrency:

- If the entity contains concurrency tokens, the stored procedure can optionally have an output parameter that returns the number of rows updated/deleted (rows affected). Such a parameter must be configured using the RowsAffectedParameter method.
  By default EF uses the return value from ExecuteNonQuery to determine how many rows were affected. Specifying a rows affected output parameter is useful if you perform any logic in your sproc that would result in the return value of ExecuteNonQuery being incorrect (from EF's perspective) at the end of execution.
- For each concurrency token there will be a parameter named **<property_name>_Original** (for example, Timestamp_Original). This will be passed the original value of this property – the value when queried from the database.
  - Concurrency tokens that are computed by the database – such as timestamps – will only have an original value parameter.

- Non-computed properties that are set as concurrency tokens will also have a parameter for the new value in the update procedure. This uses the naming conventions already discussed for new values. An example of such a token would be using a Blog's URL as a concurrency token, the new value is required because this can be updated to a new value by your code (unlike a Timestamp token which is only updated by the database).

This is an example class and update stored procedure with a timestamp concurrency token.

```
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
```

Here is an example class and update stored procedure with non-computed concurrency token.

```
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
```

Overriding the Defaults

You can optionally introduce a rows affected parameter.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures(s =>
    s.Update(u => u.RowsAffectedParameter("rows_affected")));
```

For database computed concurrency tokens – where only the original value is passed – you can just use the standard parameter renaming mechanism to rename the parameter for the original value.

```
modelBuilder
  .Entity<Blog>()
  .MapToStoredProcedures(s =>
    s.Update(u => u.Parameter(b => b.Timestamp, "blog_timestamp")));
```

For non-computed concurrency tokens – where both the original and new value are passed – you can use an overload of Parameter that allows you to supply a name for each parameter.

```
modelBuilder
 .Entity<Blog>()
 .MapToStoredProcedures(s => s.Update(u => u.Parameter(b => b.Url, "blog_url", "blog_original_url")));
```

Many to Many Relationships

We'll use the following classes as an example in this section.

```
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; }
}
```

Many to many relationships can be mapped to stored procedures with the following syntax.

```
modelBuilder
  .Entity<Post>()
  .HasMany(p => p.Tags)
  .WithMany(t => t.Posts)
  .MapToStoredProcedures();
```

If no other configuration is supplied then the following stored procedure shape is used by default.

- Two stored procedures named **<type_one><type_two>_Insert** and **<type_one><type_two>_Delete** (for example, PostTag_Insert and PostTag_Delete).
- The parameters will be the key value(s) for each type. The name of each parameter being **<type_name>_<property_name>** (for example, Post_PostId and Tag_TagId).

Here are example insert and update stored procedures.

```
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
```

Overriding the Defaults

The procedure and parameter names can be configured in a similar way to entity stored procedures.

```
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

9/27/2018 • 11 minutes to read • Edit Online

Code First Migrations is the recommended way to evolve your application's database schema if you are using the Code First workflow. Migrations provide a set of tools that allow:

1. Create an initial database that works with your EF model
2. Generating migrations to keep track of changes you make to your EF model
3. Keep your database up to date with those changes

The following walkthrough will provide an overview Code First Migrations in Entity Framework. You can either complete the entire walkthrough or skip to the topic you are interested in. The following topics are covered:

Building an Initial Model & Database

Before we start using migrations we need a project and a Code First model to work with. For this walkthrough we are going to use the canonical **Blog** and **Post** model.

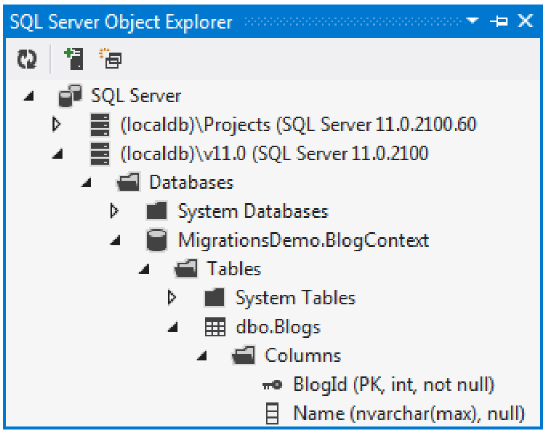
- Create a new **MigrationsDemo** Console application
- Add the latest version of the **EntityFramework** NuGet package to the project
  - **Tools –> Library Package Manager –> Package Manager Console**
  - Run the **Install-Package EntityFramework** command
- Add a **Model.cs** file with the code shown below. This code defines a single **Blog** class that makes up our domain model and a **BlogContext** class that is our EF Code First context

```
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; }
    }
}
```

- Now that we have a model it's time to use it to perform data access. Update the **Program.cs** file with the code shown below.

```
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();
        }
    }
}
```

- Run your application and you will see that a **MigrationsCodeDemo.BlogContext** database is created for you.



Enabling Migrations

It's time to make some more changes to our model.

- Let's introduce a Url property to the Blog class.

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

If you were to run the application again you would get an InvalidOperationException stating *The model backing the 'BlogContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).*

As the exception suggests, it's time to start using Code First Migrations. The first step is to enable migrations for our context.

- Run the **Enable-Migrations** command in Package Manager Console

This command has added a **Migrations** folder to our project. This new folder contains two files:

- **The Configuration class.** This class allows you to configure how Migrations behaves for your context. For this walkthrough we will just use the default configuration. *Because there is just a single Code First context in your project, Enable-Migrations has automatically filled in the context type this configuration applies to.*
- **An InitialCreate migration**. This migration was generated because we already had Code First create a database for us, before we enabled migrations. The code in this scaffolded migration represents the objects that have already been created in the database. In our case that is the **Blog** table with a **BlogId** and **Name** columns. The filename includes a timestamp to help with ordering. *If the database had not already been created this InitialCreate migration would not have been added to the project. Instead, the first time we call Add-Migration the code to create these tables would be scaffolded to a new migration.*

Multiple Models Targeting the Same Database

When using versions prior to EF6, only one Code First model could be used to generate/manage the schema of a database. This is the result of a single **__MigrationsHistory** table per database with no way to identify which entries belong to which model.

Starting with EF6, the **Configuration** class includes a **ContextKey** property. This acts as a unique identifier for each Code First model. A corresponding column in the **__MigrationsHistory** table allows entries from multiple models to share the table. By default, this property is set to the fully qualified name of your context.

Generating & Running Migrations

Code First Migrations has two primary commands that you are going to become familiar with.

- **Add-Migration** will scaffold the next migration based on changes you have made to your model since the last migration was created
- **Update-Database** will apply any pending migrations to the database

We need to scaffold a migration to take care of the new Url property we have added. The **Add-Migration** command allows us to give these migrations a name, let's just call ours **AddBlogUrl**.

- Run the **Add-Migration AddBlogUrl** command in Package Manager Console
- In the **Migrations** folder we now have a new **AddBlogUrl** migration. The migration filename is pre-fixed with a timestamp to help with ordering

```
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");
        }
    }
}
```

We could now edit or add to this migration but everything looks pretty good. Let's use **Update-Database** to apply this migration to the database.

- Run the **Update-Database** command in Package Manager Console
- Code First Migrations will compare the migrations in our **Migrations** folder with the ones that have been applied to the database. It will see that the **AddBlogUrl** migration needs to be applied, and run it.

The **MigrationsDemo.BlogContext** database is now updated to include the **Url** column in the **Blogs** table.

Customizing Migrations

So far we've generated and run a migration without making any changes. Now let's look at editing the code that gets generated by default.

- It's time to make some more changes to our model, let's add a new **Rating** property to the **Blog** class

```
public int Rating { get; set; }
```

- Let's also add a new **Post** class

```
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; }
}
```

- We'll also add a **Posts** collection to the **Blog** class to form the other end of the relationship between **Blog** and **Post**

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

We'll use the **Add-Migration** command to let Code First Migrations scaffold its best guess at the migration for us. We're going to call this migration **AddPostClass**.

- Run the **Add-Migration AddPostClass** command in Package Manager Console.

Code First Migrations did a pretty good job of scaffolding these changes, but there are some things we might want to change:

1. First up, let's add a unique index to **Posts.Title** column (Adding in line 22 & 29 in the code below).
2. We're also adding a non-nullable **Blogs.Rating** column. If there is any existing data in the table it will get assigned the CLR default of the data type for new column (Rating is integer, so that would be **0**). But we want to specify a default value of **3** so that existing rows in the **Blogs** table will start with a decent rating. (You can see the default value specified on line 24 of the code below)

```
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");
        }
    }
}
```

Our edited migration is ready to go, so let's use **Update-Database** to bring the database up-to-date. This time let's specify the **–Verbose** flag so that you can see the SQL that Code First Migrations is running.

- Run the **Update-Database –Verbose** command in Package Manager Console.

Data Motion / Custom SQL

So far we have looked at migration operations that don't change or move any data, now let's look at something that needs to move some data around. There is no native support for data motion yet, but we can run some arbitrary SQL commands at any point in our script.

- Let's add a **Post.Abstract** property to our model. Later, we're going to pre-populate the **Abstract** for existing posts using some text from the start of the **Content** column.

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

We'll use the **Add-Migration** command to let Code First Migrations scaffold its best guess at the migration for us.

- Run the **Add-Migration AddPostAbstract** command in Package Manager Console.
- The generated migration takes care of the schema changes but we also want to pre-populate the **Abstract** column using the first 100 characters of content for each post. We can do this by dropping down to SQL and running an **UPDATE** statement after the column is added. (Adding in line 12 in the code below)

```
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");
        }
    }
}
```

Our edited migration is looking good, so let's use **Update-Database** to bring the database up-to-date. We'll specify the **–Verbose** flag so that we can see the SQL being run against the database.

- Run the **Update-Database –Verbose** command in Package Manager Console.

Migrate to a Specific Version (Including Downgrade)

So far we have always upgraded to the latest migration, but there may be times when you want upgrade/downgrade to a specific migration.

Let's say we want to migrate our database to the state it was in after running our **AddBlogUrl** migration. We can use the **–TargetMigration** switch to downgrade to this migration.

- Run the **Update-Database –TargetMigration: AddBlogUrl** command in Package Manager Console.

This command will run the Down script for our **AddBlogAbstract** and **AddPostClass** migrations.

If you want to roll all the way back to an empty database then you can use the **Update-Database –TargetMigration: $InitialDatabase** command.

Getting a SQL Script

If another developer wants these changes on their machine they can just sync once we check our changes into source control. Once they have our new migrations they can just run the Update-Database command to have the changes applied locally. However if we want to push these changes out to a test server, and eventually production, we probably want a SQL script we can hand off to our DBA.

- Run the **Update-Database** command but this time specify the **–Script** flag so that changes are written to a script rather than applied. We'll also specify a source and target migration to generate the script for. We want a script to go from an empty database (**$InitialDatabase**) to the latest version (migration **AddPostAbstract**). *If you don't specify a target migration, Migrations will use the latest migration as the target. If you don't specify a source migrations, Migrations will use the current state of the database.*
- Run the **Update-Database -Script -SourceMigration: $InitialDatabase -TargetMigration: AddPostAbstract** command in Package Manager Console

Code First Migrations will run the migration pipeline but instead of actually applying the changes it will write them out to a .sql file for you. Once the script is generated, it is opened for you in Visual Studio, ready for you to view or save.

Generating Idempotent Scripts

Starting with EF6, if you specify **–SourceMigration $InitialDatabase** then the generated script will be 'idempotent'. Idempotent scripts can upgrade a database currently at any version to the latest version (or the specified version if you use **–TargetMigration**). The generated script includes logic to check the **__MigrationsHistory** table and only apply changes that haven't been previously applied.

Automatically Upgrading on Application Startup (MigrateDatabaseToLatestVersion Initializer)

If you are deploying your application you may want it to automatically upgrade the database (by applying any pending migrations) when the application launches. You can do this by registering the **MigrateDatabaseToLatestVersion** database initializer. A database initializer simply contains some logic that is used to make sure the database is setup correctly. This logic is run the first time the context is used within the application process (**AppDomain**).

We can update the **Program.cs** file, as shown below, to set the **MigrateDatabaseToLatestVersion** initializer for BlogContext before we use the context (Line 14). Note that you also need to add a using statement for the **System.Data.Entity** namespace (Line 5).

*When we create an instance of this initializer we need to specify the context type (**BlogContext**) and the migrations configuration (**Configuration**) - the migrations configuration is the class that got added to our **Migrations** folder when we enabled 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();
        }
    }
}
```

Now whenever our application runs it will first check if the database it is targeting is up-to-date, and apply any pending migrations if it is not.

Automatic Code First Migrations

9/18/2018 • 6 minutes to read • Edit Online

Automatic Migrations allows you to use Code First Migrations without having a code file in your project for each change you make. Not all changes can be applied automatically - for example column renames require the use of a code-based migration.

NOTE

This article assumes you know how to use Code First Migrations in basic scenarios. If you don't, then you'll need to read Code First Migrations before continuing.

Recommendation for Team Environments

You can intersperse automatic and code-based migrations but this is not recommended in team development scenarios. If you are part of a team of developers that use source control you should either use purely automatic migrations or purely code-based migrations. Given the limitations of automatic migrations we recommend using code-based migrations in team environments.

Building an Initial Model & Database

Before we start using migrations we need a project and a Code First model to work with. For this walkthrough we are going to use the canonical **Blog** and **Post** model.

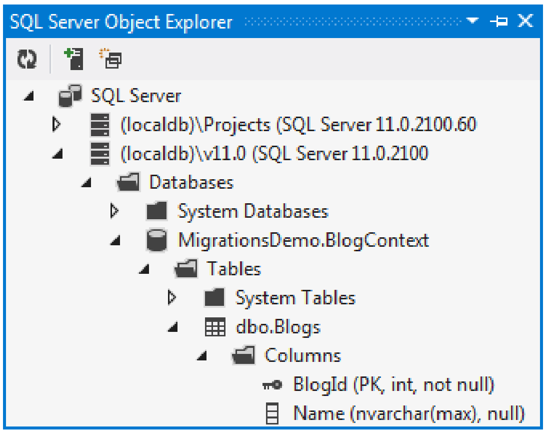
- Create a new **MigrationsAutomaticDemo** Console application
- Add the latest version of the **EntityFramework** NuGet package to the project
  - **Tools –> Library Package Manager –> Package Manager Console**
  - Run the **Install-Package EntityFramework** command
- Add a **Model.cs** file with the code shown below. This code defines a single **Blog** class that makes up our domain model and a **BlogContext** class that is our EF Code First context

```
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; }
    }
}
```

- Now that we have a model it's time to use it to perform data access. Update the **Program.cs** file with the code shown below.

```
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();
        }
    }
}
```

- Run your application and you will see that a **MigrationsAutomaticCodeDemo.BlogContext** database is created for you.



Enabling Migrations

It's time to make some more changes to our model.

- Let's introduce a Url property to the Blog class.

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

If you were to run the application again you would get an InvalidOperationException stating *The model backing the 'BlogContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).*

As the exception suggests, it's time to start using Code First Migrations. Because we want to use automatic migrations we're going to specify the **–EnableAutomaticMigrations** switch.

- Run the **Enable-Migrations –EnableAutomaticMigrations** command in Package Manager Console This command has added a **Migrations** folder to our project. This new folder contains one file:

- **The Configuration class.** This class allows you to configure how Migrations behaves for your context. For this walkthrough we will just use the default configuration. *Because there is just a single Code First context in your project, Enable-Migrations has automatically filled in the context type this configuration applies to.*

Your First Automatic Migration

Code First Migrations has two primary commands that you are going to become familiar with.

- **Add-Migration** will scaffold the next migration based on changes you have made to your model since the last migration was created
- **Update-Database** will apply any pending migrations to the database

We are going to avoid using Add-Migration (unless we really need to) and focus on letting Code First Migrations automatically calculate and apply the changes. Let's use **Update-Database** to get Code First Migrations to push the changes to our model (the new **Blog.Ur**l property) to the database.

- Run the **Update-Database** command in Package Manager Console.

The **MigrationsAutomaticDemo.BlogContext** database is now updated to include the **Url** column in the **Blogs** table.

Your Second Automatic Migration

Let's make another change and let Code First Migrations automatically push the changes to the database for us.

- Let's also add a new **Post** class

```
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; }
}
```

- We'll also add a **Posts** collection to the **Blog** class to form the other end of the relationship between **Blog** and **Post**

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

Now use **Update-Database** to bring the database up-to-date. This time let's specify the **–Verbose** flag so that you can see the SQL that Code First Migrations is running.

- Run the **Update-Database –Verbose** command in Package Manager Console.

Adding a Code Based Migration

Now let's look at something we might want to use a code-based migration for.

- Let’s add a **Rating** property to the **Blog** class

```
public int Rating { get; set; }
```

We could just run **Update-Database** to push these changes to the database. However, we're adding a non-nullable **Blogs.Rating** column, if there is any existing data in the table it will get assigned the CLR default of the data type for new column (Rating is integer, so that would be **0**). But we want to specify a default value of **3** so that existing rows in the **Blogs** table will start with a decent rating. Let’s use the Add-Migration command to write this change out to a code-based migration so that we can edit it. The **Add-Migration** command allows us to give these migrations a name, let’s just call ours **AddBlogRating**.

- Run the **Add-Migration AddBlogRating** command in Package Manager Console.
- In the **Migrations** folder we now have a new **AddBlogRating** migration. The migration filename is pre-fixed with a timestamp to help with ordering. Let’s edit the generated code to specify a default value of 3 for Blog.Rating (Line 10 in the code below)

The migration also has a code-behind file that captures some metadata. This metadata will allow Code First Migrations to replicate the automatic migrations we performed before this code-based migration. This is important if another developer wants to run our migrations or when it’s time to deploy our application.

```
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");
        }
    }
}
```

Our edited migration is looking good, so let’s use **Update-Database** to bring the database up-to-date.

- Run the **Update-Database** command in Package Manager Console.

Back to Automatic Migrations

We are now free to switch back to automatic migrations for our simpler changes. Code First Migrations will take care of performing the automatic and code-based migrations in the correct order based on the metadata it is storing in the code-behind file for each code-based migration.

- Let’s add a Post.Abstract property to our model

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

Now we can use **Update-Database** to get Code First Migrations to push this change to the database using an automatic migration.

- Run the **Update-Database** command in Package Manager Console.

Summary

In this walkthrough you saw how to use automatic migrations to push model changes to the database. You also saw how to scaffold and run code-based migrations in between automatic migrations when you need more control.

Code First Migrations with an existing database

9/13/2018 • 7 minutes to read • Edit Online

> **NOTE**
> **EF4.3 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 4.1. If you are using an earlier version, some or all of the information does not apply.

This article covers using Code First Migrations with an existing database, one that wasn't created by Entity Framework.

> **NOTE**
> This article assumes you know how to use Code First Migrations in basic scenarios. If you don't, then you'll need to read Code First Migrations before continuing.

Screencasts

If you'd rather watch a screencast than read this article, the following two videos cover the same content as this article.

Video One: "Migrations - Under the Hood"

This screencast covers how migrations tracks and uses information about the model to detect model changes.

Video Two: "Migrations - Existing Databases"

Building on the concepts from the previous video, this screencast covers how to enable and use migrations with an existing database.

Step 1: Create a model

Your first step will be to create a Code First model that targets your existing database. The Code First to an Existing Database topic provides detailed guidance on how to do this.

> **NOTE**
> It is important to follow the rest of the steps in this topic before making any changes to your model that would require changes to the database schema. The following steps require the model to be in-sync with the database schema.

Step 2: Enable Migrations

The next step is to enable migrations. You can do this by running the **Enable-Migrations** command in Package Manager Console.

This command will create a folder in your solution called Migrations, and put a single class inside it called Configuration. The Configuration class is where you configure migrations for your application, you can find out more about it in the Code First Migrations topic.

Step 3: Add an initial migration

Once migrations have been created and applied to the local database you may also want to apply these changes to other databases. For example, your local database may be a test database and you may ultimately want to also apply the changes to a production database and/or other developers test databases. There are two options for this step and the one you should pick depends whether or not the schema of any other databases is empty or currently matches the schema of the local database.

- **Option One: Use existing schema as starting point.** You should use this approach when other databases that migrations will be applied to in the future will have the same schema as your local database currently has. For example, you might use this if your local test database currently matches v1 of your production database and you will later apply these migrations to update your production database to v2.
- **Option Two: Use empty database as starting point.** You should use this approach when other databases that migrations will be applied to in the future are empty (or do not exist yet). For example, you might use this if you started developing your application using a test database but without using migrations and you will later want to create a production database from scratch.

Option One: Use existing schema as a starting point

Code First Migrations uses a snapshot of the model stored in the most recent migration to detect changes to the model (you can find detailed information about this in Code First Migrations in Team Environments). Since we are going to assume that databases already have the schema of the current model, we will generate an empty (no-op) migration that has the current model as a snapshot.

1. Run the **Add-Migration InitialCreate –IgnoreChanges** command in Package Manager Console. This creates an empty migration with the current model as a snapshot.
2. Run the **Update-Database** command in Package Manager Console. This will apply the InitialCreate migration to the database. Since the actual migration doesn't contain any changes, it will simply add a row to the __MigrationsHistory table indicating that this migration has already been applied.

Option Two: Use empty database as a starting point

In this scenario we need Migrations to be able to create the entire database from scratch – including the tables that are already present in our local database. We're going to generate an InitialCreate migration that includes logic to create the existing schema. We'll then make our existing database look like this migration has already been applied.

1. Run the **Add-Migration InitialCreate** command in Package Manager Console. This creates a migration to create the existing schema.
2. Comment out all code in the Up method of the newly created migration. This will allow us to 'apply' the migration to the local database without trying to recreate all the tables etc. that already exist.
3. Run the **Update-Database** command in Package Manager Console. This will apply the InitialCreate migration to the database. Since the actual migration doesn't contain any changes (because we temporarily commented them out), it will simply add a row to the __MigrationsHistory table indicating that this migration has already been applied.
4. Un-comment the code in the Up method. This means that when this migration is applied to future databases, the schema that already existed in the local database will be created by migrations.

Things to be aware of

There are a few things you need to be aware of when using Migrations against an existing database.

Default/calculated names may not match existing schema

Migrations explicitly specifies names for columns and tables when it scaffolds a migrations. However, there are other database objects that Migrations calculates a default name for when applying the migrations. This includes indexes and foreign key constraints. When targeting an existing schema, these calculated names may not match what actually exists in your database.

Here are some examples of when you need to be aware of this:

If you used 'Option One: Use existing schema as a starting point' from Step 3:

- If future changes in your model require changing or dropping one of the database objects that is named differently, you will need to modify the scaffolded migration to specify the correct name. The Migrations APIs have an optional Name parameter that allows you to do this. For example, your existing schema may have a Post table with a BlogId foreign key column that has an index named IndexFk_BlogId. However, by default Migrations would expect this index to be named IX_BlogId. If you make a change to your model that results in dropping this index, you will need to modify the scaffolded DropIndex call to specify the IndexFk_BlogId name.

If you used 'Option Two: Use empty database as a starting point' from Step 3:

- Trying to run the Down method of the initial migration (that is, reverting to an empty database) against your local database may fail because Migrations will try to drop indexes and foreign key constraints using the incorrect names. This will only affect your local database since other databases will be created from scratch using the Up method of the initial migration. If you want to downgrade your existing local database to an empty state it is easiest to do this manually, either by dropping the database or dropping all the tables. After this initial downgrade all database objects will be recreated with the default names, so this issue will not present itself again.
- If future changes in your model require changing or dropping one of the database objects that is named differently, this will not work against your existing local database – since the names won't match the defaults. However, it will work against databases that were created 'from scratch' since they will have used the default names chosen by Migrations. You could either make these changes manually on your local existing database, or consider having Migrations recreate your database from scratch – as it will on other machines.
- Databases created using the Up method of your initial migration may differ slightly from the local database since the calculated default names for indexes and foreign key constraints will be used. You may also end up with extra indexes as Migrations will create indexes on foreign key columns by default – this may not have been the case in your original local database.

Not all database objects are represented in the model

Database objects that are not part of your model will not be handled by Migrations. This can include views, stored procedures, permissions, tables that are not part of your model, additional indexes, etc.

Here are some examples of when you need to be aware of this:

- Regardless of the option you chose in 'Step 3', if future changes in your model require changing or dropping these additional objects Migrations will not know to make these changes. For example, if you drop a column that has an additional index on it, Migrations will not know to drop the index. You will need to manually add this to the scaffolded Migration.
- If you used 'Option Two: Use empty database as a starting point', these additional objects will not be created by the Up method of your initial migration. You can modify the Up and Down methods to take care of these additional objects if you wish. For objects that are not natively supported in the Migrations API – such as views – you can use the Sql method to run raw SQL to create/drop them.

Customizing the migrations history table

1/6/2019 • 3 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

NOTE

This article assumes you know how to use Code First Migrations in basic scenarios. If you don't, then you'll need to read Code First Migrations before continuing.

What is Migrations History Table?

Migrations history table is a table used by Code First Migrations to store details about migrations applied to the database. By default the name of the table in the database is __MigrationHistory and it is created when applying the first migration to the database. In Entity Framework 5 this table was a system table if the application used Microsoft Sql Server database. This has changed in Entity Framework 6 however and the migrations history table is no longer marked a system table.

Why customize Migrations History Table?

Migrations history table is supposed to be used solely by Code First Migrations and changing it manually can break migrations. However sometimes the default configuration is not suitable and the table needs to be customized, for instance:

- You need to change names and/or facets of the columns to enable a 3rd party Migrations provider
- You want to change the name of the table
- You need to use a non-default schema for the __MigrationHistory table
- You need to store additional data for a given version of the context and therefore you need to add an additional column to the table

Words of precaution

Changing the migration history table is powerful but you need to be careful to not overdo it. EF runtime currently does not check whether the customized migrations history table is compatible with the runtime. If it is not your application may break at runtime or behave in unpredictable ways. This is even more important if you use multiple contexts per database in which case multiple contexts can use the same migration history table to store information about migrations.

How to customize Migrations History Table?

Before you start you need to know that you can customize the migrations history table only before you apply the first migration. Now, to the code.

First, you will need to create a class derived from System.Data.Entity.Migrations.History.HistoryContext class. The HistoryContext class is derived from the DbContext class so configuring the migrations history table is very similar to configuring EF models with fluent API. You just need to override the OnModelCreating method and use fluent API to configure the table.

> **NOTE**
>
> Typically when you configure EF models you don't need to call base.OnModelCreating() from the overridden OnModelCreating method since the DbContext.OnModelCreating() has empty body. This is not the case when configuring the migrations history table. In this case the first thing to do in your OnModelCreating() override is to actually call base.OnModelCreating(). This will configure the migrations history table in the default way which you then tweak in the overriding method.

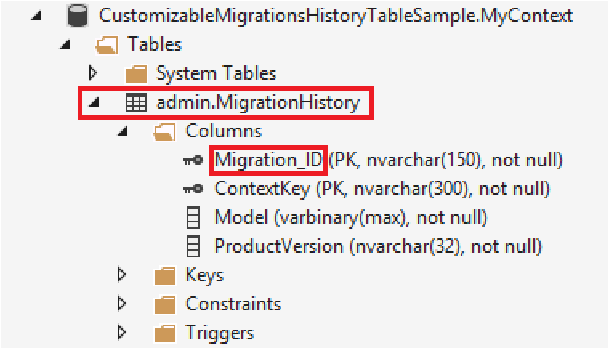
Let's say you want to rename the migrations history table and put it to a custom schema called "admin". In addition your DBA would like you to rename the MigrationId column to Migration_ID. You could achieve this by creating the following class derived from HistoryContext:

```
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");
        }
    }
}
```

Once your custom HistoryContext is ready you need to make EF aware of it by registering it via code-based configuration:

```
using System.Data.Entity;

namespace CustomizableMigrationsHistoryTableSample
{
    public class ModelConfiguration : DbConfiguration
    {
        public ModelConfiguration()
        {
            this.SetHistoryContext("System.Data.SqlClient",
                (connection, defaultSchema) => new MyHistoryContext(connection, defaultSchema));
        }
    }
}
```

That's pretty much it. Now you can go to the Package Manager Console, Enable-Migrations, Add-Migration and finally Update-Database. This should result in adding to the database a migrations history table configured according to the details you specified in your HistoryContext derived class.


CustomizableMigrationsHistoryTableSample.MyContext
Tables
System Tables
admin.MigrationHistory
Columns
Migration_ID (PK, nvarchar(150), not null)
ContextKey (PK, nvarchar(300), not null)
Model (varbinary(max), not null)
ProductVersion (nvarchar(32), not null)
Keys
Constraints
Triggers

Using migrate.exe

9/30/2018 • 4 minutes to read • Edit Online

Code First Migrations can be used to update a database from inside visual studio, but can also be executed via the command line tool migrate.exe. This page will give a quick overview on how to use migrate.exe to execute migrations against a database.

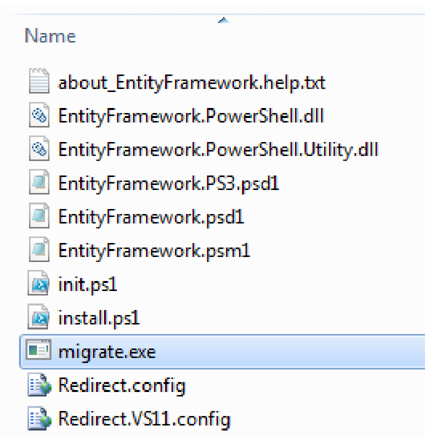
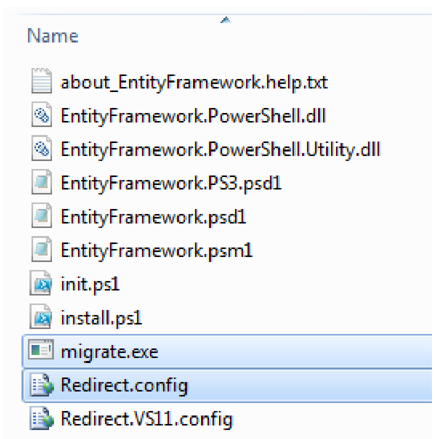
> **NOTE**
>
> This article assumes you know how to use Code First Migrations in basic scenarios. If you don't, then you'll need to read Code First Migrations before continuing.

Copy migrate.exe

When you install Entity Framework using NuGet migrate.exe will be inside the tools folder of the downloaded package. In <project folder>\packages\EntityFramework.<version>\tools

Once you have migrate.exe then you need to copy it to the location of the assembly that contains your migrations.

If your application targets .NET 4, and not 4.5, then you will need to copy the **Redirect.config** into the location as well and rename it **migrate.exe.config**. This is so that migrate.exe gets the correct binding redirects to be able to locate the Entity Framework assembly.

.NET 4.5	.NET 4.0
	

> **NOTE**
>
> migrate.exe doesn't support x64 assemblies.

Once you have moved migrate.exe to the correct folder then you should be able to use it to execute migrations against the database. All the utility is designed to do is execute migrations. It cannot generate migrations or create a SQL script.

See options

```
Migrate.exe /?
```

The above will display the help page associated with this utility, note that you will need to have the EntityFramework.dll in the same location that you are running migrate.exe in order for this to work.

Migrate to the latest migration

```
Migrate.exe MyMvcApplication.dll /startupConfigurationFile="..\\web.config"
```

When running migrate.exe the only mandatory parameter is the assembly, which is the assembly that contains the migrations that you are trying to run, but it will use all convention based settings if you do not specify the configuration file.

Migrate to a specific migration

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /targetMigration="AddTitle"
```

If you want to run migrations up to a specific migration, then you can specify the name of the migration. This will run all previous migrations as required until getting to the migration specified.

Specify working directory

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /startupDirectory="c:\\MyApp"
```

If you assembly has dependencies or reads files relative to the working directory then you will need to set startupDirectory.

Specify migration configuration to use

```
Migrate.exe MyAssembly CustomConfig /startupConfigurationFile="..\\web.config"
```

If you have multiple migration configuration classes, classes inheriting from DbMigrationConfiguration, then you need to specify which is to be used for this execution. This is specified by providing the optional second parameter without a switch as above.

Provide connection string

```
Migrate.exe BlogDemo.dll /connectionString="Data Source=localhost;Initial Catalog=BlogDemo;Integrated
Security=SSPI" /connectionProviderName="System.Data.SqlClient"
```

If you wish to specify a connection string at the command line then you must also provide the provider name. Not specifying the provider name will cause an exception.

Common Problems

ERROR MESSAGE	SOLUTION
Unhandled Exception: System.IO.FileLoadException: Could not load file or assembly 'EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' or one of its dependencies. The located assembly's manifest definition does not match the assembly reference. (Exception from HRESULT: 0x80131040)	This typically means that you are running a .NET 4 application without the Redirect.config file. You need to copy the Redirect.config to the same location as migrate.exe and rename it to migrate.exe.config.
Unhandled Exception: System.IO.FileLoadException: Could not load file or assembly 'EntityFramework, Version=4.4.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089' or one of its dependencies. The located assembly's manifest definition does not match the assembly reference. (Exception from HRESULT: 0x80131040)	This exception means that you are running a .NET 4.5 application with the Redirect.config copied to the migrate.exe location. If your app is .NET 4.5 then you do not need to have the config file with the redirects inside. Delete the migrate.exe.config file.
ERROR: Unable to update database to match the current model because there are pending changes and automatic migration is disabled. Either write the pending model changes to a code-based migration or enable automatic migration. Set DbMigrationsConfiguration.AutomaticMigrationsEnabled to true to enable automatic migration.	This error occurs if running migrate when you haven't created a migration to cope with changes made to the model, and the database does not match the model. Adding a property to a model class then running migrate.exe without creating a migration to upgrade the database is an example of this.
ERROR: Type is not resolved for member 'System.Data.Entity.Migrations.Design.ToolingFacade+UpdateRunner,EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.	This error can be caused by specifying an incorrect startup directory. This must be the location of migrate.exe
Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an object. at System.Data.Entity.Migrations.Console.Program.Main(String[] args)	This can be caused by not specifying a required parameter for a scenario that you are using. For example specifying a connection string without specifying the provider name.
ERROR: More than one migrations configuration type was found in the assembly 'ClassLibrary1'. Specify the name of the one to use.	As the error states, there is more than one configuration class in the given assembly. You must use the /configurationType switch to specify which to use.
ERROR: Could not load file or assembly '<assemblyName>' or one of its dependencies. The given assembly name or codebase was invalid. (Exception from HRESULT: 0x80131047)	This can be caused by specifying an assembly name incorrectly or not having
ERROR: Could not load file or assembly '<assemblyName>' or one of its dependencies. An attempt was made to load a program with an incorrect format.	This happens if you are trying to run migrate.exe against an x64 application. EF 5.0 and below will only work on x86.

Code First Migrations in Team Environments

12/4/2018 • 14 minutes to read • Edit Online

NOTE

This article assumes you know how to use Code First Migrations in basic scenarios. If you don't, then you'll need to read Code First Migrations before continuing.

Grab a coffee, you need to read this whole article

The issues in team environments are mostly around merging migrations when two developers have generated migrations in their local code base. While the steps to solve these are pretty simple, they require you to have a solid understanding of how migrations works. Please don't just skip ahead to the end – take the time to read the whole article to ensure you are successful.

Some general guidelines

Before we dig into how to manage merging migrations generated by multiple developers, here are some general guidelines to set you up for success.

Each team member should have a local development database

Migrations uses the **__MigrationsHistory** table to store what migrations have been applied to the database. If you have multiple developers generating different migrations while trying to target the same database (and thus share a **__MigrationsHistory** table) migrations is going to get very confused.

Of course, if you have team members that aren't generating migrations, there is no problem having them share a central development database.

Avoid automatic migrations

The bottom line is that automatic migrations initially look good in team environments, but in reality they just don't work. If you want to know why, keep reading – if not, then you can skip to the next section.

Automatic migrations allows you to have your database schema updated to match the current model without the need to generate code files (code-based migrations). Automatic migrations would work very well in a team environment if you only ever used them and never generated any code-based migrations. The problem is that automatic migrations are limited and don't handle a number of operations – property/column renames, moving data to another table, etc. To handle these scenarios you end up generating code-based migrations (and editing the scaffolded code) that are mixed in between changes that are handled by automatic migrations. This makes it near on impossible to merge changes when two developers check in migrations.

Screencasts

If you'd rather watch a screencast than read this article, the following two videos cover the same content as this article.

Video One: "Migrations - Under the Hood"

This screencast covers how migrations tracks and uses information about the model to detect model changes.

Video Two: "Migrations - Team Environments"

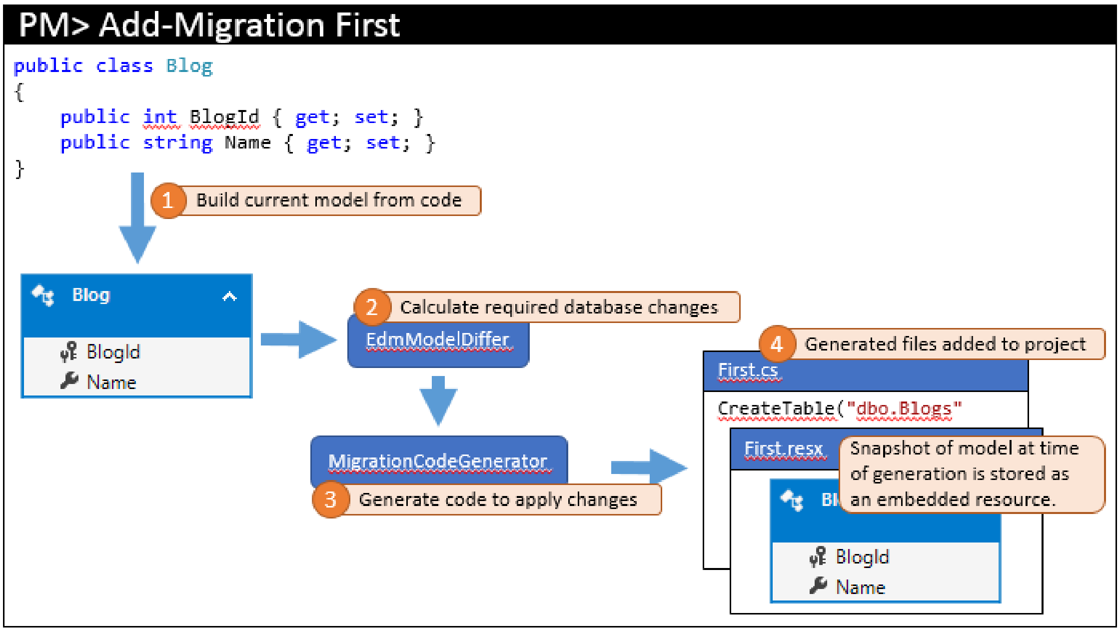
Building on the concepts from the previous video, this screencast covers the issues that arise in a team environment and how to solve them.

Understanding how migrations works

The key to successfully using migrations in a team environment is a basic understanding how migrations tracks and uses information about the model to detect model changes.

The first migration

When you add the first migration to your project, you run something like **Add-Migration First** in Package Manager Console. The high level steps that this command performs are pictured below.



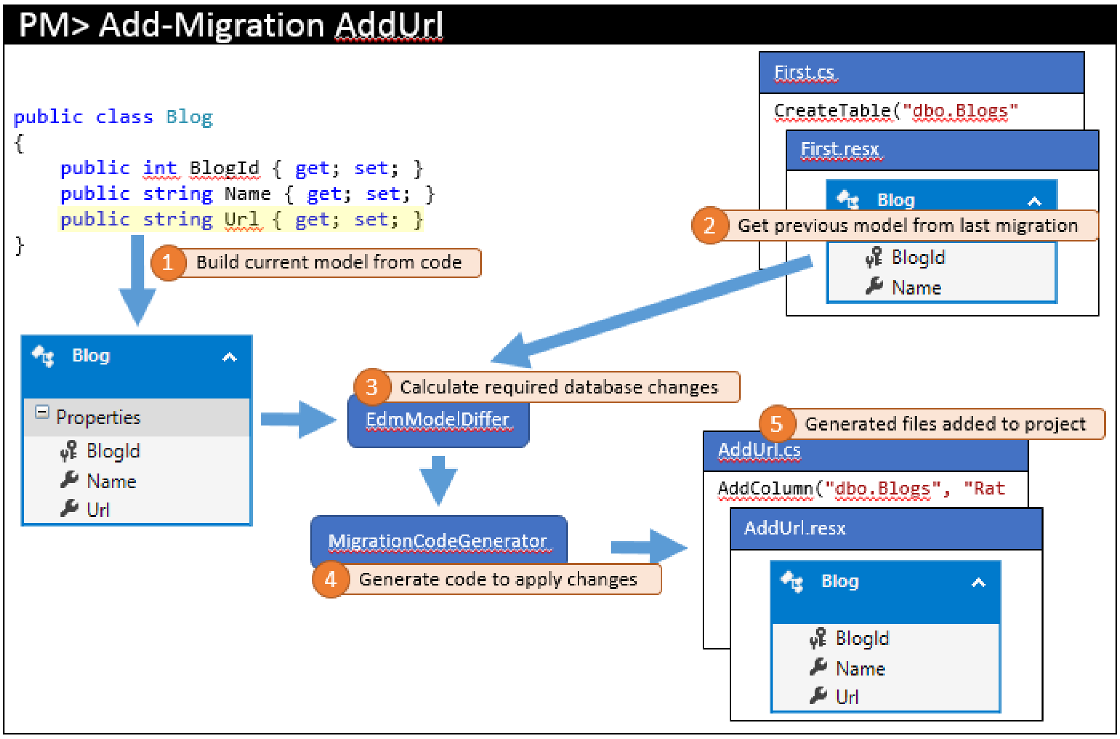
The current model is calculated from your code (1). The required database objects are then calculated by the model differ (2) – since this is the first migration the model differ just uses an empty model for the comparison. The required changes are passed to the code generator to build the required migration code (3) which is then added to your Visual Studio solution (4).

In addition to the actual migration code that is stored in the main code file, migrations also generates some additional code-behind files. These files are metadata that is used by migrations and are not something you should edit. One of these files is a resource file (.resx) that contains a snapshot of the model at the time the migration was generated. You'll see how this is used in the next step.

At this point you would probably run **Update-Database** to apply your changes to the database, and then go about implementing other areas of your application.

Subsequent migrations

Later you come back and make some changes to your model – in our example we'll add a **Url** property to **Blog**. You would then issue a command such as **Add-Migration AddUrl** to scaffold a migration to apply the corresponding database changes. The high level steps that this command performs are pictured below.



Just like last time, the current model is calculated from code (1). However, this time there are existing migrations so the previous model is retrieved from the latest migration (2). These two models are diffed to find the required database changes (3) and then the process completes as before.

This same process is used for any further migrations that you add to the project.

Why bother with the model snapshot?

You may be wondering why EF bothers with the model snapshot – why not just look at the database. If so, read on. If you're not interested then you can skip this section.

There are a number of reasons EF keeps the model snapshot around:

- It allows your database to drift from the EF model. These changes can be made directly in the database, or you can change the scaffolded code in your migrations to make the changes. Here are a couple of examples of this in practice:
  - You want to add an Inserted and Updated to column to one or more of your tables but you don't want to include these columns in the EF model. If migrations looked at the database it would continually try to drop these columns every time you scaffolded a migration. Using the model snapshot, EF will only ever detect legitimate changes to the model.
  - You want to change the body of a stored procedure used for updates to include some logging. If migrations looked at this stored procedure from the database it would continually try and reset it back to the definition that EF expects. By using the model snapshot, EF will only ever scaffold code to alter the stored procedure when you change the shape of the procedure in the EF model.
  - These same principles apply to adding extra indexes, including extra tables in your database, mapping EF to a database view that sits over a table, etc.
- The EF model contains more than just the shape of the database. Having the entire model allows migrations to look at information about the properties and classes in your model and how they map to the columns and tables. This information allows migrations to be more intelligent in the code that it scaffolds. For example, if you change the name of the column that a property maps to migrations can detect the rename by seeing that it's the same property – something that can't be done if you only have the database schema.

What causes issues in team environments

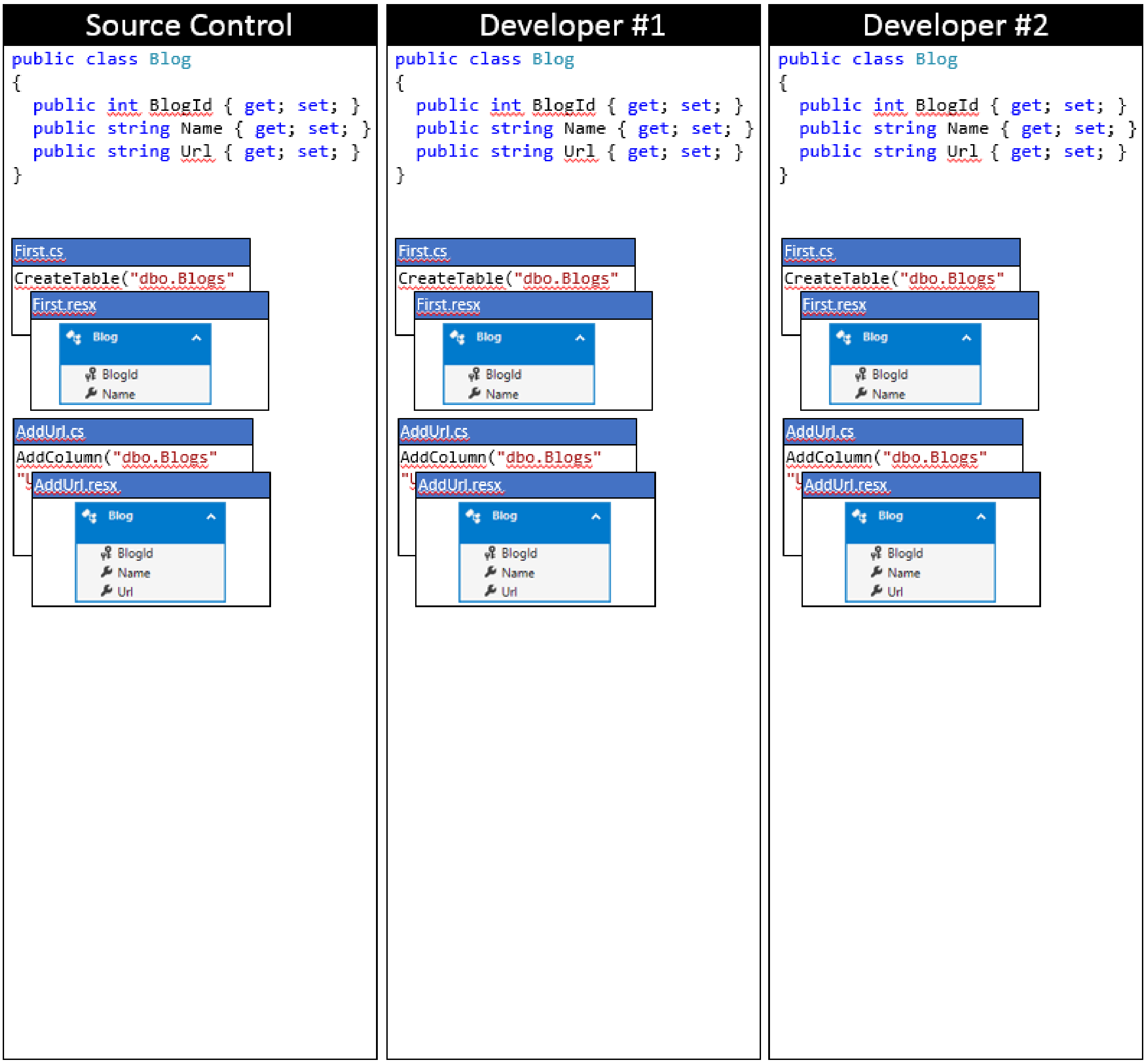
The workflow covered in the previous section works great when you are a single developer working on an application. It also works well in a team environment if you are the only person making changes to the model. In this scenario you can make model changes, generate migrations and submit them to your source control. Other developers can sync your changes and run **Update-Database** to have the schema changes applied.

Issues start to arise when you have multiple developers making changes to the EF model and submitting to source control at the same time. What EF lacks is a first class way to merge your local migrations with migrations that another developer has submitted to source control since you last synced.

An example of a merge conflict

First let's look at a concrete example of such a merge conflict. We'll continue on with the example we looked at earlier. As a starting point let's assume the changes from the previous section were checked in by the original developer. We'll track two developers as they make changes to code base.

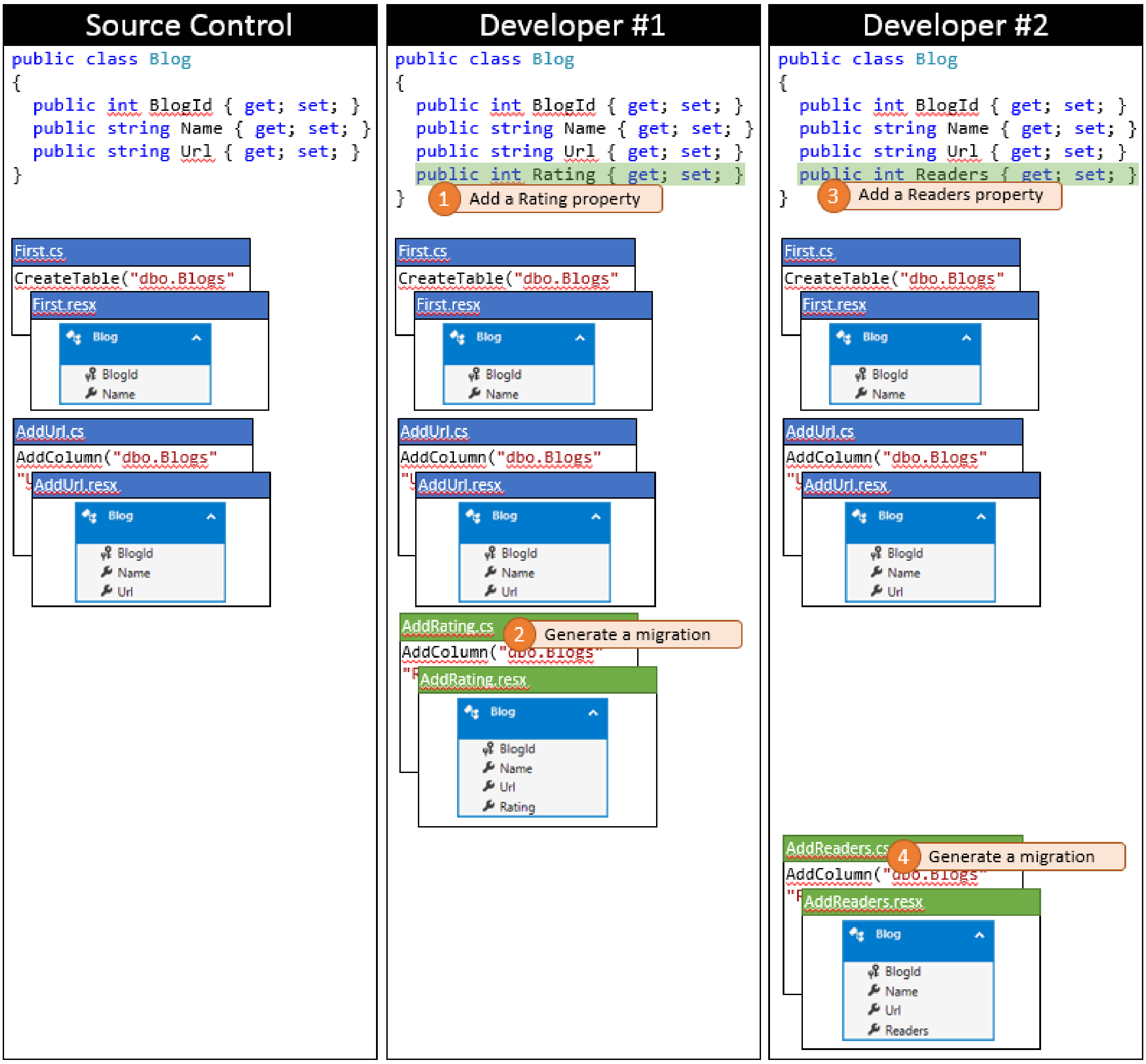
We'll track the EF model and the migrations thru a number of changes. For a starting point, both developers have synced to the source control repository, as depicted in the following graphic.



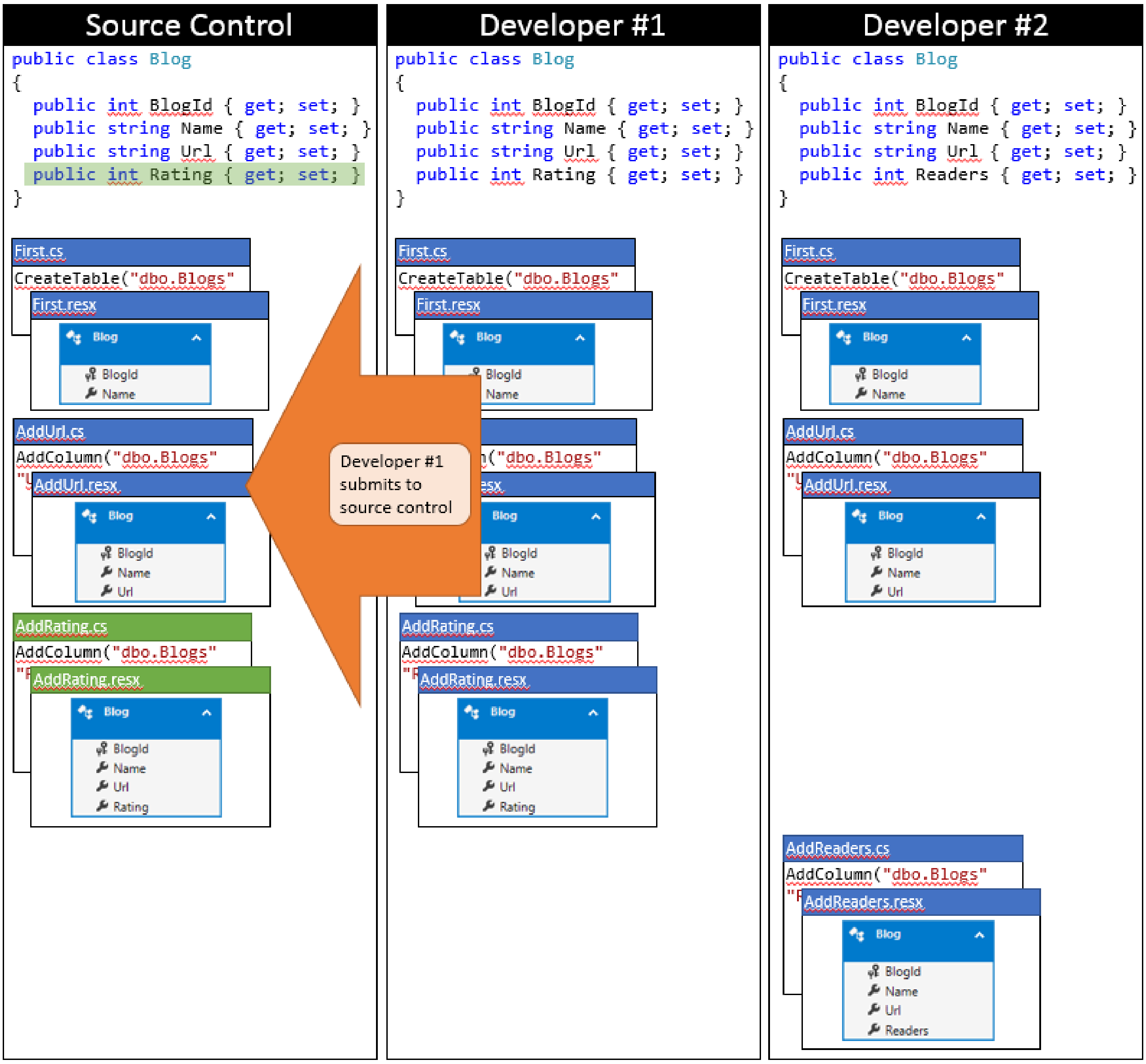
Developer #1 and developer #2 now makes some changes to the EF model in their local code base. Developer #1 adds a **Rating** property to **Blog** – and generates an **AddRating** migration to apply the changes to the database. Developer #2 adds a **Readers** property to **Blog** – and generates the corresponding **AddReaders** migration. Both developers run **Update-Database**, to apply the changes to their local databases, and then continue developing the application.

NOTE

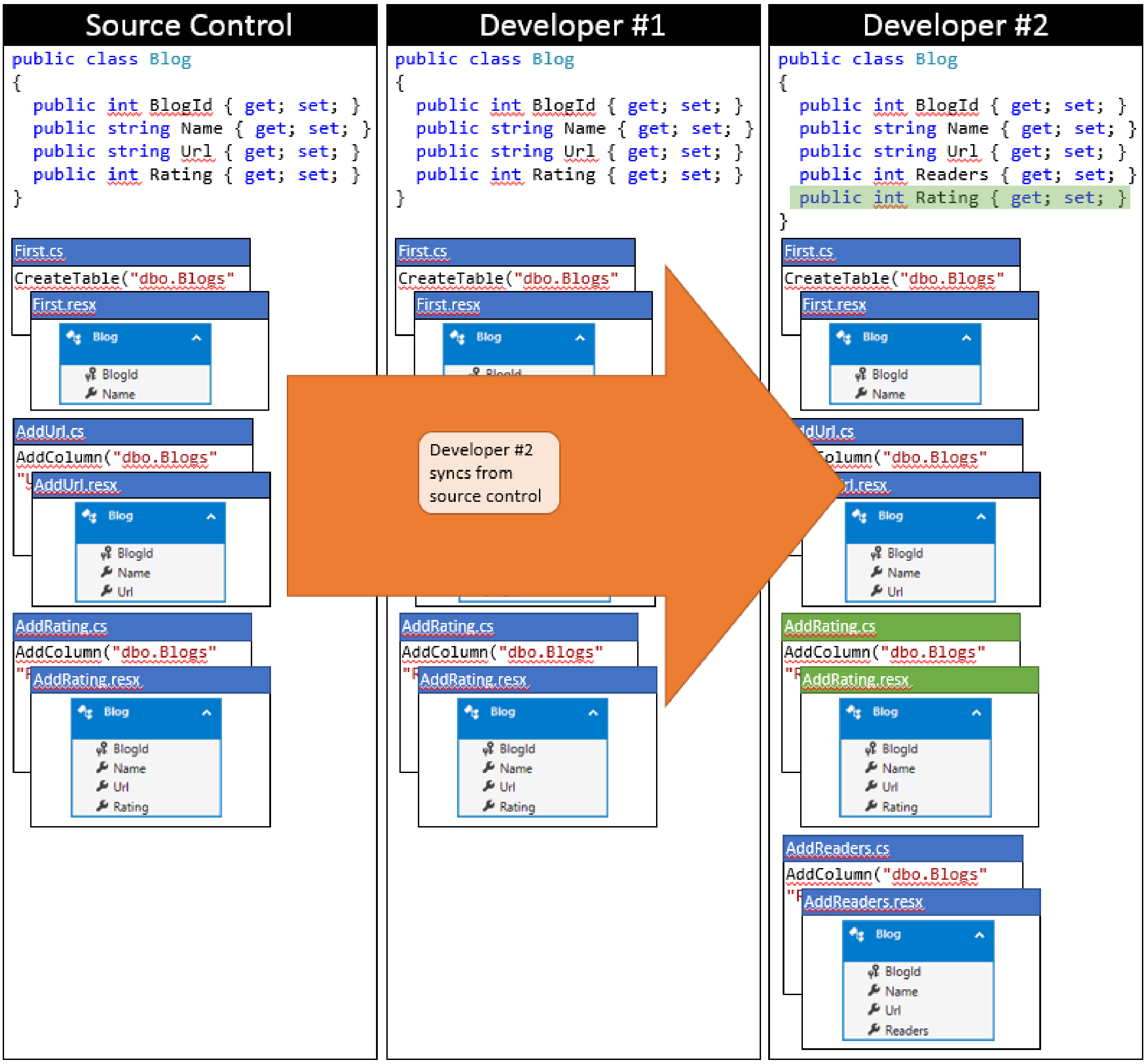
Migrations are prefixed with a timestamp, so our graphic represents that the AddReaders migration from Developer #2 comes after the AddRating migration from Developer #1. Whether developer #1 or #2 generated the migration first makes no difference to the issues of working in a team, or the process for merging them that we'll look at in the next section.



It's a lucky day for Developer #1 as they happen to submit their changes first. Because no one else has checked in since they synced their repository, they can just submit their changes without performing any merging.



Now it's time for Developer #2 to submit. They aren't so lucky. Because someone else has submitted changes since they synced, they will need to pull down the changes and merge. The source control system will likely be able to automatically merge the changes at the code level since they are very simple. The state of Developer #2's local repository after syncing is depicted in the following graphic.



At this stage Developer #2 can run **Update-Database** which will detect the new **AddRating** migration (which hasn't been applied to Developer #2's database) and apply it. Now the **Rating** column is added to the **Blogs** table and the database is in sync with the model.

There are a couple of problems though:

1. Although **Update-Database** will apply the **AddRating** migration it will also raise a warning: *Unable to update database to match the current model because there are pending changes and automatic migration is disabled…* The problem is that the model snapshot stored in the last migration (**AddReader**) is missing the **Rating** property on **Blog** (since it wasn't part of the model when the migration was generated). Code First detects that the model in the last migration doesn't match the current model and raises the warning.
2. Running the application would result in an InvalidOperationException stating that "*The model backing the 'BloggingContext' context has changed since the database was created. Consider using Code First Migrations to update the database…*" Again, the problem is the model snapshot stored in the last migration doesn't match the current model.
3. Finally, we would expect running **Add-Migration** now would generate an empty migration (since there are no changes to apply to the database). But because migrations compares the current model to the one from the last migration (which is missing the **Rating** property) it will actually scaffold another **AddColumn** call to add in the **Rating** column. Of course, this migration would fail during **Update-Database** because the **Rating** column already exists.

Resolving the merge conflict

The good news is that it's not too hard to deal with the merge manually – provided you have an understanding of how migrations works. So if you've skipped ahead to this section… sorry, you need to go back and read the rest of the article first!

There are two options, the easiest is to generate a blank migration that has the correct current model as a snapshot. The second option is to update the snapshot in the last migration to have the correct model snapshot. The second option is a little harder and can't be used in every scenario, but it's also cleaner because it doesn't involve adding an extra migration.

Option 1: Add a blank 'merge' migration

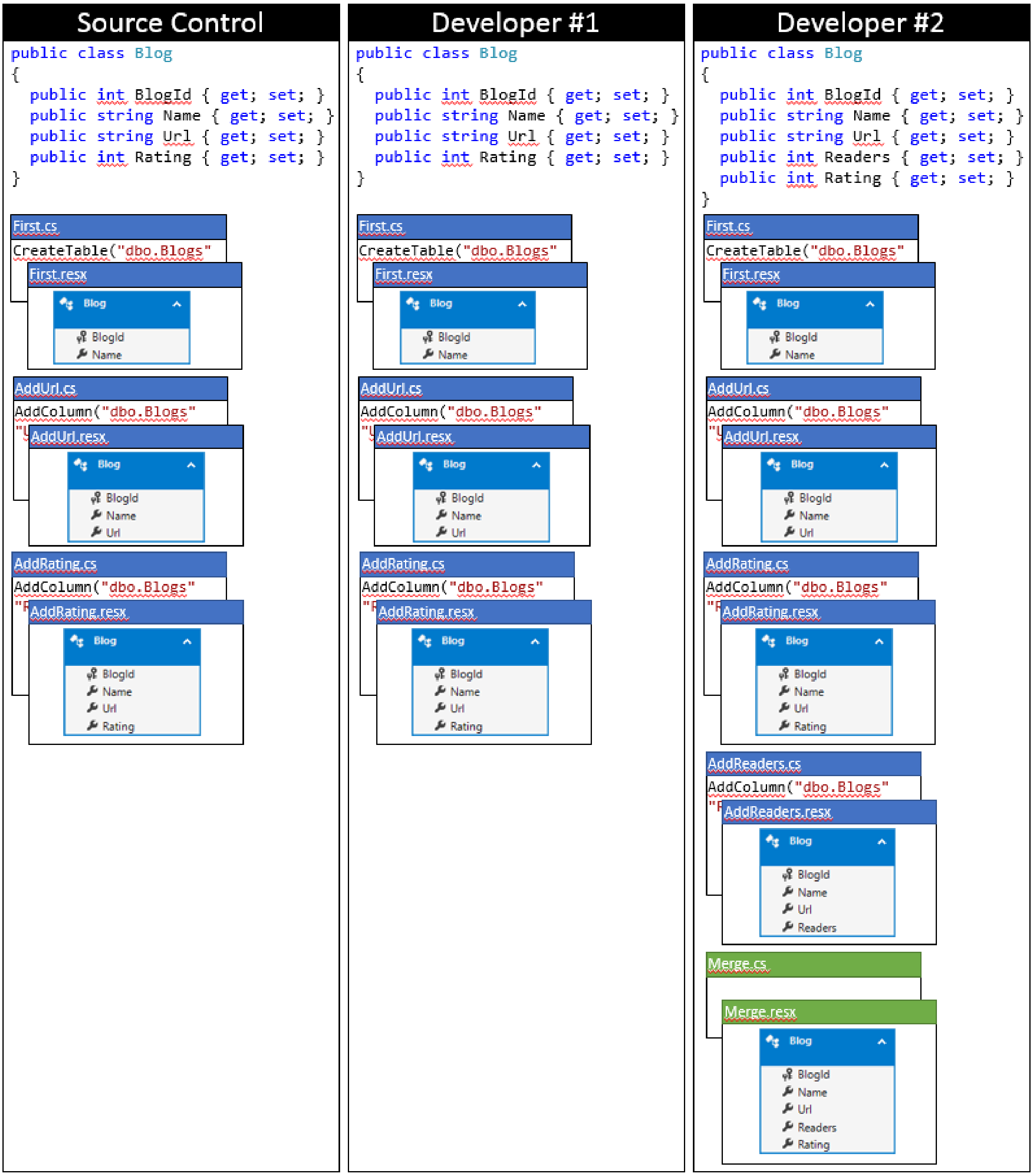
In this option we generate a blank migration solely for the purpose of making sure the latest migration has the correct model snapshot stored in it.

This option can be used regardless of who generated the last migration. In the example we've been following Developer #2 is taking care of the merge and they happened to generate the last migration. But these same steps can be used if Developer #1 generated the last migration. The steps also apply if there are multiple migrations involved – we've just been looking at two in order to keep it simple.

The following process can be used for this approach, starting from the time you realize you have changes that need to be synced from source control.

1. Ensure any pending model changes in your local code base have been written to a migration. This step ensures you don't miss any legitimate changes when it comes time to generate the blank migration.
2. Sync with source control.
3. Run **Update-Database** to apply any new migrations that other developers have checked in. ***Note:*** *if you don't get any warnings from the Update-Database command then there were no new migrations from other developers and there is no need to perform any further merging.*
4. Run **Add-Migration <pick_a_name> –IgnoreChanges** (for example, **Add-Migration Merge –IgnoreChanges**). This generates a migration with all the metadata (including a snapshot of the current model) but will ignore any changes it detects when comparing the current model to the snapshot in the last migrations (meaning you get a blank **Up** and **Down** method).
5. Continue developing, or submit to source control (after running your unit tests of course).

Here is the state of Developer #2's local code base after using this approach.



Option 2: Update the model snapshot in the last migration

This option is very similar to option 1 but removes the extra blank migration – because let's face it, who wants extra code files in their solution.

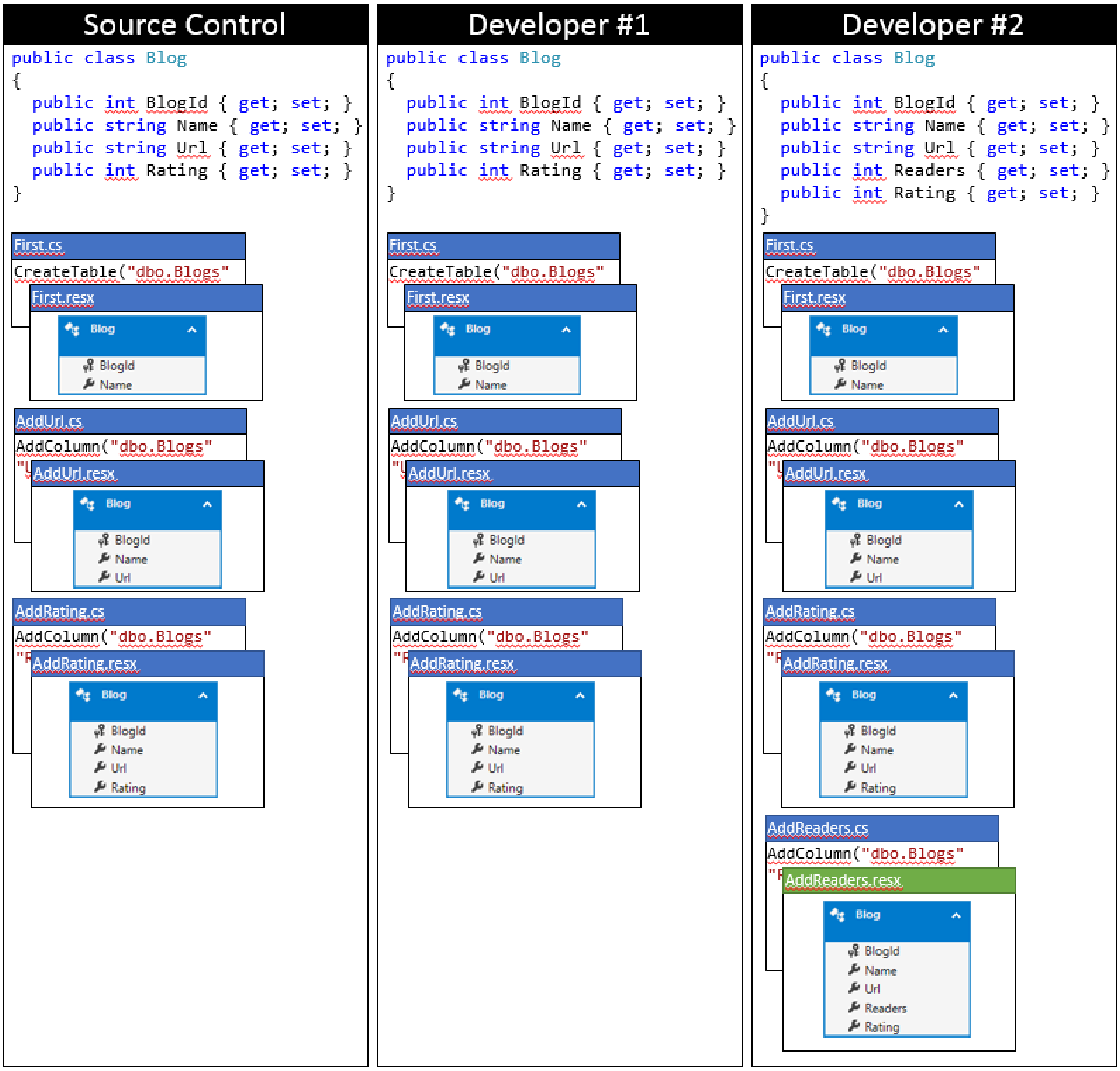
This approach is only feasible if the latest migration exists only in your local code base and has not yet been submitted to source control (for example, if the last migration was generated by the user doing the merge). Editing the metadata of migrations that other developers may have already applied to their development database – or even worse applied to a production database – can result in unexpected side effects. During the process we're going to roll back the last migration in our local database and re-apply it with updated metadata.

While the last migration needs to just be in the local code base there are no restrictions to the number or order of migrations that proceed it. There can be multiple migrations from multiple different developers and the same steps apply– we've just been looking at two in order to keep it simple.

The following process can be used for this approach, starting from the time you realize you have changes that need to be synced from source control.

1. Ensure any pending model changes in your local code base have been written to a migration. This step ensures you don’t miss any legitimate changes when it comes time to generate the blank migration.
2. Sync with the source control.
3. Run **Update-Database** to apply any new migrations that other developers have checked in. ***Note:*** *if you don’t get any warnings from the Update-Database command then there were no new migrations from other developers and there is no need to perform any further merging.*
4. Run **Update-Database –TargetMigration <second_last_migration>** (in the example we’ve been following this would be **Update-Database –TargetMigration AddRating**). This roles the database back to the state of the second last migration – effectively ‘un-applying’ the last migration from the database. ***Note:*** *This step is required to make it safe to edit the metadata of the migration since the metadata is also stored in the __MigrationsHistoryTable of the database. This is why you should only use this option if the last migration is only in your local code base. If other databases had the last migration applied you would also have to roll them back and re-apply the last migration to update the metadata.*
5. Run **Add-Migration <full_name_including_timestamp_of_last_migration**> (in the example we’ve been following this would be something like **Add-Migration 201311062215252_AddReaders**). ***Note:*** *You need to include the timestamp so that migrations knows you want to edit the existing migration rather than scaffolding a new one.* This will update the metadata for the last migration to match the current model. You’ll get the following warning when the command completes, but that’s exactly what you want. “*Only the Designer Code for migration '201311062215252_AddReaders' was re-scaffolded. To re-scaffold the entire migration, use the -Force parameter.*”
6. Run **Update-Database** to re-apply the latest migration with the updated metadata.
7. Continue developing, or submit to source control (after running your unit tests of course).

Here is the state of Developer #2’s local code base after using this approach.



Summary

There are some challenges when using Code First Migrations in a team environment. However, a basic understanding of how migrations works and some simple approaches for resolving merge conflicts make it easy to overcome these challenges.

The fundamental issue is incorrect metadata stored in the latest migration. This causes Code First to incorrectly detect that the current model and database schema don't match and to scaffold incorrect code in the next migration. This situation can be overcome by generating a blank migration with the correct model, or updating the metadata in the latest migration.

Model First

9/18/2018 • 7 minutes to read • Edit Online

This video and step-by-step walkthrough provide an introduction to Model First development using Entity Framework. Model First allows you to create a new model using the Entity Framework Designer and then generate a database schema from the model. The model is stored in an EDMX file (.edmx extension) and can be viewed and edited in the Entity Framework Designer. The classes that you interact with in your application are automatically generated from the EDMX file.

Watch the video

This video and step-by-step walkthrough provide an introduction to Model First development using Entity Framework. Model First allows you to create a new model using the Entity Framework Designer and then generate a database schema from the model. The model is stored in an EDMX file (.edmx extension) and can be viewed and edited in the Entity Framework Designer. The classes that you interact with in your application are automatically generated from the EDMX file.

Presented By: Rowan Miller

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have Visual Studio 2010 or Visual Studio 2012 installed to complete this walkthrough.

If you are using Visual Studio 2010, you will also need to have NuGet installed.

1. Create the Application

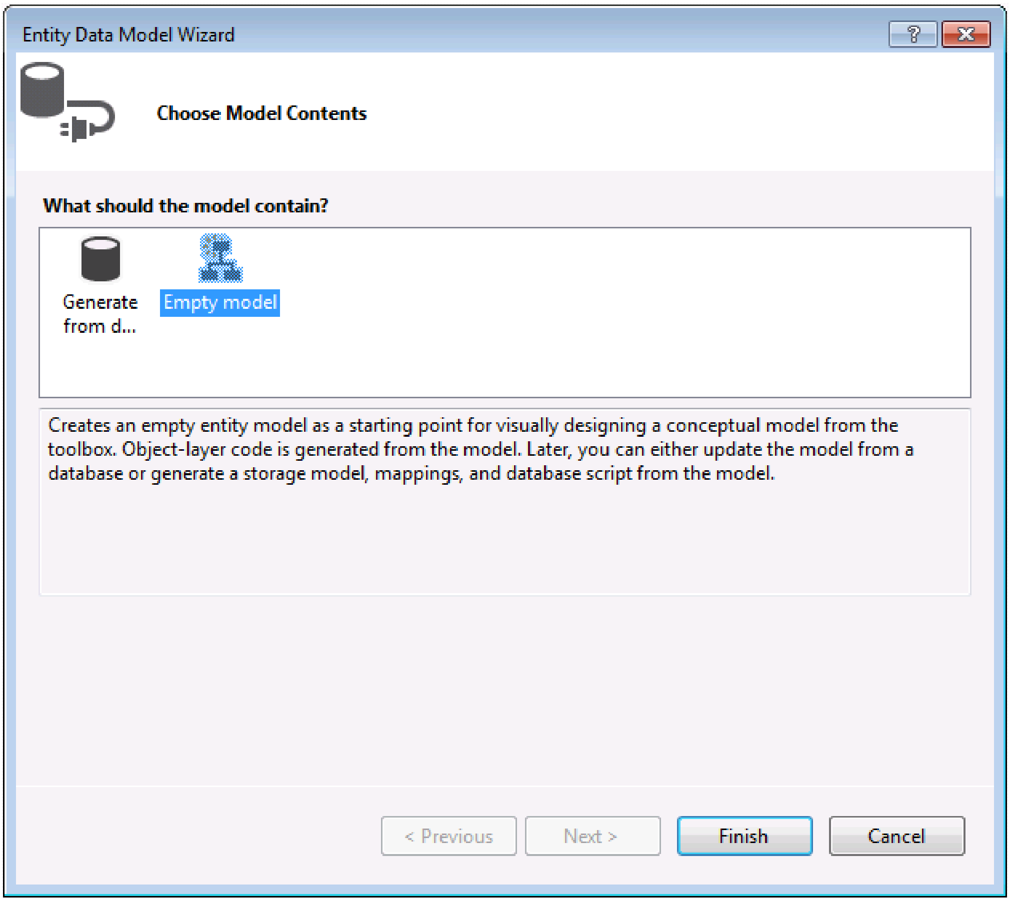
To keep things simple we're going to build a basic console application that uses the Model First to perform data access:

- Open Visual Studio
- **File -> New -> Project...**
- Select **Windows** from the left menu and **Console Application**
- Enter **ModelFirstSample** as the name
- Select **OK**

2. Create Model

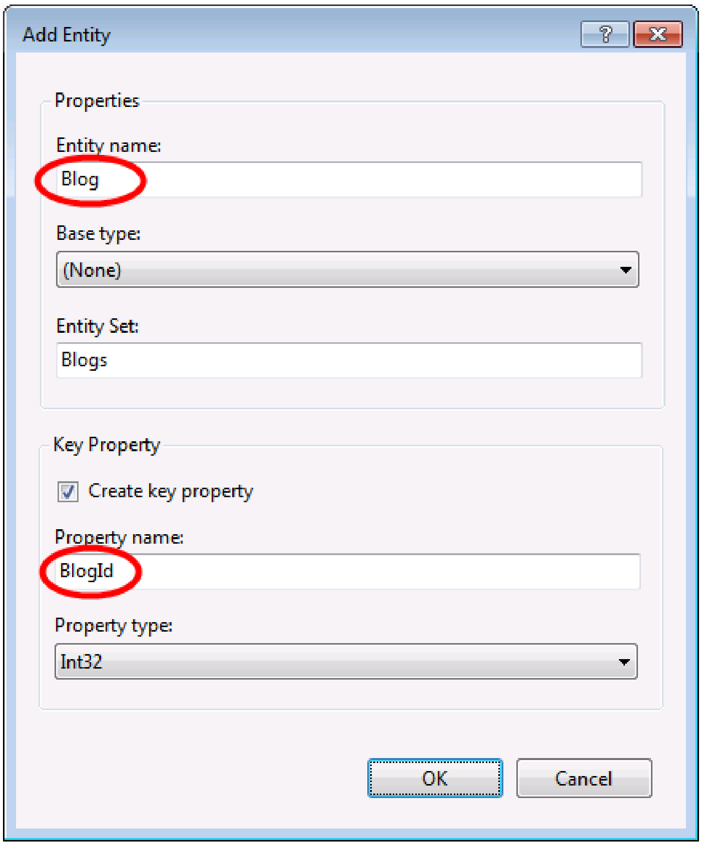
We're going to make use of Entity Framework Designer, which is included as part of Visual Studio, to create our model.

- **Project -> Add New Item...**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **BloggingModel** as the name and click **OK**, this launches the Entity Data Model Wizard
- Select **Empty Model** and click **Finish**



The Entity Framework Designer is opened with a blank model. Now we can start adding entities, properties and associations to the model.

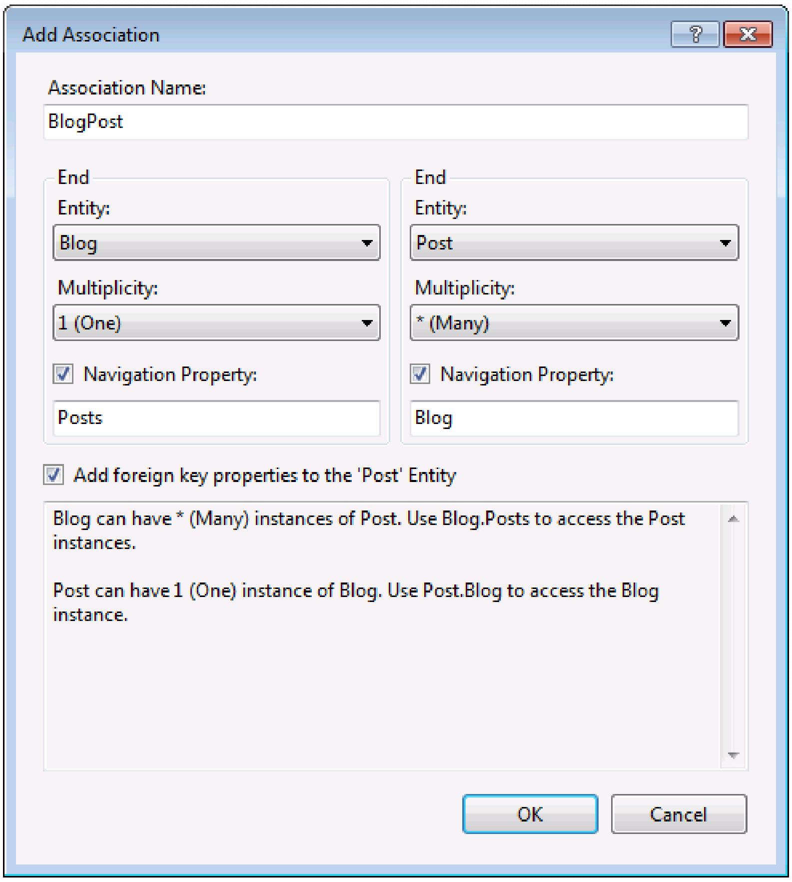
- Right-click on the design surface and select **Properties**
- In the Properties window change the **Entity Container Name** to **BloggingContext** *This is the name of the derived context that will be generated for you, the context represents a session with the database, allowing us to query and save data*
- Right-click on the design surface and select **Add New -> Entity...**
- Enter **Blog** as the entity name and **BlogId** as the key name and click **OK**



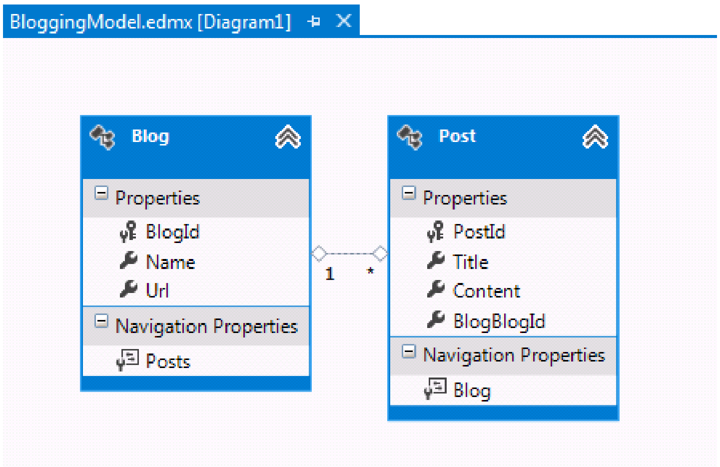
- Right-click on the new entity on the design surface and select **Add New -> Scalar Property**, enter **Name** as the name of the property.
- Repeat this process to add a **Url** property.
- Right-click on the **Url** property on the design surface and select **Properties**, in the Properties window change the **Nullable** setting to **True** *This allows us to save a Blog to the database without assigning it a Url*
- Using the techniques you just learnt, add a **Post** entity with a **PostId** key property
- Add **Title** and **Content** scalar properties to the **Post** entity

Now that we have a couple of entities, it's time to add an association (or relationship) between them.

- Right-click on the design surface and select **Add New -> Association...**
- Make one end of the relationship point to **Blog** with a multiplicity of **One** and the other end point to **Post** with a multiplicity of **Many** *This means that a Blog has many Posts and a Post belongs to one Blog*
- Ensure the **Add foreign key properties to 'Post' Entity** box is checked and click **OK**



We now have a simple model that we can generate a database from and use to read and write data.



Additional Steps in Visual Studio 2010

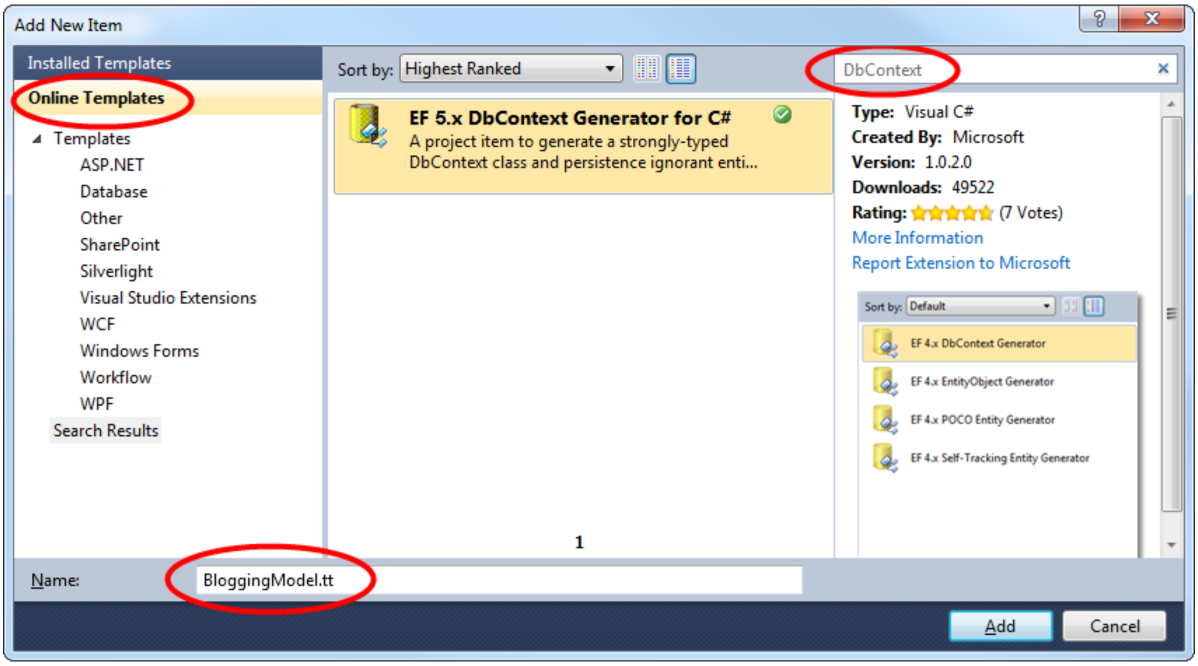
If you are working in Visual Studio 2010 there are some additional steps you need to follow to upgrade to the latest version of Entity Framework. Upgrading is important because it gives you access to an improved API surface, that is much easier to use, as well as the latest bug fixes.

First up, we need to get the latest version of Entity Framework from NuGet.

- **Project –> Manage NuGet Packages...** *If you don't have the **Manage NuGet Packages...** option you should install the latest version of NuGet*
- Select the **Online** tab
- Select the **EntityFramework** package
- Click **Install**

Next, we need to swap our model to generate code that makes use of the DbContext API, which was introduced in later versions of Entity Framework.

- Right-click on an empty spot of your model in the EF Designer and select **Add Code Generation Item...**
- Select **Online Templates** from the left menu and search for **DbContext**
- Select the EF **5.x DbContext Generator for C#**, enter **BloggingModel** as the name and click **Add**



3. Generating the Database

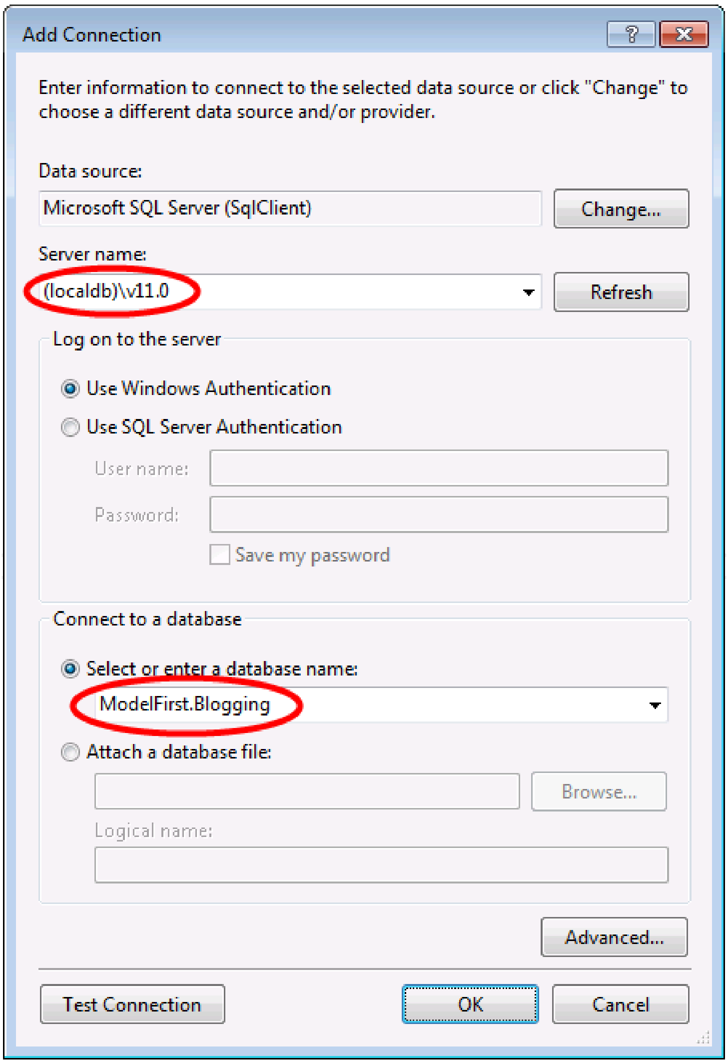
Given our model, Entity Framework can calculate a database schema that will allow us to store and retrieve data using the model.

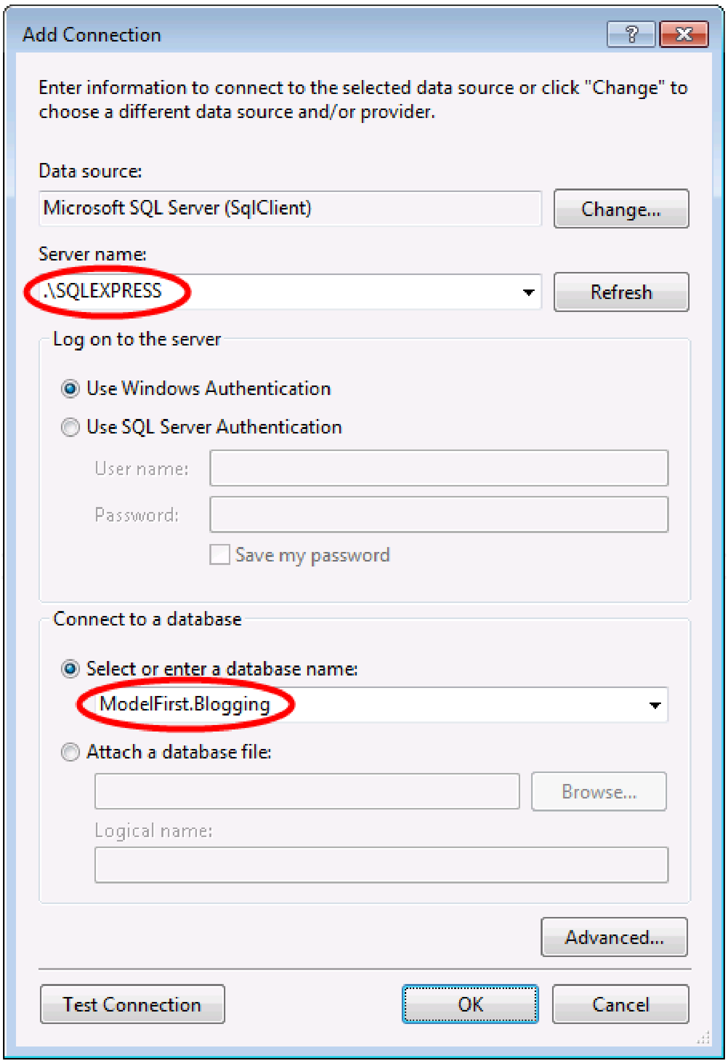
The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

- If you are using Visual Studio 2010 you'll be creating a SQL Express database.
- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.

Let's go ahead and generate the database.

- Right-click on the design surface and select **Generate Database from Model...**
- Click **New Connection...** and specify either LocalDB or SQL Express, depending on which version of Visual Studio you are using, enter **ModelFirst.Blogging** as the database name.


Add Connection
Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.
Data source:
Microsoft SQL Server (SqlClient)
Change...
Server name:
(localdb)\v11.0
Refresh
Log on to the server
Use Windows Authentication
Use SQL Server Authentication
User name:
Password:
Save my password
Connect to a database
Select or enter a database name:
ModelFirst.Blogging
Attach a database file:
Browse...
Logical name:
Advanced...
Test Connection
OK
Cancel

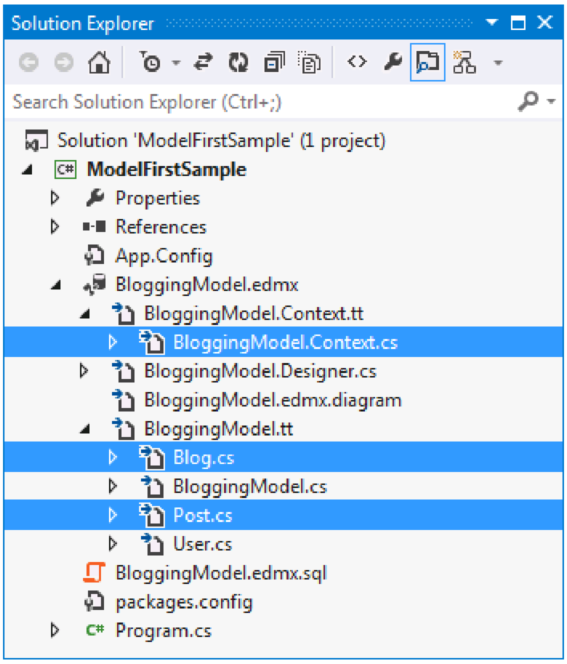


- Select **OK** and you will be asked if you want to create a new database, select **Yes**
- Select **Next** and the Entity Framework Designer will calculate a script to create the database schema
- Once the script is displayed, click **Finish** and the script will be added to your project and opened
- Right-click on the script and select **Execute**, you will be prompted to specify the database to connect to, specify LocalDB or SQL Server Express, depending on which version of Visual Studio you are using

4. Reading & Writing Data

Now that we have a model it's time to use it to access some data. The classes we are going to use to access data are being automatically generated for you based on the EDMX file.

This screen shot is from Visual Studio 2012, if you are using Visual Studio 2010 the BloggingModel.tt and BloggingModel.Context.tt files will be directly under your project rather than nested under the EDMX file.



Implement the Main method in Program.cs as shown below. This code creates a new instance of our context and then uses it to insert a new Blog. Then it uses a LINQ query to retrieve all Blogs from the database ordered alphabetically by 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();
        }
    }
}
```

You can now run the application and test it out.

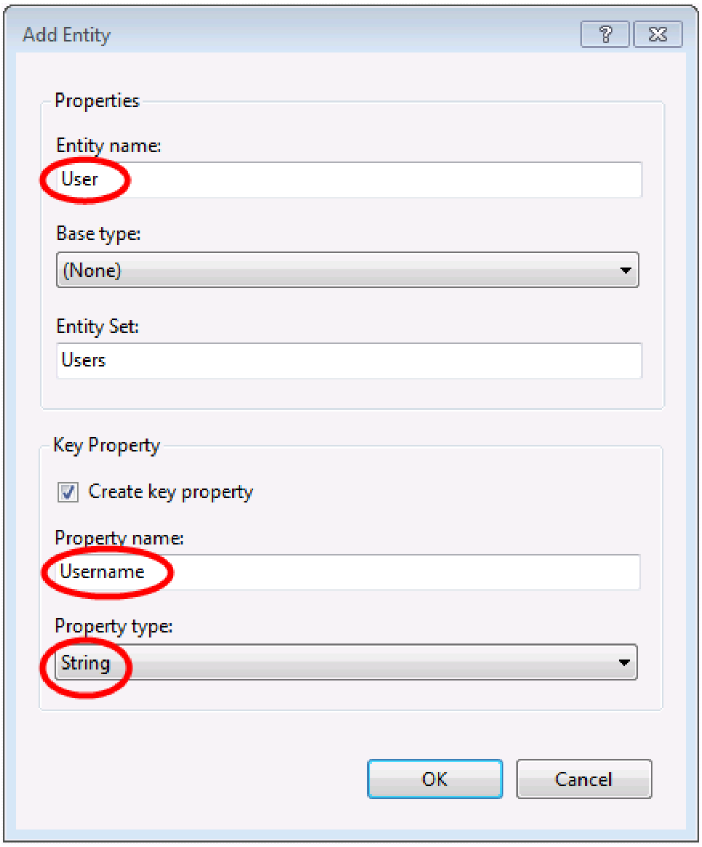
```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

5. Dealing with Model Changes

Now it's time to make some changes to our model, when we make these changes we also need to update the database schema.

We'll start by adding a new User entity to our model.

- Add a new **User** entity name with **Username** as the key name and **String** as the property type for the key



- Right-click on the **Username** property on the design surface and select **Properties**, In the Properties window change the **MaxLength** setting to **50** *This restricts the data that can be stored in username to 50 characters*
- Add a **DisplayName** scalar property to the **User** entity

We now have an updated model and we are ready to update the database to accommodate our new User entity type.

- Right-click on the design surface and select **Generate Database from Model...**, Entity Framework will calculate a script to recreate a schema based on the updated model.
- Click **Finish**
- You may receive warnings about overwriting the existing DDL script and the mapping and storage parts of the model, click **Yes** for both these warnings
- The updated SQL script to create the database is opened for you
  The script that is generated will drop all existing tables and then recreate the schema from scratch. This may work for local development but is not a viable for pushing changes to a database that has already been deployed. If you need to publish changes to a database that has already been deployed, you will need to edit the script or use a schema compare tool to calculate a migration script.
- Right-click on the script and select **Execute**, you will be prompted to specify the database to connect to, specify LocalDB or SQL Server Express, depending on which version of Visual Studio you are using

Summary

In this walkthrough we looked at Model First development, which allowed us to create a model in the EF Designer and then generate a database from that model. We then used the model to read and write some data from the database. Finally, we updated the model and then recreated the database schema to match the model.

Database First

9/18/2018 • 6 minutes to read • Edit Online

This video and step-by-step walkthrough provide an introduction to Database First development using Entity Framework. Database First allows you to reverse engineer a model from an existing database. The model is stored in an EDMX file (.edmx extension) and can be viewed and edited in the Entity Framework Designer. The classes that you interact with in your application are automatically generated from the EDMX file.

Watch the video

This video provides an introduction to Database First development using Entity Framework. Database First allows you to reverse engineer a model from an existing database. The model is stored in an EDMX file (.edmx extension) and can be viewed and edited in the Entity Framework Designer. The classes that you interact with in your application are automatically generated from the EDMX file.

Presented By: Rowan Miller

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have at least Visual Studio 2010 or Visual Studio 2012 installed to complete this walkthrough.

If you are using Visual Studio 2010, you will also need to have NuGet installed.

1. Create an Existing Database

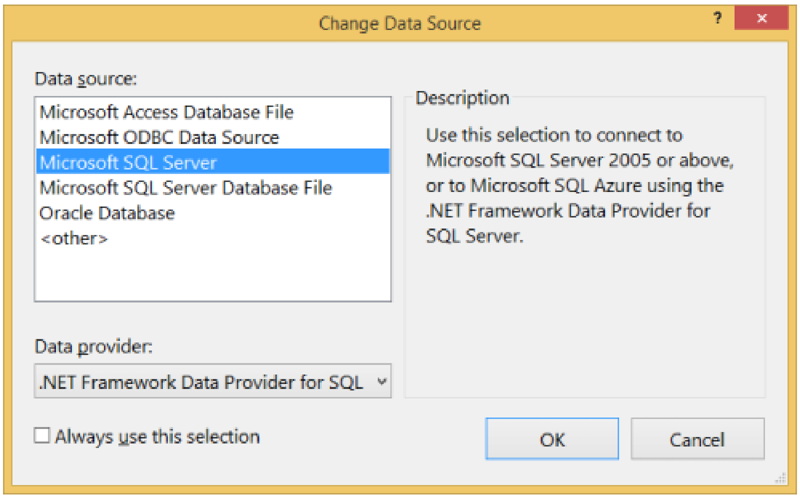
Typically when you are targeting an existing database it will already be created, but for this walkthrough we need to create a database to access.

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

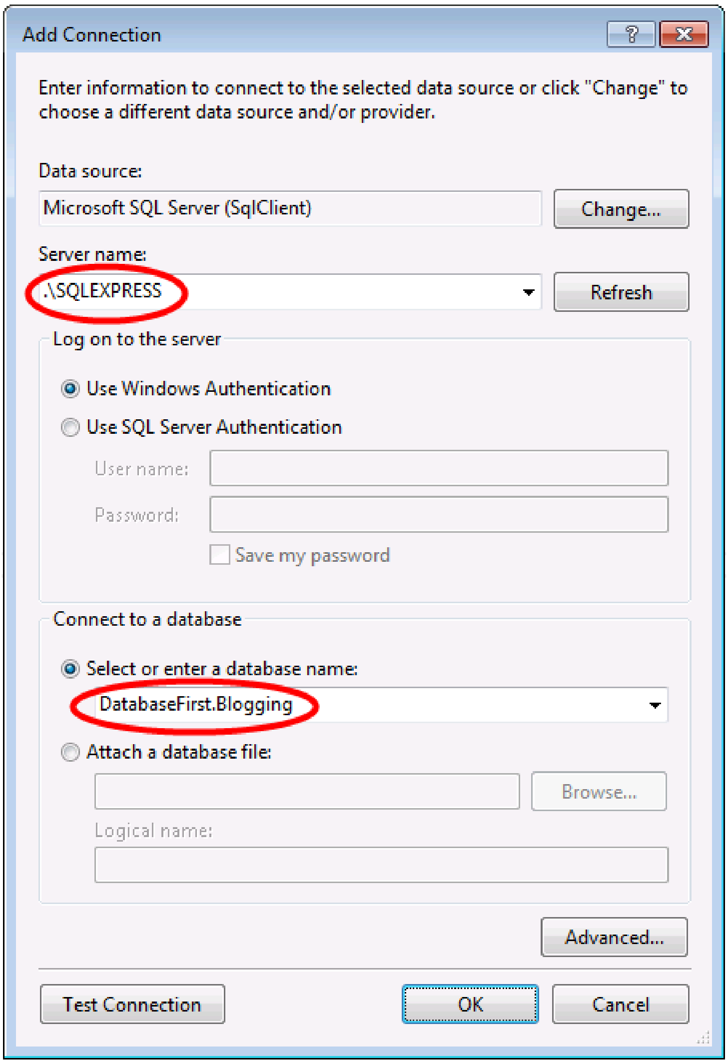
- If you are using Visual Studio 2010 you'll be creating a SQL Express database.
- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.

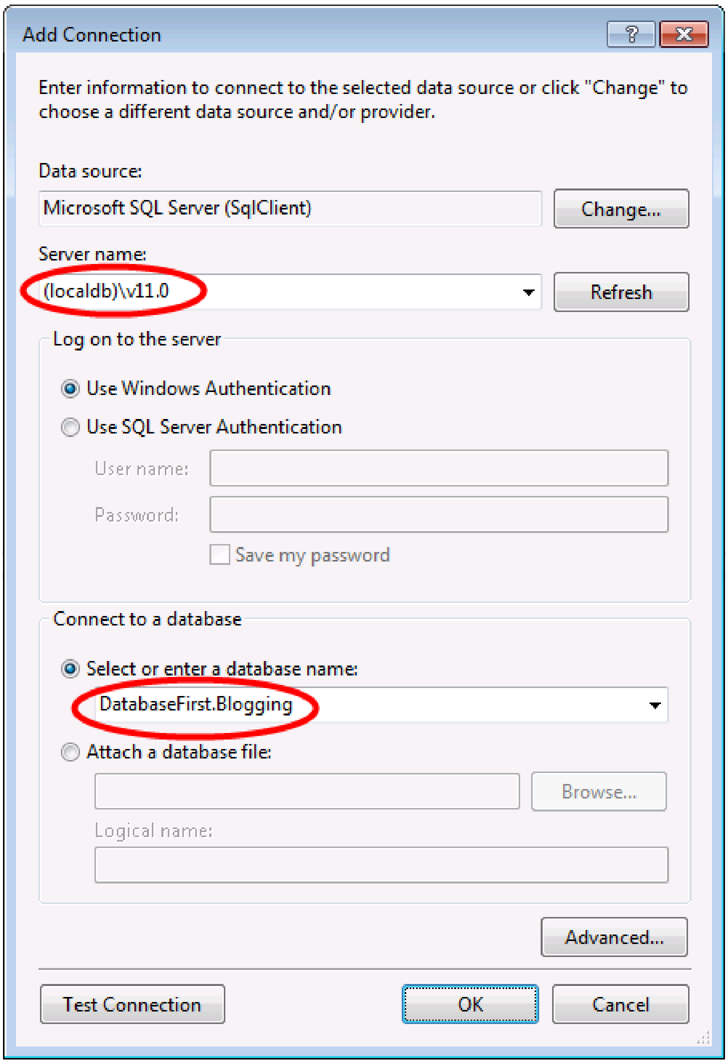
Let's go ahead and generate the database.

- Open Visual Studio
- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection...**
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source

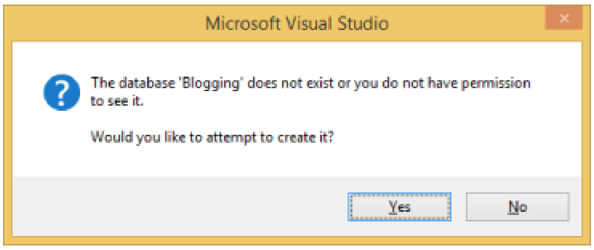


- Connect to either LocalDB or SQL Express, depending on which one you have installed, and enter **DatabaseFirst.Blogging** as the database name





- Select **OK** and you will be asked if you want to create a new database, select **Yes**



- The new database will now appear in Server Explorer, right-click on it and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **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
);
```

2. Create the Application

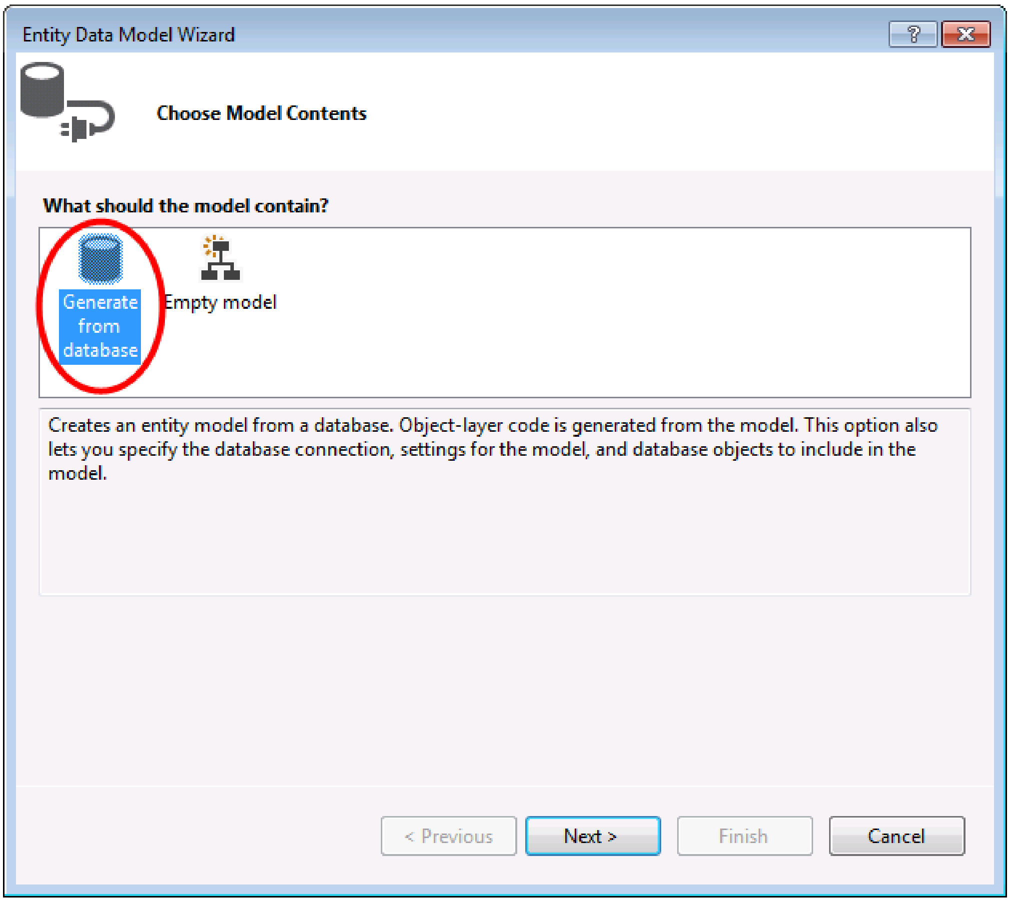
To keep things simple we're going to build a basic console application that uses the Database First to perform data access:

- Open Visual Studio
- **File -> New -> Project...**
- Select **Windows** from the left menu and **Console Application**
- Enter **DatabaseFirstSample** as the name
- Select **OK**

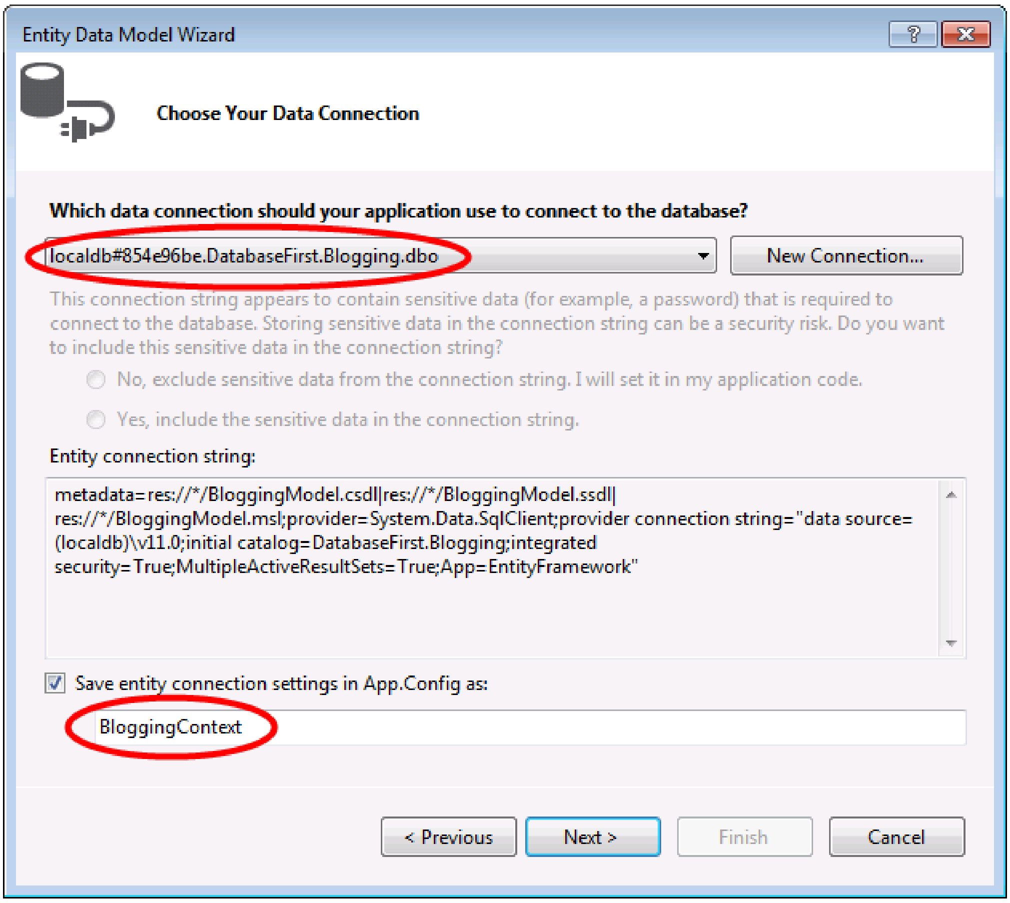
3. Reverse Engineer Model

We're going to make use of Entity Framework Designer, which is included as part of Visual Studio, to create our model.

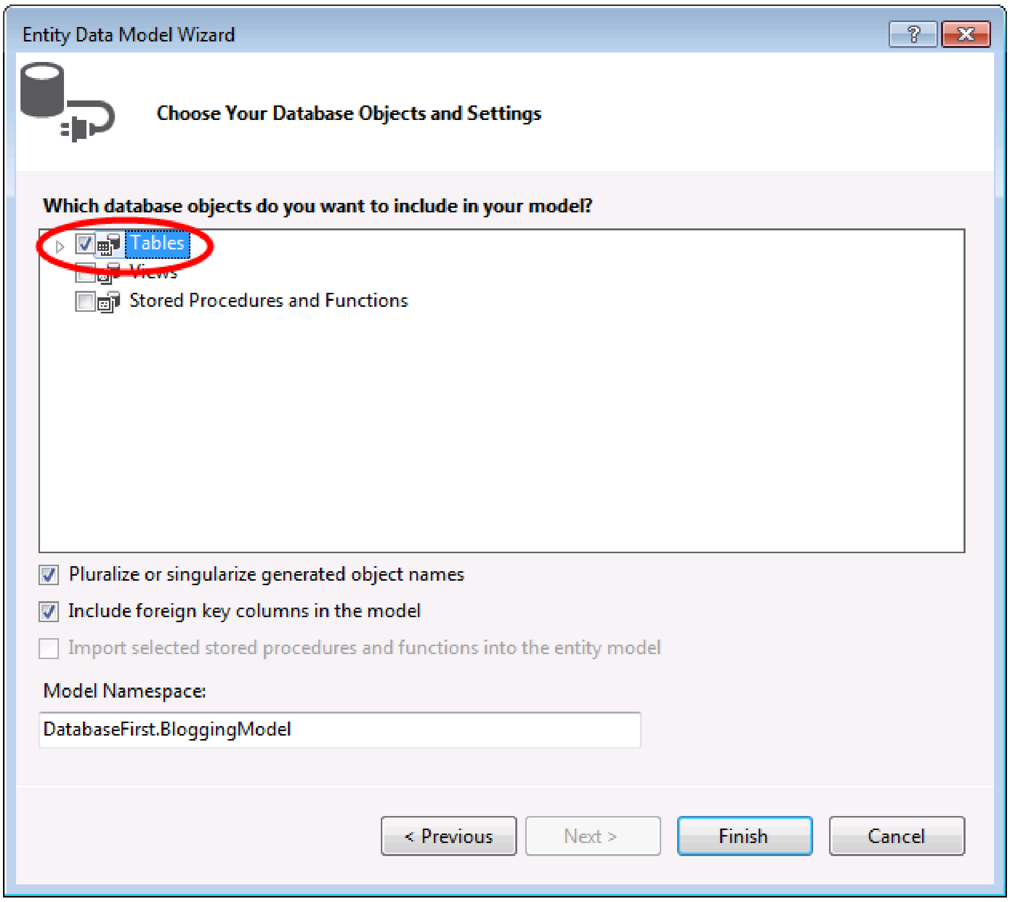
- **Project -> Add New Item...**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **BloggingModel** as the name and click **OK**
- This launches the **Entity Data Model Wizard**
- Select **Generate from Database** and click **Next**



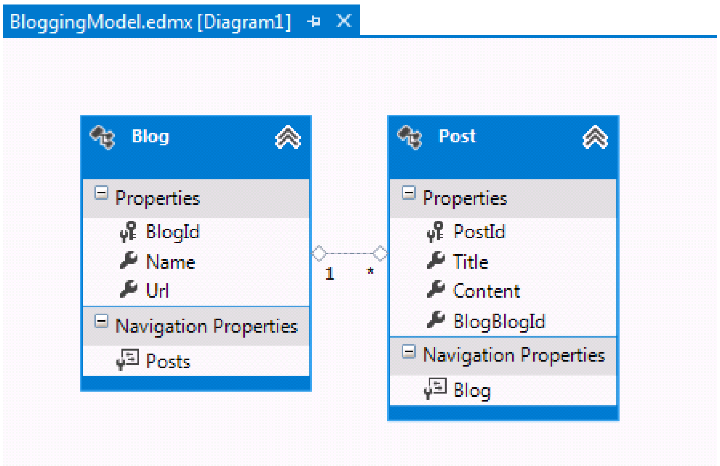
- Select the connection to the database you created in the first section, enter **BloggingContext** as the name of the connection string and click **Next**



- Click the checkbox next to ‘Tables’ to import all tables and click ‘Finish’



Once the reverse engineer process completes the new model is added to your project and opened up for you to view in the Entity Framework Designer. An App.config file has also been added to your project with the connection details for the database.



Additional Steps in Visual Studio 2010

If you are working in Visual Studio 2010 there are some additional steps you need to follow to upgrade to the latest version of Entity Framework. Upgrading is important because it gives you access to an improved API surface, that is much easier to use, as well as the latest bug fixes.

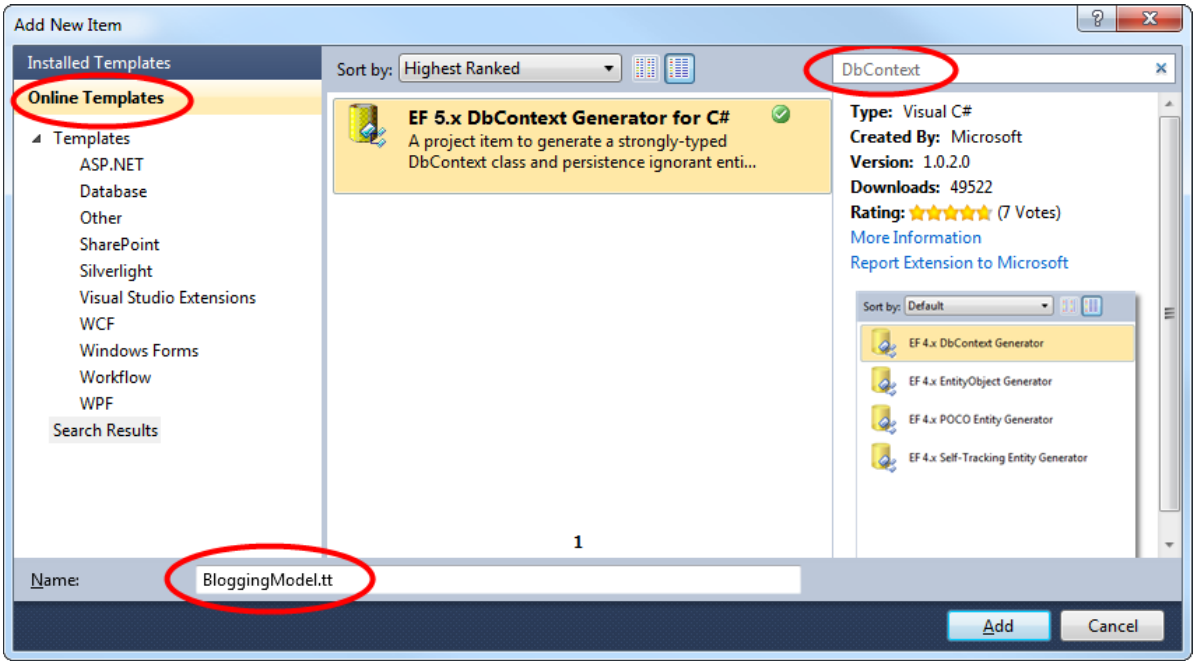
First up, we need to get the latest version of Entity Framework from NuGet.

- **Project –> Manage NuGet Packages...** *If you don't have the **Manage NuGet Packages...** option you should install the latest version of NuGet*
- Select the **Online** tab
- Select the **EntityFramework** package

- Click **Install**

Next, we need to swap our model to generate code that makes use of the DbContext API, which was introduced in later versions of Entity Framework.

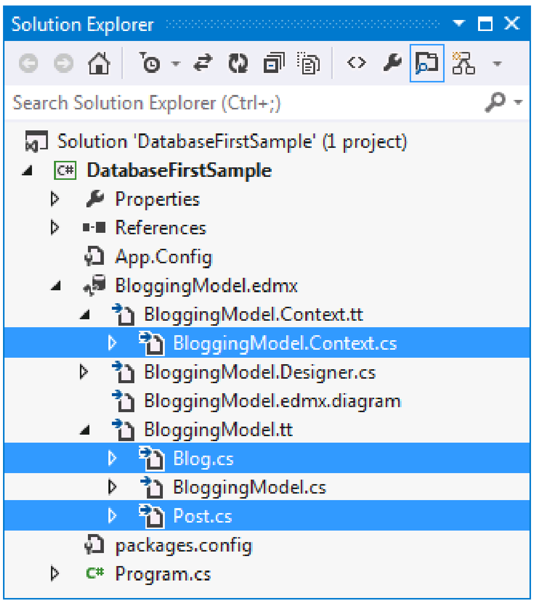
- Right-click on an empty spot of your model in the EF Designer and select **Add Code Generation Item...**
- Select **Online Templates** from the left menu and search for **DbContext**
- Select the EF **5.x DbContext Generator for C#**, enter **BloggingModel** as the name and click **Add**



4. Reading & Writing Data

Now that we have a model it's time to use it to access some data. The classes we are going to use to access data are being automatically generated for you based on the EDMX file.

This screen shot is from Visual Studio 2012, if you are using Visual Studio 2010 the BloggingModel.tt and BloggingModel.Context.tt files will be directly under your project rather than nested under the EDMX file.



Implement the Main method in Program.cs as shown below. This code creates a new instance of our context and then uses it to insert a new Blog. Then it uses a LINQ query to retrieve all Blogs from the database ordered alphabetically by 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();
        }
    }
}
```

You can now run the application and test it out.

```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

5. Dealing with Database Changes

Now it's time to make some changes to our database schema, when we make these changes we also need to update our model to reflect those changes.

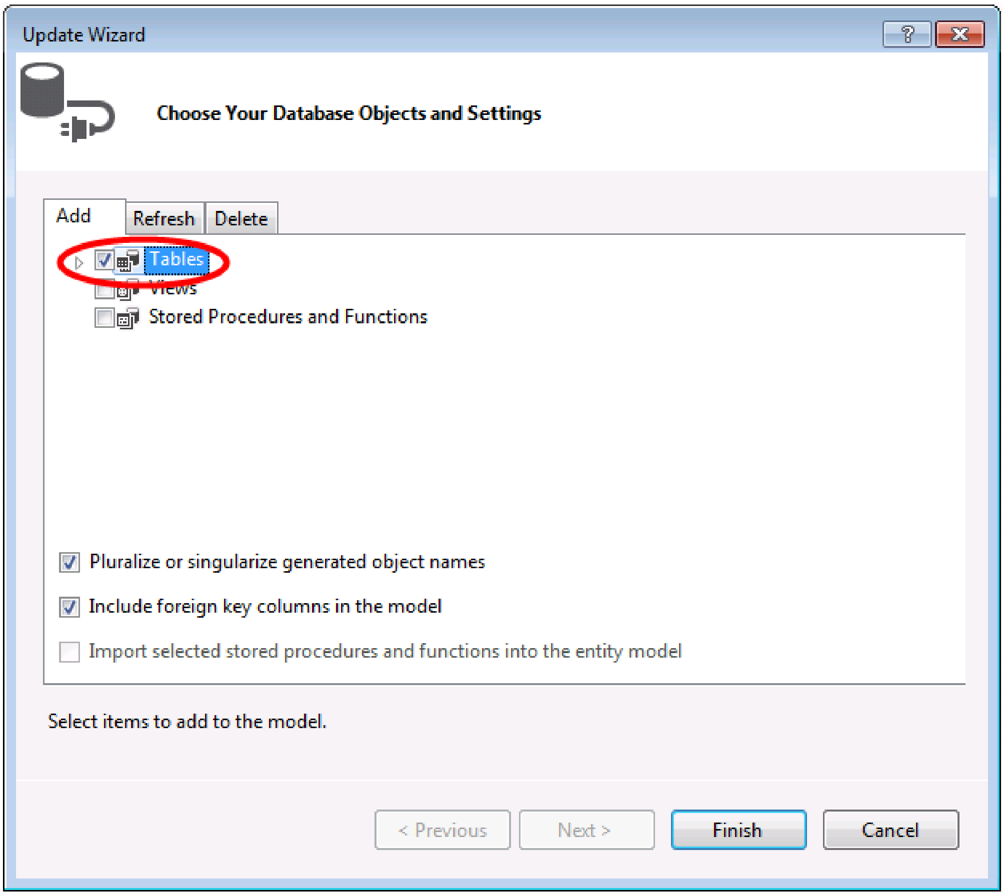
The first step is to make some changes to the database schema. We're going to add a Users table to the schema.

- Right-click on the **DatabaseFirst.Blogging** database in Server Explorer and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **Execute**

```
CREATE TABLE [dbo].[Users]
(
    [Username] NVARCHAR(50) NOT NULL PRIMARY KEY,
    [DisplayName] NVARCHAR(MAX) NULL
)
```

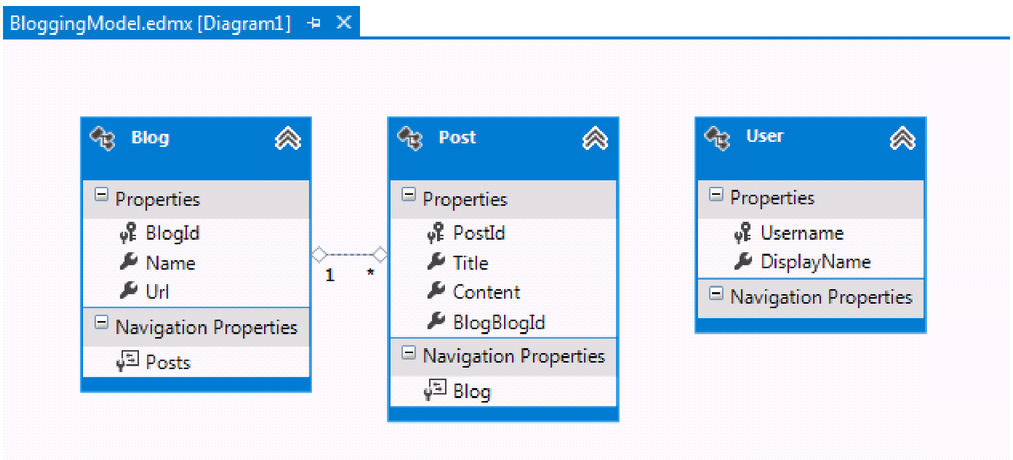
Now that the schema is updated, it's time to update the model with those changes.

- Right-click on an empty spot of your model in the EF Designer and select 'Update Model from Database…', this will launch the Update Wizard
- On the Add tab of the Update Wizard check the box next to Tables, this indicates that we want to add any new tables from the schema. *The Refresh tab shows any existing tables in the model that will be checked for changes during the update. The Delete tabs show any tables that have been removed from the schema and will also be removed from the model as part of the update. The information on these two tabs is automatically detected and is provided for informational purposes only, you cannot change any settings.*



- Click Finish on the Update Wizard

The model is now updated to include a new User entity that maps to the Users table we added to the database.



Summary

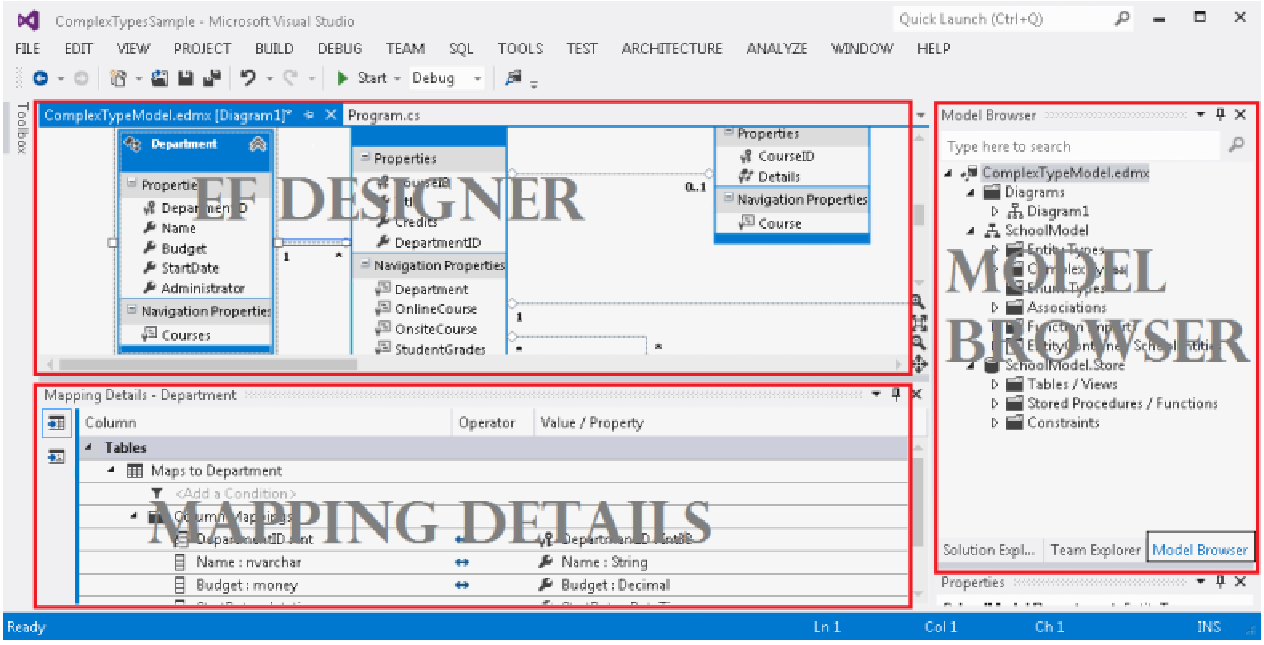
In this walkthrough we looked at Database First development, which allowed us to create a model in the EF Designer based on an existing database. We then used that model to read and write some data from the database. Finally, we updated the model to reflect changes we made to the database schema.

Complex Types - EF Designer

9/13/2018 • 6 minutes to read • Edit Online

This topic shows how to map complex types with the Entity Framework Designer (EF Designer) and how to query for entities that contain properties of complex type.

The following image shows the main windows that are used when working with the EF Designer.



> **NOTE**
> When you build the conceptual model, warnings about unmapped entities and associations may appear in the Error List. You can ignore these warnings because after you choose to generate the database from the model, the errors will go away.

What is a Complex Type

Complex types are non-scalar properties of entity types that enable scalar properties to be organized within entities. Like entities, complex types consist of scalar properties or other complex type properties.

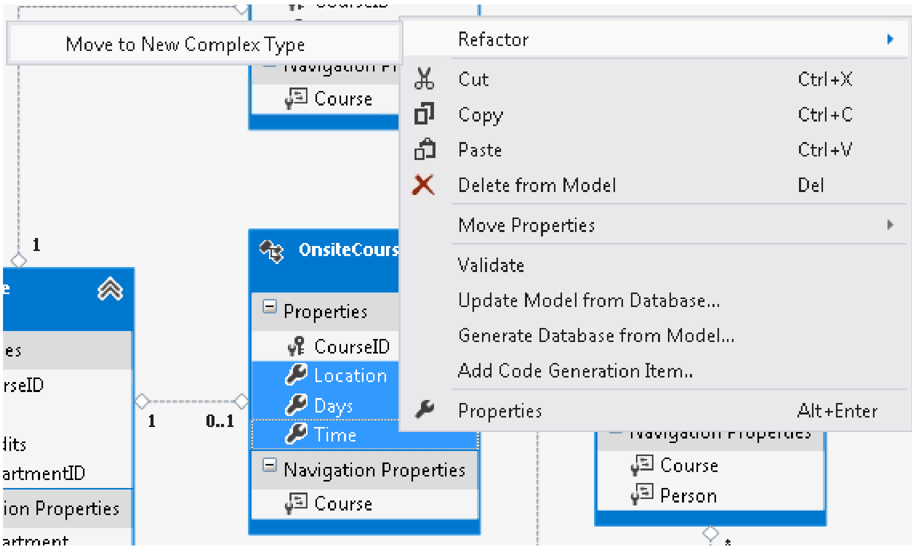
When you work with objects that represent complex types, be aware of the following:

- Complex types do not have keys and therefore cannot exist independently. Complex types can only exist as properties of entity types or other complex types.
- Complex types cannot participate in associations and cannot contain navigation properties.
- Complex type properties cannot be **null**. An **InvalidOperationException **occurs when **DbContext.SaveChanges** is called and a null complex object is encountered. Scalar properties of complex objects can be **null**.
- Complex types cannot inherit from other complex types.
- You must define the complex type as a **class**.
- EF detects changes to members on a complex type object when **DbContext.DetectChanges** is called. Entity Framework calls **DetectChanges** automatically when the following members are called: **DbSet.Find**, **DbSet.Local**, **DbSet.Remove**, **DbSet.Add**, **DbSet.Attach**, **DbContext.SaveChanges**, **DbContext.GetValidationErrors**, **DbContext.Entry**, **DbChangeTracker.Entries**.

Refactor an Entity's Properties into New Complex Type

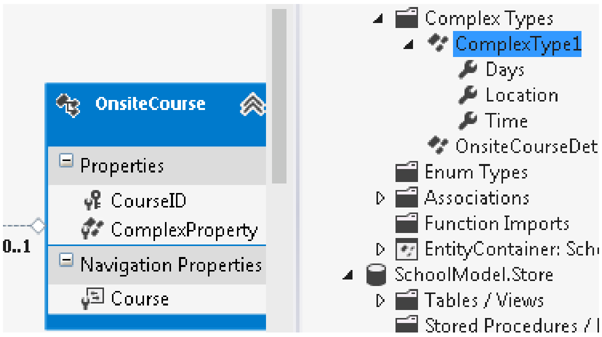
If you already have an entity in your conceptual model you may want to refactor some of the properties into a complex type property.

On the designer surface, select one or more properties (excluding navigation properties) of an entity, then right-click and select **Refactor -> Move to New Complex Type**.



A new complex type with the selected properties is added to the **Model Browser**. The complex type is given a default name.

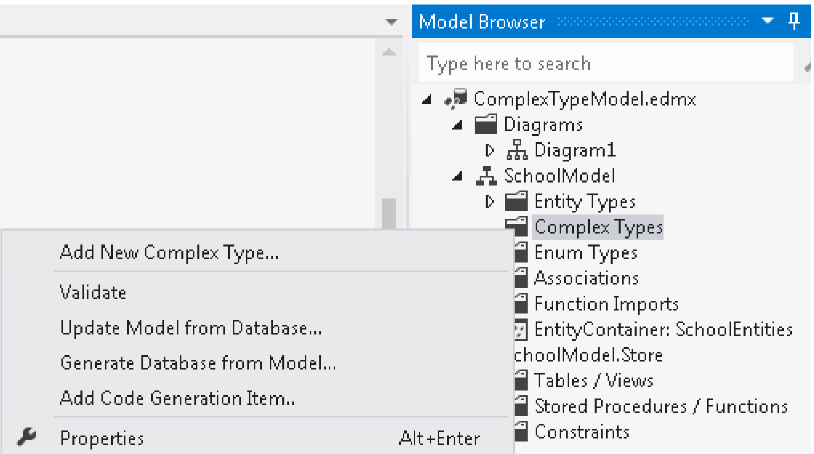
A complex property of the newly created type replaces the selected properties. All property mappings are preserved.



Create a New Complex Type

You can also create a new complex type that does not contain properties of an existing entity.

Right-click the **Complex Types** folder in the Model Browser, point to **AddNew Complex Type...**. Alternatively, you can select the **Complex Types** folder and press the **Insert** key on your keyboard.



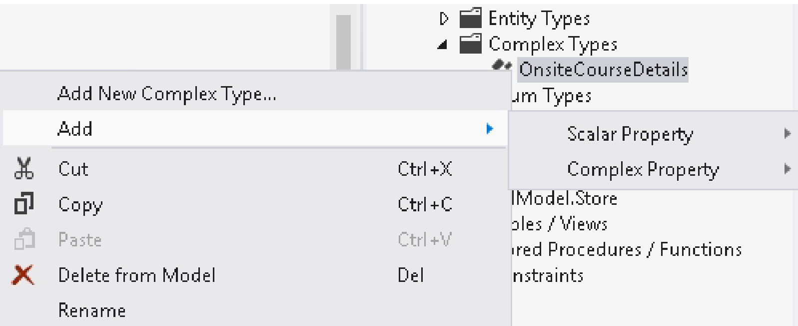
A new complex type is added to the folder with a default name. You can now add properties to the type.

Add Properties to a Complex Type

Properties of a complex type can be scalar types or existing complex types. However, complex type properties cannot have circular references. For example, a complex type **OnsiteCourseDetails** cannot have a property of complex type **OnsiteCourseDetails**.

You can add a property to a complex type in any of the ways listed below.

- Right-click a complex type in the Model Browser, point to **Add**, then point to **Scalar Property** or **Complex Property**, then select the desired property type. Alternatively, you can select a complex type and then press the **Insert** key on your keyboard.

  A new property is added to the complex type with a default name.

- OR -
- Right-click an entity property on the **EF Designer** surface and select **Copy**, then right-click the complex type in the **Model Browser** and select **Paste**.

Rename a Complex Type

When you rename a complex type, all references to the type are updated throughout the project.

- Slowly double-click a complex type in the **Model Browser**. The name will be selected and in edit mode.
- OR -
- Right-click a complex type in the **Model Browser** and select **Rename**.
- OR -
- Select a complex type in the Model Browser and press the F2 key.
- OR -

- Right-click a complex type in the **Model Browser** and select **Properties**. Edit the name in the **Properties** window.

Add an Existing Complex Type to an Entity and Map its Properties to Table Columns

1. Right-click an entity, point to **Add New**, and select **Complex Property**. A complex type property with a default name is added to the entity. A default type (chosen from the existing complex types) is assigned to the property.

2. Assign the desired type to the property in the **Properties** window. After adding a complex type property to an entity, you must map its properties to table columns.

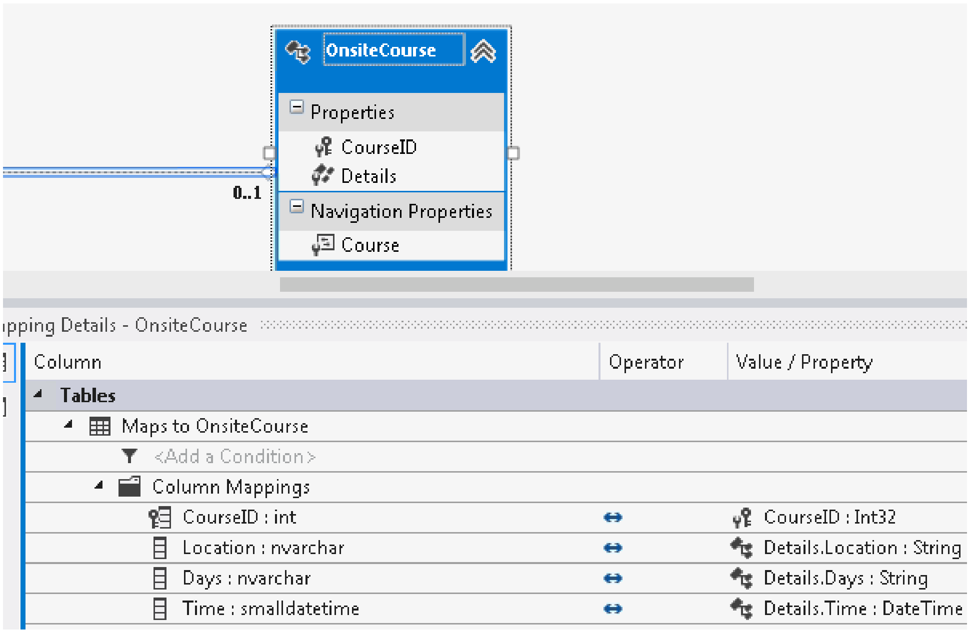
3. Right-click an entity type on the design surface or in the **Model Browser** and select **Table Mappings**. The table mappings are displayed in the **Mapping Details** window.

4. Expand the **Maps to <Table Name>** node. A **Column Mappings** node appears.

5. Expand the **Column Mappings** node. A list of all the columns in the table appears. The default properties (if any) to which the columns map are listed under the **Value/Property** heading.

6. Select the column you want to map, and then right-click the corresponding **Value/Property** field. A drop-down list of all the scalar properties is displayed.

7. Select the appropriate property.



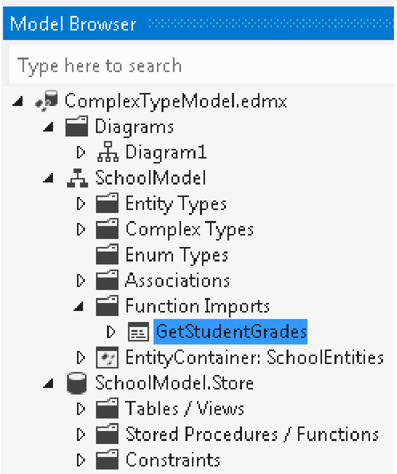
8. Repeat steps 6 and 7 for each table column.

> **NOTE**
> To delete a column mapping, select the column that you want to map, and then click the **Value/Property** field. Then, select **Delete** from the drop-down list.

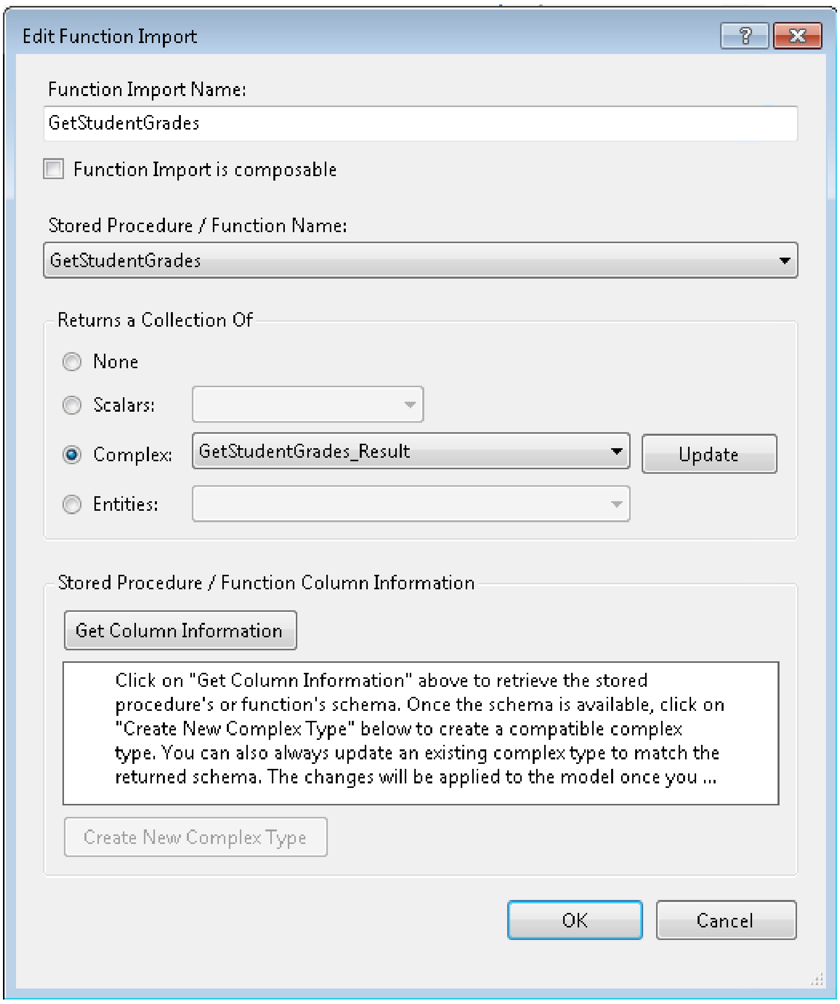
Map a Function Import to a Complex Type

Function imports are based on stored procedures. To map a function import to a complex type, the columns returned by the corresponding stored procedure must match the properties of the complex type in number and must have storage types that are compatible with the property types.

- Double-click on an imported function that you want to map to a complex type.



- Fill in the settings for the new function import, as follows:
  - Specify the stored procedure for which you are creating a function import in the **Stored Procedure Name** field. This field is a drop-down list that displays all the stored procedures in the storage model.
  - Specify the name of the function import in the **Function Import Name** field.
  - Select **Complex** as the return type and then specify the specific complex return type by choosing the appropriate type from the drop-down list.



- Click **OK**. The function import entry is created in the conceptual model.

Customize Column Mapping for Function Import

- Right-click the function import in the Model Browser and select **Function Import Mapping**. The **Mapping**

Details window appears and shows the default mapping for the function import. Arrows indicate the mappings between column values and property values. By default, the column names are assumed to be the same as the complex type's property names. The default column names appear in gray text.

- If necessary, change the column names to match the column names that are returned by the stored procedure that corresponds to the function import.

Delete a Complex Type

When you delete a complex type, the type is deleted from the conceptual model and mappings for all instances of the type are deleted. However, references to the type are not updated. For example, if an entity has a complex type property of type ComplexType1 and ComplexType1 is deleted in the **Model Browser**, the corresponding entity property is not updated. The model will not validate because it contains an entity that references a deleted complex type. You can update or delete references to deleted complex types by using the Entity Designer.

- Right-click a complex type in the Model Browser and select **Delete**.
- OR -
- Select a complex type in the Model Browser and press the Delete key on your keyboard.

Query for Entities Containing Properties of Complex Type

The following code shows how to execute a query that returns a collection of entity type objects that contain a complex type property.

```
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);
    }
}
```

Enum Support - EF Designer

9/18/2018 • 5 minutes to read • Edit Online

> **NOTE**
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

This video and step-by-step walkthrough shows how to use enum types with the Entity Framework Designer. It also demonstrates how to use enums in a LINQ query.

This walkthrough will use Model First to create a new database, but the EF Designer can also be used with the Database First workflow to map to an existing database.

Enum support was introduced in Entity Framework 5. To use the new features like enums, spatial data types, and table-valued functions, you must target .NET Framework 4.5. Visual Studio 2012 targets .NET 4.5 by default.

In Entity Framework, an enumeration can have the following underlying types: **Byte**, **Int16**, **Int32**, **Int64** , or **SByte**.

Watch the Video

This video shows how to use enum types with the Entity Framework Designer. It also demonstrates how to use enums in a LINQ query.

Presented By: Julia Kornich

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have Visual Studio 2012, Ultimate, Premium, Professional, or Web Express edition installed to complete this walkthrough.

Set up the Project

1. Open Visual Studio 2012
2. On the **File** menu, point to **New**, and then click **Project**
3. In the left pane, click **Visual C#**, and then select the **Console** template
4. Enter **EnumEFDesigner** as the name of the project and click **OK**

Create a New Model using the EF Designer

1. Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**
2. Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane
3. Enter **EnumTestModel.edmx** for the file name, and then click **Add**
4. On the Entity Data Model Wizard page, select **Empty Model** in the Choose Model Contents dialog box
5. Click **Finish**

The Entity Designer, which provides a design surface for editing your model, is displayed.

The wizard performs the following actions:

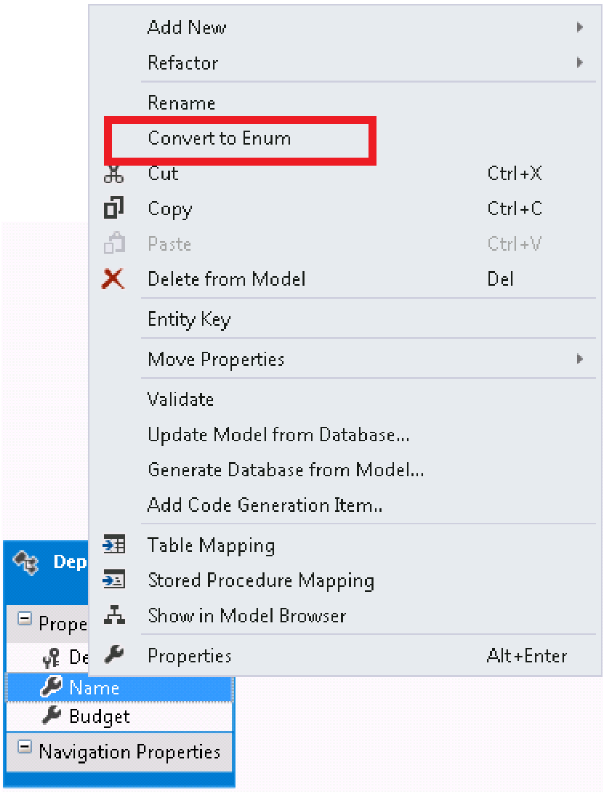
- Generates the EnumTestModel.edmx file that defines the conceptual model, the storage model, and the mapping between them. Sets the Metadata Artifact Processing property of the .edmx file to Embed in Output Assembly so the generated metadata files get embedded into the assembly.
- Adds a reference to the following assemblies: EntityFramework, System.ComponentModel.DataAnnotations, and System.Data.Entity.
- Creates EnumTestModel.tt and EnumTestModel.Context.tt files and adds them under the .edmx file. These T4 template files generate the code that defines the DbContext derived type and POCO types that map to the entities in the .edmx model.

Add a New Entity Type

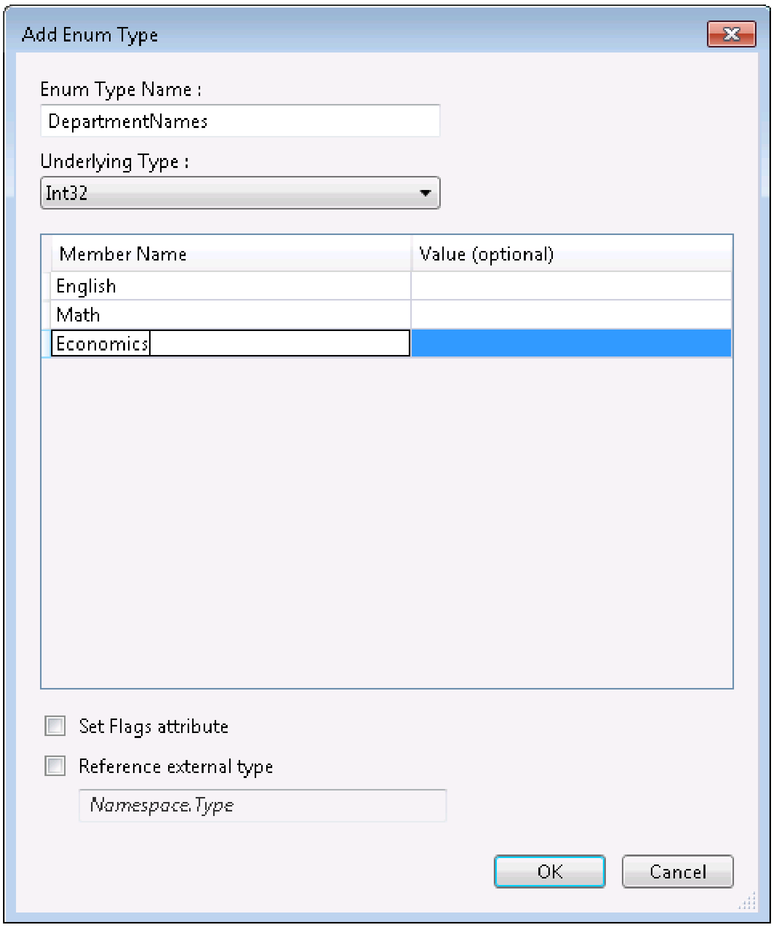
1. Right-click an empty area of the design surface, select **Add -> Entity**, the New Entity dialog box appears
2. Specify **Department** for the type name and specify **DepartmentID** for the key property name, leave the type as **Int32**
3. Click **OK**
4. Right-click the entity and select **Add New -> Scalar Property**
5. Rename the new property to **Name**
6. Change the type of the new property to **Int32** (by default, the new property is of String type) To change the type, open the Properties window and change the Type property to **Int32**
7. Add another scalar property and rename it to **Budget**, change the type to **Decimal**

Add an Enum Type

1. In the Entity Framework Designer, right-click the Name property, select **Convert to enum**



2. In the **Add Enum** dialog box type **DepartmentNames** for the Enum Type Name, change the Underlying Type to **Int32**, and then add the following members to the type: English, Math, and Economics



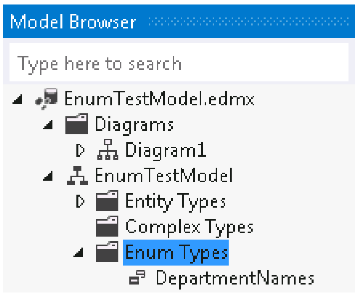
3. Press **OK**

4. Save the model and build the project

> **NOTE**
>
> When you build, warnings about unmapped entities and associations may appear in the Error List. You can ignore these warnings because after we choose to generate the database from the model, the errors will go away.

If you look at the Properties window, you will notice that the type of the Name property was changed to **DepartmentNames** and the newly added enum type was added to the list of types.

If you switch to the Model Browser window, you will see that the type was also added to the Enum Types node.



> **NOTE**
>
> You can also add new enum types from this window by clicking the right mouse button and selecting **Add Enum Type**. Once the type is created it will appear in the list of types and you would be able to associate with a property

Generate Database from Model

Now we can generate a database that is based on the model.

1. Right-click an empty space on the Entity Designer surface and select **Generate Database from Model**
2. The Choose Your Data Connection Dialog Box of the Generate Database Wizard is displayed Click the **New Connection** button Specify **(localdb)\mssqllocaldb** for the server name and **EnumTest** for the database and click **OK**
3. A dialog asking if you want to create a new database will pop up, click **Yes**.
4. Click **Next** and the Create Database Wizard generates data definition language (DDL) for creating a database The generated DDL is displayed in the Summary and Settings Dialog Box Note, that the DDL does not contain a definition for a table that maps to the enumeration type
5. Click **Finish** Clicking Finish does not execute the DDL script.
6. The Create Database Wizard does the following: Opens the **EnumTest.edmx.sql** in T-SQL Editor Generates the store schema and mapping sections of the EDMX file Adds connection string information to the App.config file
7. Click the right mouse button in T-SQL Editor and select **Execute** The Connect to Server dialog appears, enter the connection information from step 2 and click **Connect**
8. To view the generated schema, right-click on the database name in SQL Server Object Explorer and select **Refresh**

Persist and Retrieve Data

Open the Program.cs file where the Main method is defined. Add the following code into the Main function. The code adds a new Department object to the context. It then saves the data. The code also executes a LINQ query that returns a Department where the name is 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);
}
```

Compile and run the application. The program produces the following output:

```
DepartmentID: 1 Name: English
```

To view data in the database, right-click on the database name in SQL Server Object Explorer and select **Refresh**. Then, click the right mouse button on the table and select **View Data**.

Summary

In this walkthrough we looked at how to map enum types using the Entity Framework Designer and how to use enums in code.

Spatial - EF Designer

9/18/2018 • 5 minutes to read • Edit Online

> **NOTE**
>
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

The video and step-by-step walkthrough shows how to map spatial types with the Entity Framework Designer. It also demonstrates how to use a LINQ query to find a distance between two locations.

This walkthrough will use Model First to create a new database, but the EF Designer can also be used with the Database First workflow to map to an existing database.

Spatial type support was introduced in Entity Framework 5. Note that to use the new features like spatial type, enums, and Table-valued functions, you must target .NET Framework 4.5. Visual Studio 2012 targets .NET 4.5 by default.

To use spatial data types you must also use an Entity Framework provider that has spatial support. See provider support for spatial types for more information.

There are two main spatial data types: geography and geometry. The geography data type stores ellipsoidal data (for example, GPS latitude and longitude coordinates). The geometry data type represents Euclidean (flat) coordinate system.

Watch the video

This video shows how to map spatial types with the Entity Framework Designer. It also demonstrates how to use a LINQ query to find a distance between two locations.

Presented By: Julia Kornich

Video: WMV | MP4 | WMV (ZIP)

Pre-Requisites

You will need to have Visual Studio 2012, Ultimate, Premium, Professional, or Web Express edition installed to complete this walkthrough.

Set up the Project

1. Open Visual Studio 2012
2. On the **File** menu, point to **New**, and then click **Project**
3. In the left pane, click **Visual C#**, and then select the **Console** template
4. Enter **SpatialEFDesigner** as the name of the project and click **OK**

Create a New Model using the EF Designer

1. Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**
2. Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane
3. Enter **UniversityModel.edmx** for the file name, and then click **Add**

4. On the Entity Data Model Wizard page, select **Empty Model** in the Choose Model Contents dialog box
5. Click **Finish**

The Entity Designer, which provides a design surface for editing your model, is displayed.

The wizard performs the following actions:

- Generates the EnumTestModel.edmx file that defines the conceptual model, the storage model, and the mapping between them. Sets the Metadata Artifact Processing property of the .edmx file to Embed in Output Assembly so the generated metadata files get embedded into the assembly.
- Adds a reference to the following assemblies: EntityFramework, System.ComponentModel.DataAnnotations, and System.Data.Entity.
- Creates UniversityModel.tt and UniversityModel.Context.tt files and adds them under the .edmx file. These T4 template files generate the code that defines the DbContext derived type and POCO types that map to the entities in the .edmx model

Add a New Entity Type

1. Right-click an empty area of the design surface, select **Add -> Entity**, the New Entity dialog box appears
2. Specify **University** for the type name and specify **UniversityID** for the key property name, leave the type as **Int32**
3. Click **OK**
4. Right-click the entity and select **Add New -> Scalar Property**
5. Rename the new property to **Name**
6. Add another scalar property and rename it to **Location** Open the Properties window and change the type of the new property to **Geography**
7. Save the model and build the project

> **NOTE**
>
> When you build, warnings about unmapped entities and associations may appear in the Error List. You can ignore these warnings because after we choose to generate the database from the model, the errors will go away.

Generate Database from Model

Now we can generate a database that is based on the model.

1. Right-click an empty space on the Entity Designer surface and select **Generate Database from Model**
2. The Choose Your Data Connection Dialog Box of the Generate Database Wizard is displayed Click the **New Connection** button Specify **(localdb)\mssqllocaldb** for the server name and **University** for the database and click **OK**
3. A dialog asking if you want to create a new database will pop up, click **Yes**.
4. Click **Next** and the Create Database Wizard generates data definition language (DDL) for creating a database The generated DDL is displayed in the Summary and Settings Dialog Box Note, that the DDL does not contain a definition for a table that maps to the enumeration type
5. Click **Finish** Clicking Finish does not execute the DDL script.
6. The Create Database Wizard does the following: Opens the **UniversityModel.edmx.sql** in T-SQL Editor Generates the store schema and mapping sections of the EDMX file Adds connection string information to the App.config file
7. Click the right mouse button in T-SQL Editor and select **Execute** The Connect to Server dialog appears, enter the connection information from step 2 and click **Connect**
8. To view the generated schema, right-click on the database name in SQL Server Object Explorer and select

Refresh

Persist and Retrieve Data

Open the Program.cs file where the Main method is defined. Add the following code into the Main function.

The code adds two new University objects to the context. Spatial properties are initialized by using the DbGeography.FromText method. The geography point represented as WellKnownText is passed to the method. The code then saves the data. Then, the LINQ query that that returns a University object where its location is closest to the specified location, is constructed and executed.

```
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);
}
```

Compile and run the application. The program produces the following output:

```
The closest University to you is: School of Fine Art.
```

To view data in the database, right-click on the database name in SQL Server Object Explorer and select **Refresh**. Then, click the right mouse button on the table and select **View Data**.

Summary

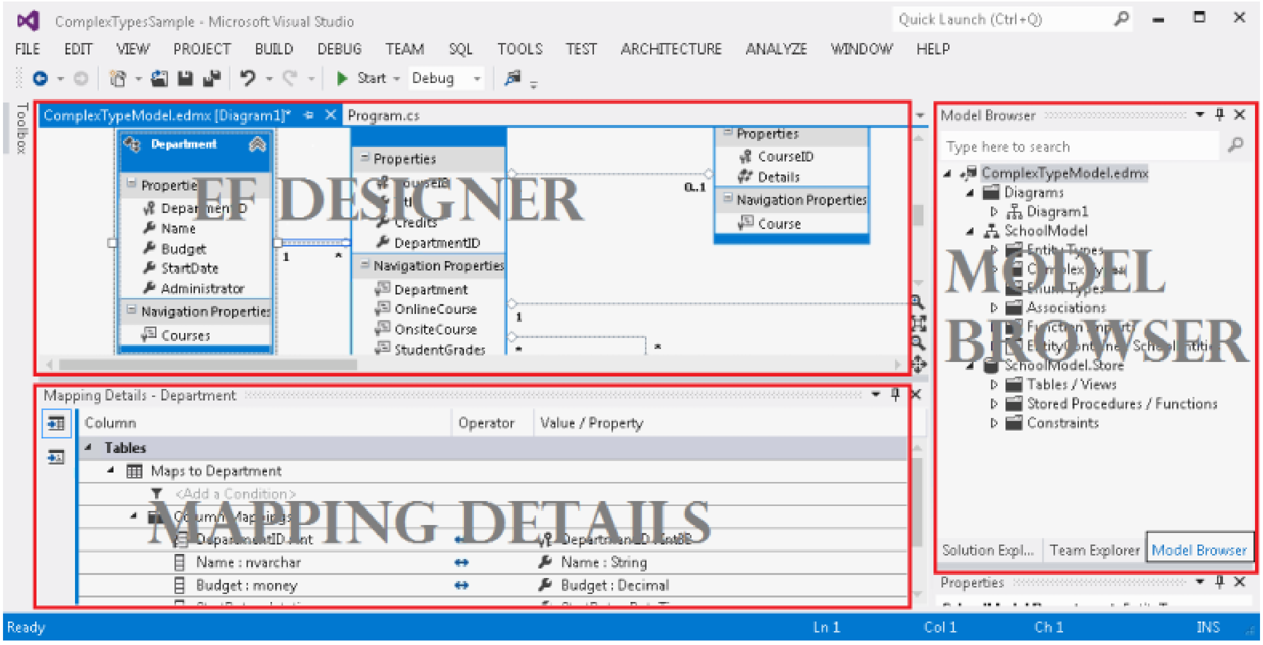
In this walkthrough we looked at how to map spatial types using the Entity Framework Designer and how to use spatial types in code.

Designer Entity Splitting

9/13/2018 • 4 minutes to read • Edit Online

This walkthrough shows how to map an entity type to two tables by modifying a model with the Entity Framework Designer (EF Designer). You can map an entity to multiple tables when the tables share a common key. The concepts that apply to mapping an entity type to two tables are easily extended to mapping an entity type to more than two tables.

The following image shows the main windows that are used when working with the EF Designer.



Prerequisites

Visual Studio 2012 or Visual Studio 2010, Ultimate, Premium, Professional, or Web Express edition.

Create the Database

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.
- If you are using Visual Studio 2010 you'll be creating a SQL Express database.

First we'll create a database with two tables that we are going to combine into a single entity.

- Open Visual Studio
- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection…**
- If you haven't connected to a database from Server Explorer before you'll need to select **Microsoft SQL Server** as the data source
- Connect to either LocalDB or SQL Express, depending on which one you have installed
- Enter **EntitySplitting** as the database name
- Select **OK** and you will be asked if you want to create a new database, select **Yes**
- The new database will now appear in Server Explorer

- If you are using Visual Studio 2012
  - Right-click on the database in Server Explorer and select **New Query**
  - Copy the following SQL into the new query, then right-click on the query and select **Execute**
- If you are using Visual Studio 2010
  - Select **Data -> Transact SQL Editor -> New Query Connection...**
  - Enter **.\SQLEXPRESS** as the server name and click **OK**
  - Select the **EntitySplitting** database from the drop down at the top of the query editor
  - Copy the following SQL into the new query, then right-click on the query and select **Execute 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
);
```

Create the Project

- On the **File** menu, point to **New**, and then click **Project**.
- In the left pane, click **Visual C#**, and then select the **Console Application** template.
- Enter **MapEntityToTablesSample** as the name of the project and click **OK**.
- Click **No** if prompted to save the SQL query created in the first section.

Create a Model based on the Database

- Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **MapEntityToTablesModel.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next.**
- Select the **EntitySplitting** connection from the drop down and click **Next**.
- In the Choose Your Database Objects dialog box, check the box next to the **Tables** node. This will add all the tables from the **EntitySplitting** database to the model.
- Click **Finish**.

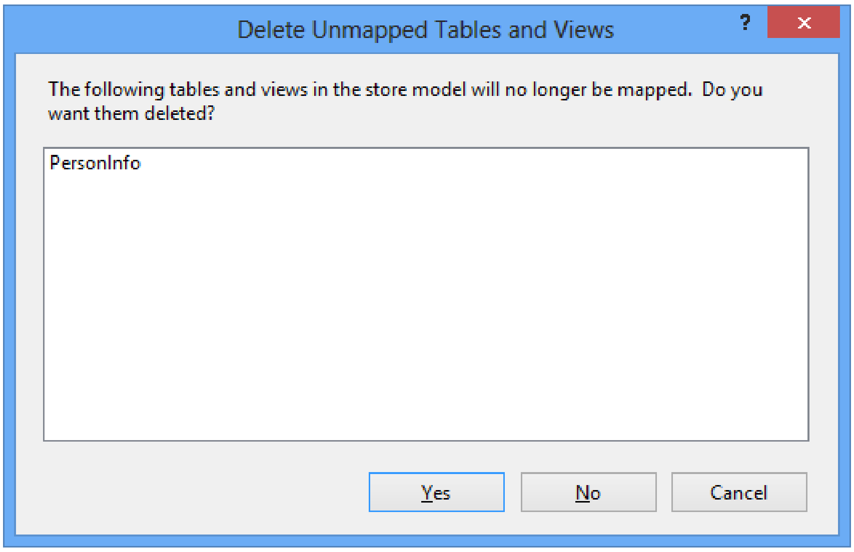
The Entity Designer, which provides a design surface for editing your model, is displayed.

Map an Entity to Two Tables

In this step we will update the **Person** entity type to combine data from the **Person** and **PersonInfo** tables.

- Select the **Email** and **Phone** properties of the **PersonInfo **entity and press **Ctrl+X** keys.
- Select the **Person **entity and press **Ctrl+V** keys.
- On the design surface, select the **PersonInfo** entity and press **Delete** button on the keyboard.

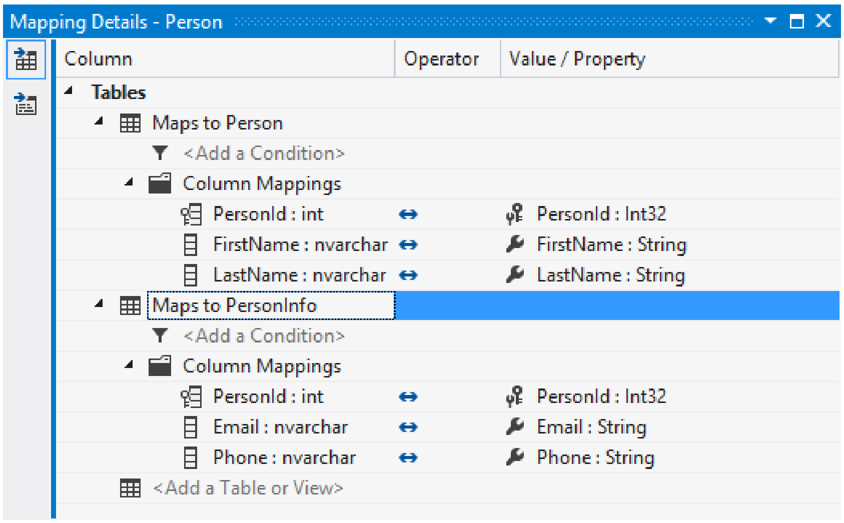
- Click **No** when asked if you want to remove the **PersonInfo** table from the model, we are about to map it to the **Person** entity.



The next steps require the **Mapping Details** window. If you cannot see this window, right-click the design surface and select **Mapping Details**.

- Select the **Person** entity type and click **<Add a Table or View>** in the **Mapping Details** window.
- Select **PersonInfo ** from the drop-down list. The **Mapping Details** window is updated with default column mappings, these are fine for our scenario.

The **Person** entity type is now mapped to the **Person** and **PersonInfo** tables.



Use the Model

- Paste the following code in the Main method.

```
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);
    }
}
```

- Compile and run the application.

The following T-SQL statements were executed against the database as a result of running this application.

- The following two **INSERT** statements were executed as a result of executing context.SaveChanges(). They take the data from the **Person** entity and split it between the **Person** and **PersonInfo** tables.

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]([Persor
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

- The following **SELECT** was executed as a result of enumerating the people in the database. It combines the data from the **Person** and **PersonInfo** table.

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

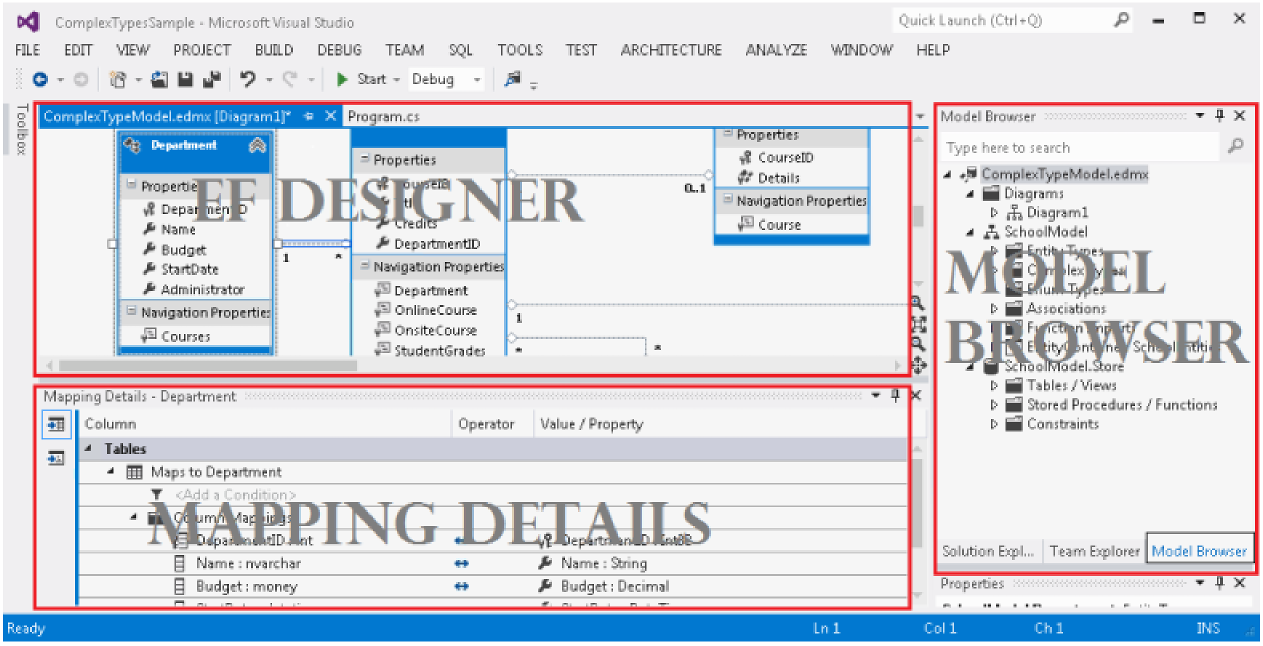
Designer Table Splitting

9/13/2018 • 3 minutes to read • Edit Online

This walkthrough shows how to map multiple entity types to a single table by modifying a model with the Entity Framework Designer (EF Designer).

One reason you may want to use table splitting is delaying the loading of some properties when using lazy loading to load your objects. You can separate the properties that might contain very large amount of data into a seperate entity and only load it when required.

The following image shows the main windows that are used when working with the EF Designer.



Prerequisites

To complete this walkthrough, you will need:

- A recent version of Visual Studio.
- The School sample database.

Set up the Project

This walkthrough is using Visual Studio 2012.

- Open Visual Studio 2012.
- On the **File** menu, point to **New**, and then click **Project**.
- In the left pane, click Visual C#, and then select the Console Application template.
- Enter **TableSplittingSample** as the name of the project and click **OK**.

Create a Model based on the School Database

- Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **TableSplittingModel.edmx** for the file name, and then click **Add**.

- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next.**
- Click New Connection. In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**. The Choose Your Data Connection dialog box is updated with your database connection setting.
- In the Choose Your Database Objects dialog box, unfold the **Tables** node and check the **Person** table. This will add the specified table to the **School** model.
- Click **Finish**.

The Entity Designer, which provides a design surface for editing your model, is displayed. All the objects that you selected in the **Choose Your Database Objects** dialog box are added to the model.

Map Two Entities to a Single Table

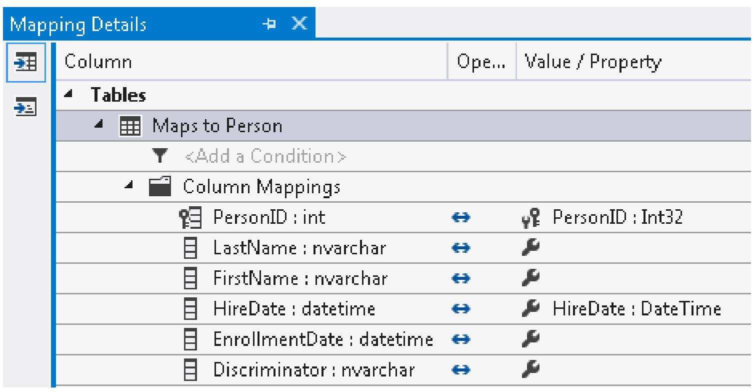
In this section you will split the **Person** entity into two entities and then map them to a single table.

> **NOTE**
>
> The **Person** entity does not contain any properties that may contain large amount of data; it is just used as an example.

- Right-click an empty area of the design surface, point to **Add New**, and click **Entity**. The **New Entity** dialog box appears.
- Type **HireInfo** for the **Entity name** and **PersonID** for the **Key Property** name.
- Click **OK**.
- A new entity type is created and displayed on the design surface.
- Select the **HireDate** property of the **Person** entity type and press **Ctrl+X** keys.
- Select the **HireInfo** entity and press **Ctrl+V** keys.
- Create an association between **Person** and **HireInfo**. To do this, right-click an empty area of the design surface, point to **Add New**, and click **Association**.
- The **Add Association** dialog box appears. The **PersonHireInfo** name is given by default.
- Specify multiplicity **1(One)** on both ends of the relationship.
- Press **OK**.

The next step requires the **Mapping Details** window. If you cannot see this window, right-click the design surface and select **Mapping Details**.

- Select the **HireInfo** entity type and click **<Add a Table or View>** in the **Mapping Details** window.
- Select **Person** from the **<Add a Table or View>** field drop-down list. The list contains tables or views to which the selected entity can be mapped. The appropriate properties should be mapped by default.



- Select the **PersonHireInfo** association on the design surface.
- Right-click the association on the design surface and select **Properties**.

- In the **Properties** window, select the **Referential Constraints** property and click the ellipses button.
- Select **Person** from the **Principal** drop-down list.
- Press **OK**.

Use the Model

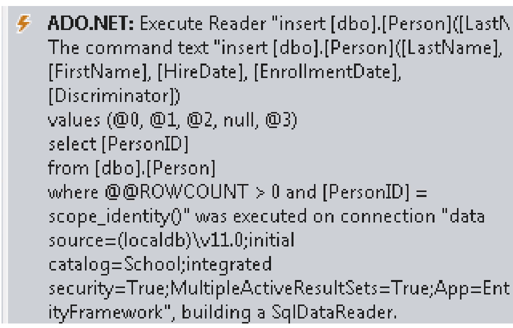
- Paste the following code in the Main method.

```
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);
}
```

- Compile and run the application.

The following T-SQL statements were executed against the **School** database as a result of running this application.

- The following **INSERT** was executed as a result of executing context.SaveChanges() and combines data from the **Person** and **HireInfo** entities



ADO.NET: Execute Reader "insert [dbo].[Person]([LastN
The command text "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.

- The following **SELECT** was executed as a result of executing context.People.FirstOrDefault() and selects just the columns mapped to **Person**

```
ADO.NET: Execute Reader "SELECT TOP (1)  [c].[Person
The command text "SELECT TOP (1)
[c].[PersonID] AS [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=Ent
ityFramework", building a SqlDataReader.
```

- The following **SELECT** was executed as a result of accessing the navigation property existingPerson.Instructor and selects just the columns mapped to **HireInfo**

```
ADO.NET: Execute Reader "SELECT  [Extent1].[PersonID
The command text "SELECT
[Extent1].[PersonID] AS [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=Ent
ityFramework", building a SqlDataReader.
```

Designer TPH Inheritance

9/13/2018 • 5 minutes to read • Edit Online

This step-by-step walkthrough shows how to implement table-per-hierarchy (TPH) inheritance in your conceptual model with the Entity Framework Designer (EF Designer). TPH inheritance uses one database table to maintain data for all of the entity types in an inheritance hierarchy.

In this walkthrough we will map the Person table to three entity types: Person (the base type), Student (derives from Person), and Instructor (derives from Person). We'll create a conceptual model from the database (Database First) and then alter the model to implement the TPH inheritance using the EF Designer.

It is possible to map to a TPH inheritance using Model First but you would have to write your own database generation workflow which is complex. You would then assign this workflow to the **Database Generation Workflow** property in the EF Designer. An easier alternative is to use Code First.

Other Inheritance Options

Table-per-Type (TPT) is another type of inheritance in which separate tables in the database are mapped to entities that participate in the inheritance. For information about how to map Table-per-Type inheritance with the EF Designer, see EF Designer TPT Inheritance.

Table-per-Concrete Type Inheritance (TPC) and mixed inheritance models are supported by the Entity Framework runtime but are not supported by the EF Designer. If you want to use TPC or mixed inheritance, you have two options: use Code First, or manually edit the EDMX file. If you choose to work with the EDMX file, the Mapping Details Window will be put into "safe mode" and you will not be able to use the designer to change the mappings.

Prerequisites

To complete this walkthrough, you will need:

- A recent version of Visual Studio.
- The School sample database.

Set up the Project

- Open Visual Studio 2012.
- Select **File-> New -> Project**
- In the left pane, click **Visual C#**, and then select the **Console** template.
- Enter **TPHDBFirstSample** as the name.
- Select **OK**.

Create a Model

- Right-click the project name in Solution Explorer, and select **Add -> New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **TPHModel.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next**.
- Click **New Connection**. In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**. The Choose Your Data Connection dialog box is updated with your database connection setting.

- In the Choose Your Database Objects dialog box, under the Tables node, select the **Person** table.
- Click **Finish**.

The Entity Designer, which provides a design surface for editing your model, is displayed. All the objects that you selected in the Choose Your Database Objects dialog box are added to the model.

That is how the **Person** table looks in the database.

dbo.Person [Design]

Update Script File: dbo.Person.sql

Name	Data Type	Allow Nulls
PersonID	int	☐
LastName	nvarchar(50)	☐
FirstName	nvarchar(50)	☐
HireDate	datetime	☑
EnrollmentDate	datetime	☑
Discriminator	nvarchar(50)	☐
		☐

Implement Table-per-Hierarchy Inheritance

The **Person** table has the **Discriminator** column, which can have one of two values: "Student" and "Instructor". Depending on the value the **Person** table will be mapped to the **Student** entity or the **Instructor** entity. The **Person** table also has two columns, **HireDate** and **EnrollmentDate**, which must be **nullable** because a person cannot be a student and an instructor at the same time (at least not in this walkthrough).

Add new Entities

- Add a new entity. To do this, right-click on an empty space of the design surface of the Entity Framework Designer, and select **Add->Entity**.
- Type **Instructor** for the **Entity name** and select **Person** from the drop-down list for the **Base type**.
- Click **OK**.
- Add another new entity. Type **Student** for the **Entity name** and select **Person** from the drop-down list for the **Base type**.

Two new entity types were added to the design surface. An arrow points from the new entity types to the **Person** entity type; this indicates that **Person** is the base type for the new entity types.

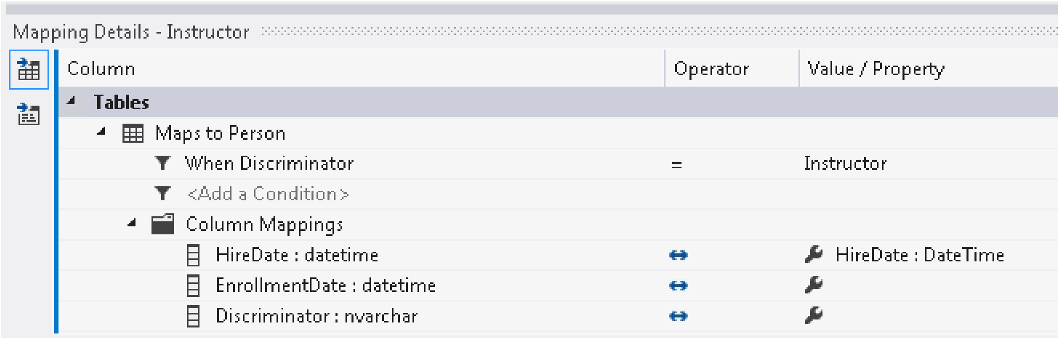
- Right-click the **HireDate** property of the **Person** entity. Select **Cut** (or use the Ctrl-X key).
- Right-click the **Instructor** entity and select **Paste** (or use the Ctrl-V key).
- Right-click the **HireDate** property and select **Properties**.
- In the **Properties** window, set the **Nullable** property to **false**.
- Right-click the **EnrollmentDate** property of the **Person** entity. Select **Cut** (or use the Ctrl-X key).
- Right-click the **Student** entity and select **Paste(or use the Ctrl-V key).**
- Select the **EnrollmentDate** property and set the **Nullable** property to **false**.
- Select the **Person** entity type. In the **Properties** window, set its **Abstract** property to **true**.
- Delete the **Discriminator** property from **Person**. The reason it should be deleted is explained in the following section.

Map the entities

- Right-click the **Instructor** and select **Table Mapping.** The Instructor entity is selected in the Mapping Details window.

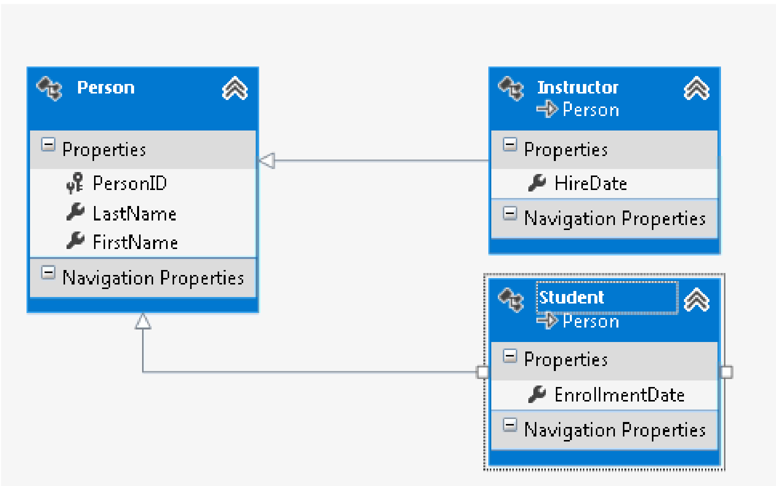
- Click **<Add a Table or View>** in the **Mapping Details** window. The **<Add a Table or View>** field becomes a drop-down list of tables or views to which the selected entity can be mapped.

- Select **Person** from the drop-down list.
- The **Mapping Details** window is updated with default column mappings and an option for adding a condition.
- Click on **<Add a Condition>**. The **<Add a Condition>** field becomes a drop-down list of columns for which conditions can be set.
- Select **Discriminator** from the drop-down list.
- In the **Operator** column of the **Mapping Details** window, select = from the drop-down list.
- In the **Value/Property** column, type **Instructor**. The end result should look like this:



- Repeat these steps for the **Student** entity type, but make the condition equal to **Student** value.
  *The reason we wanted to remove the **Discriminator** property, is because you cannot map a table column more than once. This column will be used for conditional mapping, so it cannot be used for property mapping as well. The only way it can be used for both, if a condition uses an **Is Null** or **Is Not Null** comparison.*

Table-per-hierarchy inheritance is now implemented.



Use the Model

Open the **Program.cs** file where the **Main** method is defined. Paste the following code into the **Main** function. The code executes three queries. The first query brings back all **Person** objects. The second query uses the **OfType** method to return **Instructor** objects. The third query uses the **OfType** method to return **Student** objects.

```
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);
    }
}
```

Designer TPT Inheritance

9/13/2018 • 3 minutes to read • Edit Online

This step-by-step walkthrough shows how to implement table-per-type (TPT) inheritance in your model using the Entity Framework Designer (EF Designer). Table-per-type inheritance uses a separate table in the database to maintain data for non-inherited properties and key properties for each type in the inheritance hierarchy.

In this walkthrough we will map the **Course** (base type), **OnlineCourse** (derives from Course), and **OnsiteCourse** (derives from **Course**) entities to tables with the same names. We'll create a model from the database and then alter the model to implement the TPT inheritance.

You can also start with the Model First and then generate the database from the model. The EF Designer uses the TPT strategy by default and so any inheritance in the model will be mapped to separate tables.

Other Inheritance Options

Table-per-Hierarchy (TPH) is another type of inheritance in which one database table is used to maintain data for all of the entity types in an inheritance hierarchy. For information about how to map Table-per-Hierarchy inheritance with the Entity Designer, see EF Designer TPH Inheritance.

Note that, the Table-per-Concrete Type Inheritance (TPC) and mixed inheritance models are supported by the Entity Framework runtime but are not supported by the EF Designer. If you want to use TPC or mixed inheritance, you have two options: use Code First, or manually edit the EDMX file. If you choose to work with the EDMX file, the Mapping Details Window will be put into “safe mode” and you will not be able to use the designer to change the mappings.

Prerequisites

To complete this walkthrough, you will need:

- A recent version of Visual Studio.
- The School sample database.

Set up the Project

- Open Visual Studio 2012.
- Select **File-> New -> Project**
- In the left pane, click **Visual C#**, and then select the **Console** template.
- Enter **TPTDBFirstSample** as the name.
- Select **OK**.

Create a Model

- Right-click the project in Solution Explorer, and select **Add -> New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **TPTModel.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select** Generate from database**, and then click **Next**.
- Click **New Connection**. In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**. The Choose Your Data Connection dialog box is updated with your database connection setting.

- In the Choose Your Database Objects dialog box, under the Tables node, select the **Department**, **Course, OnlineCourse, and OnsiteCourse** tables.
- Click **Finish**.

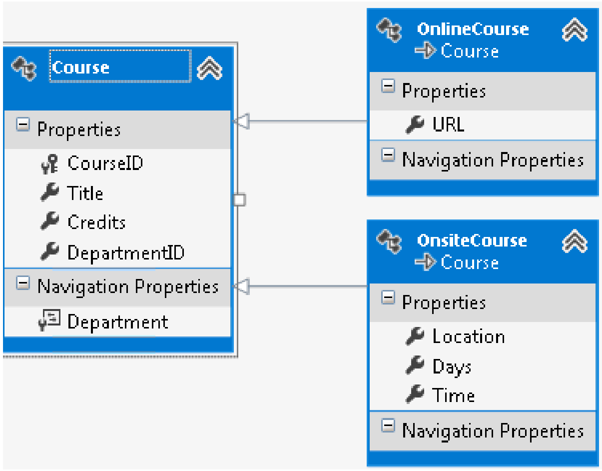
The Entity Designer, which provides a design surface for editing your model, is displayed. All the objects that you selected in the Choose Your Database Objects dialog box are added to the model.

Implement Table-per-Type Inheritance

- On the design surface, right-click the **OnlineCourse** entity type and select **Properties**.
- In the **Properties** window, set the Base Type property to **Course**.
- Right-click the **OnsiteCourse** entity type and select **Properties**.
- In the **Properties** window, set the Base Type property to **Course**.
- Right-click the association (the line) between the **OnlineCourse** and **Course** entity types. Select **Delete from Model**.
- Right-click the association between the **OnsiteCourse** and **Course** entity types. Select **Delete from Model**.

We will now delete the **CourseID** property from **OnlineCourse** and **OnsiteCourse** because these classes inherit **CourseID** from the **Course** base type.

- Right-click the **CourseID** property of the **OnlineCourse** entity type, and then select **Delete from Model**.
- Right-click the **CourseID** property of the **OnsiteCourse** entity type, and then select **Delete from Model**
- Table-per-type inheritance is now implemented.



Use the Model

Open the **Program.cs** file where the **Main** method is defined. Paste the following code into the **Main** function. The code executes three queries. The first query brings back all **Courses** related to the specified department. The second query uses the **OfType** method to return **OnlineCourses** related to the specified department. The third query returns **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);
        }
    }
}
```

Designer Query Stored Procedures

9/18/2018 • 3 minutes to read • Edit Online

This step-by-step walkthrough show how to use the Entity Framework Designer (EF Designer) to import stored procedures into a model and then call the imported stored procedures to retrieve results.

Note, that Code First does not support mapping to stored procedures or functions. However, you can call stored procedures or functions by using the System.Data.Entity.DbSet.SqlQuery method. For example:

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]")`;
```

Prerequisites

To complete this walkthrough, you will need:

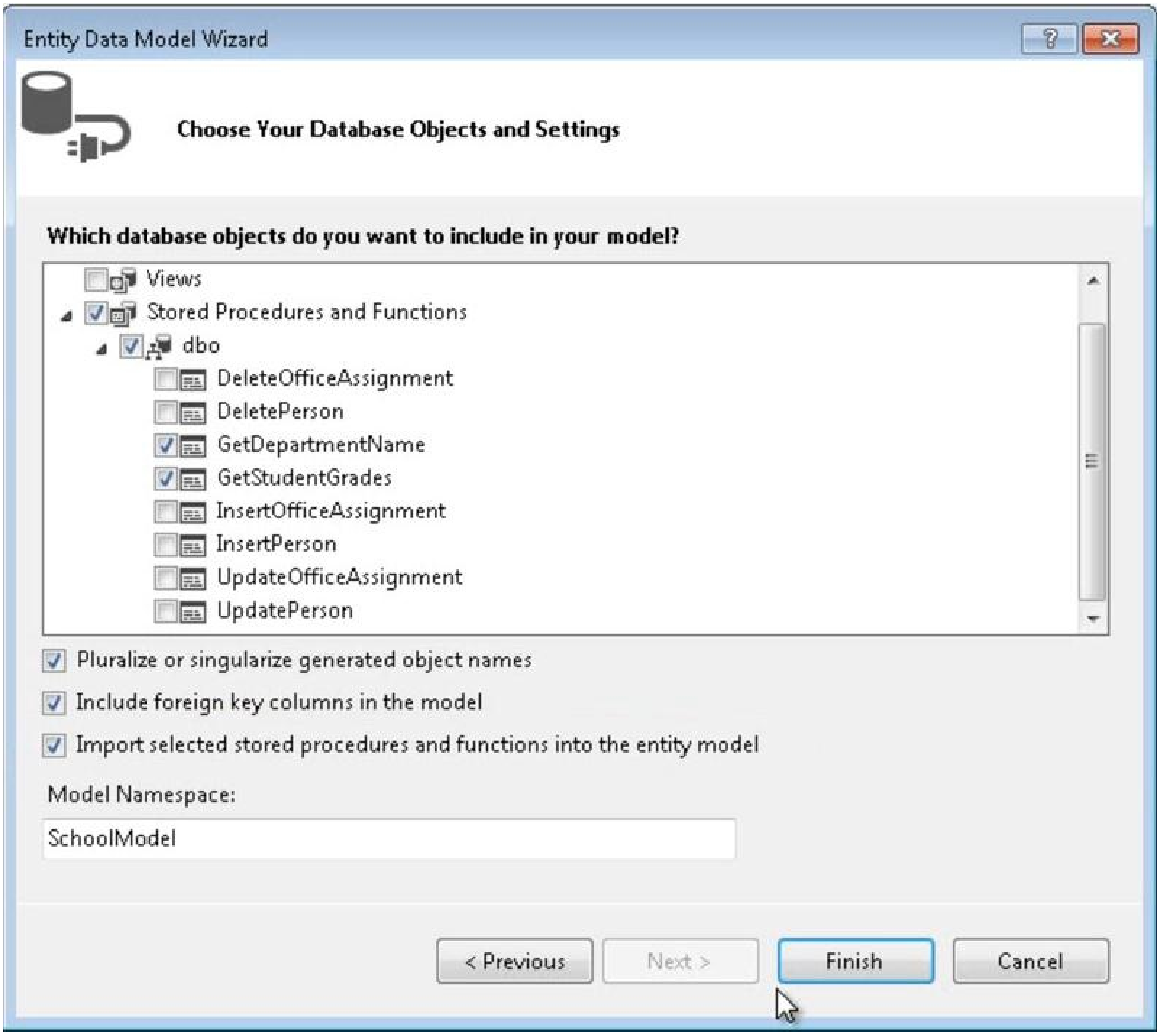
- A recent version of Visual Studio.
- The School sample database.

Set up the Project

- Open Visual Studio 2012.
- Select **File-> New -> Project**
- In the left pane, click **Visual C#**, and then select the **Console** template.
- Enter **EFwithSProcsSample** as the name.
- Select **OK**.

Create a Model

- Right-click the project in Solution Explorer and select **Add -> New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **EFwithSProcsModel.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next**.
- Click **New Connection**.
  In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**.
  The Choose Your Data Connection dialog box is updated with your database connection setting.
- In the Choose Your Database Objects dialog box, check the **Tables** checkbox to select all the tables.
  Also, select the following stored procedures under the **Stored Procedures and Functions** node: **GetStudentGrades** and **GetDepartmentName**.



*Starting with Visual Studio 2012 the EF Designer supports bulk import of stored procedures. The **Import selected stored procedures and functions into theentity model** is checked by default.*

- Click **Finish**.

By default, the result shape of each imported stored procedure or function that returns more than one column will automatically become a new complex type. In this example we want to map the results of the **GetStudentGrades** function to the **StudentGrade** entity and the results of the **GetDepartmentName** to **none** (**none** is the default value).

For a function import to return an entity type, the columns returned by the corresponding stored procedure must exactly match the scalar properties of the returned entity type. A function import can also return collections of simple types, complex types, or no value.

- Right-click the design surface and select **Model Browser**.
- In **Model Browser**, select **Function Imports**, and then double-click the **GetStudentGrades** function.
- In the Edit Function Import dialog box, select **Entities** and choose **StudentGrade**.
  *The **Function Import is composable** checkbox at the top of the **Function Imports** dialog will let you map to composable functions. If you do check this box, only composable functions (Table-valued Functions) will appear in the **Stored Procedure / Function Name** drop-down list. If you do not check this box, only non-composable functions will be shown in the list.*

Use the Model

Open the **Program.cs** file where the **Main** method is defined. Add the following code into the Main function.

The code calls two stored procedures: **GetStudentGrades** (returns **StudentGrades** for the specified *StudentId*) and **GetDepartmentName** (returns the name of the department in the output parameter).

```
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);

}
```

Compile and run the application. The program produces the following output:

```
StudentID: 2
Student grade: 4.00
StudentID: 2
Student grade: 3.50
The department name is Engineering
```

Output Parameters

If output parameters are used, their values will not be available until the results have been read completely. This is due to the underlying behavior of DbDataReader, see Retrieving Data Using a DataReader for more details.

Designer CUD Stored Procedures

9/13/2018 • 5 minutes to read • Edit Online

This step-by-step walkthrough show how to map the create\insert, update, and delete (CUD) operations of an entity type to stored procedures using the Entity Framework Designer (EF Designer). By default, the Entity Framework automatically generates the SQL statements for the CUD operations, but you can also map stored procedures to these operations.

Note, that Code First does not support mapping to stored procedures or functions. However, you can call stored procedures or functions by using the System.Data.Entity.DbSet.SqlQuery method. For example:

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]");
```

Considerations when Mapping the CUD Operations to Stored Procedures

When mapping the CUD operations to stored procedures, the following considerations apply:

- If you are mapping one of the CUD operations to a stored procedure, map all of them. If you do not map all three, the unmapped operations will fail if executed and an **UpdateException** will be thrown.
- You must map every parameter of the stored procedure to entity properties.
- If the server generates the primary key value for the inserted row, you must map this value back to the entity's key property. In the example that follows, the **InsertPerson** stored procedure returns the newly created primary key as part of the stored procedure's result set. The primary key is mapped to the entity key (**PersonID**) using the **<Add Result Bindings>** feature of the EF Designer.
- The stored procedure calls are mapped 1:1 with the entities in the conceptual model. For example, if you implement an inheritance hierarchy in your conceptual model and then map the CUD stored procedures for the **Parent** (base) and the **Child** (derived) entities, saving the **Child** changes will only call the **Child**'s stored procedures, it will not trigger the **Parent**'s stored procedures calls.

Prerequisites

To complete this walkthrough, you will need:

- A recent version of Visual Studio.
- The School sample database.

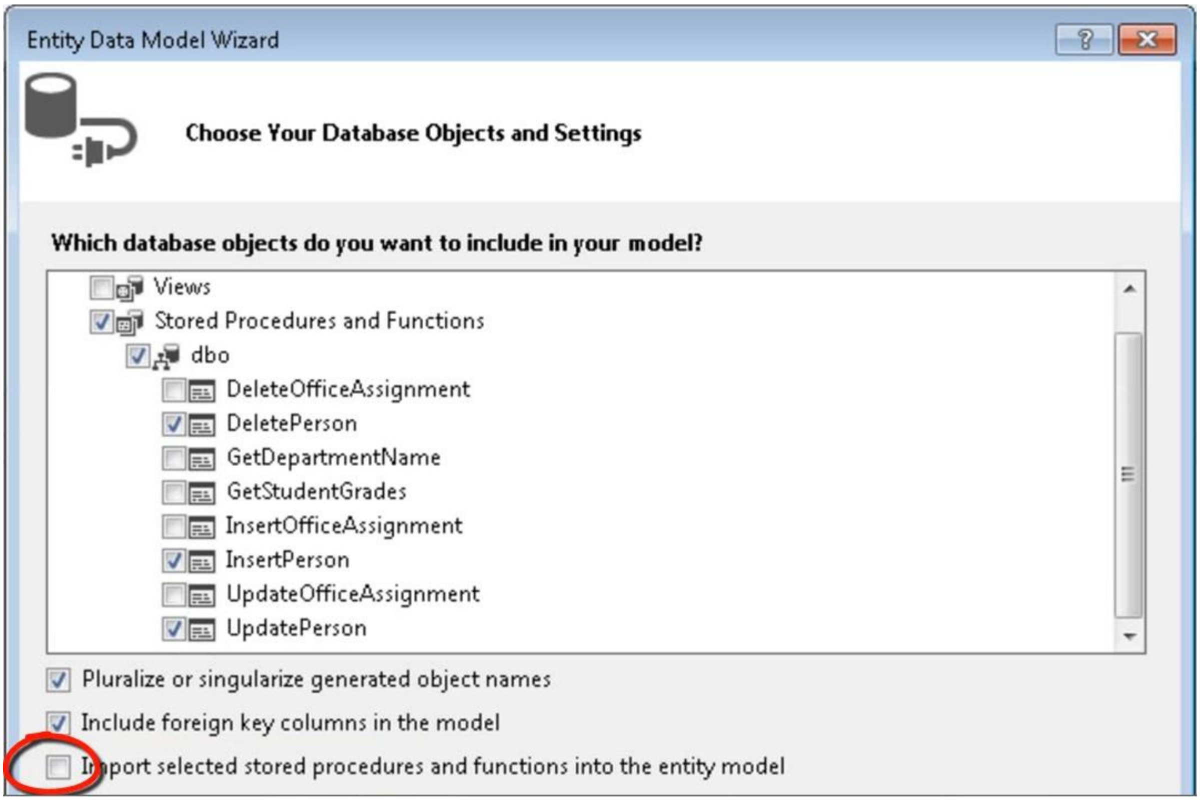
Set up the Project

- Open Visual Studio 2012.
- Select **File-> New -> Project**
- In the left pane, click **Visual C#**, and then select the **Console** template.
- Enter **CUDSProcsSample** as the name.
- Select **OK**.

Create a Model

- Right-click the project name in Solution Explorer, and select **Add -> New Item**.

- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **CUDSProcs.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next**.
- Click **New Connection**. In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**. The Choose Your Data Connection dialog box is updated with your database connection setting.
- In the Choose Your Database Objects dialog box, under the **Tables** node, select the **Person** table.
- Also, select the following stored procedures under the **Stored Procedures and Functions** node: **DeletePerson**, **InsertPerson**, and **UpdatePerson**.
- Starting with Visual Studio 2012 the EF Designer supports bulk import of stored procedures. The **Import selected stored procedures and functions into the entity model** is checked by default. Since in this example we have stored procedures that insert, update, and delete entity types, we do not want to import them and will uncheck this checkbox.

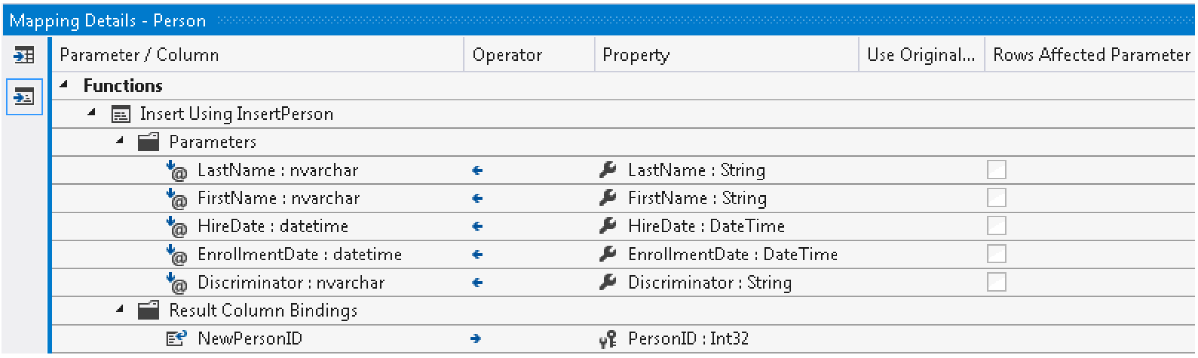


- Click **Finish**. The EF Designer, which provides a design surface for editing your model, is displayed.

Map the Person Entity to Stored Procedures

- Right-click the **Person** entity type and select **Stored Procedure Mapping**.
- The stored procedure mappings appear in the **Mapping Details** window.
- Click **<Select Insert Function>**. The field becomes a drop-down list of the stored procedures in the storage model that can be mapped to entity types in the conceptual model. Select **InsertPerson** from the drop-down list.
- Default mappings between stored procedure parameters and entity properties appear. Note that arrows indicate the mapping direction: Property values are supplied to stored procedure parameters.
- Click **<Add Result Binding>**.

- Type **NewPersonID**, the name of the parameter returned by the **InsertPerson** stored procedure. Make sure not to type leading or trailing spaces.
- Press **Enter**.
- By default, **NewPersonID** is mapped to the entity key **PersonID**. Note that an arrow indicates the direction of the mapping: The value of the result column is supplied to the property.



- Click **<Select Update Function>** and select **UpdatePerson** from the resulting drop-down list.
- Default mappings between stored procedure parameters and entity properties appear.
- Click **<Select Delete Function>** and select **DeletePerson** from the resulting drop-down list.
- Default mappings between stored procedure parameters and entity properties appear.

The insert, update, and delete operations of the **Person** entity type are now mapped to stored procedures.

If you want to enable concurrency checking when updating or deleting an entity with stored procedures, use one of the following options:

- Use an **OUTPUT** parameter to return the number of affected rows from the stored procedure and check the **Rows Affected Parameter** checkbox next to the parameter name. If the value returned is zero when the operation is called, an **OptimisticConcurrencyException** will be thrown.
- Check the **Use Original Value** checkbox next to a property that you want to use for concurrency checking. When an update is attempted, the value of the property that was originally read from the database will be used when writing data back to the database. If the value does not match the value in the database, an **OptimisticConcurrencyException** will be thrown.

Use the Model

Open the **Program.cs** file where the **Main** method is defined. Add the following code into the Main function.

The code creates a new **Person** object, then updates the object, and finally deletes the object.

```
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);
}
```

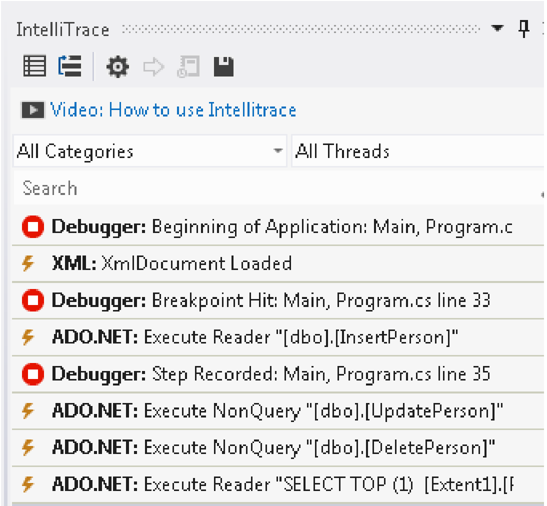
- Compile and run the application. The program produces the following output *

> **NOTE**

> PersonID is auto-generated by the server, so you will most likely see a different number*

```
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.
```

If you are working with the Ultimate version of Visual Studio, you can use Intellitrace with the debugger to see the SQL statements that get executed.


IntelliTrace
Video: How to use Intellitrace
All Categories
All Threads
Search
Debugger: Beginning of Application: Main, Program.c
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].[

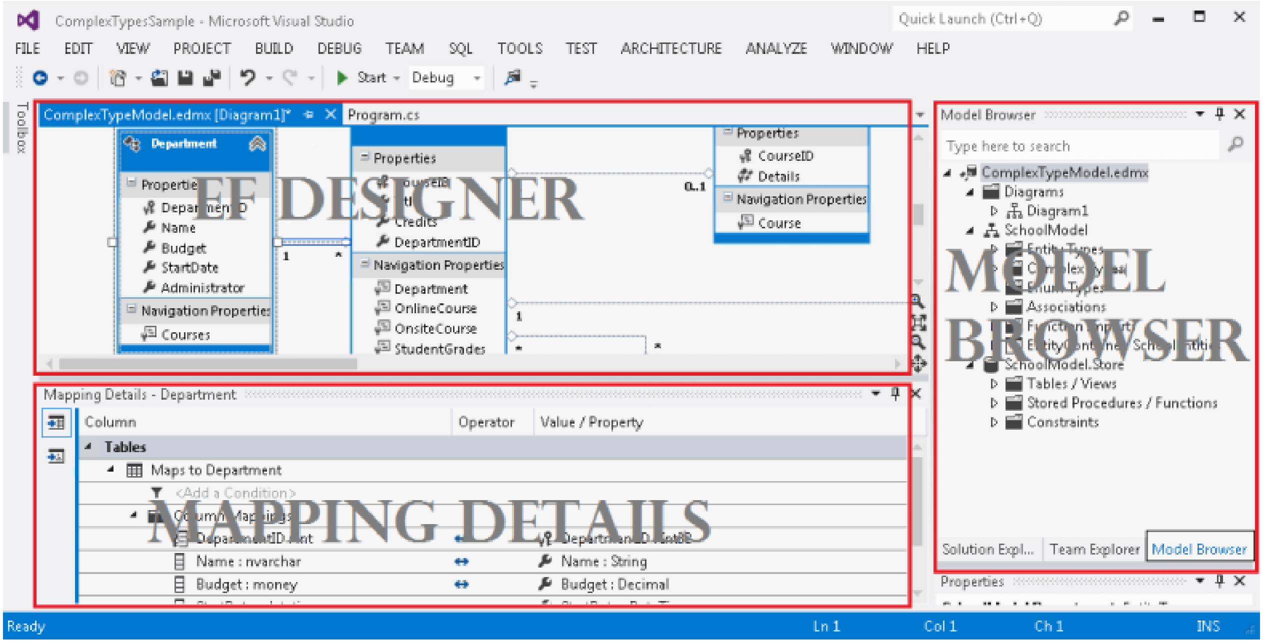
Relationships - EF Designer

9/13/2018 • 5 minutes to read • Edit Online

NOTE

This page provides information about setting up relationships in your model using the EF Designer. For general information about relationships in EF and how to access and manipulate data using relationships, see Relationships & Navigation Properties.

Associations define relationships between entity types in a model. This topic shows how to map associations with the Entity Framework Designer (EF Designer). The following image shows the main windows that are used when working with the EF Designer.



NOTE

When you build the conceptual model, warnings about unmapped entities and associations may appear in the Error List. You can ignore these warnings because after you choose to generate the database from the model, the errors will go away.

Associations Overview

When you design your model using the EF Designer, an .edmx file represents your model. In the .edmx file, an **Association** element defines a relationship between two entity types. An association must specify the entity types that are involved in the relationship and the possible number of entity types at each end of the relationship, which is known as the multiplicity. The multiplicity of an association end can have a value of one (1), zero or one (0..1), or many (*). This information is specified in two child **End** elements.

At run time, entity type instances at one end of an association can be accessed through navigation properties or foreign keys (if you choose to expose foreign keys in your entities). With foreign keys exposed, the relationship between the entities is managed with a **ReferentialConstraint** element (a child element of the **Association** element). It is recommended that you always expose foreign keys for relationships in your entities.

> **NOTE**
>
> In many-to-many (*:*) you cannot add foreign keys to the entities. In a *:* relationship, the association information is managed with an independent object.

For information about CSDL elements (**ReferentialConstraint**, **Association**, etc.) see the CSDL specification.

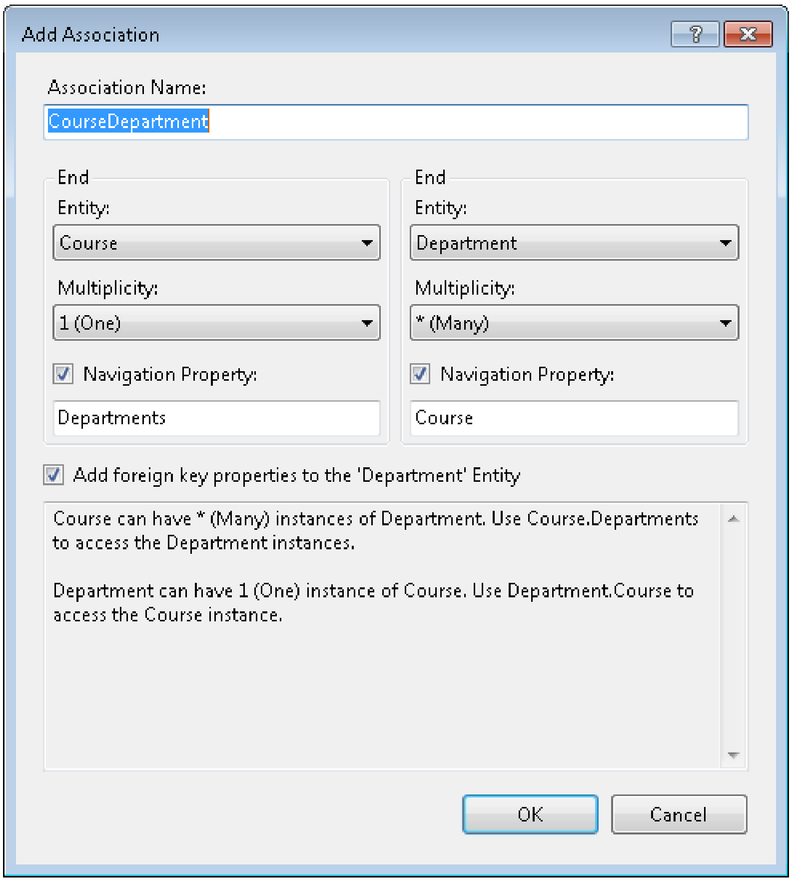
Create and Delete Associations

Creating an association with the EF Designer updates the model content of the .edmx file. After creating an association, you must create the mappings for the association (discussed later in this topic).

> **NOTE**
>
> This section assumes that you already added the entities you wish to create an association between to your model.

To create an association

1. Right-click an empty area of the design surface, point to **Add New**, and select **Association...**.
2. Fill in the settings for the association in the **Add Association** dialog.



> **NOTE**
>
> You can choose to not add navigation properties or foreign key properties to the entities at the ends of the association by clearing the **Navigation Property **and **Add foreign key properties to the <entity type name> Entity **checkboxes. If you add only one navigation property, the association will be traversable in only one direction. If you add no navigation properties, you must choose to add foreign key properties in order to access entities at the ends of the association.

3. Click **OK**.

To delete an association

To delete an association do one of the following:

- Right-click the association on the EF Designer surface and select **Delete**.
- OR -
- Select one or more associations and press the DELETE key.

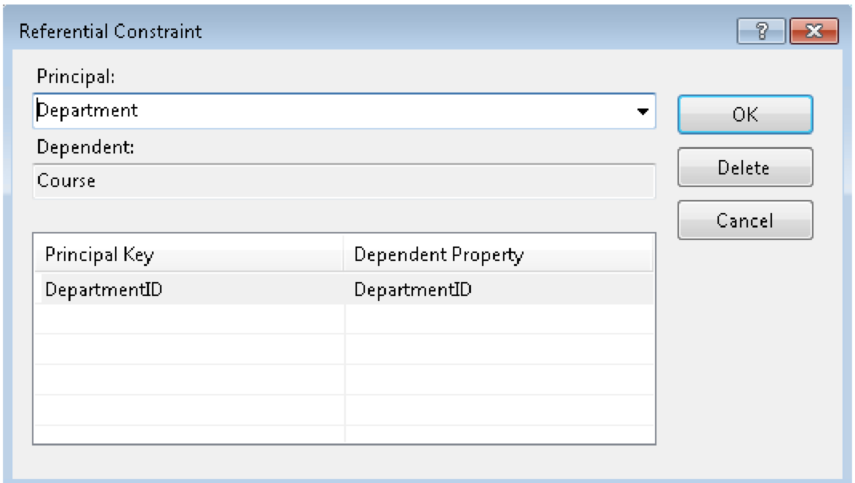
Include Foreign Key Properties in Your Entities (Referential Constraints)

It is recommended that you always expose foreign keys for relationships in your entities. Entity Framework uses a referential constraint to identify that a property acts as the foreign key for a relationship.

If you checked the ***Add foreign key properties to the <entity type name> Entity*** checkbox when creating a relationship, this referential constraint was added for you.

When you use the EF Designer to add or edit a referential constraint, the EF Designer adds or modifies a **ReferentialConstraint** element in the CSDL content of the .edmx file.

- Double-click the association that you want to edit. The **Referential Constraint** dialog box appears.
- From the **Principal** drop-down list, select the principal entity in the referential constraint. The entity's key properties are added to the **Principal Key** list in the dialog box.
- From the **Dependent** drop-down list, select the dependent entity in the referential constraint.
- For each principal key that has a dependent key, select a corresponding dependent key from the drop-down lists in the **Dependent Key** column.



- Click **OK**.

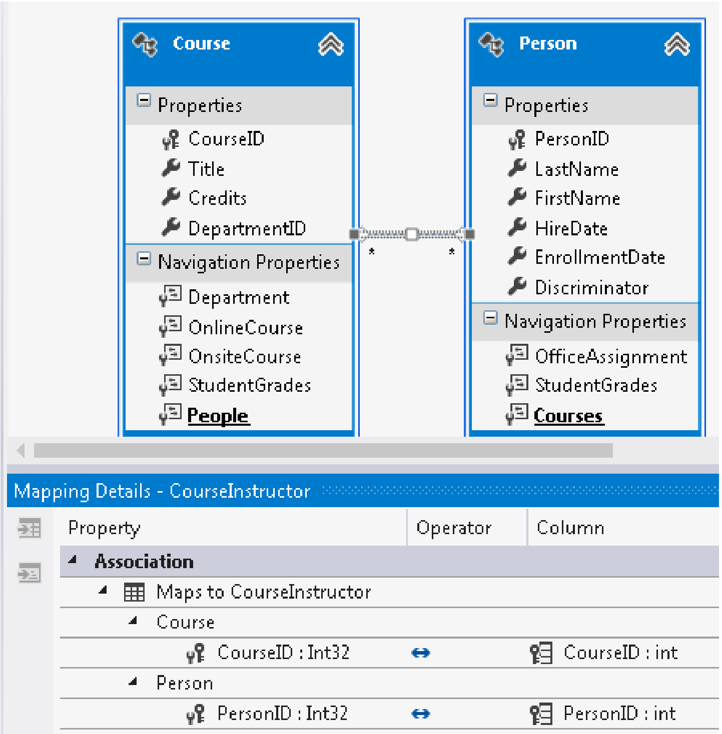
Create and Edit Association Mappings

You can specify how an association maps to the database in the **Mapping Details** window of the EF Designer.

> **NOTE**
>
> You can only map details for the associations that do not have a referential constraint specified. If a referential constraint is specified then a foreign key property is included in the entity and you can use the Mapping Details for the entity to control which column the foreign key maps to.

Create an association mapping

- Right-click an association in the design surface and select **Table Mapping**. This displays the association mapping in the **Mapping Details** window.
- Click **Add a Table or View**. A drop-down list appears that includes all the tables in the storage model.
- Select the table to which the association will map. The **Mapping Details** window displays both ends of the association and the key properties for the entity type at each **End**.
- For each key property, click the **Column** field, and select the column to which the property will map.



Edit an association mapping

- Right-click an association in the design surface and select **Table Mapping**. This displays the association mapping in the **Mapping Details** window.
- Click **Maps to <Table Name>**. A drop-down list appears that includes all the tables in the storage model.
- Select the table to which the association will map. The **Mapping Details** window displays both ends of the association and the key properties for the entity type at each End.
- For each key property, click the **Column** field, and select the column to which the property will map.

Edit and Delete Navigation Properties

Navigation properties are shortcut properties that are used to locate the entities at the ends of an association in a model. Navigation properties can be created when you create an association between two entity types.

To edit navigation properties

- Select a navigation property on the EF Designer surface. Information about the navigation property is displayed in the Visual Studio **Properties** window.
- Change the property settings in the **Properties** window.

To delete navigation properties

- If foreign keys are not exposed on entity types in the conceptual model, deleting a navigation property may make the corresponding association traversable in only one direction or not traversable at all.
- Right-click a navigation property on the EF Designer surface and select **Delete**.

Multiple Diagrams per Model

9/18/2018 • 4 minutes to read • Edit Online

> **NOTE**
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

This video and page shows how to split a model into multiple diagrams using the Entity Framework Designer (EF Designer). You might want to use this feature when your model becomes too large to view or edit.

In earlier versions of the EF Designer you could only have one diagram per the EDMX file. Starting with Visual Studio 2012, you can use the EF Designer to split your EDMX file into multiple diagrams.

Watch the video

This video shows how to split a model into multiple diagrams using the Entity Framework Designer (EF Designer). You might want to use this feature when your model becomes too large to view or edit.

Presented By: Julia Kornich

Video: WMV | MP4 | WMV (ZIP)

EF Designer Overview

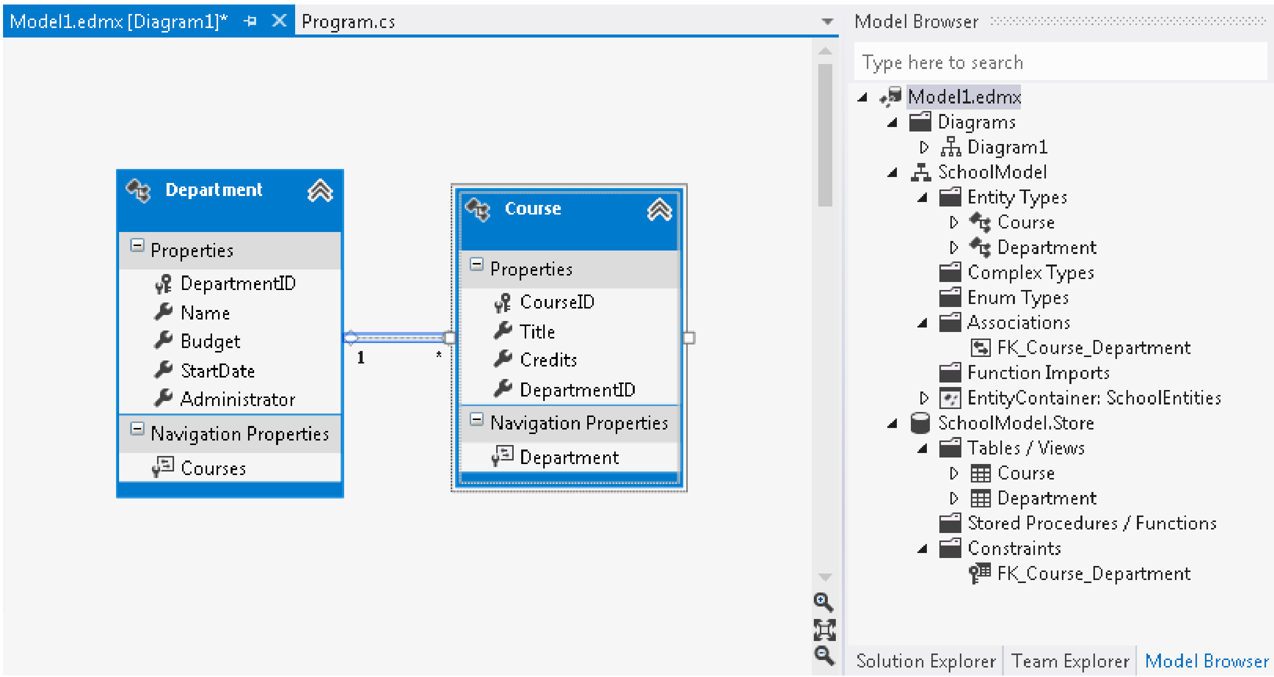
When you create a model using the EF Designer's Entity Data Model Wizard, an .edmx file is created and added to your solution. This file defines the shape of your entities and how they map to the database.

The EF Designer consists of the following components:

- A visual design surface for editing the model. You can create, modify, or delete entities and associations.
- A **Model Browser** window that provides tree views of the model. The entities and their associations are located under the *[ModelName]* folder. The database tables and constraints are located under the *[ModelName]*.Store folder.
- A **Mapping Details** window for viewing and editing mappings. You can map entity types or associations to database tables, columns, and stored procedures.

The visual design surface window is automatically opened when the Entity Data Model Wizard finishes. If the Model Browser is not visible, right-click the main design surface and select **Model Browser**.

The following screenshot shows an .edmx file opened in the EF Designer. The screenshot shows the visual design surface (to the left) and the **Model Browser** window (to the right).



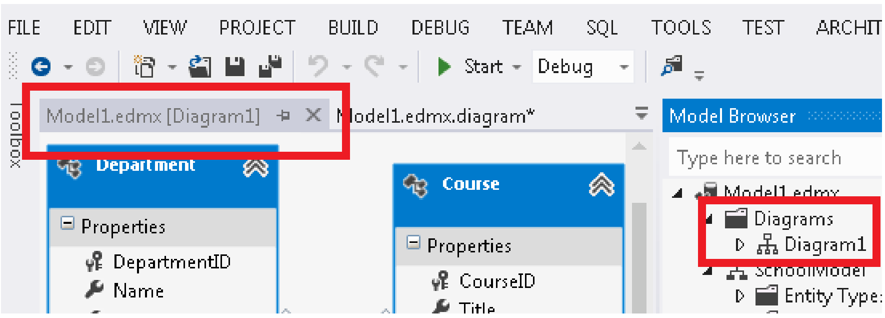
To undo an operation done in the EF Designer, click Ctrl-Z.

Working with Diagrams

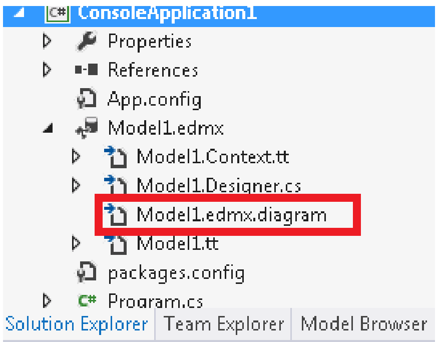
By default the EF Designer creates one diagram called Diagram1. If you have a diagram with a large number of entities and associations, you will most like want to split them up logically. Starting with Visual Studio 2012, you can view your conceptual model in multiple diagrams.

As you add new diagrams, they appear under the Diagrams folder in the Model Browser window. To rename a diagram: select the diagram in the Model Browser window, click once on the name, and type the new name. You can also right-click the diagram name and select **Rename**.

The diagram name is displayed next to the .edmx file name, in the Visual Studio editor. For example Model1.edmx[Diagram1].



The diagrams content (shape and color of entities and associations) is stored in the .edmx.diagram file. To view this file, select Solution Explorer and unfold the .edmx file.



You should not edit the .edmx.diagram file manually, the content of this file maybe overwritten by the EF Designer.

Splitting Entities and Associations into a New Diagram

You can select entities on the existing diagram (hold Shift to select multiple entities). Click the right mouse button and select **Move to new Diagram**. The new diagram is created and the selected entities and their associations are moved to the diagram.

Alternatively, you can right-click the Diagrams folder in Model Browser and select **Add new Diagram.** You can then drag and drop entities from under the Entity Types folder in Model Browser onto the design surface.

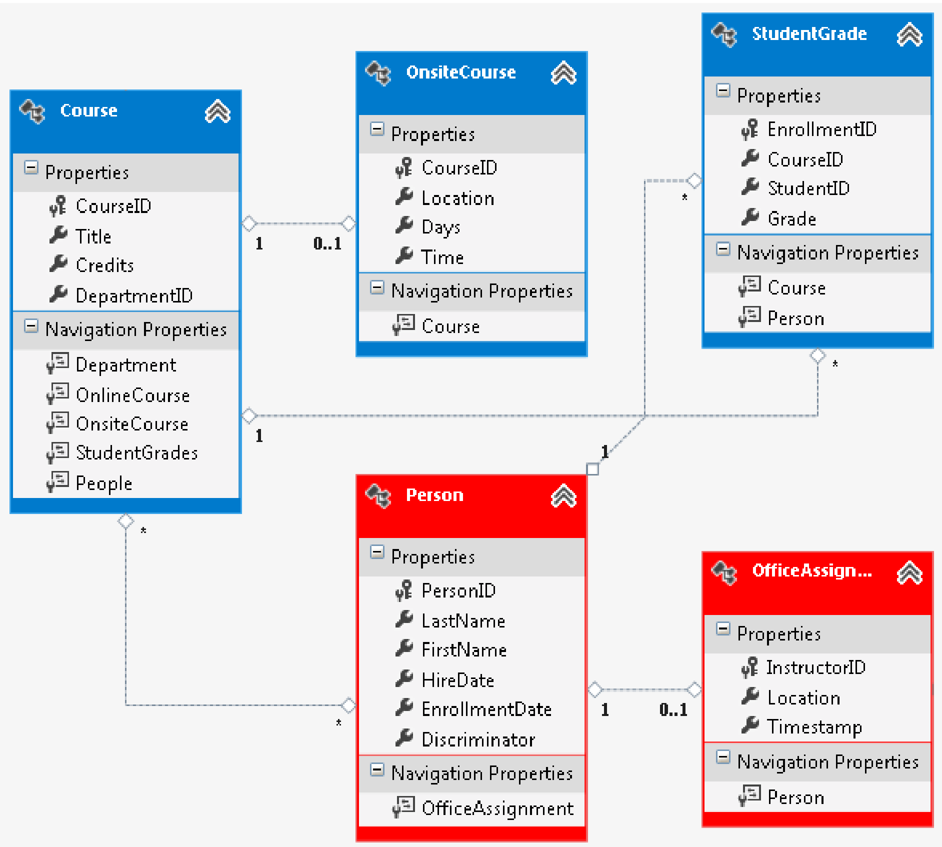
You can also cut or copy entities (using Ctrl-X or Ctrl-C keys) from one diagram and paste (using Ctrl-V key) on the other. If the diagram into which you are pasting an entity already contains an entity with the same name, a new entity will be created and added to the model. For example: Diagram2 contains the Department entity. Then, you paste another Department on Diagram2. The Department1 entity is created and added to the conceptual model.

To include related entities in a diagram, rick-click the entity and select **Include Related**. This will make a copy of the related entities and associations in the specified diagram.

Changing the Color of Entities

In addition to splitting a model into multiple diagrams, you can also change colors of your entities.

To change the color, select an entity (or multiple entities) on the design surface. Then, click the right mouse button and select **Properties**. In the Properties window, select the **Fill Color** property. Specify the color using either a valid color name (for example, Red) or a valid RGB (for example, 255, 128, 128).



Summary

In this topic we looked at how to split a model into multiple diagrams and also how to specify a different color for an entity using the Entity Framework Designer.

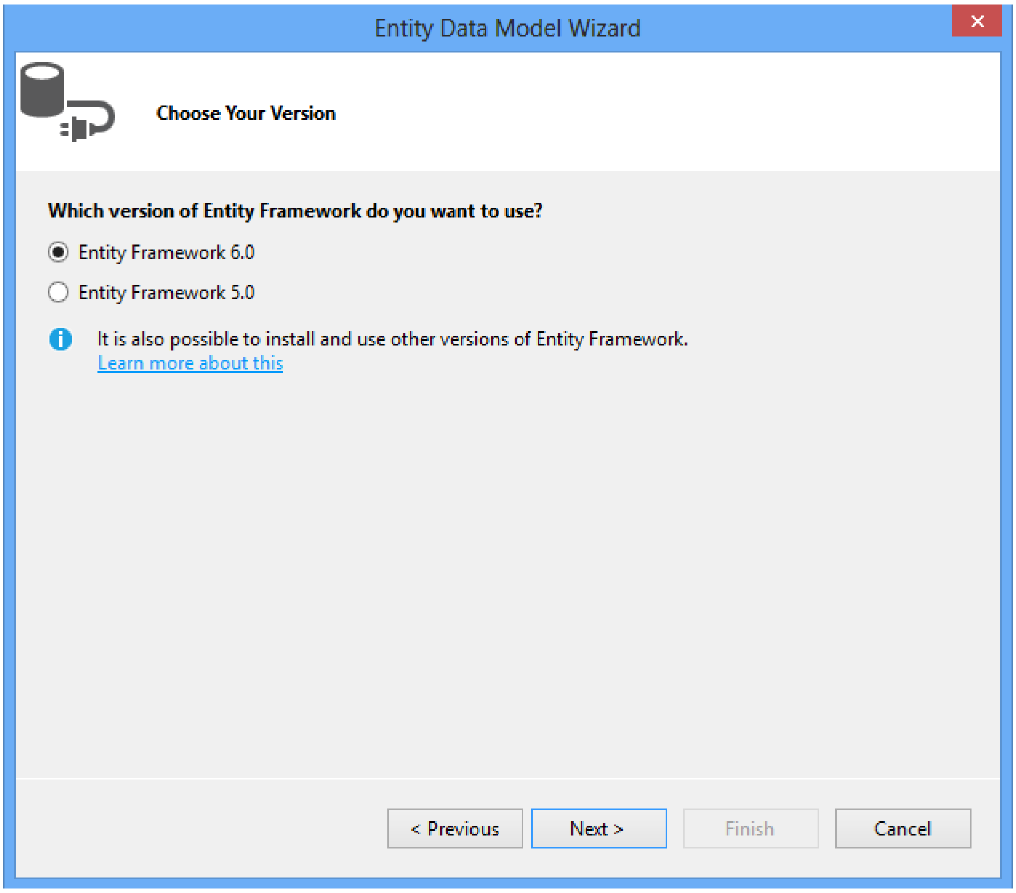
Selecting Entity Framework Runtime Version for EF Designer Models

9/13/2018 • 2 minutes to read • Edit Online

NOTE

EF6 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

Starting with EF6 the following screen was added to the EF Designer to allow you to select the version of the runtime you wish to target when creating a model. The screen will appear when the latest version of Entity Framework is not already installed in the project. If the latest version is already installed it will just be used by default.



Targeting EF6.x

You can choose EF6 from the 'Choose Your Version' screen to add the EF6 runtime to your project. Once you've added EF6, you'll stop seeing this screen in the current project.

EF6 will be disabled if you already have an older version of EF installed (since you can't target multiple versions of the runtime from the same project). If EF6 option is not enabled here, follow these steps to upgrade your project to EF6:

1. Right-click on your project in Solution Explorer and select **Manage NuGet Packages...**
2. Select **Updates**

3. Select **EntityFramework** (make sure it is going to update it to the version you want)
4. Click **Update**

Targeting EF5.x

You can choose EF5 from the 'Choose Your Version' screen to add the EF5 runtime to your project. Once you've added EF5, you’ll still see the screen with the EF6 option disabled.

If you have an EF4.x version of the runtime already installed then you will see that version of EF listed in the screen rather than EF5. In this situation you can upgrade to EF5 using the following steps:

1. Select **Tools -> Library Package Manager -> Package Manager Console**
2. Run **Install-Package EntityFramework -version 5.0.0**

Targeting EF4.x

You can install the EF4.x runtime to your project using the following steps:

1. Select **Tools -> Library Package Manager -> Package Manager Console**
2. Run **Install-Package EntityFramework -version 4.3.0**

Designer Code Generation Templates

9/18/2018 • 7 minutes to read • Edit Online

When you create a model using the Entity Framework Designer your classes and derived context are automatically generated for you. In addition to the default code generation we also provide a number of templates that can be used to customize the code that gets generated. These templates are provided as T4 Text Templates, allowing you to customize the templates if needed.

The code that gets generated by default depends on which version of Visual Studio you create your model in:

- Models created in Visual Studio 2012 & 2013 will generate simple POCO entity classes and a context that derives from the simplified DbContext.
- Models created in Visual Studio 2010 will generate entity classes that derive from EntityObject and a context that derives from ObjectContext.

> **NOTE**
>
> We recommend switching to the DbContext Generator template once you've added your model.

This page covers the available templates and then provides instructions for adding a template to your model.

Available Templates

The following templates are provided by the Entity Framework team:

DbContext Generator

This template will generate simple POCO entity classes and a context that derives from DbContext using EF6. This is the recommended template unless you have a reason to use one of the other templates listed below. It is also the code generation template you get by default if you are using recent versions of Visual Studio (Visual Studio 2013 onwards): When you create a new model this template is used by default and the T4 files (.tt) are nested under your .edmx file.

Older versions of Visual Studio

- **Visual Studio 2012:** To get the **EF 6.x DbContextGenerator** templates you will need to install the latest **Entity Framework Tools for Visual Studio** - see the Get Entity Framework page for more information.
- **Visual Studio 2010:** The **EF 6.x DbContextGenerator** templates are not available for Visual Studio 2010.

DbContext Generator for EF 5.x

If you are using an older version of the EntityFramework NuGet package (one with a major version of 5) you will need to use the **EF 5.x DbContext Generator** template.

If you are using Visual Studio 2013 or 2012 this template is already installed.

If you are using Visual Studio 2010 you will need to select the **Online** tab when adding the template to download it from Visual Studio Gallery. Alternatively you can install the template directly from Visual Studio Gallery ahead of time. Because the templates are included in later versions of Visual Studio the versions on the gallery can only be installed on Visual Studio 2010.

- EF 5.x DbContext Generator for C#
- EF 5.x DbContext Generator for C# Web Sites
- EF 5.x DbContext Generator for VB.NET

- EF 5.x DbContext Generator for VB.NET Web Sites

DbContext Generator for EF 4.x

If you are using an older version of the EntityFramework NuGet package (one with a major version of 4) you will need to use the **EF 4.x DbContext Generator** template. This can be found in the **Online** tab when adding the template, or you can install the template directly from Visual Studio Gallery ahead of time.

- EF 4.x DbContext Generator for C#
- EF 4.x DbContext Generator for C# Web Sites
- EF 4.x DbContext Generator for VB.NET
- EF 4.x DbContext Generator for VB.NET Web Sites

EntityObject Generator

This template will generate entity classes that derive from EntityObject and a context that derives from ObjectContext.

> **NOTE**
>
> Consider using the DbContext Generator

The DbContext Generator is now the recommended template for new applications. The DbContext Generator takes advantage of the simpler DbContext API. The EntityObject Generator continues to be available to support existing applications.

Visual Studio 2010, 2012 & 2013

You will need to select the **Online** tab when adding the template to download it from Visual Studio Gallery. Alternatively you can install the template directly from Visual Studio Gallery ahead of time.

- EF 6.x EntityObject Generator for C#
- EF 6.x EntityObject Generator for C# Web Sites
- EF 6.x EntityObject Generator for VB.NET
- EF 6.x EntityObject Generator for VB.NET Web Sites

EntityObject Generator for EF 5.x

If you are using Visual Studio 2012 or 2013 you will need to select the **Online** tab when adding the template to download it from Visual Studio Gallery. Alternatively you can install the template directly from Visual Studio Gallery ahead of time. Because the templates are included in Visual Studio 2010 the versions on the gallery can only be installed on Visual Studio 2012 & 2013.

- EF 5.x EntityObject Generator for C#
- EF 5.x EntityObject Generator for C# Web Sites
- EF 5.x EntityObject Generator for VB.NET
- EF 5.x EntityObject Generator for VB.NET Web Sites

If you just want ObjectContext code generation without needing to edit the template you can revert to EntityObject code generation.

If you are using Visual Studio 2010 this template is already installed. If you create a new model in Visual Studio 2010 this template is used by default but the .tt files are not included in your project. If you want to customize the template you will need to add it to your project.

Self-Tracking Entities (STE) Generator

This template will generate Self-Tracking Entity classes and a context that derives from ObjectContext. In an EF application, a context is responsible for tracking changes in the entities. However, in N-Tier scenarios, the context might not be available on the tier that modifies the entities. Self-tracking entities help you track changes in any tier. For more information, see Self-Tracking Entities.

> **NOTE**
>
> STE Template Not Recommended

We no longer recommend using the STE template in new applications, it continues to be available to support existing applications. Visit the disconnected entities article for other options we recommend for N-Tier scenarios.

> **NOTE**
>
> There is no EF 6.x version of the STE template.

> **NOTE**
>
> There is no Visual Studio 2013 version of the STE template.

Visual Studio 2012

If you are using Visual Studio 2012 you will need to select the **Online** tab when adding the template to download it from Visual Studio Gallery. Alternatively you can install the template directly from Visual Studio Gallery ahead of time. Because the templates are included in Visual Studio 2010 the versions on the gallery can only be installed on Visual Studio 2012.

- EF 5.x STE Generator for C#
- EF 5.x STE Generator for C# Web Sites
- EF 5.x STE Generator for VB.NET
- EF 5.x STE Generator for VB.NET Web Sites

Visual Studio 2010**

If you are using Visual Studio 2010 this template is already installed.

POCO Entity Generator

This template will generate POCO entity classes and a context that derives from ObjectContext

> **NOTE**
>
> Consider using the DbContext Generator

The DbContext Generator is now the recommended template for generating POCO classes in new applications. The DbContext Generator takes advantage of the new DbContext API and can generate simpler POCO classes. The POCO Entity Generator continues to be available to support existing applications.

> **NOTE**
>
> There is no EF 5.x or EF 6.x version of the STE template.

> **NOTE**
>
> There is no Visual Studio 2013 version of the POCO template.

Visual Studio 2012 & Visual Studio 2010

You will need to select the **Online** tab when adding the template to download it from Visual Studio Gallery. Alternatively you can install the template directly from Visual Studio Gallery ahead of time.

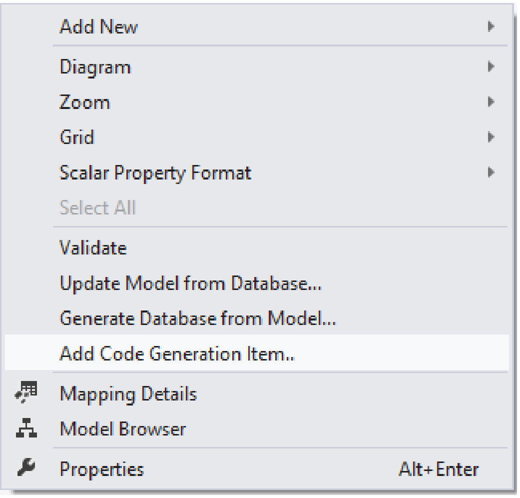
- EF 4.x POCO Generator for C#
- EF 4.x POCO Generator for C# Web Sites
- EF 4.x POCO Generator for VB.NET
- EF 4.x POCO Generator for VB.NET Web Sites

What are the "Web Sites" Templates

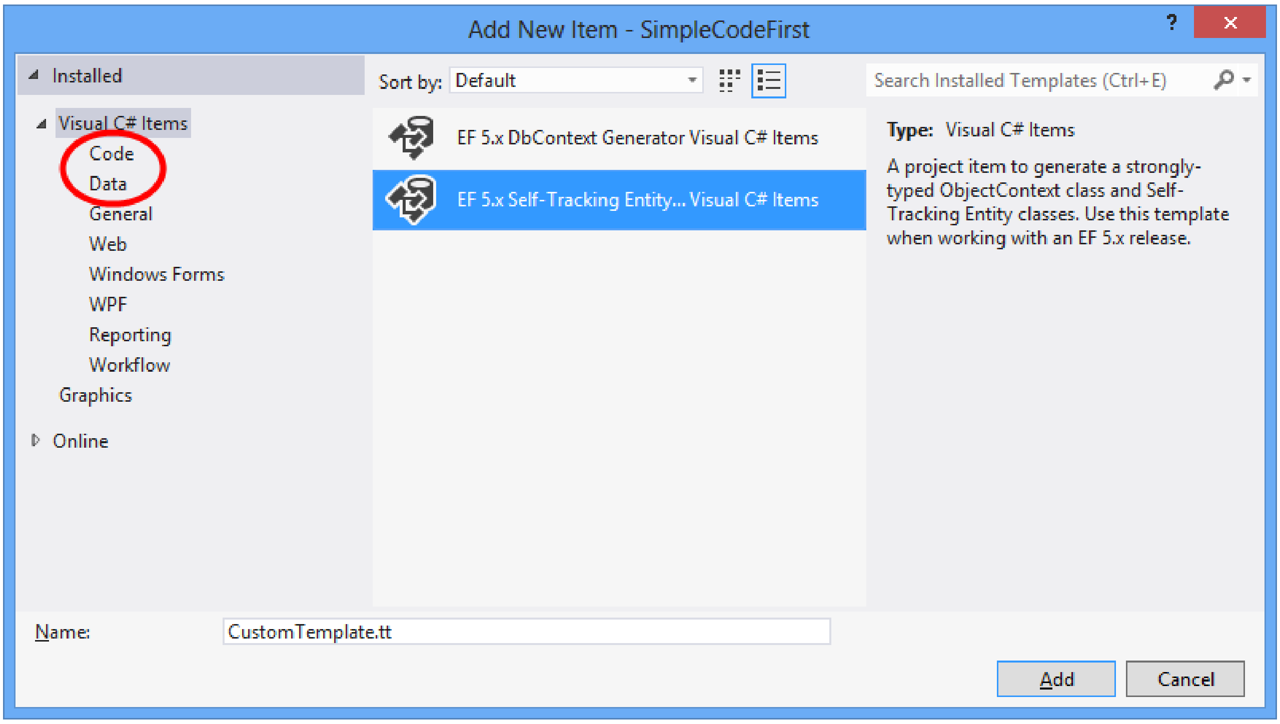
The "Web Sites" templates (for example, **EF 5.x DbContext Generator for C# Web Sites**) are for use in Web Site projects created via **File -> New -> Web Site...**. These are different from Web Applications, created via **File -> New -> Project...**, which use the standard templates. We provide separate templates because the item template system in Visual Studio requires them.

Using a Template

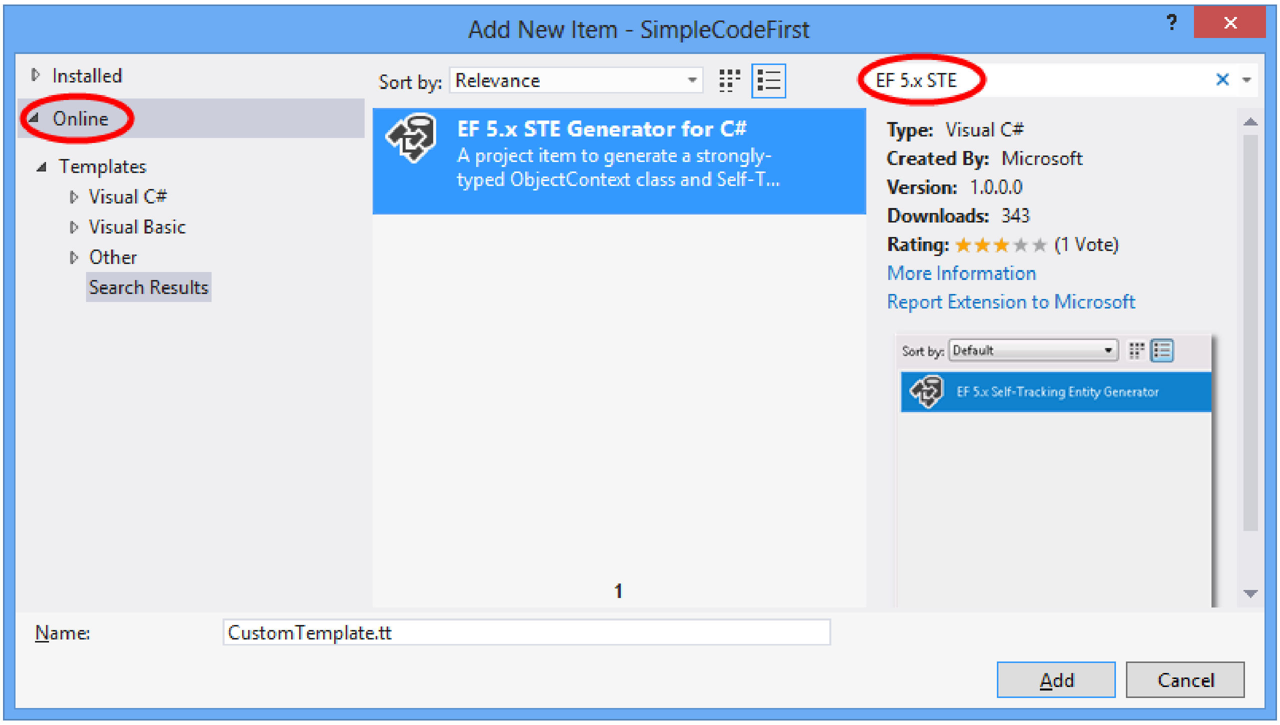
To start using a code generation template, right-click an empty spot on the design surface in the EF Designer and select **Add Code Generation Item...**.



If you've already installed the template you want to use (or it was included in Visual Studio), then it will be available under either the **Code** or **Data** section from the left menu.



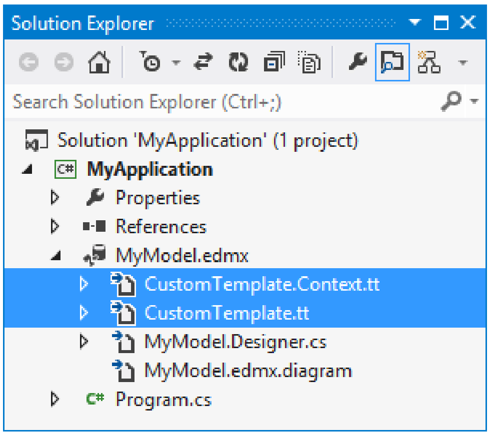
If you don't already have the template installed, select **Online** from the left menu and search for the template you want.



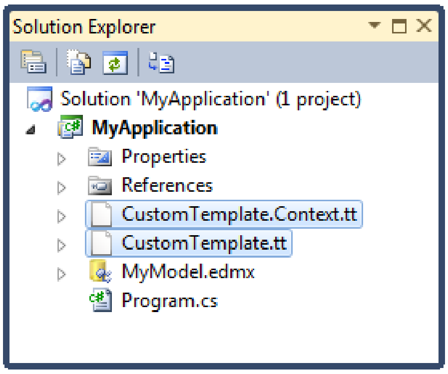
If you are using Visual Studio 2012, the new .tt files will be nested under the .edmx file.*

NOTE

For models created in Visual Studio 2012 you will need to delete the templates used for default code generation, otherwise you will have duplicate classes and context generated. The default files are **<model name>.tt** and **<model name>.context.tt**.



If you are using Visual Studio 2010, the tt files are added directly to your project.



Reverting to ObjectContext in Entity Framework Designer

9/13/2018 • 2 minutes to read • Edit Online

With previous version of Entity Framework a model created with the EF Designer would generate a context that derived from ObjectContext and entity classes that derived from EntityObject.

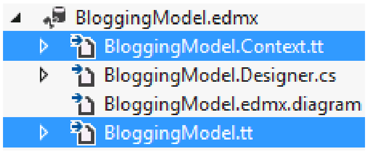
Starting with EF4.1 we recommended swapping to a code generation template that generates a context deriving from DbContext and POCO entity classes.

In Visual Studio 2012 you get DbContext code generated by default for all new models created with the EF Designer. Existing models will continue to generate ObjectContext based code unless you decide to swap to the DbContext based code generator.

Reverting Back to ObjectContext Code Generation

1. Disable DbContext Code Generation

Generation of the derived DbContext and POCO classes is handled by two .tt files in you project, if you expand the .edmx file in solution explorer you will see these files. Delete both of these files from your project.

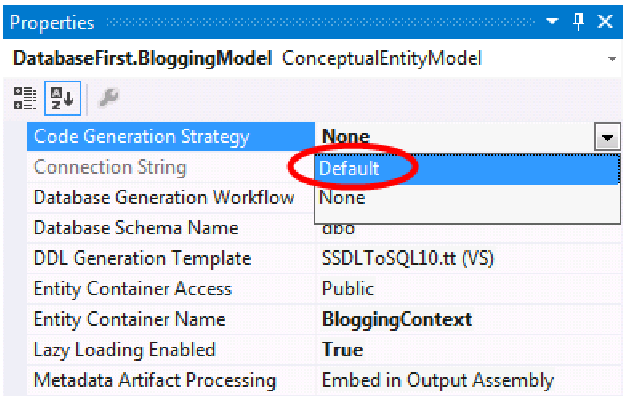


If you are using VB.NET you will need to select the **Show All Files** button to see the nested files.

2. Re-Enable ObjectContext Code Generation

Open you model in the EF Designer, right-click on a blank section of the design surface and select **Properties**.

In the Properties window change the **Code Generation Strategy** from **None** to **Default**.



CSDL Specification

10/1/2018 • 61 minutes to read • Edit Online

Conceptual schema definition language (CSDL) is an XML-based language that describes the entities, relationships, and functions that make up a conceptual model of a data-driven application. This conceptual model can be used by the Entity Framework or WCF Data Services. The metadata that is described with CSDL is used by the Entity Framework to map entities and relationships that are defined in a conceptual model to a data source. For more information, see SSDL Specification and MSL Specification.

CSDL is the Entity Framework's implementation of the Entity Data Model.

In an Entity Framework application, conceptual model metadata is loaded from a .csdl file (written in CSDL) into an instance of the System.Data.Metadata.Edm.EdmItemCollection and is accessible by using methods in the System.Data.Metadata.Edm.MetadataWorkspace class. Entity Framework uses conceptual model metadata to translate queries against the conceptual model to data source-specific commands.

The EF Designer stores conceptual model information in an .edmx file at design time. At build time, the EF Designer uses information in an .edmx file to create the .csdl file that is needed by Entity Framework at runtime.

Versions of CSDL are differentiated by XML namespaces.

CSDL VERSION	XML NAMESPACE
CSDL v1	http://schemas.microsoft.com/ado/2006/04/edm
CSDL v2	http://schemas.microsoft.com/ado/2008/09/edm
CSDL v3	http://schemas.microsoft.com/ado/2009/11/edm

Association Element (CSDL)

An **Association** element defines a relationship between two entity types. An association must specify the entity types that are involved in the relationship and the possible number of entity types at each end of the relationship, which is known as the multiplicity. The multiplicity of an association end can have a value of one (1), zero or one (0..1), or many (*). This information is specified in two child End elements.

Entity type instances at one end of an association can be accessed through navigation properties or foreign keys, if they are exposed on an entity type.

In an application, an instance of an association represents a specific association between instances of entity types. Association instances are logically grouped in an association set.

An **Association** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- End (exactly 2 elements)
- ReferentialConstraint (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **Association** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the association.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Association** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that defines the **CustomerOrders** association when foreign keys have not been exposed on the **Customer** and **Order** entity types. The **Multiplicity** values for each **End** of the association indicate that many **Orders** can be associated with a **Customer**, but only one **Customer** can be associated with an **Order**. Additionally, the **OnDelete** element indicates that all **Orders** that are related to a particular **Customer** and have been loaded into the ObjectContext will be deleted if the **Customer** is deleted.

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" >
        <OnDelete Action="Cascade" />
  </End>
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

The following example shows an **Association** element that defines the **CustomerOrders** association when foreign keys have been exposed on the **Customer** and **Order** entity types. With foreign keys exposed, the relationship between the entities is managed with a **ReferentialConstraint** element. A corresponding AssociationSetMapping element is not necessary to map this association to the data source.

```
<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>
```

AssociationSet Element (CSDL)

The **AssociationSet** element in conceptual schema definition language (CSDL) is a logical container for association instances of the same type. An association set provides a definition for grouping association instances so that they can be mapped to a data source.

The **AssociationSet** element can have the following child elements (in the order listed):

- Documentation (zero or one elements allowed)
- End (exactly two elements required)
- Annotation elements (zero or more elements allowed)

The **Association** attribute specifies the type of association that an association set contains. The entity sets that make up the ends of an association set are specified with exactly two child **End** elements.

Applicable Attributes

The table below describes the attributes that can be applied to the **AssociationSet** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity set. The value of the **Name** attribute cannot be the same as the value of the **Association** attribute.
Association	Yes	The fully-qualified name of the association that the association set contains instances of. The association must be in the same namespace as the association set.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **AssociationSet** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element with two **AssociationSet** elements:

```
<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>
```

CollectionType Element (CSDL)

The **CollectionType** element in conceptual schema definition language (CSDL) specifies that a function parameter or function return type is a collection. The **CollectionType** element can be a child of the Parameter element or the ReturnType (Function) element. The type of collection can be specified by using either the **Type** attribute or one of the following child elements:

- **CollectionType**
- ReferenceType
- RowType
- TypeRef

> **NOTE**
> A model will not validate if the type of a collection is specified with both the **Type** attribute and a child element.

Applicable Attributes

The following table describes the attributes that can be applied to the **CollectionType** element. Note that the **DefaultValue**, **MaxLength**, **FixedLength**, **Precision**, **Scale**, **Unicode**, and **Collation** attributes are only applicable to collections of **EDMSimpleTypes**.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	No	The type of the collection.
Nullable	No	**True** (the default value) or **False** depending on whether the property can have a null value. [!NOTE]
> In the CSDL v1, a complex type property must have `Nullable="False"` .		
DefaultValue	No	The default value of the property.
MaxLength	No	The maximum length of the property value.
FixedLength	No	**True** or **False** depending on whether the property value will be stored as a fixed length string.
Precision	No	The precision of the property value.
Scale	No	The scale of the property value.

ATTRIBUTE NAME	IS REQUIRED	VALUE
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server)
Unicode	No	**True** or **False** depending on whether the property value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **CollectionType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a model-defined function that that uses a **CollectionType** element to specify that the function returns a collection of **Person** entity types (as specified with the **ElementType** attribute).

```
<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>
```

The following example shows a model-defined function that uses a **CollectionType** element to specify that the function returns a collection of rows (as specified in the **RowType** element).

```
<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>
```

The following example shows a model-defined function that uses the **CollectionType** element to specify that the function accepts as a parameter a collection of **Department** entity types.

```
<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>
```

ComplexType Element (CSDL)

A **ComplexType** element defines a data structure composed of **EdmSimpleType** properties or other complex types. A complex type can be a property of an entity type or another complex type. A complex type is similar to an entity type in that a complex type defines data. However, there are some key differences between complex types and entity types:

- Complex types do not have identities (or keys) and therefore cannot exist independently. Complex types can only exist as properties of entity types or other complex types.
- Complex types cannot participate in associations. Neither end of an association can be a complex type, and therefore navigation properties cannot be defined for complex types.
- A complex type property cannot have a null value, though the scalar properties of a complex type may each be set to null.

A **ComplexType** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Property (zero or more elements)
- Annotation elements (zero or more elements)

The table below describes the attributes that can be applied to the **ComplexType** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the complex type. The name of a complex type cannot be the same as the name of another complex type, entity type, or association that is within the scope of the model.
BaseType	No	The name of another complex type that is the base type of the complex type that is being defined. [!NOTE]
> This attribute is not applicable in CSDL v1. Inheritance for complex types is not supported in that version.		
Abstract	No	**True** or **False** (the default value) depending on whether the complex type is an abstract type. [!NOTE]
> This attribute is not applicable in CSDL v1. Complex types in that version cannot be abstract types.		

NOTE

Any number of annotation attributes (custom XML attributes) may be applied to the **ComplexType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a complex type, **Address**, with the **EdmSimpleType** properties **StreetAddress**, **City**, **StateOrProvince**, **Country**, and **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>
```

To define the complex type **Address** (above) as a property of an entity type, you must declare the property type in the entity type definition. The following example shows the **Address** property as a complex type on an entity type (**Publisher**):

```
<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>
```

DefiningExpression Element (CSDL)

The **DefiningExpression** element in conceptual schema definition language (CSDL) contains an Entity SQL expression that defines a function in the conceptual model.

> **NOTE**
>
> For validation purposes, a **DefiningExpression** element can contain arbitrary content. However, Entity Framework will throw an exception at runtime if a **DefiningExpression** element does not contain valid Entity SQL.

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **DefiningExpression** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example uses a **DefiningExpression** element to define a function that returns the number of years since a book was published. The content of the **DefiningExpression** element is written in 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>
```

Dependent Element (CSDL)

The **Dependent** element in conceptual schema definition language (CSDL) is a child element to the ReferentialConstraint element and defines the dependent end of a referential constraint. A **ReferentialConstraint** element defines functionality that is similar to a referential integrity constraint in a relational database. In the same way that a column (or columns) from a database table can reference the primary key of another table, a property (or properties) of an entity type can reference the entity key of another entity type. The entity type that is referenced is called the *principal end* of the constraint. The entity type that references the principal end is called the *dependent end* of the constraint. **PropertyRef** elements are used to specify which keys reference the principal end.

The **Dependent** element can have the following child elements (in the order listed):

- PropertyRef (one or more elements)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **Dependent** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Role	Yes	The name of the entity type on the dependent end of the association.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Dependent** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **ReferentialConstraint** element being used as part of the definition of the **PublishedBy** association. The **PublisherId** property of the **Book** entity type makes up the dependent end of the referential constraint.

```
<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>
```

Documentation Element (CSDL)

The **Documentation** element in conceptual schema definition language (CSDL) can be used to provide information about an object that is defined in a parent element. In an .edmx file, when the **Documentation** element is a child of an element that appears as an object on the design surface of the EF Designer (such as an entity, association, or property), the contents of the **Documentation** element will appear in the Visual Studio **Properties** window for the object.

The **Documentation** element can have the following child elements (in the order listed):

- **Summary**: A brief description of the parent element. (zero or one element)

- **LongDescription**: An extensive description of the parent element. (zero or one element)
- Annotation elements. (zero or more elements)

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **Documentation** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows the **Documentation** element as a child element of an EntityType element. If the snippet below were in the CSDL content of an .edmx file, the contents of the **Summary** and **LongDescription** elements would appear in the Visual Studio **Properties** window when you click on the `Customer` entity type.

```
<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>
```

End Element (CSDL)

The **End** element in conceptual schema definition language (CSDL) can be a child of the Association element or the AssociationSet element. In each case, the role of the **End** element is different and the applicable attributes are different.

End Element as a Child of the Association Element

An **End** element (as a child of the **Association** element) identifies the entity type on one end of an association and the number of entity type instances that can exist at that end of an association. Association ends are defined as part of an association; an association must have exactly two association ends. Entity type instances at one end of an association can be accessed through navigation properties or foreign keys if they are exposed on an entity type.

An **End** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- OnDelete (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The following table describes the attributes that can be applied to the **End** element when it is the child of an **Association** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	Yes	The name of the entity type at one end of the association.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Role	No	A name for the association end. If no name is provided, the name of the entity type on the association end will be used.
Multiplicity	Yes	**1**, **0..1**, or ***** depending on the number of entity type instances that can be at the end of the association. **1** indicates that exactly one entity type instance exists at the association end. **0..1** indicates that zero or one entity type instances exist at the association end. ***** indicates that zero, one, or more entity type instances exist at the association end.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **End** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that defines the **CustomerOrders** association. The **Multiplicity** values for each **End** of the association indicate that many **Orders** can be associated with a **Customer**, but only one **Customer** can be associated with an **Order**. Additionally, the **OnDelete** element indicates that all **Orders** that are related to a particular **Customer** and that have been loaded into the ObjectContext will be deleted if the **Customer** is deleted.

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" />
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*">
        <OnDelete Action="Cascade" />
  </End>
</Association>
```

End Element as a Child of the AssociationSet Element

The **End** element specifies one end of an association set. The **AssociationSet** element must contain two **End** elements. The information contained in an **End** element is used in mapping an association set to a data source.

An **End** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Annotation elements (zero or more elements)

> **NOTE**
> Annotation elements must appear after all other child elements. Annotation elements are only allowed in CSDL v2 and later.

Applicable Attributes

The following table describes the attributes that can be applied to the **End** element when it is the child of an **AssociationSet** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
EntitySet	Yes	The name of the **EntitySet** element that defines one end of the parent **AssociationSet** element. The **EntitySet** element must be defined in the same entity container as the parent **AssociationSet** element.
Role	No	The name of the association set end. If the **Role** attribute is not used, the name of the association set end will be the name of the entity set.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **End** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element with two **AssociationSet** elements, each with two **End** elements:

```
<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>
```

EntityContainer Element (CSDL)

The **EntityContainer** element in conceptual schema definition language (CSDL) is a logical container for entity sets, association sets, and function imports. A conceptual model entity container maps to a storage model entity container through the EntityContainerMapping element. A storage model entity container describes the structure of the database: entity sets describe tables, association sets describe foreign key constraints, and function imports describe stored procedures in a database.

An **EntityContainer** element can have zero or one Documentation elements. If a **Documentation** element is present, it must precede all **EntitySet**, **AssociationSet**, and **FunctionImport** elements.

An **EntityContainer** element can have zero or more of the following child elements (in the order listed):

- EntitySet
- AssociationSet
- FunctionImport
- Annotation elements

You can extend an **EntityContainer** element to include the contents of another **EntityContainer** that is within the same namespace. To include the contents of another **EntityContainer**, in the referencing **EntityContainer** element, set the value of the **Extends** attribute to the name of the **EntityContainer** element that you want to include. All child elements of the included **EntityContainer** element will be treated as child elements of the referencing **EntityContainer** element.

Applicable Attributes

The table below describes the attributes that can be applied to the **Using** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity container.
Extends	No	The name of another entity container within the same namespace.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntityContainer** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element that defines three entity sets and two association sets.

```
<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>
```

EntitySet Element (CSDL)

The **EntitySet** element in conceptual schema definition language is a logical container for instances of an entity type and instances of any type that is derived from that entity type. The relationship between an entity type and an entity set is analogous to the relationship between a row and a table in a relational database. Like a row, an entity type defines a set of related data, and, like a table, an entity set contains instances of that definition. An entity set provides a construct for grouping entity type instances so that they can be mapped to related data structures in a data source.

More than one entity set for a particular entity type may be defined.

> **NOTE**
>
> The EF Designer does not support conceptual models that contain multiple entity sets per type.

The **EntitySet** element can have the following child elements (in the order listed):

- Documentation Element (zero or one elements allowed)
- Annotation elements (zero or more elements allowed)

Applicable Attributes

The table below describes the attributes that can be applied to the **EntitySet** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity set.
EntityType	Yes	The fully-qualified name of the entity type for which the entity set contains instances.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntitySet** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element with three **EntitySet** elements:

```
<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>
```

It is possible to define multiple entity sets per type (MEST). The following example defines an entity container with two entity sets for the **Book** entity type:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="FictionBooks" 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="BookAuthor" Association="BooksModel.BookAuthor">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

EntityType Element (CSDL)

The **EntityType** element represents the structure of a top-level concept, such as a customer or order, in a conceptual model. An entity type is a template for instances of entity types in an application. Each template contains the following information:

- A unique name. (Required.)
- An entity key that is defined by one or more properties. (Required.)
- Properties for containing data. (Optional.)

- Navigation properties that allow for navigation from one end of an association to the other end. (Optional.)

In an application, an instance of an entity type represents a specific object (such as a specific customer or order). Each instance of an entity type must have a unique entity key within an entity set.

Two entity type instances are considered equal only if they are of the same type and the values of their entity keys are the same.

An **EntityType** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Key (zero or one element)
- Property (zero or more elements)
- NavigationProperty (zero or more elements)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **EntityType** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity type.
BaseType	No	The name of another entity type that is the base type of the entity type that is being defined.
Abstract	No	**True** or **False**, depending on whether the entity type is an abstract type.
OpenType	No	**True** or **False** depending on whether the entity type is an open entity type. [!NOTE]
> The **OpenType** attribute is only applicable to entity types that are defined in conceptual models that are used with ADO.NET Data Services.		

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntityType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityType** element with three **Property** elements and two **NavigationProperty** elements:

```
<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>
```

EnumType Element (CSDL)

The **EnumType** element represents an enumerated type.

An **EnumType** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Member (zero or more elements)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **EnumType** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity type.
IsFlags	No	**True** or **False**, depending on whether the enum type can be used as a set of flags. The default value is **False.**.
UnderlyingType	No	**Edm.Byte**, **Edm.Int16**, **Edm.Int32**, **Edm.Int64** or **Edm.SByte** defining the range of values of the type. The default underlying type of enumeration elements is **Edm.Int32.**.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **EnumType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EnumType** element with three **Member** elements:

```
<EnumType Name="Color" IsFlags=”false” UnderlyingTyp=”Edm.Byte”>
  <Member Name="Red" />
  <Member Name="Green" />
  <Member Name="Blue" />
</EntityType>
```

Function Element (CSDL)

The **Function** element in conceptual schema definition language (CSDL) is used to define or declare functions in the conceptual model. A function is defined by using a DefiningExpression element.

A **Function** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Parameter (zero or more elements)
- DefiningExpression (zero or one element)
- ReturnType (Function) (zero or one element)
- Annotation elements (zero or more elements)

A return type for a function must be specified with either the **ReturnType** (Function) element or the **ReturnType** attribute (see below), but not both. The possible return types are any EdmSimpleType, entity type, complex type, row type, or ref type (or a collection of one of these types).

Applicable Attributes

The table below describes the attributes that can be applied to the **Function** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the function.
ReturnType	No	The type returned by the function.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Function** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example uses a **Function** element to define a function that returns the number of years since an instructor was hired.

```
<Function Name="YearsSince" ReturnType="Edm.Int32">
  <Parameter Name="date" Type="Edm.DateTime" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(date)
  </DefiningExpression>
</Function>
```

FunctionImport Element (CSDL)

The **FunctionImport** element in conceptual schema definition language (CSDL) represents a function that is defined in the data source but available to objects through the conceptual model. For example, a Function element in the storage model can be used to represent a stored procedure in a database. A **FunctionImport** element in the conceptual model represents the corresponding function in an Entity Framework application and is mapped to the storage model function by using the FunctionImportMapping element. When the function is called in the application, the corresponding stored procedure is executed in the database.

The **FunctionImport** element can have the following child elements (in the order listed):

- Documentation (zero or one elements allowed)
- Parameter (zero or more elements allowed)
- Annotation elements (zero or more elements allowed)
- ReturnType (FunctionImport) (zero or more elements allowed)

One **Parameter** element should be defined for each parameter that the function accepts.

A return type for a function must be specified with either the **ReturnType** (FunctionImport) element or the **ReturnType** attribute (see below), but not both. The return type value must be a collection of EdmSimpleType, EntityType, or ComplexType.

Applicable Attributes

The table below describes the attributes that can be applied to the **FunctionImport** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the imported function.
ReturnType	No	The type that the function returns. Do not use this attribute if the function does not return a value. Otherwise, the value must be a collection of ComplexType, EntityType, or EDMSimpleType.
EntitySet	No	If the function returns a collection of entity types, the value of the **EntitySet** must be the entity set to which the collection belongs. Otherwise, the **EntitySet** attribute must not be used.
IsComposable	No	If the value is set to true, the function is composable (Table-valued Function) and can be used in a LINQ query. The default is **false**.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **FunctionImport** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **FunctionImport** element that accepts one parameter and returns a collection of entity types:

```
<FunctionImport Name="GetStudentGrades"
                EntitySet="StudentGrade"
                ReturnType="Collection(SchoolModel.StudentGrade)">
       <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Key Element (CSDL)

The **Key** element is a child element of the EntityType element and defines an *entity key* (a property or a set of properties of an entity type that determine identity). The properties that make up an entity key are chosen at design time. The values of entity key properties must uniquely identify an entity type instance within an entity set at run time. The properties that make up an entity key should be chosen to guarantee uniqueness of instances in an entity set. The **Key** element defines an entity key by referencing one or more of the properties of an entity type.

The **Key** element can have the following child elements:

- PropertyRef (one or more elements)
- Annotation elements (zero or more elements)

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **Key** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The example below defines an entity type named **Book**. The entity key is defined by referencing the **ISBN** property of the entity type.

```
<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>
```

The **ISBN** property is a good choice for the entity key because an International Standard Book Number (ISBN) uniquely identifies a book.

The following example shows an entity type (**Author**) that has an entity key that consists of two properties, **Name** and **Address**.

```
<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>
```

Using **Name** and **Address** for the entity key is a reasonable choice, because two authors of the same name are unlikely to live at the same address. However, this choice for an entity key does not absolutely guarantee unique entity keys in an entity set. Adding a property, such as **AuthorId**, that could be used to uniquely identify an author would be recommended in this case.

Member Element (CSDL)

The **Member** element is a child element of the EnumType element and defines a member of the enumerated type.

Applicable Attributes

The table below describes the attributes that can be applied to the **FunctionImport** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the member.
Value	No	The value of the member. By default, the first member has the value 0, and the value of each successive enumerator is incremented by 1. Multiple members with the same values may exist.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **FunctionImport** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EnumType** element with three **Member** elements:

```
<EnumType Name="Color">
  <Member Name="Red" Value=”1”/>
  <Member Name="Green" Value=”3” />
  <Member Name="Blue" Value=”5”/>
</EntityType>
```

NavigationProperty Element (CSDL)

A **NavigationProperty** element defines a navigation property, which provides a reference to the other end of an association. Unlike properties defined with the Property element, navigation properties do not define the shape and characteristics of data. They provide a way to navigate an association between two entity types.

Note that navigation properties are optional on both entity types at the ends of an association. If you define a navigation property on one entity type at the end of an association, you do not have to define a navigation property on the entity type at the other end of the association.

The data type returned by a navigation property is determined by the multiplicity of its remote association end. For example, suppose a navigation property, **OrdersNavProp**, exists on a **Customer** entity type and navigates a one-to-many association between **Customer** and **Order**. Because the remote association end for the navigation property has multiplicity many (*), its data type is a collection (of **Order**). Similarly, if a navigation property, **CustomerNavProp**, exists on the **Order** entity type, its data type would be **Customer** since the multiplicity of the remote end is one (1).

A **NavigationProperty** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **NavigationProperty** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the navigation property.
Relationship	Yes	The name of an association that is within the scope of the model.

ATTRIBUTE NAME	IS REQUIRED	VALUE
ToRole	Yes	The end of the association at which navigation ends. The value of the **ToRole** attribute must be the same as the value of one of the **Role** attributes defined on one of the association ends (defined in the AssociationEnd element).
FromRole	Yes	The end of the association from which navigation begins. The value of the **FromRole** attribute must be the same as the value of one of the **Role** attributes defined on one of the association ends (defined in the AssociationEnd element).

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **NavigationProperty** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example defines an entity type (**Book**) with two navigation properties (**PublishedBy** and **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>
```

OnDelete Element (CSDL)

The **OnDelete** element in conceptual schema definition language (CSDL) defines behavior that is connected with an association. If the **Action** attribute is set to **Cascade** on one end of an association, related entity types on the other end of the association are deleted when the entity type on the first end is deleted. If the association between two entity types is a primary key-to-primary key relationship, then a loaded dependent object is deleted when the principal object on the other end of the association is deleted regardless of the **OnDelete** specification.

> **NOTE**
> The **OnDelete** element only affects the runtime behavior of an application; it does not affect behavior in the data source. The behavior defined in the data source should be the same as the behavior defined in the application.

An **OnDelete** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **OnDelete** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Action	Yes	**Cascade** or **None**. If **Cascade**, dependent entity types will be deleted when the principal entity type is deleted. If **None**, dependent entity types will not be deleted when the principal entity type is deleted.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Association** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that defines the **CustomerOrders** association. The **OnDelete** element indicates that all **Orders** that are related to a particular **Customer** and have been loaded into the ObjectContext will be deleted when the **Customer** is deleted.

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1">
        <OnDelete Action="Cascade" />
  </End>
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

Parameter Element (CSDL)

The **Parameter** element in conceptual schema definition language (CSDL) can be a child of the FunctionImport element or the Function element.

FunctionImport Element Application

A **Parameter** element (as a child of the **FunctionImport** element) is used to define input and output parameters for function imports that are declared in CSDL.

The **Parameter** element can have the following child elements (in the order listed):

- Documentation (zero or one elements allowed)
- Annotation elements (zero or more elements allowed)

Applicable Attributes

The following table describes the attributes that can be applied to the **Parameter** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the parameter.
Type	Yes	The parameter type. The value must be an **EDMSimpleType** or a complex type that is within the scope of the model.
Mode	No	**In**, **Out**, or **InOut** depending on whether the parameter is an input, output, or input/output parameter.
MaxLength	No	The maximum allowed length of the parameter.
Precision	No	The precision of the parameter.
Scale	No	The scale of the parameter.
SRID	No	Spatial System Reference Identifier. Valid only for parameters of spatial types. For more information, see SRID and SRID (SQL Server).

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Parameter** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **FunctionImport** element with one **Parameter** child element. The function accepts one input parameter and returns a collection of entity types.

```
<FunctionImport Name="GetStudentGrades"
                EntitySet="StudentGrade"
                ReturnType="Collection(SchoolModel.StudentGrade)">
       <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Function Element Application

A **Parameter** element (as a child of the **Function** element) defines parameters for functions that are defined or declared in a conceptual model.

The **Parameter** element can have the following child elements (in the order listed):

- Documentation (zero or one elements)
- CollectionType (zero or one elements)
- ReferenceType (zero or one elements)
- RowType (zero or one elements)

> **NOTE**
> Only one of the **CollectionType**, **ReferenceType**, or **RowType** elements can be a child element of a **Property** element.

- Annotation elements (zero or more elements allowed)

> **NOTE**
> Annotation elements must appear after all other child elements. Annotation elements are only allowed in CSDL v2 and later.

Applicable Attributes

The following table describes the attributes that can be applied to the **Parameter** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the parameter.
Type	No	The parameter type. A parameter can be any of the following types (or collections of these types): **EdmSimpleType** entity type complex type row type reference type
Nullable	No	**True** (the default value) or **False** depending on whether the property can have a **null** value.
DefaultValue	No	The default value of the property.
MaxLength	No	The maximum length of the property value.
FixedLength	No	**True** or **False** depending on whether the property value will be stored as a fixed length string.
Precision	No	The precision of the property value.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Scale	No	The scale of the property value.
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server).
Unicode	No	**True** or **False** depending on whether the property value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Parameter** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **Function** element that uses one **Parameter** child element to define a function parameter.

```
<Function Name="GetYearsEmployed" ReturnType="Edm.Int32">
<Parameter Name="Instructor" Type="SchoolModel.Person" />
  <DefiningExpression>
  Year(CurrentDateTime()) - Year(cast(Instructor.HireDate as DateTime))
  </DefiningExpression>
</Function>
```

Principal Element (CSDL)

The **Principal** element in conceptual schema definition language (CSDL) is a child element to the ReferentialConstraint element that defines the principal end of a referential constraint. A **ReferentialConstraint** element defines functionality that is similar to a referential integrity constraint in a relational database. In the same way that a column (or columns) from a database table can reference the primary key of another table, a property (or properties) of an entity type can reference the entity key of another entity type. The entity type that is referenced is called the *principal end* of the constraint. The entity type that references the principal end is called the *dependent end* of the constraint. **PropertyRef** elements are used to specify which keys are referenced by the dependent end.

The **Principal** element can have the following child elements (in the order listed):

- PropertyRef (one or more elements)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **Principal** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Role	Yes	The name of the entity type on the principal end of the association.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Principal** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **ReferentialConstraint** element that is part of the definition of the **PublishedBy** association. The **Id** property of the **Publisher** entity type makes up the principal end of the referential constraint.

```
<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>
```

Property Element (CSDL)

The **Property** element in conceptual schema definition language (CSDL) can be a child of the EntityType element, the ComplexType element, or the RowType element.

EntityType and ComplexType Element Applications

Property elements (as children of **EntityType** or **ComplexType** elements) define the shape and characteristics of data that an entity type instance or complex type instance will contain. Properties in a conceptual model are analogous to properties that are defined on a class. In the same way that properties on a class define the shape of the class and carry information about objects, properties in a conceptual model define the shape of an entity type and carry information about entity type instances.

The **Property** element can have the following child elements (in the order listed):

- Documentation Element (zero or one elements allowed)
- Annotation elements (zero or more elements allowed)

The following facets can be applied to a **Property** element: **Nullable**, **DefaultValue**, **MaxLength**, **FixedLength**,

Precision, **Scale**, **Unicode**, **Collation**, **ConcurrencyMode**. Facets are XML attributes that provide information about how property values are stored in the data store.

> **NOTE**
>
> Facets can only be applied to properties of type **EDMSimpleType**.

Applicable Attributes

The following table describes the attributes that can be applied to the **Property** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the property.
Type	Yes	The type of the property value. The property value type must be an **EDMSimpleType** or a complex type (indicated by a fully-qualified name) that is within scope of the model.
Nullable	No	**True** (the default value) or **False** depending on whether the property can have a null value. [!NOTE]
> In the CSDL v1 a complex type property must have `Nullable="False"`.		
DefaultValue	No	The default value of the property.
MaxLength	No	The maximum length of the property value.
FixedLength	No	**True** or **False** depending on whether the property value will be stored as a fixed length string.
Precision	No	The precision of the property value.
Scale	No	The scale of the property value.
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server).
Unicode	No	**True** or **False** depending on whether the property value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.

ATTRIBUTE NAME	IS REQUIRED	VALUE
ConcurrencyMode	No	**None** (the default value) or **Fixed**. If the value is set to **Fixed**, the property value will be used in optimistic concurrency checks.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Property** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityType** element with three **Property** elements:

```
<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>
```

The following example shows a **ComplexType** element with five **Property** elements:

```
<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>
```

RowType Element Application

Property elements (as the children of a **RowType** element) define the shape and characteristics of data that can be passed to or returned from a model-defined function.

The **Property** element can have exactly one of the following child elements:

- CollectionType
- ReferenceType
- RowType

The **Property** element can have any number child annotation elements.

> **NOTE**
> Annotation elements are only allowed in CSDL v2 and later.

Applicable Attributes

The following table describes the attributes that can be applied to the **Property** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the property.
Type	Yes	The type of the property value.
Nullable	No	**True** (the default value) or **False** depending on whether the property can have a null value. [!NOTE]
> In CSDL v1 a complex type property must have `Nullable="False"`.		
DefaultValue	No	The default value of the property.
MaxLength	No	The maximum length of the property value.
FixedLength	No	**True** or **False** depending on whether the property value will be stored as a fixed length string.
Precision	No	The precision of the property value.
Scale	No	The scale of the property value.
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server).
Unicode	No	**True** or **False** depending on whether the property value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Property** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows **Property** elements used to define the shape of the return type of a model-defined function.

```
<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>
```

PropertyRef Element (CSDL)

The **PropertyRef** element in conceptual schema definition language (CSDL) references a property of an entity type to indicate that the property will perform one of the following roles:

- Part of the entity's key (a property or a set of properties of an entity type that determine identity). One or more **PropertyRef** elements can be used to define an entity key.
- The dependent or principal end of a referential constraint.

The **PropertyRef** element can only have annotation elements (zero or more) as child elements.

> **NOTE**
> Annotation elements are only allowed in CSDL v2 and later.

Applicable Attributes

The table below describes the attributes that can be applied to the **PropertyRef** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the referenced property.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **PropertyRef** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The example below defines an entity type (**Book**). The entity key is defined by referencing the **ISBN** property of the entity type.

```
<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>
```

In the next example, two **PropertyRef** elements are used to indicate that two properties (**Id** and **PublisherId**) are the principal and dependent ends of a referential constraint.

```
<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>
```

ReferenceType Element (CSDL)

The **ReferenceType** element in conceptual schema definition language (CSDL) specifies a reference to an entity type. The **ReferenceType** element can be a child of the following elements:

- ReturnType (Function)
- Parameter
- CollectionType

The **ReferenceType** element is used when defining a parameter or return type for a function.

A **ReferenceType** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The table below describes the attributes that can be applied to the **ReferenceType** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	Yes	The name of the entity type being referenced.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **ReferenceType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows the **ReferenceType** element used as a child of a **Parameter** element in a model-defined function that accepts a reference to a **Person** entity type:

```
<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>
```

The following example shows the **ReferenceType** element used as a child of a **ReturnType** (Function) element in a model-defined function that returns a reference to a **Person** entity type:

```
<Function Name="GetPersonReference">
    <Parameter Name="p" Type="SchoolModel.Person" />
    <ReturnType>
        <ReferenceType Type="SchoolModel.Person" />
    </ReturnType>
    <DefiningExpression>
          REF(p)
    </DefiningExpression>
</Function>
```

ReferentialConstraint Element (CSDL)

A **ReferentialConstraint** element in conceptual schema definition language (CSDL) defines functionality that is similar to a referential integrity constraint in a relational database. In the same way that a column (or columns) from a database table can reference the primary key of another table, a property (or properties) of an entity type can reference the entity key of another entity type. The entity type that is referenced is called the *principal end* of the constraint. The entity type that references the principal end is called the *dependent end* of the constraint.

If a foreign key that is exposed on one entity type references a property on another entity type, the **ReferentialConstraint** element defines an association between the two entity types. Because the **ReferentialConstraint** element provides information about how two entity types are related, no corresponding AssociationSetMapping element is necessary in the mapping specification language (MSL). An association between two entity types that do not have foreign keys exposed must have a corresponding **AssociationSetMapping** element in order to map association information to the data source.

If a foreign key is not exposed on an entity type, the **ReferentialConstraint** element can only define a primary key-to-primary key constraint between the entity type and another entity type.

A **ReferentialConstraint** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Principal (exactly one element)
- Dependent (exactly one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The **ReferentialConstraint** element can have any number of annotation attributes (custom XML attributes). However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **ReferentialConstraint** element being used as part of the definition of the **PublishedBy** association.

```
<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>
```

ReturnType (Function) Element (CSDL)

The **ReturnType** (Function) element in conceptual schema definition language (CSDL) specifies the return type for a function that is defined in a Function element. A function return type can also be specified with a **ReturnType** attribute.

Return types can be any **EdmSimpleType**, entity type, complex type, row type, ref type, or a collection of one of these types.

The return type of a function can be specified with either the **Type** attribute of the **ReturnType** (Function) element, or with one of the following child elements:

- CollectionType
- ReferenceType
- RowType

> **NOTE**
> A model will not validate if you specify a function return type with both the **Type** attribute of the **ReturnType** (Function) element and one of the child elements.

Applicable Attributes

The following table describes the attributes that can be applied to the **ReturnType** (Function) element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
ReturnType	No	The type returned by the function.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **ReturnType** (Function) element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example uses a **Function** element to define a function that returns the number of years a book has been in print. Note that the return type is specified by the **Type** attribute of a **ReturnType** (Function) element.

```
<Function Name="GetYearsInPrint">
  <ReturnType Type=="Edm.Int32">
  <Parameter Name="book" Type="BooksModel.Book" />
  <DefiningExpression>
   Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
  </DefiningExpression>
</Function>
```

ReturnType (FunctionImport) Element (CSDL)

The **ReturnType** (FunctionImport) element in conceptual schema definition language (CSDL) specifies the return type for a function that is defined in a FunctionImport element. A function return type can also be specified with a **ReturnType** attribute.

Return types can be any collection of entity type, complex type,or **EdmSimpleType**,

The return type of a function is specified with the **Type** attribute of the **ReturnType** (FunctionImport) element.

Applicable Attributes

The following table describes the attributes that can be applied to the **ReturnType** (FunctionImport) element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	No	The type that the function returns. The value must be a collection of ComplexType, EntityType, or EDMSimpleType.
EntitySet	No	If the function returns a collection of entity types, the value of the **EntitySet** must be the entity set to which the collection belongs. Otherwise, the **EntitySet** attribute must not be used.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **ReturnType** (FunctionImport) element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example uses a **FunctionImport** that returns books and publishers. Note that the function returns two result sets and therefore two **ReturnType** (FunctionImport) elements are specified.

```
<FunctionImport Name="GetBooksAndPublishers">
  <ReturnType Type=="Collection(BooksModel.Book )" EntitySet=”Books”>
  <ReturnType Type=="Collection(BooksModel.Publisher)" EntitySet=”Publishers”>
</FunctionImport>
```

RowType Element (CSDL)

A **RowType** element in conceptual schema definition language (CSDL) defines an unnamed structure as a parameter or return type for a function defined in the conceptual model.

A **RowType** element can be the child of the following elements:

- CollectionType
- Parameter
- ReturnType (Function)

A **RowType** element can have the following child elements (in the order listed):

- Property (one or more)

- Annotation elements (zero or more)

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **RowType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a model-defined function that uses a **CollectionType** element to specify that the function returns a collection of rows (as specified in the **RowType** element).

```
<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>
```

Schema Element (CSDL)

The **Schema** element is the root element of a conceptual model definition. It contains definitions for the objects, functions, and containers that make up a conceptual model.

The **Schema** element may contain zero or more of the following child elements:

- Using
- EntityContainer
- EntityType
- EnumType
- Association
- ComplexType
- Function

A **Schema** element may contain zero or one Annotation elements.

> **NOTE**
>
> The **Function** element and annotation elements are only allowed in CSDL v2 and later.

The **Schema** element uses the **Namespace** attribute to define the namespace for the entity type, complex type, and association objects in a conceptual model. Within a namespace, no two objects can have the same name. Namespaces can span multiple **Schema** elements and multiple .csdl files.

A conceptual model namespace is different from the XML namespace of the **Schema** element. A conceptual model namespace (as defined by the **Namespace** attribute) is a logical container for entity types, complex types, and association types. The XML namespace (indicated by the **xmlns** attribute) of a **Schema** element is the default namespace for child elements and attributes of the **Schema** element. XML namespaces of the form http://schemas.microsoft.com/ado/YYYY/MM/edm (where YYYY and MM represent a year and month respectively) are reserved for CSDL. Custom elements and attributes cannot be in namespaces that have this form.

Applicable Attributes

The table below describes the attributes can be applied to the **Schema** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Namespace	Yes	The namespace of the conceptual model. The value of the **Namespace** attribute is used to form the fully qualified name of a type. For example, if an **EntityType** named *Customer* is in the Simple.Example.Model namespace, then the fully qualified name of the **EntityType** is SimpleExampleModel.Customer. The following strings cannot be used as the value for the **Namespace** attribute: **System**, **Transient**, or **Edm**. The value for the **Namespace** attribute cannot be the same as the value for the **Namespace** attribute in the SSDL Schema element.
Alias	No	An identifier used in place of the namespace name. For example, if an **EntityType** named *Customer* is in the Simple.Example.Model namespace and the value of the **Alias** attribute is *Model*, then you can use Model.Customer as the fully qualified name of the **EntityType.**

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **Schema** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **Schema** element that contains an **EntityContainer** element, two **EntityType** elements, and one **Association** element.

```xml
<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">
        <EntityContainer Name="ExampleModelContainer">
          <EntitySet Name="Customers"
                     EntityType="ExampleModel.Customer" />
          <EntitySet Name="Orders" EntityType="ExampleModel.Order" />
          <AssociationSet
                      Name="CustomerOrder"
                      Association="ExampleModel.CustomerOrders">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Order" EntitySet="Orders" />
          </AssociationSet>
        </EntityContainer>
        <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>
```

TypeRef Element (CSDL)

The **TypeRef** element in conceptual schema definition language (CSDL) provides a reference to an existing named type. The **TypeRef** element can be a child of the CollectionType element, which is used to specify that a function has a collection as a parameter or return type.

A **TypeRef** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The following table describes the attributes that can be applied to the **TypeRef** element. Note that the **DefaultValue**, **MaxLength**, **FixedLength**, **Precision**, **Scale**, **Unicode**, and **Collation** attributes are only applicable to **EDMSimpleTypes**.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	No	The name of the type being referenced.
Nullable	No	**True** (the default value) or **False** depending on whether the property can have a null value. [!NOTE]
> In CSDL v1 a complex type property must have `Nullable="False"`.		
DefaultValue	No	The default value of the property.
MaxLength	No	The maximum length of the property value.
FixedLength	No	**True** or **False** depending on whether the property value will be stored as a fixed length string.
Precision	No	The precision of the property value.
Scale	No	The scale of the property value.
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server).
Unicode	No	**True** or **False** depending on whether the property value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **CollectionType** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a model-defined function that uses the **TypeRef** element (as a child of a **CollectionType** element) to specify that the function accepts a collection of **Department** entity types.

```
<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>
```

Using Element (CSDL)

The **Using** element in conceptual schema definition language (CSDL) imports the contents of a conceptual model that exists in a different namespace. By setting the value of the **Namespace** attribute, you can refer to entity types, complex types, and association types that are defined in another conceptual model. More than one **Using** element can be a child of a **Schema** element.

> **NOTE**
>
> The **Using** element in CSDL does not function exactly like a **using** statement in a programming language. By importing a namespace with a **using** statement in a programming language, you do not affect objects in the original namespace. In CSDL, an imported namespace can contain an entity type that is derived from an entity type in the original namespace. This can affect entity sets declared in the original namespace.

The **Using** element can have the following child elements:

- Documentation (zero or one elements allowed)
- Annotation elements (zero or more elements allowed)

Applicable Attributes

The table below describes the attributes can be applied to the **Using** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Namespace	Yes	The name of the imported namespace.
Alias	Yes	An identifier used in place of the namespace name. Although this attribute is required, it is not required that it be used in place of the namespace name to qualify object names.

NOTE

Any number of annotation attributes (custom XML attributes) may be applied to the **Using** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example demonstrates the **Using** element being used to import a namespace that is defined elsewhere. Note that the namespace for the **Schema** element shown is `BooksModel`. The `Address` property on the `Publisher` **EntityType** is a complex type that is defined in the `ExtendedBooksModel` namespace (imported with the **Using** element).

```
<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" >
      <EntitySet 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>
```

Annotation Attributes (CSDL)

Annotation attributes in conceptual schema definition language (CSDL) are custom XML attributes in the conceptual model. In addition to having valid XML structure, the following must be true of annotation attributes:

- Annotation attributes must not be in any XML namespace that is reserved for CSDL.
- More than one annotation attribute may be applied to a given CSDL element.
- The fully-qualified names of any two annotation attributes must not be the same.

Annotation attributes can be used to provide extra metadata about the elements in a conceptual model. Metadata contained in annotation elements can be accessed at runtime by using classes in the System.Data.Metadata.Edm namespace.

Example

The following example shows an **EntityType** element with an annotation attribute (**CustomAttribute**). The example also shows an annotation element applied to the entity type element.

```
<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>
```

The following code retrieves the metadata in the annotation attribute and writes it to the console:

```
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());
}
```

The code above assumes that the `School.csdl` file is in the project's output directory and that you have added the following `Imports` and `Using` statements to your project:

```
using System.Data.Metadata.Edm;
```

Annotation Elements (CSDL)

Annotation elements in conceptual schema definition language (CSDL) are custom XML elements in the conceptual model. In addition to having valid XML structure, the following must be true of annotation elements:

- Annotation elements must not be in any XML namespace that is reserved for CSDL.
- More than one annotation element may be a child of a given CSDL element.

- The fully-qualified names of any two annotation elements must not be the same.
- Annotation elements must appear after all other child elements of a given CSDL element.

Annotation elements can be used to provide extra metadata about the elements in a conceptual model. Starting with the .NET Framework version 4, metadata contained in annotation elements can be accessed at runtime by using classes in the System.Data.Metadata.Edm namespace.

Example

The following example shows an **EntityType** element with an annotation element (**CustomElement**). The example also show an annotation attribute applied to the entity type element.

```
<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>
```

The following code retrieves the metadata in the annotation element and writes it to the console:

```
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());
}
```

The code above assumes that the School.csdl file is in the project's output directory and that you have added the following `Imports` and `Using` statements to your project:

```
using System.Data.Metadata.Edm;
```

Conceptual Model Types (CSDL)

Conceptual schema definition language (CSDL) supports a set of abstract primitive data types, called **EDMSimpleTypes**, that define properties in a conceptual model. **EDMSimpleTypes** are proxies for primitive data types that are supported in the storage or hosting environment.

The table below lists the primitive data types that are supported by CSDL. The table also lists the facets that can be applied to each **EDMSimpleType**.

EDMSIMPLETYPE	DESCRIPTION	APPLICABLE FACETS
Edm.Binary	Contains binary data.	MaxLength, FixedLength, Nullable, Default
Edm.Boolean	Contains the value **true** or **false**.	Nullable, Default
Edm.Byte	Contains an unsigned 8-bit integer value.	Precision, Nullable, Default
Edm.DateTime	Represents a date and time.	Precision, Nullable, Default
Edm.DateTimeOffset	Contains a date and time as an offset in minutes from GMT.	Precision, Nullable, Default
Edm.Decimal	Contains a numeric value with fixed precision and scale.	Precision, Nullable, Default
Edm.Double	Contains a floating point number with 15-digit precision	Precision, Nullable, Default
Edm.Float	Contains a floating point number with 7-digit precision.	Precision, Nullable, Default
Edm.Guid	Contains a 16-byte unique identifier.	Precision, Nullable, Default
Edm.Int16	Contains a signed 16-bit integer value.	Precision, Nullable, Default
Edm.Int32	Contains a signed 32-bit integer value.	Precision, Nullable, Default
Edm.Int64	Contains a signed 64-bit integer value.	Precision, Nullable, Default
Edm.SByte	Contains a signed 8-bit integer value.	Precision, Nullable, Default
Edm.String	Contains character data.	Unicode, FixedLength, MaxLength, Collation, Precision, Nullable, Default
Edm.Time	Contains a time of day.	Precision, Nullable, Default
Edm.Geography		Nullable, Default, SRID
Edm.GeographyPoint		Nullable, Default, SRID

EDMSIMPLETYPE	DESCRIPTION	APPLICABLE FACETS
Edm.GeographyLineString		Nullable, Default, SRID
Edm.GeographyPolygon		Nullable, Default, SRID
Edm.GeographyMultiPoint		Nullable, Default, SRID
Edm.GeographyMultiLineString		Nullable, Default, SRID
Edm.GeographyMultiPolygon		Nullable, Default, SRID
Edm.GeographyCollection		Nullable, Default, SRID
Edm.Geometry		Nullable, Default, SRID
Edm.GeometryPoint		Nullable, Default, SRID
Edm.GeometryLineString		Nullable, Default, SRID
Edm.GeometryPolygon		Nullable, Default, SRID
Edm.GeometryMultiPoint		Nullable, Default, SRID
Edm.GeometryMultiLineString		Nullable, Default, SRID
Edm.GeometryMultiPolygon		Nullable, Default, SRID
Edm.GeometryCollection		Nullable, Default, SRID

Facets (CSDL)

Facets in conceptual schema definition language (CSDL) represent constraints on properties of entity types and complex types. Facets appear as XML attributes on the following CSDL elements:

- Property
- TypeRef
- Parameter

The following table describes the facets that are supported in CSDL. All facets are optional. Some facets listed below are used by the Entity Framework when generating a database from a conceptual model.

> **NOTE**
> For information about data types in a conceptual model, see Conceptual Model Types (CSDL).

FACET	DESCRIPTION	APPLIES TO	USED FOR THE DATABASE GENERATION	USED BY THE RUNTIME

FACET	DESCRIPTION	APPLIES TO	USED FOR THE DATABASE GENERATION	USED BY THE RUNTIME
Collation	Specifies the collating sequence (or sorting sequence) to be used when performing comparison and ordering operations on values of the property.	**Edm.String**	Yes	No
ConcurrencyMode	Indicates that the value of the property should be used for optimistic concurrency checks.	All **EDMSimpleType** properties	No	Yes
Default	Specifies the default value of the property if no value is supplied upon instantiation.	All **EDMSimpleType** properties	Yes	Yes
FixedLength	Specifies whether the length of the property value can vary.	**Edm.Binary**, **Edm.String**	Yes	No
MaxLength	Specifies the maximum length of the property value.	**Edm.Binary**, **Edm.String**	Yes	No
Nullable	Specifies whether the property can have a **null** value.	All **EDMSimpleType** properties	Yes	Yes
Precision	For properties of type **Decimal**, specifies the number of digits a property value can have. For properties of type **Time**, **DateTime**, and **DateTimeOffset**, specifies the number of digits for the fractional part of seconds of the property value.	**Edm.DateTime**, **Edm.DateTimeOffset**, **Edm.Decimal**, **Edm.Time**	Yes	No
Scale	Specifies the number of digits to the right of the decimal point for the property value.	**Edm.Decimal**	Yes	No

FACET	DESCRIPTION	APPLIES TO	USED FOR THE DATABASE GENERATION	USED BY THE RUNTIME
SRID	Specifies the Spatial System Reference System ID. For more information, see SRID and 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**	No	Yes
Unicode	Indicates whether the property value is stored as Unicode.	**Edm.String**	Yes	Yes

> **NOTE**
> When generating a database from a conceptual model, the Generate Database Wizard will recognize the value of the **StoreGeneratedPattern** attribute on a **Property** element if it is in the following namespace: http://schemas.microsoft.com/ado/2009/02/edm/annotation. The supported values for the attribute are **Identity** and **Computed**. A value of **Identity** will produce a database column with an identity value that is generated in the database. A value of **Computed** will produce a column with a value that is computed in the database.

Example

The following example shows facets applied to the properties of an entity type:

```xml
<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>
```

MSL Specification

10/1/2018 • 40 minutes to read • Edit Online

Mapping specification language (MSL) is an XML-based language that describes the mapping between the conceptual model and storage model of an Entity Framework application.

In an Entity Framework application, mapping metadata is loaded from an .msl file (written in MSL) at build time. Entity Framework uses mapping metadata at runtime to translate queries against the conceptual model to store-specific commands.

The Entity Framework Designer (EF Designer) stores mapping information in an .edmx file at design time. At build time, the Entity Designer uses information in an .edmx file to create the .msl file that is needed by Entity Framework at runtime

Names of all conceptual or storage model types that are referenced in MSL must be qualified by their respective namespace names. For information about the conceptual model namespace name, see CSDL Specification. For information about the storage model namespace name, see SSDL Specification.

Versions of MSL are differentiated by XML namespaces.

MSL VERSION	XML NAMESPACE
MSL v1	urn:schemas-microsoft-com:windows:storage:mapping:CS
MSL v2	http://schemas.microsoft.com/ado/2008/09/mapping/cs
MSL v3	http://schemas.microsoft.com/ado/2009/11/mapping/cs

Alias Element (MSL)

The **Alias** element in mapping specification language (MSL) is a child of the Mapping element that is used to define aliases for conceptual model and storage model namespaces. Names of all conceptual or storage model types that are referenced in MSL must be qualified by their respective namespace names. For information about the conceptual model namespace name, see Schema Element (CSDL). For information about the storage model namespace name, see Schema Element (SSDL).

The **Alias** element cannot have child elements.

Applicable Attributes

The table below describes the attributes that can be applied to the **Alias** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Key	Yes	The alias for the namespace that is specified by the **Value** attribute.
Value	Yes	The namespace for which the value of the **Key** element is an alias.

Example

The following example shows an **Alias** element that defines an alias, `c`, for types that are defined in the conceptual model.

```
<Mapping Space="C-S"
         xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
                          CdmEntityContainer="SchoolModelEntities">
    <EntitySetMapping Name="Courses">
      <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>
```

AssociationEnd Element (MSL)

The **AssociationEnd** element in mapping specification language (MSL) is used when the modification functions of an entity type in the conceptual model are mapped to stored procedures in the underlying database. If a modification stored procedure takes a parameter whose value is held in an association property, the **AssociationEnd** element maps the property value to the parameter. For more information, see the example below.

For more information about mapping modification functions of entity types to stored procedures, see ModificationFunctionMapping Element (MSL) and Walkthrough: Mapping an Entity to Stored Procedures.

The **AssociationEnd** element can have the following child elements:

- ScalarProperty

Applicable Attributes

The following table describes the attributes that are applicable to the **AssociationEnd** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
AssociationSet	Yes	The name of the association that is being mapped.
From	Yes	The value of the **FromRole** attribute of the navigation property that corresponds to the association being mapped. For more information, see NavigationProperty Element (CSDL).

ATTRIBUTE NAME	IS REQUIRED	VALUE
To	Yes	The value of the **ToRole** attribute of the navigation property that corresponds to the association being mapped. For more information, see NavigationProperty Element (CSDL).

Example

Consider the following conceptual model entity type:

```
<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>
```

Also consider the following stored procedure:

```
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;
```

In order to map the update function of the `Course` entity to this stored procedure, you must supply a value to the **DepartmentID** parameter. The value for `DepartmentID` does not correspond to a property on the entity type; it is contained in an independent association whose mapping is shown here:

```
<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>
```

The following code shows the **AssociationEnd** element used to map the **DepartmentID** property of the **FK_Course_Department** association to the **UpdateCourse** stored procedure (to which the update function of the **Course** entity type is mapped):

```
<EntitySetMapping Name="Courses">
  <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>
</EntitySetMapping>
```

AssociationSetMapping Element (MSL)

The **AssociationSetMapping** element in mapping specification language (MSL) defines the mapping between an association in the conceptual model and table columns in the underlying database.

Associations in the conceptual model are types whose properties represent primary and foreign key columns in the underlying database. The **AssociationSetMapping** element uses two EndProperty elements to define the mappings between association type properties and columns in the database. You can place conditions on these mappings with the Condition element. Map the insert, update, and delete functions for associations to stored procedures in the database with the ModificationFunctionMapping element. Define read-only mappings between associations and table columns by using an Entity SQL string in a QueryView element.

> **NOTE**
>
> If a referential constraint is defined for an association in the conceptual model, the association does not need to be mapped with an **AssociationSetMapping** element. If an **AssociationSetMapping** element is present for an association that has a referential constraint, the mappings defined in the **AssociationSetMapping** element will be ignored. For more information, see ReferentialConstraint Element (CSDL).

The **AssociationSetMapping** element can have the following child elements

- QueryView (zero or one)
- EndProperty (zero or two)
- Condition (zero or more)
- ModificationFunctionMapping (zero or one)

Applicable Attributes

The following table describes the attributes that can be applied to the **AssociationSetMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the conceptual model association set that is being mapped.
TypeName	No	The namespace-qualified name of the conceptual model association type that is being mapped.
StoreEntitySet	No	The name of the table that is being mapped.

Example

The following example shows an **AssociationSetMapping** element in which the **FK_Course_Department** association set in the conceptual model is mapped to the **Course** table in the database. Mappings between association type properties and table columns are specified in child **EndProperty** elements.

```
<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>
```

ComplexProperty Element (MSL)

A **ComplexProperty** element in mapping specification language (MSL) defines the mapping between a complex type property on a conceptual model entity type and table columns in the underlying database. The property-column mappings are specified in child ScalarProperty elements.

The **ComplexType** property element can have the following child elements:

- ScalarProperty (zero or more)
- **ComplexProperty** (zero or more)
- ComplextTypeMapping (zero or more)
- Condition (zero or more)

Applicable Attributes

The following table describes the attributes that are applicable to the **ComplexProperty** element:

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the complex property of an entity type in the conceptual model that is being mapped.
TypeName	No	The namespace-qualified name of the conceptual model property type.

Example

The following example is based on the School model. The following complex type has been added to the conceptual model:

```
<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>
```

The **LastName** and **FirstName** properties of the **Person** entity type have been replaced with one complex property, **Name**:

```
<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>
```

The following MSL shows the **ComplexProperty** element used to map the **Name** property to columns in the underlying database:

```
<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>
```

ComplexTypeMapping Element (MSL)

The **ComplexTypeMapping** element in mapping specification language (MSL) is a child of the ResultMapping element and defines the mapping between a function import in the conceptual model and a stored procedure in the underlying database when the following are true:

- The function import returns a conceptual complex type.
- The names of the columns returned by the stored procedure do not exactly match the names of the properties on the complex type.

By default, the mapping between the columns returned by a stored procedure and a complex type is based on column and property names. If column names do not exactly match property names, you must use the **ComplexTypeMapping** element to define the mapping. For an example of the default mapping, see FunctionImportMapping Element (MSL).

The **ComplexTypeMapping** element can have the following child elements:

- ScalarProperty (zero or more)

Applicable Attributes

The following table describes the attributes that are applicable to the **ComplexTypeMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
TypeName	Yes	The namespace-qualified name of the complex type that is being mapped.

Example

Consider the following stored procedure:

```
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
```

Also consider the following conceptual model complex type:

```
<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>
```

In order to create a function import that returns instances of the previous complex type, the mapping between the columns returned by the stored procedure and the entity type must be defined in a **ComplexTypeMapping** element:

```
<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>
```

Condition Element (MSL)

The **Condition** element in mapping specification language (MSL) places conditions on mappings between the conceptual model and the underlying database. The mapping that is defined within an XML node is valid if all conditions, as specified in child **Condition** elements, are met. Otherwise, the mapping is not valid. For example, if a MappingFragment element contains one or more **Condition** child elements, the mapping defined within the **MappingFragment** node will only be valid if all the conditions of the child **Condition** elements are met.

Each condition can apply to either a **Name** (the name of a conceptual model entity property, specified by the **Name** attribute), or a **ColumnName** (the name of a column in the database, specified by the **ColumnName** attribute). When the **Name** attribute is set, the condition is checked against an entity property value. When the **ColumnName** attribute is set, the condition is checked against a column value. Only one of the **Name** or **ColumnName** attribute can be specified in a **Condition** element.

> **NOTE**
> When the **Condition** element is used within a FunctionImportMapping element, only the **Name** attribute is not applicable.

The **Condition** element can be a child of the following elements:

- AssociationSetMapping
- ComplexProperty
- EntitySetMapping
- MappingFragment
- EntityTypeMapping

The **Condition** element can have no child elements.

Applicable Attributes

The following table describes the attributes that are applicable to the **Condition** element:

ATTRIBUTE NAME	IS REQUIRED	VALUE
ColumnName	No	The name of the table column whose value is used to evaluate the condition.
IsNull	No	**True** or **False**. If the value is **True** and the column value is **null**, or if the value is **False** and the column value is not **null**, the condition is true. Otherwise, the condition is false. The **IsNull** and **Value** attributes cannot be used at the same time.
Value	No	The value with which the column value is compared. If the values are the same, the condition is true. Otherwise, the condition is false. The **IsNull** and **Value** attributes cannot be used at the same time.
Name	No	The name of the conceptual model entity property whose value is used to evaluate the condition. This attribute is not applicable if the **Condition** element is used within a FunctionImportMapping element.

Example

The following example shows **Condition** elements as children of **MappingFragment** elements. When **HireDate** is not null and **EnrollmentDate** is null, data is mapped between the **SchoolModel.Instructor** type and the **PersonID** and **HireDate** columns of the **Person** table. When **EnrollmentDate** is not null and **HireDate** is null, data is mapped between the **SchoolModel.Student** type and the **PersonID** and **Enrollment** columns of the **Person** table.

```
<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>
```

DeleteFunction Element (MSL)

The **DeleteFunction** element in mapping specification language (MSL) maps the delete function of an entity type or association in the conceptual model to a stored procedure in the underlying database. Stored procedures to which modification functions are mapped must be declared in the storage model. For more information, see Function Element (SSDL).

> **NOTE**
>
> If you do not map all three of the insert, update, or delete operations of a entity type to stored procedures, the unmapped operations will fail if executed at runtime and an UpdateException is thrown.

DeleteFunction Applied to EntityTypeMapping

When applied to the EntityTypeMapping element, the **DeleteFunction** element maps the delete function of an entity type in the conceptual model to a stored procedure.

The **DeleteFunction** element can have the following child elements when applied to an **EntityTypeMapping** element:

- AssociationEnd (zero or more)
- ComplexProperty (zero or more)
- ScarlarProperty (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **DeleteFunction** element when it is applied to an **EntityTypeMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionName	Yes	The namespace-qualified name of the stored procedure to which the delete function is mapped. The stored procedure must be declared in the storage model.
RowsAffectedParameter	No	The name of the output parameter that returns the number of rows affected.

Example

The following example is based on the School model and shows the **DeleteFunction** element mapping the delete function of the **Person** entity type to the **DeletePerson** stored procedure. The **DeletePerson** stored procedure is declared in the storage model.

```
<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>
```

DeleteFunction Applied to AssociationSetMapping

When applied to the AssociationSetMapping element, the **DeleteFunction** element maps the delete function of an association in the conceptual model to a stored procedure.

The **DeleteFunction** element can have the following child elements when applied to the

AssociationSetMapping element:

- EndProperty

Applicable Attributes

The following table describes the attributes that can be applied to the **DeleteFunction** element when it is applied to the **AssociationSetMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionName	Yes	The namespace-qualified name of the stored procedure to which the delete function is mapped. The stored procedure must be declared in the storage model.
RowsAffectedParameter	No	The name of the output parameter that returns the number of rows affected.

Example

The following example is based on the School model and shows the **DeleteFunction** element used to map delete function of the **CourseInstructor** association to the **DeleteCourseInstructor** stored procedure. The **DeleteCourseInstructor** stored procedure is declared in the storage model.

```
<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>
```

EndProperty Element (MSL)

The **EndProperty** element in mapping specification language (MSL) defines the mapping between an end or a modification function of a conceptual model association and the underlying database. The property-column mapping is specified in a child ScalarProperty element.

When an **EndProperty** element is used to define the mapping for the end of a conceptual model association, it is a child of an AssociationSetMapping element. When the **EndProperty** element is used to define the mapping for a modification function of a conceptual model association, it is a child of an InsertFunction element or DeleteFunction element.

The **EndProperty** element can have the following child elements:

- ScalarProperty (zero or more)

Applicable Attributes

The following table describes the attributes that are applicable to the **EndProperty** element:

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the association end that is being mapped.

Example

The following example shows an **AssociationSetMapping** element in which the **FK_Course_Department** association in the conceptual model is mapped to the **Course** table in the database. Mappings between association type properties and table columns are specified in child **EndProperty** elements.

```
<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>
```

Example

The following example shows the **EndProperty** element mapping the insert and delete functions of an association (**CourseInstructor**) to stored procedures in the underlying database. The functions that are mapped to are declared in the storage model.

```
<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>
```

EntityContainerMapping Element (MSL)

The **EntityContainerMapping** element in mapping specification language (MSL) maps the entity container in the conceptual model to the entity container in the storage model. The **EntityContainerMapping** element is a child of the Mapping element.

The **EntityContainerMapping** element can have the following child elements (in the order listed):

- EntitySetMapping (zero or more)
- AssociationSetMapping (zero or more)
- FunctionImportMapping (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **EntityContainerMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
StorageModelContainer	Yes	The name of the storage model entity container that is being mapped.
CdmEntityContainer	Yes	The name of the conceptual model entity container that is being mapped.
GenerateUpdateViews	No	**True** or **False**. If **False**, no update views are generated. This attribute should be set to **False** when you have a read-only mapping that would be invalid because data may not round-trip successfully. The default value is **True**.

Example

The following example shows an **EntityContainerMapping** element that maps the **SchoolModelEntities** container (the conceptual model entity container) to the **SchoolModelStoreContainer** container (the storage model entity container):

```
<EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
                        CdmEntityContainer="SchoolModelEntities">
  <EntitySetMapping Name="Courses">
    <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>
```

EntitySetMapping Element (MSL)

The **EntitySetMapping** element in mapping specification language (MSL) maps all types in a conceptual model entity set to entity sets in the storage model. An entity set in the conceptual model is a logical container for instances of entities of the same type (and derived types). An entity set in the storage model represents a table or view in the underlying database. The conceptual model entity set is specified by the value of the **Name** attribute of the **EntitySetMapping** element. The mapped-to table or view is specified by the **StoreEntitySet** attribute in each child MappingFragment element or in the **EntitySetMapping** element itself.

The **EntitySetMapping** element can have the following child elements:

- EntityTypeMapping (zero or more)
- QueryView (zero or one)
- MappingFragment (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **EntitySetMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the conceptual model entity set that is being mapped.
TypeName 1	No	The name of the conceptual model entity type that is being mapped.
StoreEntitySet 1	No	The name of the storage model entity set that is being mapped to.

ATTRIBUTE NAME	IS REQUIRED	VALUE
MakeColumnsDistinct	No	**True** or **False** depending on whether only distinct rows are returned. If this attribute is set to **True**, the **GenerateUpdateViews** attribute of the EntityContainerMapping element must be set to **False**.

1 The **TypeName** and **StoreEntitySet** attributes can be used in place of the EntityTypeMapping and MappingFragment child elements to map a single entity type to a single table.

Example

The following example shows an **EntitySetMapping** element that maps three types (a base type and two derived types) in the **Courses** entity set of the conceptual model to three different tables in the underlying database. The tables are specified by the **StoreEntitySet** attribute in each **MappingFragment** element.

```
<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>
```

EntityTypeMapping Element (MSL)

The **EntityTypeMapping** element in mapping specification language (MSL) defines the mapping between an entity type in the conceptual model and tables or views in the underlying database. For information about conceptual model entity types and underlying database tables or views, see EntityType Element (CSDL) and EntitySet Element (SSDL). The conceptual model entity type that is being mapped is specified by the **TypeName** attribute of the **EntityTypeMapping** element. The table or view that is being mapped is specified by the **StoreEntitySet** attribute of the child MappingFragment element.

The ModificationFunctionMapping child element can be used to map the insert, update, or delete functions of entity types to stored procedures in the database.

The **EntityTypeMapping** element can have the following child elements:

- MappingFragment (zero or more)

- ModificationFunctionMapping (zero or one)
- ScalarProperty
- Condition

> **NOTE**
>
> **MappingFragment** and **ModificationFunctionMapping** elements cannot be child elements of the **EntityTypeMapping** element at the same time.

> **NOTE**
>
> The **ScalarProperty** and **Condition** elements can only be child elements of the **EntityTypeMapping** element when it is used within a FunctionImportMapping element.

Applicable Attributes

The following table describes the attributes that can be applied to the **EntityTypeMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
TypeName	Yes	The namespace-qualified name of the conceptual model entity type that is being mapped. If the type is abstract or a derived type, the value must be `IsOfType(Namespace-qualified_type_name)` .

Example

The following example shows an EntitySetMapping element with two child **EntityTypeMapping** elements. In the first **EntityTypeMapping** element, the **SchoolModel.Person** entity type is mapped to the **Person** table. In the second **EntityTypeMapping** element, the update functionality of the **SchoolModel.Person** type is mapped to a stored procedure, **UpdatePerson**, in the database.

```
<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>
```

Example

The next example shows the mapping of a type hierarchy in which the root type is abstract. Note the use of the `IsOfType` syntax for the **TypeName** attributes.

```
<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>
```

FunctionImportMapping Element (MSL)

The **FunctionImportMapping** element in mapping specification language (MSL) defines the mapping between a function import in the conceptual model and a stored procedure or function in the underlying database. Function imports must be declared in the conceptual model and stored procedures must be declared in the storage model. For more information, see FunctionImport Element (CSDL) and Function Element (SSDL).

> **NOTE**
> By default, if a function import returns a conceptual model entity type or complex type, then the names of the columns returned by the underlying stored procedure must exactly match the names of the properties on the conceptual model type. If the column names do not exactly match the property names, the mapping must be defined in a ResultMapping element.

The **FunctionImportMapping** element can have the following child elements:

- ResultMapping (zero or more)

Applicable Attributes

The following table describes the attributes that are applicable to the **FunctionImportMapping** element:

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionImportName	Yes	The name of the function import in the conceptual model that is being mapped.
FunctionName	Yes	The namespace-qualified name of the function in the storage model that is being mapped.

Example

The following example is based on the School model. Consider the following function in the storage model:

```
<Function Name="GetStudentGrades" Aggregate="false"
          BuiltIn="false" NiladicFunction="false"
          IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
          Schema="dbo">
  <Parameter Name="StudentID" Type="int" Mode="In" />
</Function>
```

Also consider this function import in the conceptual model:

```
<FunctionImport Name="GetStudentGrades" EntitySet="StudentGrades"
                ReturnType="Collection(SchoolModel.StudentGrade)">
  <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

The following example show a **FunctionImportMapping** element used to map the function and function import above to each other:

```
<FunctionImportMapping FunctionImportName="GetStudentGrades"
                       FunctionName="SchoolModel.Store.GetStudentGrades" />
```

InsertFunction Element (MSL)

The **InsertFunction** element in mapping specification language (MSL) maps the insert function of an entity type or association in the conceptual model to a stored procedure in the underlying database. Stored procedures to which modification functions are mapped must be declared in the storage model. For more information, see Function Element (SSDL).

> **NOTE**
> If you do not map all three of the insert, update, or delete operations of a entity type to stored procedures, the unmapped operations will fail if executed at runtime and an UpdateException is thrown.

The **InsertFunction** element can be a child of the ModificationFunctionMapping element and applied to the EntityTypeMapping element or the AssociationSetMapping element.

InsertFunction Applied to EntityTypeMapping

When applied to the EntityTypeMapping element, the **InsertFunction** element maps the insert function of an entity type in the conceptual model to a stored procedure.

The **InsertFunction** element can have the following child elements when applied to an **EntityTypeMapping** element:

- AssociationEnd (zero or more)
- ComplexProperty (zero or more)
- ResultBinding (zero or one)
- ScarlarProperty (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **InsertFunction** element when applied to an **EntityTypeMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionName	Yes	The namespace-qualified name of the stored procedure to which the insert function is mapped. The stored procedure must be declared in the storage model.
RowsAffectedParameter	No	The name of the output parameter that returns the number of affected rows.

Example

The following example is based on the School model and shows the **InsertFunction** element used to map insert function of the Person entity type to the **InsertPerson** stored procedure. The **InsertPerson** stored procedure is declared in the storage model.

```xml
<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>
```

InsertFunction Applied to AssociationSetMapping

When applied to the AssociationSetMapping element, the **InsertFunction** element maps the insert function of an association in the conceptual model to a stored procedure.

The **InsertFunction** element can have the following child elements when applied to the **AssociationSetMapping** element:

- EndProperty

Applicable Attributes

The following table describes the attributes that can be applied to the **InsertFunction** element when it is applied to the **AssociationSetMapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionName	Yes	The namespace-qualified name of the stored procedure to which the insert function is mapped. The stored procedure must be declared in the storage model.
RowsAffectedParameter	No	The name of the output parameter that returns the number of rows affected.

Example

The following example is based on the School model and shows the **InsertFunction** element used to map insert function of the **CourseInstructor** association to the **InsertCourseInstructor** stored procedure. The **InsertCourseInstructor** stored procedure is declared in the storage model.

```
<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>
```

Mapping Element (MSL)

The **Mapping** element in mapping specification language (MSL) contains information for mapping objects that are defined in a conceptual model to a database (as described in a storage model). For more information, see CSDL Specification and SSDL Specification.

The **Mapping** element is the root element for a mapping specification. The XML namespace for mapping specifications is http://schemas.microsoft.com/ado/2009/11/mapping/cs.

The mapping element can have the following child elements (in the order listed):

- Alias (zero or more)
- EntityContainerMapping (exactly one)

Names of conceptual and storage model types that are referenced in MSL must be qualified by their respective namespace names. For information about the conceptual model namespace name, see Schema Element (CSDL). For information about the storage model namespace name, see Schema Element (SSDL). Aliases for namespaces that are used in MSL can be defined with the Alias element.

Applicable Attributes

The table below describes the attributes that can be applied to the **Mapping** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Space	Yes	**C-S**. This is a fixed value and cannot be changed.

Example

The following example shows a **Mapping** element that is based on part of the School model. For more information about the School model, see Quickstart (Entity Framework):

```xml
<Mapping Space="C-S"
         xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
                          CdmEntityContainer="SchoolModelEntities">
    <EntitySetMapping Name="Courses">
      <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>
```

MappingFragment Element (MSL)

The **MappingFragment** element in mapping specification language (MSL) defines the mapping between the properties of a conceptual model entity type and a table or view in the database. For information about conceptual model entity types and underlying database tables or views, see EntityType Element (CSDL) and EntitySet Element (SSDL). The **MappingFragment** can be a child element of the EntityTypeMapping element or the EntitySetMapping element.

The **MappingFragment** element can have the following child elements:

- ComplexType (zero or more)
- ScalarProperty (zero or more)
- Condition (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **MappingFragment** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
StoreEntitySet	Yes	The name of the table or view that is being mapped.
MakeColumnsDistinct	No	**True** or **False** depending on whether only distinct rows are returned. If this attribute is set to **True**, the **GenerateUpdateViews** attribute of the EntityContainerMapping element must be set to **False**.

Example

The following example shows a **MappingFragment** element as the child of an **EntityTypeMapping** element. In this example, properties of the **Course** type in the conceptual model are mapped to columns of the **Course** table in the database.

```
<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>
```

Example

The following example shows a **MappingFragment** element as the child of an **EntitySetMapping** element. As in the example above, properties of the **Course** type in the conceptual model are mapped to columns of the **Course** table in the database.

```
<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>
```

ModificationFunctionMapping Element (MSL)

The **ModificationFunctionMapping** element in mapping specification language (MSL) maps the insert, update, and delete functions of a conceptual model entity type to stored procedures in the underlying database. The **ModificationFunctionMapping** element can also map the insert and delete functions for many-to-many associations in the conceptual model to stored procedures in the underlying database. Stored procedures to which modification functions are mapped must be declared in the storage model. For more information, see Function Element (SSDL).

NOTE

If you do not map all three of the insert, update, or delete operations of a entity type to stored procedures, the unmapped operations will fail if executed at runtime and an UpdateException is thrown.

NOTE

If the modification functions for one entity in an inheritance hierarchy are mapped to stored procedures, then modification functions for all types in the hierarchy must be mapped to stored procedures.

The **ModificationFunctionMapping** element can be a child of the EntityTypeMapping element or the AssociationSetMapping element.

The **ModificationFunctionMapping** element can have the following child elements:

- DeleteFunction (zero or one)
- InsertFunction (zero or one)

- UpdateFunction (zero or one)

No attributes are applicable to the **ModificationFunctionMapping** element.

Example

The following example shows the entity set mapping for the **People** entity set in the School model. In addition to the column mapping for the **Person** entity type, the mapping of the insert, update, and delete functions of the **Person** type are shown. The functions that are mapped to are declared in the storage model.

```
<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>
```

Example

The following example shows the association set mapping for the **CourseInstructor** association set in the School model. In addition to the column mapping for the **CourseInstructor** association, the mapping of the insert and delete functions of the **CourseInstructor** association are shown. The functions that are mapped to are declared in the storage model.

```xml
<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>
```

QueryView Element (MSL)

The **QueryView** element in mapping specification language (MSL) defines a read-only mapping between an entity type or association in the conceptual model and a table in the underlying database. The mapping is defined with an Entity SQL query that is evaluated against the storage model, and you express the result set in terms of an entity or association in the conceptual model. Because query views are read-only, you cannot use standard update commands to update types that are defined by query views. You can make updates to these types by using modification functions. For more information, see How to: Map Modification Functions to Stored Procedures.

> **NOTE**
> In the **QueryView** element, Entity SQL expressions that contain **GroupBy**, group aggregates, or navigation properties are not supported.

The **QueryView** element can be a child of the EntitySetMapping element or the AssociationSetMapping element. In the former case, the query view defines a read-only mapping for an entity in the conceptual model. In the latter case, the query view defines a read-only mapping for an association in the conceptual model.

> **NOTE**
> If the **AssociationSetMapping** element is for an association with a referential constraint, the **AssociationSetMapping** element is ignored. For more information, see ReferentialConstraint Element (CSDL).

The **QueryView** element cannot have any child elements.

Applicable Attributes

The following table describes the attributes that can be applied to the **QueryView** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
TypeName	No	The name of the conceptual model type that is being mapped by the query view.

Example

The following example shows the **QueryView** element as a child of the **EntitySetMapping** element and defines a query view mapping for the **Department** entity type in the School Model.

```
<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>
```

Because the query only returns a subset of the members of the **Department** type in the storage model, the **Department** type in the School model has been modified based on this mapping as follows:

```
<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>
```

Example

The next example shows the **QueryView** element as the child of an **AssociationSetMapping** element and defines a read-only mapping for the `FK_Course_Department` association in the School model.

```
<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>
```

Comments

You can define query views to enable the following scenarios:

- Define an entity in the conceptual model that doesn't include all the properties of the entity in the storage model. This includes properties that do not have default values and do not support **null** values.
- Map computed columns in the storage model to properties of entity types in the conceptual model.
- Define a mapping where conditions used to partition entities in the conceptual model are not based on equality. When you specify a conditional mapping using the **Condition** element, the supplied condition must equal the specified value. For more information, see Condition Element (MSL).
- Map the same column in the storage model to multiple types in the conceptual model.
- Map multiple types to the same table.
- Define associations in the conceptual model that are not based on foreign keys in the relational schema.
- Use custom business logic to set the value of properties in the conceptual model. For example, you could map the string value "T" in the data source to a value of **true**, a Boolean, in the conceptual model.
- Define conditional filters for query results.
- Enforce fewer restrictions on data in the conceptual model than in the storage model. For example, you could make a property in the conceptual model nullable even if the column to which it is mapped does not support **null**values.

The following considerations apply when you define query views for entities:

- Query views are read-only. You can only make updates to entities by using modification functions.
- When you define an entity type by a query view, you must also define all related entities by query views.
- When you map a many-to-many association to an entity in the storage model that represents a link table in the relational schema, you must define a **QueryView** element in the **AssociationSetMapping** element for this link table.

- Query views must be defined for all types in a type hierarchy. You can do this in the following ways:
  - With a single **QueryView** element that specifies a single Entity SQL query that returns a union of all of the entity types in the hierarchy.
  - With a single **QueryView** element that specifies a single Entity SQL query that uses the CASE operator to return a specific entity type in the hierarchy based on a specific condition.
  - With an additional **QueryView** element for a specific type in the hierarchy. In this case, use the **TypeName** attribute of the **QueryView** element to specify the entity type for each view.
- When a query view is defined, you cannot specify the **StorageSetName** attribute on the **EntitySetMapping** element.
- When a query view is defined, the **EntitySetMapping**element cannot also contain **Property** mappings.

ResultBinding Element (MSL)

The **ResultBinding** element in mapping specification language (MSL) maps column values that are returned by stored procedures to entity properties in the conceptual model when entity type modification functions are mapped to stored procedures in the underlying database. For example, when the value of an identity column is returned by an insert stored procedure, the **ResultBinding** element maps the returned value to an entity type property in the conceptual model.

The **ResultBinding** element can be child of the InsertFunction element or the UpdateFunction element.

The **ResultBinding** element cannot have any child elements.

Applicable Attributes

The following table describes the attributes that are applicable to the **ResultBinding** element:

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity property in the conceptual model that is being mapped.
ColumnName	Yes	The name of the column being mapped.

Example

The following example is based on the School model and shows an **InsertFunction** element used to map the insert function of the **Person** entity type to the **InsertPerson** stored procedure. (The **InsertPerson** stored procedure is shown below and is declared in the storage model.) A **ResultBinding** element is used to map a column value that is returned by the stored procedure (**NewPersonID**) to an entity type property (**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>
```

The following Transact-SQL describes the **InsertPerson** stored procedure:

```
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;
```

ResultMapping Element (MSL)

The **ResultMapping** element in mapping specification language (MSL) defines the mapping between a function import in the conceptual model and a stored procedure in the underlying database when the following are true:

- The function import returns a conceptual model entity type or complex type.
- The names of the columns returned by the stored procedure do not exactly match the names of the properties on the entity type or complex type.

By default, the mapping between the columns returned by a stored procedure and an entity type or complex type is based on column and property names. If column names do not exactly match property names, you must use the **ResultMapping** element to define the mapping. For an example of the default mapping, see FunctionImportMapping Element (MSL).

The **ResultMapping** element is a child element of the FunctionImportMapping element.

The **ResultMapping** element can have the following child elements:

- EntityTypeMapping (zero or more)
- ComplexTypeMapping

No attributes are applicable to the **ResultMapping** Element.

Example

Consider the following stored procedure:

```
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
```

Also consider the following conceptual model entity type:

```
<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>
```

In order to create a function import that returns instances of the previous entity type, the mapping between the columns returned by the stored procedure and the entity type must be defined in a **ResultMapping** element:

```
<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>
```

ScalarProperty Element (MSL)

The **ScalarProperty** element in mapping specification language (MSL) maps a property on a conceptual model entity type, complex type, or association to a table column or stored procedure parameter in the underlying database.

> **NOTE**
> Stored procedures to which modification functions are mapped must be declared in the storage model. For more information, see Function Element (SSDL).

The **ScalarProperty** element can be a child of the following elements:

- MappingFragment
- InsertFunction
- UpdateFunction
- DeleteFunction
- EndProperty
- ComplexProperty
- ResultMapping

As a child of the **MappingFragment**, **ComplexProperty**, or **EndProperty** element, the **ScalarProperty** element maps a property in the conceptual model to a column in the database. As a child of the **InsertFunction**, **UpdateFunction**, or **DeleteFunction** element, the **ScalarProperty** element maps a property in the conceptual model to a stored procedure parameter.

The **ScalarProperty** element cannot have any child elements.

Applicable Attributes

The attributes that apply to the **ScalarProperty** element differ depending on the role of the element.

The following table describes the attributes that are applicable when the **ScalarProperty** element is used to map a conceptual model property to a column in the database:

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the conceptual model property that is being mapped.
ColumnName	Yes	The name of the table column that is being mapped.

The following table describes the attributes that are applicable to the **ScalarProperty** element when it is used to map a conceptual model property to a stored procedure parameter:

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the conceptual model property that is being mapped.
ParameterName	Yes	The name of the parameter that is being mapped.
Version	No	**Current** or **Original** depending on whether the current value or the original value of the property should be used for concurrency checks.

Example

The following example shows the **ScalarProperty** element used in two ways:

- To map the properties of the **Person** entity type to the columns of the **Person**table.
- To map the properties of the **Person** entity type to the parameters of the **UpdatePerson** stored procedure. The stored procedures are declared in the storage model.

```
<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>
```

Example

The next example shows the **ScalarProperty** element used to map the insert and delete functions of a conceptual model association to stored procedures in the database. The stored procedures are declared in the storage model.

```
<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>
```

UpdateFunction Element (MSL)

The **UpdateFunction** element in mapping specification language (MSL) maps the update function of an entity type in the conceptual model to a stored procedure in the underlying database. Stored procedures to which modification functions are mapped must be declared in the storage model. For more information, see Function Element (SSDL).

> **NOTE**
>
> If you do not map all three of the insert, update, or delete operations of a entity type to stored procedures, the unmapped operations will fail if executed at runtime and an UpdateException is thrown.

The **UpdateFunction** element can be a child of the ModificationFunctionMapping element and applied to the EntityTypeMapping element.

The **UpdateFunction** element can have the following child elements:

- AssociationEnd (zero or more)
- ComplexProperty (zero or more)
- ResultBinding (zero or one)
- ScarlarProperty (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **UpdateFunction** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE

ATTRIBUTE NAME	IS REQUIRED	VALUE
FunctionName	Yes	The namespace-qualified name of the stored procedure to which the update function is mapped. The stored procedure must be declared in the storage model.
RowsAffectedParameter	No	The name of the output parameter that returns the number of rows affected.

Example

The following example is based on the School model and shows the **UpdateFunction** element used to map update function of the **Person** entity type to the **UpdatePerson** stored procedure. The **UpdatePerson** stored procedure is declared in the storage model.

```
<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>
```

SSDL Specification

9/13/2018 • 32 minutes to read • Edit Online

Store schema definition language (SSDL) is an XML-based language that describes the storage model of an Entity Framework application.

In an Entity Framework application, storage model metadata is loaded from a .ssdl file (written in SSDL) into an instance of the System.Data.Metadata.Edm.StoreItemCollection and is accessible by using methods in the System.Data.Metadata.Edm.MetadataWorkspace class. Entity Framework uses storage model metadata to translate queries against the conceptual model to store-specific commands.

The Entity Framework Designer (EF Designer) stores storage model information in an .edmx file at design time. At build time the Entity Designer uses information in an .edmx file to create the .ssdl file that is needed by Entity Framework at runtime.

Versions of SSDL are differentiated by XML namespaces.

SSDL VERSION	XML NAMESPACE
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

Association Element (SSDL)

An **Association** element in store schema definition language (SSDL) specifies table columns that participate in a foreign key constraint in the underlying database. Two required child End elements specify tables at the ends of the association and the multiplicity at each end. An optional ReferentialConstraint element specifies the principal and dependent ends of the association as well as the participating columns. If no **ReferentialConstraint** element is present, an AssociationSetMapping element must be used to specify the column mappings for the association.

The **Association** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- End (exactly two)
- ReferentialConstraint (zero or one)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **Association** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the corresponding foreign key constraint in the underlying database.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Association** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that uses a **ReferentialConstraint** element to specify the columns that participate in the **FK_CustomerOrders** foreign key constraint:

```
<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>
```

AssociationSet Element (SSDL)

The **AssociationSet** element in store schema definition language (SSDL) represents a foreign key constraint between two tables in the underlying database. The table columns that participate in the foreign key constraint are specified in an Association element. The **Association** element that corresponds to a given **AssociationSet** element is specified in the **Association** attribute of the **AssociationSet** element.

SSDL association sets are mapped to CSDL association sets by an AssociationSetMapping element. However, if the CSDL association for a given CSDL association set is defined by using a ReferentialConstraint element , no corresponding **AssociationSetMapping** element is necessary. In this case, if an **AssociationSetMapping** element is present, the mappings it defines will be overridden by the **ReferentialConstraint** element.

The **AssociationSet** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- End (zero or two)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **AssociationSet** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the foreign key constraint that the association set represents.
Association	Yes	The name of the association that defines the columns that participate in the foreign key constraint.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **AssociationSet** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **AssociationSet** element that represents the `FK_CustomerOrders` foreign key constraint in the underlying database:

```
<AssociationSet Name="FK_CustomerOrders"
                Association="ExampleModel.Store.FK_CustomerOrders">
  <End Role="Customers" EntitySet="Customers" />
  <End Role="Orders" EntitySet="Orders" />
</AssociationSet>
```

CollectionType Element (SSDL)

The **CollectionType** element in store schema definition language (SSDL) specifies that a function's return type is a collection. The **CollectionType** element is a child of the ReturnType element. The type of collection is specified by using the RowType child element:

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **CollectionType** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a function that uses a **CollectionType** element to specify that the function returns a collection of rows.

```
  <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>
```

CommandText Element (SSDL)

The **CommandText** element in store schema definition language (SSDL) is a child of the Function element that allows you to define a SQL statement that is executed at the database. The **CommandText** element allows you to add functionality that is similar to a stored procedure in the database, but you define the **CommandText** element in the storage model.

The **CommandText** element cannot have child elements. The body of the **CommandText** element must be a valid SQL statement for the underlying database.

No attributes are applicable to the **CommandText** element.

Example

The following example shows a **Function** element with a child **CommandText** element. Expose the **UpdateProductInOrder** function as a method on the ObjectContext by importing it into the conceptual model.

```
<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>
```

DefiningQuery Element (SSDL)

The **DefiningQuery** element in store schema definition language (SSDL) allows you to execute a SQL statement directly in the underlying database. The **DefiningQuery** element is commonly used like a database view, but the view is defined in the storage model instead of the database. The view defined in a **DefiningQuery** element can be mapped to an entity type in the conceptual model through an EntitySetMapping element. These mappings are read-only.

The following SSDL syntax shows the declaration of an **EntitySet** followed by the **DefiningQuery** element that contains a query used to retrieve the view.

```
<Schema>
    <EntitySet Name="Tables" EntityType="Self.STable">
        <DefiningQuery>
          SELECT  TABLE_CATALOG,
                  'test' as TABLE_SCHEMA,
                  TABLE_NAME
          FROM    INFORMATION_SCHEMA.TABLES
        </DefiningQuery>
    </EntitySet>
</Schema>
```

You can use stored procedures in the Entity Framework to enable read-write scenarios over views. You can use either a data source view or an Entity SQL view as the base table for retrieving data and for change processing by stored procedures.

You can use the **DefiningQuery** element to target Microsoft SQL Server Compact 3.5. Though SQL Server Compact 3.5 does not support stored procedures, you can implement similar functionality with the **DefiningQuery** element. Another place where it can be useful is in creating stored procedures to overcome a mismatch between the data types used in the programming language and those of the data source. You could write a **DefiningQuery** that takes a certain set of parameters and then calls a stored procedure with a different set of parameters, for example, a stored procedure that deletes data.

Dependent Element (SSDL)

The **Dependent** element in store schema definition language (SSDL) is a child element to the

ReferentialConstraint element that defines the dependent end of a foreign key constraint (also called a referential constraint). The **Dependent** element specifies the column (or columns) in a table that reference a primary key column (or columns). **PropertyRef** elements specify which columns are referenced. The Principal element specifies the primary key columns that are referenced by columns that are specified in the **Dependent** element.

The **Dependent** element can have the following child elements (in the order listed):

- PropertyRef (one or more)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **Dependent** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Role	Yes	The same value as the **Role** attribute (if used) of the corresponding End element; otherwise, the name of the table that contains the referencing column.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Dependent** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an Association element that uses a **ReferentialConstraint** element to specify the columns that participate in the **FK_CustomerOrders** foreign key constraint. The **Dependent** element specifies the **CustomerId** column of the **Order** table as the dependent end of the constraint.

```
<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>
```

Documentation Element (SSDL)

The **Documentation** element in store schema definition language (SSDL) can be used to provide information about an object that is defined in a parent element.

The **Documentation** element can have the following child elements (in the order listed):

- **Summary**: A brief description of the parent element. (zero or one element)

- **LongDescription**: An extensive description of the parent element. (zero or one element)

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **Documentation** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows the **Documentation** element as a child element of an EntityType element.

```
<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>
```

End Element (SSDL)

The **End** element in store schema definition language (SSDL) specifies the table and number of rows at one end of a foreign key constraint in the underlying database. The **End** element can be a child of the Association element or the AssociationSet element. In each case, the possible child elements and applicable attributes are different.

End Element as a Child of the Association Element

An **End** element (as a child of the **Association** element) specifies the table and number of rows at the end of a foreign key constraint with the **Type** and **Multiplicity** attributes respectively. Ends of a foreign key constraint are defined as part of an SSDL association; an SSDL association must have exactly two ends.

An **End** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- OnDelete (zero or one element)
- Annotation elements (zero or more elements)

Applicable Attributes

The following table describes the attributes that can be applied to the **End** element when it is the child of an **Association** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Type	Yes	The fully qualified name of the SSDL entity set that is at the end of the foreign key constraint.
Role	No	The value of the **Role** attribute in either the Principal or Dependent element of the corresponding ReferentialConstraint element (if used).

ATTRIBUTE NAME	IS REQUIRED	VALUE
Multiplicity	Yes	**1**, **0..1**, or ***** depending on the number of rows that can be at the end of the foreign key constraint. **1** indicates that exactly one row exists at the foreign key constraint end. **0..1** indicates that zero or one row exists at the foreign key constraint end. ***** indicates that zero, one, or more rows exist at the foreign key constraint end.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **End** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that defines the **FK_CustomerOrders** foreign key constraint. The **Multiplicity** values specified on each **End** element indicate that many rows in the **Orders** table can be associated with a row in the **Customers** table, but only one row in the **Customers** table can be associated with a row in the **Orders** table. Additionally, the **OnDelete** element indicates that all rows in the **Orders** table that reference a particular row in the **Customers** table will be deleted if the row in the **Customers** table is deleted.

```
<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>
```

End Element as a Child of the AssociationSet Element

The **End** element (as a child of the **AssociationSet** element) specifies a table at one end of a foreign key constraint in the underlying database.

An **End** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **End** element when it is the child of an **AssociationSet** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
EntitySet	Yes	The name of the SSDL entity set that is at the end of the foreign key constraint.
Role	No	The value of one of the **Role** attributes specified on one **End** element of the corresponding Association element.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **End** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element with an **AssociationSet** element with two **End** elements:

```
<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>
```

EntityContainer Element (SSDL)

An **EntityContainer** element in store schema definition language (SSDL) describes the structure of the underlying data source in an Entity Framework application: SSDL entity sets (defined in EntitySet elements) represent tables in a database, SSDL entity types (defined in EntityType elements) represent rows in a table, and association sets (defined in AssociationSet elements) represent foreign key constraints in a database. A storage model entity container maps to a conceptual model entity container through the EntityContainerMapping element.

An **EntityContainer** element can have zero or one Documentation elements. If a **Documentation** element is present, it must precede all other child elements.

An **EntityContainer** element can have zero or more of the following child elements (in the order listed):

- EntitySet
- AssociationSet
- Annotation elements

Applicable Attributes

The table below describes the attributes that can be applied to the **EntityContainer** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity container. This name cannot contain periods (.).

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntityContainer** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element that defines two entity sets and one association set. Note that entity type and association type names are qualified by the conceptual model namespace name.

```
<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>
```

EntitySet Element (SSDL)

An **EntitySet** element in store schema definition language (SSDL) represents a table or view in the underlying database. An EntityType element in SSDL represents a row in the table or view. The **EntityType** attribute of an **EntitySet** element specifies the particular SSDL entity type that represents rows in an SSDL entity set. The mapping between a CSDL entity set and an SSDL entity set is specified in an EntitySetMapping element.

The **EntitySet** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- DefiningQuery (zero or one element)
- Annotation elements

Applicable Attributes

The following table describes the attributes that can be applied to the **EntitySet** element.

> **NOTE**
> Some attributes (not listed here) may be qualified with the **store** alias. These attributes are used by the Update Model Wizard when updating a model.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity set.

ATTRIBUTE NAME	IS REQUIRED	VALUE
EntityType	Yes	The fully-qualified name of the entity type for which the entity set contains instances.
Schema	No	The database schema.
Table	No	The database table.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntitySet** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityContainer** element that has two **EntitySet** elements and one **AssociationSet** element:

```
<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>
```

EntityType Element (SSDL)

An **EntityType** element in store schema definition language (SSDL) represents a row in a table or view of the underlying database. An EntitySet element in SSDL represents the table or view in which rows occur. The **EntityType** attribute of an **EntitySet** element specifies the particular SSDL entity type that represents rows in an SSDL entity set. The mapping between an SSDL entity type and a CSDL entity type is specified in an EntityTypeMapping element.

The **EntityType** element can have the following child elements (in the order listed):

- Documentation (zero or one element)
- Key (zero or one element)
- Annotation elements

Applicable Attributes

The table below describes the attributes that can be applied to the **EntityType** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the entity type. This value is usually the same as the name of the table in which the entity type represents a row. This value can contain no periods (.).

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **EntityType** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityType** element with two properties:

```
<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>
```

Function Element (SSDL)

The **Function** element in store schema definition language (SSDL) specifies a stored procedure that exists in the underlying database.

The **Function** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- Parameter (zero or more)
- CommandText (zero or one)
- ReturnType (zero or more)
- Annotation elements (zero or more)

A return type for a function must be specified with either the **ReturnType** element or the **ReturnType** attribute (see below), but not both.

Stored procedures that are specified in the storage model can be imported into the conceptual model of an application. For more information, see Querying with Stored Procedures. The **Function** element can also be used to define custom functions in the storage model.

Applicable Attributes

The following table describes the attributes that can be applied to the **Function** element.

> **NOTE**
>
> Some attributes (not listed here) may be qualified with the **store** alias. These attributes are used by the Update Model Wizard when updating a model.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the stored procedure.
ReturnType	No	The return type of the stored procedure.
Aggregate	No	**True** if the stored procedure returns an aggregate value; otherwise **False**.
BuiltIn	No	**True** if the function is a built-in[1] function; otherwise **False**.
StoreFunctionName	No	The name of the stored procedure.
NiladicFunction	No	**True** if the function is a niladic[2] function; **False** otherwise.
IsComposable	No	**True** if the function is a composable[3] function; **False** otherwise.
ParameterTypeSemantics	No	The enumeration that defines the type semantics used to resolve function overloads. The enumeration is defined in the provider manifest per function definition. The default value is **AllowImplicitConversion**.
Schema	No	The name of the schema in which the stored procedure is defined.

[1] A built-in function is a function that is defined in the database. For information about functions that are defined in the storage model, see CommandText Element (SSDL).

[2] A niladic function is a function that accepts no parameters and, when called, does not require parentheses.

[3] Two functions are composable if the output of one function can be the input for the other function.

> **NOTE**
>
> Any number of annotation attributes (custom XML attributes) may be applied to the **Function** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **Function** element that corresponds to the **UpdateOrderQuantity** stored procedure. The stored procedure accepts two parameters and does not return a value.

```
<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>
```

Key Element (SSDL)

The **Key** element in store schema definition language (SSDL) represents the primary key of a table in the underlying database. **Key** is a child element of an EntityType element, which represents a row in a table. The primary key is defined in the **Key** element by referencing one or more Property elements that are defined on the **EntityType** element.

The **Key** element can have the following child elements (in the order listed):

- PropertyRef (one or more)
- Annotation elements

No attributes are applicable to the **Key** element.

Example

The following example shows an **EntityType** element with a key that references one property:

```
<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>
```

OnDelete Element (SSDL)

The **OnDelete** element in store schema definition language (SSDL) reflects the database behavior when a row that participates in a foreign key constraint is deleted. If the action is set to **Cascade**, then rows that reference a row that is being deleted will also be deleted. If the action is set to **None**, then rows that reference a row that is being deleted are not also deleted. An **OnDelete** element is a child element of an End element.

An **OnDelete** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **OnDelete** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Action	Yes	**Cascade** or **None**. (The value **Restricted** is valid but has the same behavior as **None**.)

NOTE

Any number of annotation attributes (custom XML attributes) may be applied to the **OnDelete** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that defines the **FK_CustomerOrders** foreign key constraint. The **OnDelete** element indicates that all rows in the **Orders** table that reference a particular row in the **Customers** table will be deleted if the row in the **Customers** table is deleted.

```
<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>
```

Parameter Element (SSDL)

The **Parameter** element in store schema definition language (SSDL) is a child of the Function element that specifies parameters for a stored procedure in the database.

The **Parameter** element can have the following child elements (in the order listed):

- Documentation (zero or one)
- Annotation elements (zero or more)

Applicable Attributes

The table below describes the attributes that can be applied to the **Parameter** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the parameter.
Type	Yes	The parameter type.
Mode	No	**In**, **Out**, or **InOut** depending on whether the parameter is an input, output, or input/output parameter.

ATTRIBUTE NAME	IS REQUIRED	VALUE
MaxLength	No	The maximum length of the parameter.
Precision	No	The precision of the parameter.
Scale	No	The scale of the parameter.
SRID	No	Spatial System Reference Identifier. Valid only for parameters of spatial types. For more information, see SRID and SRID (SQL Server).

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Parameter** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **Function** element that has two **Parameter** elements that specify input parameters:

```
<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>
```

Principal Element (SSDL)

The **Principal** element in store schema definition language (SSDL) is a child element to the ReferentialConstraint element that defines the principal end of a foreign key constraint (also called a referential constraint). The **Principal** element specifies the primary key column (or columns) in a table that is referenced by another column (or columns). **PropertyRef** elements specify which columns are referenced. The Dependent element specifies columns that reference the primary key columns that are specified in the **Principal** element.

The **Principal** element can have the following child elements (in the order listed):

- PropertyRef (one or more)
- Annotation elements (zero or more)

Applicable Attributes

The following table describes the attributes that can be applied to the **Principal** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE

ATTRIBUTE NAME	IS REQUIRED	VALUE
Role	Yes	The same value as the **Role** attribute (if used) of the corresponding End element; otherwise, the name of the table that contains the referenced column.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **Principal** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an Association element that uses a **ReferentialConstraint** element to specify the columns that participate in the **FK_CustomerOrders** foreign key constraint. The **Principal** element specifies the **CustomerId** column of the **Customer** table as the principal end of the constraint.

```
<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>
```

Property Element (SSDL)

The **Property** element in store schema definition language (SSDL) represents a column in a table in the underlying database. **Property** elements are children of EntityType elements, which represent rows in a table. Each **Property** element defined on an **EntityType** element represents a column.

A **Property** element cannot have any child elements.

Applicable Attributes

The following table describes the attributes that can be applied to the **Property** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the corresponding column.
Type	Yes	The type of the corresponding column.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Nullable	No	**True** (the default value) or **False** depending on whether the corresponding column can have a null value.
DefaultValue	No	The default value of the corresponding column.
MaxLength	No	The maximum length of the corresponding column.
FixedLength	No	**True** or **False** depending on whether the corresponding column value will be stored as a fixed length string.
Precision	No	The precision of the corresponding column.
Scale	No	The scale of the corresponding column.
Unicode	No	**True** or **False** depending on whether the corresponding column value will be stored as a Unicode string.
Collation	No	A string that specifies the collating sequence to be used in the data source.
SRID	No	Spatial System Reference Identifier. Valid only for properties of spatial types. For more information, see SRID and SRID (SQL Server).
StoreGeneratedPattern	No	**None**, **Identity** (if the corresponding column value is an identity that is generated in the database), or **Computed** (if the corresponding column value is computed in the database). Not Valid for RowType properties.

NOTE

Any number of annotation attributes (custom XML attributes) may be applied to the **Property** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **EntityType** element with two child **Property** elements:

```
<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>
```

PropertyRef Element (SSDL)

The **PropertyRef** element in store schema definition language (SSDL) references a property defined on an EntityType element to indicate that the property will perform one of the following roles:

- Be part of the primary key of the table that the **EntityType** represents. One or more **PropertyRef** elements can be used to define a primary key. For more information, see Key element.
- Be the dependent or principal end of a referential constraint. For more information, see ReferentialConstraint element.

The **PropertyRef** element can only have the following child elements:

- Documentation (zero or one)
- Annotation elements

Applicable Attributes

The table below describes the attributes that can be applied to the **PropertyRef** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Name	Yes	The name of the referenced property.

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **PropertyRef** element. However, custom attributes may not belong to any XML namespace that is reserved for CSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows a **PropertyRef** element used to define a primary key by referencing a property that is defined on an **EntityType** element.

```
<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>
```

ReferentialConstraint Element (SSDL)

The **ReferentialConstraint** element in store schema definition language (SSDL) represents a foreign key constraint (also called a referential integrity constraint) in the underlying database. The principal and dependent ends of the constraint are specified by the Principal and Dependent child elements, respectively. Columns that participate in the principal and dependent ends are referenced with PropertyRef elements.

The **ReferentialConstraint** element is an optional child element of the Association element. If a **ReferentialConstraint** element is not used to map the foreign key constraint that is specified in the **Association** element, an AssociationSetMapping element must be used to do this.

The **ReferentialConstraint** element can have the following child elements:

- Documentation (zero or one)
- Principal (exactly one)
- Dependent (exactly one)
- Annotation elements (zero or more)

Applicable Attributes

Any number of annotation attributes (custom XML attributes) may be applied to the **ReferentialConstraint** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example shows an **Association** element that uses a **ReferentialConstraint** element to specify the columns that participate in the **FK_CustomerOrders** foreign key constraint:

```
<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>
```

ReturnType Element (SSDL)

The **ReturnType** element in store schema definition language (SSDL) specifies the return type for a function that is defined in a **Function** element. A function return type can also be specified with a **ReturnType** attribute.

The return type of a function is specified with the **Type** attribute or the **ReturnType** element.

The **ReturnType** element can have the following child elements:

- CollectionType (one)

> **NOTE**
> Any number of annotation attributes (custom XML attributes) may be applied to the **ReturnType** element. However, custom attributes may not belong to any XML namespace that is reserved for SSDL. The fully-qualified names for any two custom attributes cannot be the same.

Example

The following example uses a **Function** that returns a collection of rows.

```
<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>
```

RowType Element (SSDL)

A **RowType** element in store schema definition language (SSDL) defines an unnamed structure as a return type for a function defined in the store.

A **RowType** element is the child element of **CollectionType** element:

A **RowType** element can have the following child elements:

- Property (one or more)

Example

The following example shows a store function that uses a **CollectionType** element to specify that the function returns a collection of rows (as specified in the **RowType** element).

```
<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>
```

Schema Element (SSDL)

The **Schema** element in store schema definition language (SSDL) is the root element of a storage model definition. It contains definitions for the objects, functions, and containers that make up a storage model.

The **Schema** element may contain zero or more of the following child elements:

- Association
- EntityType
- EntityContainer
- Function

The **Schema** element uses the **Namespace** attribute to define the namespace for the entity type and association objects in a storage model. Within a namespace, no two objects can have the same name.

A storage model namespace is different from the XML namespace of the **Schema** element. A storage model namespace (as defined by the **Namespace** attribute) is a logical container for entity types and association types. The XML namespace (indicated by the **xmlns** attribute) of a **Schema** element is the default namespace for child elements and attributes of the **Schema** element. XML namespaces of the form http://schemas.microsoft.com/ado/YYYY/MM/edm/ssdl (where YYYY and MM represent a year and month respectively) are reserved for SSDL. Custom elements and attributes cannot be in namespaces that have this form.

Applicable Attributes

The table below describes the attributes can be applied to the **Schema** element.

ATTRIBUTE NAME	IS REQUIRED	VALUE
Namespace	Yes	The namespace of the storage model. The value of the **Namespace** attribute is used to form the fully qualified name of a type. For example, if an **EntityType** named *Customer* is in the ExampleModel.Store namespace, then the fully qualified name of the **EntityType** is ExampleModel.Store.Customer. The following strings cannot be used as the value for the **Namespace** attribute: **System**, **Transient**, or **Edm**. The value for the **Namespace** attribute cannot be the same as the value for the **Namespace** attribute in the CSDL Schema element.
Alias	No	An identifier used in place of the namespace name. For example, if an **EntityType** named *Customer* is in the ExampleModel.Store namespace and the value of the **Alias** attribute is *StorageModel*, then you can use StorageModel.Customer as the fully qualified name of the **EntityType.**
Provider	Yes	The data provider.
ProviderManifestToken	Yes	A token that indicates to the provider which provider manifest to return. No format for the token is defined. Values for the token are defined by the provider. For information about SQL Server provider manifest tokens, see SqlClient for Entity Framework.

Example

The following example shows a **Schema** element that contains an **EntityContainer** element, two **EntityType** elements, and one **Association** element.

```
<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">
    <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>
  <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>
```

Annotation Attributes

Annotation attributes in store schema definition language (SSDL) are custom XML attributes in the storage model that provide extra metadata about the elements in the storage model. In addition to having valid XML structure, the following constraints apply to annotation attributes:

- Annotation attributes must not be in any XML namespace that is reserved for SSDL.
- The fully-qualified names of any two annotation attributes must not be the same.

More than one annotation attribute may be applied to a given SSDL element. Metadata contained in annotation elements can be accessed at runtime by using classes in the System.Data.Metadata.Edm namespace.

Example

The following example shows an EntityType element that has an annotation attribute applied to the **OrderId** property. The example also show an annotation element added to the **EntityType** element.

```
<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>
```

Annotation Elements (SSDL)

Annotation elements in store schema definition language (SSDL) are custom XML elements in the storage model that provide extra metadata about the storage model. In addition to having valid XML structure, the following constraints apply to annotation elements:

- Annotation elements must not be in any XML namespace that is reserved for SSDL.
- The fully-qualified names of any two annotation elements must not be the same.
- Annotation elements must appear after all other child elements of a given SSDL element.

More than one annotation element may be a child of a given SSDL element. Starting with the .NET Framework version 4, metadata contained in annotation elements can be accessed at runtime by using classes in the

System.Data.Metadata.Edm namespace.

Example

The following example shows an EntityType element that has an annotation element (**CustomElement**). The example also shows an annotation attribute applied to the **OrderId** property.

```
<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>
```

Facets (SSDL)

Facets in store schema definition language (SSDL) represent constraints on column types that are specified in Property elements. Facets are implemented as XML attributes on **Property** elements.

The following table describes the facets that are supported in SSDL:

FACET	DESCRIPTION
Collation	Specifies the collating sequence (or sorting sequence) to be used when performing comparison and ordering operations on values of the property.
FixedLength	Specifies whether the length of the column value can vary.
MaxLength	Specifies the maximum length of the column value.
Precision	For properties of type **Decimal**, specifies the number of digits a property value can have. For properties of type **Time**, **DateTime**, and **DateTimeOffset**, specifies the number of digits for the fractional part of seconds of the column value.
Scale	Specifies the number of digits to the right of the decimal point for the column value.
Unicode	Indicates whether the column value is stored as Unicode.

Defining Query - EF Designer

9/13/2018 • 5 minutes to read • Edit Online

This walkthrough demonstrates how to add a defining query and a corresponding entity type to a model using the EF Designer. A defining query is commonly used to provide functionality similar to that provided by a database view, but the view is defined in the model, not the database. A defining query allows you to execute a SQL statement that is specified in the **DefiningQuery** element of an .edmx file. For more information, see **DefiningQuery** in the SSDL Specification.

When using defining queries, you also have to define an entity type in your model. The entity type is used to surface data exposed by the defining query. Note that data surfaced through this entity type is read-only.

Parameterized queries cannot be executed as defining queries. However, the data can be updated by mapping the insert, update, and delete functions of the entity type that surfaces the data to stored procedures. For more information, see Insert, Update, and Delete with Stored Procedures.

This topic shows how to perform the following tasks.

- Add a Defining Query
- Add an Entity Type to the Model
- Map the Defining Query to the Entity Type

Prerequisites

To complete this walkthrough, you will need:

- A recent version of Visual Studio.
- The School sample database.

Set up the Project

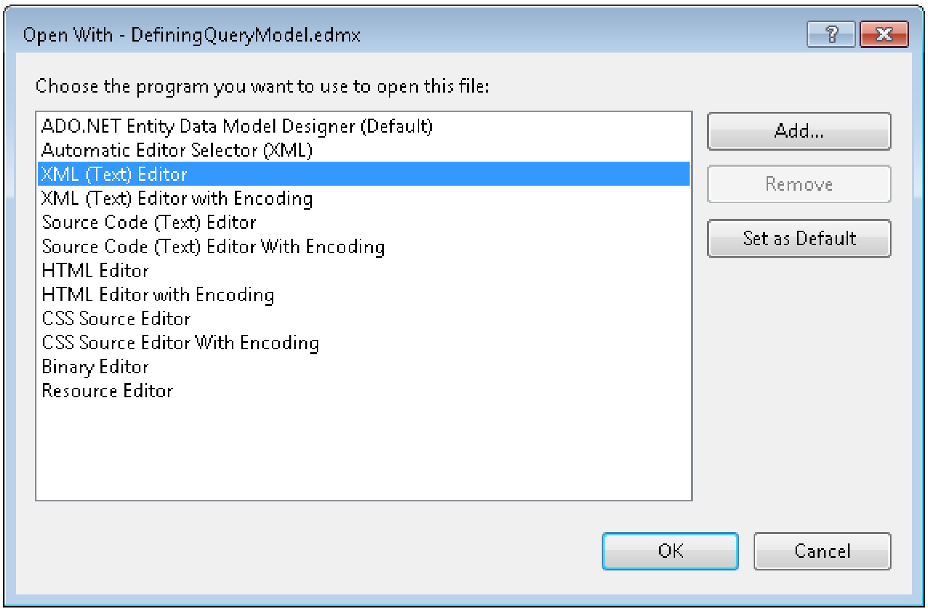
This walkthrough is using Visual Studio 2012 or newer.

- Open Visual Studio.
- On the **File** menu, point to **New**, and then click **Project**.
- In the left pane, click **Visual C#**, and then select the **Console Application** template.
- Enter **DefiningQuerySample** as the name of the project and click **OK**.

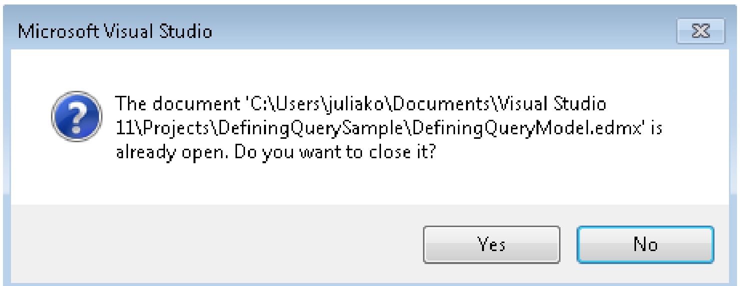
Create a Model based on the School Database

- Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**.
- Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the Templates pane.
- Enter **DefiningQueryModel.edmx** for the file name, and then click **Add**.
- In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next**.
- Click New Connection. In the Connection Properties dialog box, enter the server name (for example, **(localdb)\mssqllocaldb**), select the authentication method, type **School** for the database name, and then click **OK**. The Choose Your Data Connection dialog box is updated with your database connection setting.

- In the Choose Your Database Objects dialog box, check the **Tables** node. This will add all the tables to the **School** model.
- Click **Finish**.
- In Solution Explorer, right-click the **DefiningQueryModel.edmx** file and select **Open With...**.
- Select **XML (Text) Editor**.



- Click **Yes** if prompted with the following message:



Add a Defining Query

In this step we will use the XML Editor to add a defining query and an entity type to the SSDL section of the .edmx file.

- Add an **EntitySet** element to the SSDL section of the .edmx file (line 5 thru 13). Specify the following:
  - Only the **Name** and **EntityType** attributes of the **EntitySet** element are specified.
  - The fully-qualified name of the entity type is used in the **EntityType** attribute.
  - The SQL statement to be executed is specified in the **DefiningQuery** element.

```
    <!-- 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">
           <EntitySet 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>
          </EntitySet>
          <EntitySet Name="Course" EntityType="SchoolModel.Store.Course" store:Type="Tables" Schema="dbo" />
```

- Add the **EntityType** element to the SSDL section of the .edmx. file as shown below. Note the following:
  - The value of the **Name** attribute corresponds to the value of the **EntityType** attribute in the **EntitySet** element above, although the fully-qualified name of the entity type is used in the **EntityType** attribute.
  - The property names correspond to the column names returned by the SQL statement in the **DefiningQuery** element (above).
  - In this example, the entity key is composed of three properties to ensure a unique key value.

```
    <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

If later you run the **Update Model Wizard** dialog, any changes made to the storage model, including defining queries, will be overwritten.

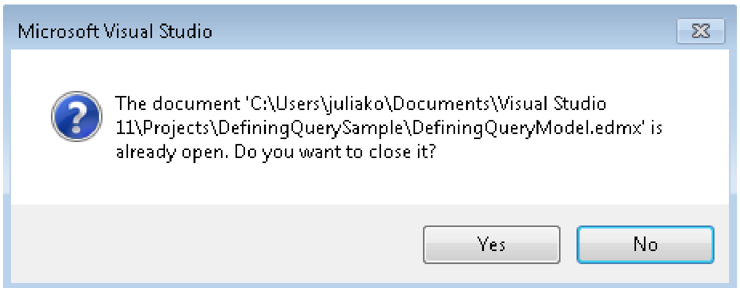
Add an Entity Type to the Model

In this step we will add the entity type to the conceptual model using the EF Designer. Note the following:

- The **Name** of the entity corresponds to the value of the **EntityType** attribute in the **EntitySet** element above.
- The property names correspond to the column names returned by the SQL statement in the **DefiningQuery** element above.
- In this example, the entity key is composed of three properties to ensure a unique key value.

Open the model in the EF Designer.

- Double-click the DefiningQueryModel.edmx.
- Say **Yes** to the following message:



The Entity Designer, which provides a design surface for editing your model, is displayed.

- Right-click the designer surface and select **Add New**->**Entity…**.
- Specify **GradeReport** for the entity name and **CourseID** for the **Key Property**.
- Right-click the **GradeReport** entity and select **Add New**-> **Scalar Property**.
- Change the default name of the property to **FirstName**.
- Add another scalar property and specify **LastName** for the name.
- Add another scalar property and specify **Grade** for the name.
- In the **Properties** window, change the **Grade**'s **Type** property to **Decimal**.
- Select the **FirstName** and **LastName** properties.
- In the **Properties** window, change the **EntityKey** property value to **True**.

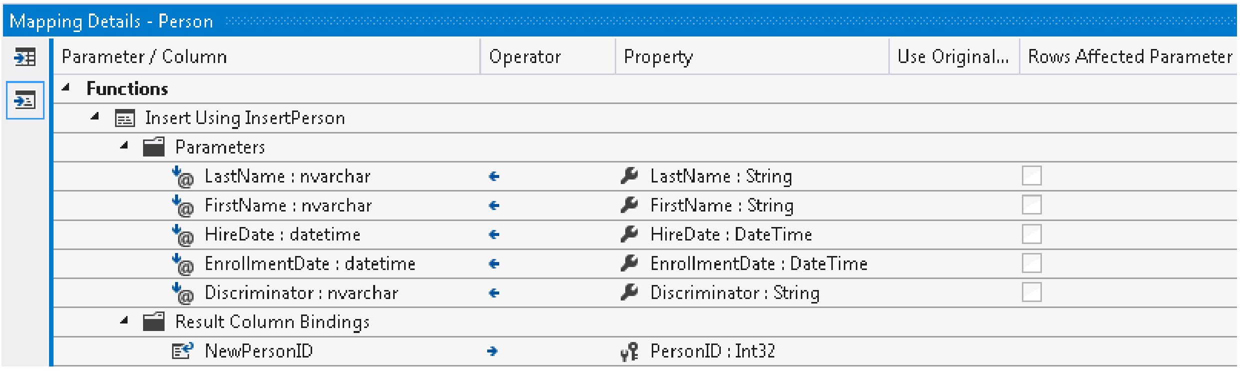
As a result, the following elements were added to the **CSDL** section of the .edmx file.

```
<EntitySet Name="GradeReport" EntityType="SchoolModel.GradeReport" />

<EntityType Name="GradeReport">
. . .
</EntityType>
```

Map the Defining Query to the Entity Type

In this step, we will use the Mapping Details window to map the conceptual and storage entity types.

- Right-click the **GradeReport** entity on the design surface and select **Table Mapping**.
  The **Mapping Details** window is displayed.
- Select **GradeReport** from the **<Add a Table or View>** dropdown list (located under **Table**s).
  Default mappings between the conceptual and storage **GradeReport** entity type appear.



As a result, the **EntitySetMapping** element is added to the mapping section of the .edmx file.

```
<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>
```

- Compile the application.

Call the Defining Query in your Code

You can now execute the defining query by using the **GradeReport** entity type.

```
using (var context = new SchoolEntities())
{
    var report = context.GradeReports.FirstOrDefault();
    Console.WriteLine("{0} {1} got {2}",
        report.FirstName, report.LastName, report.Grade);
}
```

Stored Procedures with Multiple Result Sets

9/13/2018 • 5 minutes to read • Edit Online

Sometimes when using stored procedures you will need to return more than one result set. This scenario is commonly used to reduce the number of database round trips required to compose a single screen. Prior to EF5, Entity Framework would allow the stored procedure to be called but would only return the first result set to the calling code.

This article will show you two ways that you can use to access more than one result set from a stored procedure in Entity Framework. One that uses just code and works with both Code first and the EF Designer and one that only works with the EF Designer. The tooling and API support for this should improve in future versions of Entity Framework.

Model

The examples in this article use a basic Blog and Posts model where a blog has many posts and a post belongs to a single blog. We will use a stored procedure in the database that returns all blogs and posts, something like this:

```
CREATE PROCEDURE [dbo].[GetAllBlogsAndPosts]
AS
    SELECT * FROM dbo.Blogs
    SELECT * FROM dbo.Posts
```

Accessing Multiple Result Sets with Code

We can execute use code to issue a raw SQL command to execute our stored procedure. The advantage of this approach is that it works with both Code first and the EF Designer.

In order to get multiple result sets working we need to drop to the ObjectContext API by using the IObjectContextAdapter interface.

Once we have an ObjectContext then we can use the Translate method to translate the results of our stored procedure into entities that can be tracked and used in EF as normal. The following code sample demonstrates this in action.

```
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();
    }
}
```

The Translate method accepts the reader that we received when we executed the procedure, an EntitySet name, and a MergeOption. The EntitySet name will be the same as the DbSet property on your derived context. The MergeOption enum controls how results are handled if the same entity already exists in memory.

Here we iterate through the collection of blogs before we call NextResult, this is important given the above code because the first result set must be consumed before moving to the next result set.

Once the two translate methods are called then the Blog and Post entities are tracked by EF the same way as any other entity and so can be modified or deleted and saved as normal.

> **NOTE**
>
> EF does not take any mapping into account when it creates entities using the Translate method. It will simply match column names in the result set with property names on your classes.

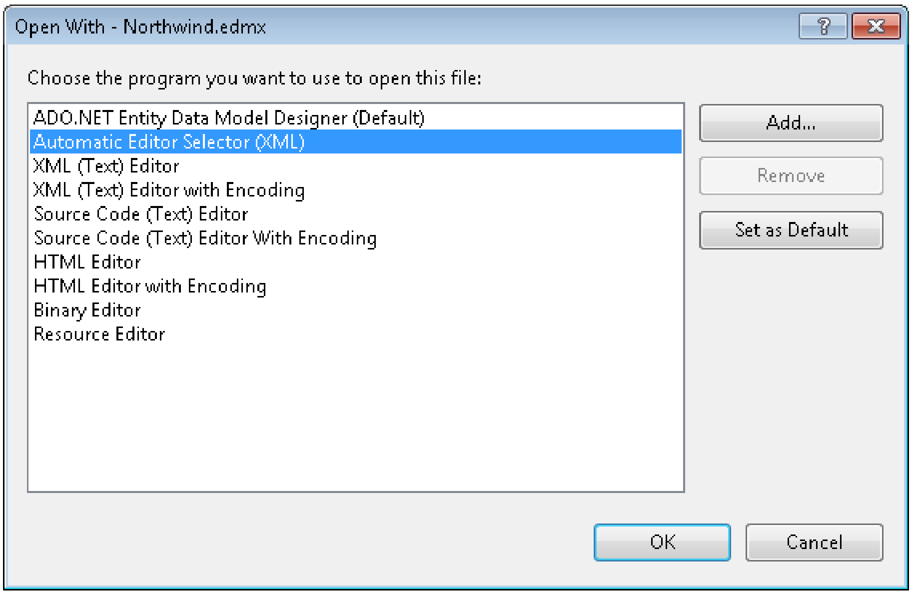
> **NOTE**
>
> That if you have lazy loading enabled, accessing the posts property on one of the blog entities then EF will connect to the database to lazily load all posts, even though we have already loaded them all. This is because EF cannot know whether or not you have loaded all posts or if there are more in the database. If you want to avoid this then you will need to disable lazy loading.

Multiple Result Sets with Configured in EDMX

> **NOTE**
>
> You must target .NET Framework 4.5 to be able to configure multiple result sets in EDMX. If you are targeting .NET 4.0 you can use the code-based method shown in the previous section.

If you are using the EF Designer, you can also modify your model so that it knows about the different result sets that will be returned. One thing to know before hand is that the tooling is not multiple result set aware, so you will need to manually edit the edmx file. Editing the edmx file like this will work but it will also break the validation of the model in VS. So if you validate your model you will always get errors.

- In order to do this you need to add the stored procedure to your model as you would for a single result set query.
- Once you have this then you need to right click on your model and select **Open With..** then **Xml**



Once you have the model opened as XML then you need to do the following steps:

- Find the complex type and function import in your model:

```
    <!-- 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>
```

- Remove the complex type
- Update the function import so that it maps to your entities, in our case it will look like the following:

```
    <FunctionImport Name="GetAllBlogsAndPosts">
      <ReturnType EntitySet="Blogs" Type="Collection(BlogModel.Blog)" />
      <ReturnType EntitySet="Posts" Type="Collection(BlogModel.Post)" />
    </FunctionImport>
```

This tells the model that the stored procedure will return two collections, one of blog entries and one of post entries.

- Find the function mapping element:

```
    <!-- 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>
```

- Replace the result mapping with one for each entity being returned, such as the following:

```
<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>
```

It is also possible to map the result sets to complex types, such as the one created by default. To do this you create a new complex type, instead of removing them, and use the complex types everywhere that you had used the entity names in the examples above.

Once these mappings have been changed then you can save the model and execute the following code to use the stored procedure:

```
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

If you manually edit the edmx file for your model it will be overwritten if you ever regenerate the model from the database.

Summary

Here we have shown two different methods of accessing multiple result sets using Entity Framework. Both of them are equally valid depending on your situation and preferences and you should choose the one that seems best for your circumstances. It is planned that the support for multiple result sets will be improved in future versions of Entity Framework and that performing the steps in this document will no longer be necessary.

Table-Valued Functions (TVFs)

9/18/2018 • 3 minutes to read • Edit Online

> **NOTE**
>
> **EF5 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 5. If you are using an earlier version, some or all of the information does not apply.

The video and step-by-step walkthrough shows how to map table-valued functions (TVFs) using the Entity Framework Designer. It also demonstrates how to call a TVF from a LINQ query.

TVFs are currently only supported in the Database First workflow.

TVF support was introduced in Entity Framework version 5. Note that to use the new features like table-valued functions, enums, and spatial types you must target .NET Framework 4.5. Visual Studio 2012 targets .NET 4.5 by default.

TVFs are very similar to stored procedures with one key difference: the result of a TVF is composable. That means the results from a TVF can be used in a LINQ query while the results of a stored procedure cannot.

Watch the video

Presented By: Julia Kornich

WMV | MP4 | WMV (ZIP)

Pre-Requisites

To complete this walkthrough, you need to:

- Install the School database.
- Have a recent version of Visual Studio

Set up the Project

1. Open Visual Studio
2. On the **File** menu, point to **New**, and then click **Project**
3. In the left pane, click **Visual C#**, and then select the **Console** template
4. Enter **TVF** as the name of the project and click **OK**

Add a TVF to the Database

- Select **View -> SQL Server Object Explorer**
- If LocalDB is not in the list of servers: Right-click on **SQL Server** and select **Add SQL Server** Use the default **Windows Authentication** to connect to the LocalDB server
- Expand the LocalDB node
- Under the Databases node, right-click the School database node and select **New Query...**
- In T-SQL Editor, paste the following TVF definition

```
CREATE FUNCTION [dbo].[GetStudentGradesForCourse]

(@CourseID INT)

RETURNS TABLE

RETURN
    SELECT [EnrollmentID],
           [CourseID],
           [StudentID],
           [Grade]
    FROM   [dbo].[StudentGrade]
    WHERE  CourseID = @CourseID
```

- Click the right mouse button on the T-SQL editor and select **Execute**
- The GetStudentGradesForCourse function is added to the School database

Create a Model

1. Right-click the project name in Solution Explorer, point to **Add**, and then click **New Item**
2. Select **Data** from the left menu and then select **ADO.NET Entity Data Model** in the **Templates** pane
3. Enter **TVFModel.edmx** for the file name, and then click **Add**
4. In the Choose Model Contents dialog box, select **Generate from database**, and then click **Next**
5. Click **New Connection** Enter **(localdb)\mssqllocaldb** in the Server name text box Enter **School** for the database name Click **OK**
6. In the Choose Your Database Objects dialog box, under the **Tables** node, select the **Person**, **StudentGrade**, and **Course** tables
7. Select the **GetStudentGradesForCourse** function located under the **Stored Procedures and Functions** node Note, that starting with Visual Studio 2012, the Entity Designer allows you to batch import your Stored Procedures and Functions
8. Click **Finish**
9. The Entity Designer, which provides a design surface for editing your model, is displayed. All the objects that you selected in the **Choose Your Database Objects** dialog box are added to the model.
10. By default, the result shape of each imported stored procedure or function will automatically become a new complex type in your entity model. But we want to map the results of the GetStudentGradesForCourse function to the StudentGrade entity: Right-click the design surface and select **Model Browser** In Model Browser, select **Function Imports**, and then double-click the **GetStudentGradesForCourse** function In the Edit Function Import dialog box, select **Entities** and choose **StudentGrade**

Persist and Retrieve Data

Open the file where the Main method is defined. Add the following code into the Main function.

The following code demonstrates how to build a query that uses a Table-valued Function. The query projects the results into an anonymous type that contains the related Course title and related students with a grade greater or equal to 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);
    }
}
```

Compile and run the application. The program produces the following output:

```
Couse: Microeconomics, Student: Arturo Anand
Couse: Microeconomics, Student: Carson Bryant
```

Summary

In this walkthrough we looked at how to map Table-valued Functions (TVFs) using the Entity Framework Designer. It also demonstrated how to call a TVF from a LINQ query.

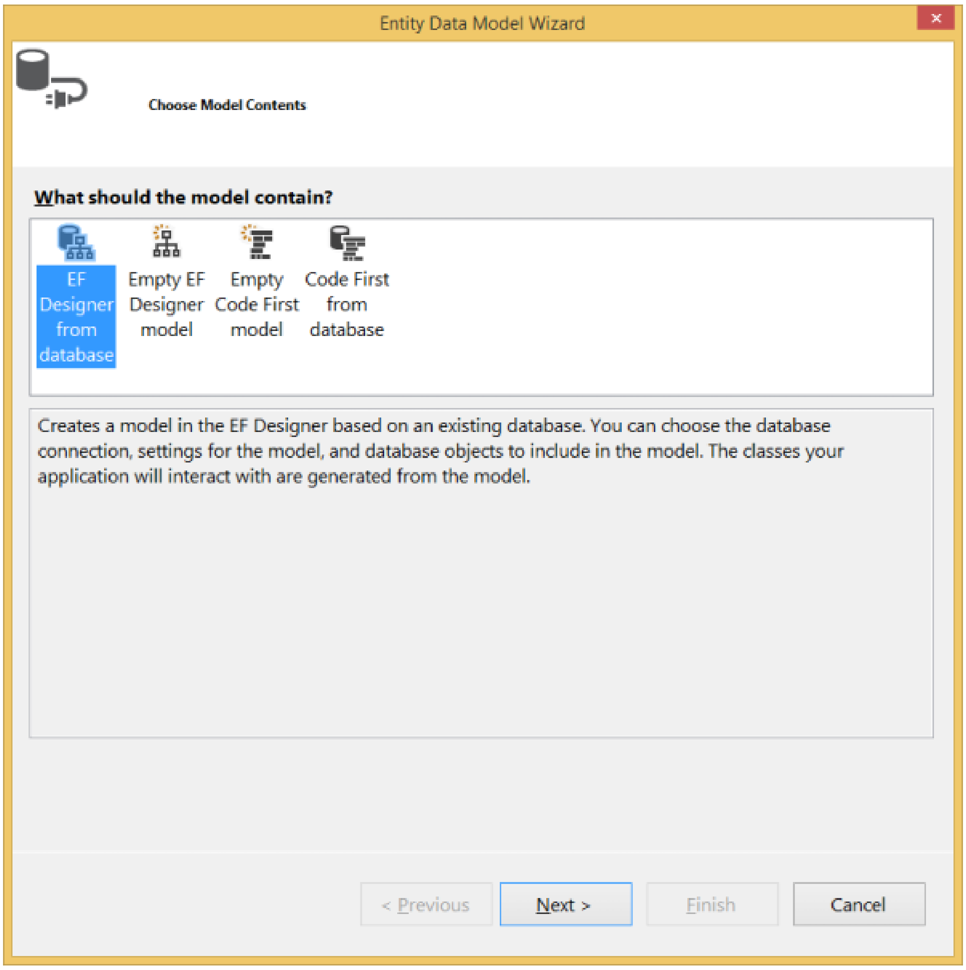
Entity Framework Designer Keyboard Shortcuts

9/13/2018 • 6 minutes to read • Edit Online

This page provides a list of keyboard shorcuts that are available in the various screens of the Entity Framework Tools for Visual Studio.

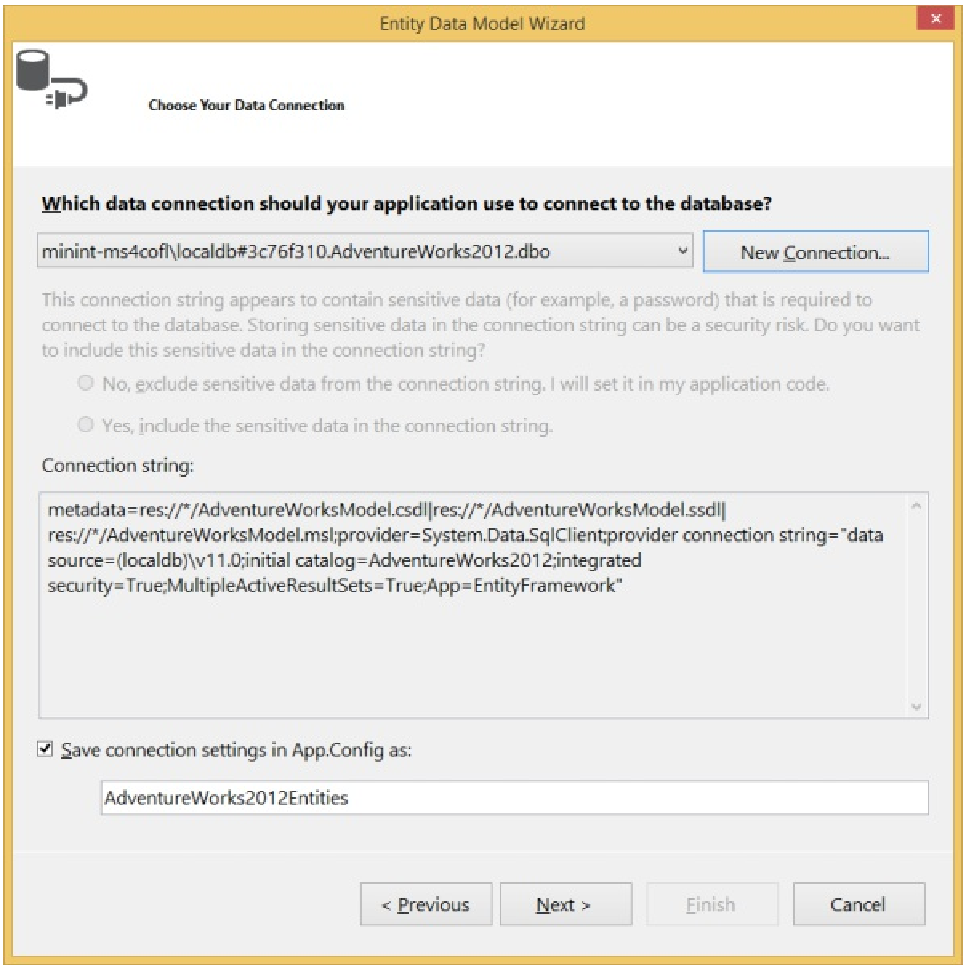
ADO.NET Entity Data Model Wizard

Step One: Choose Model Contents



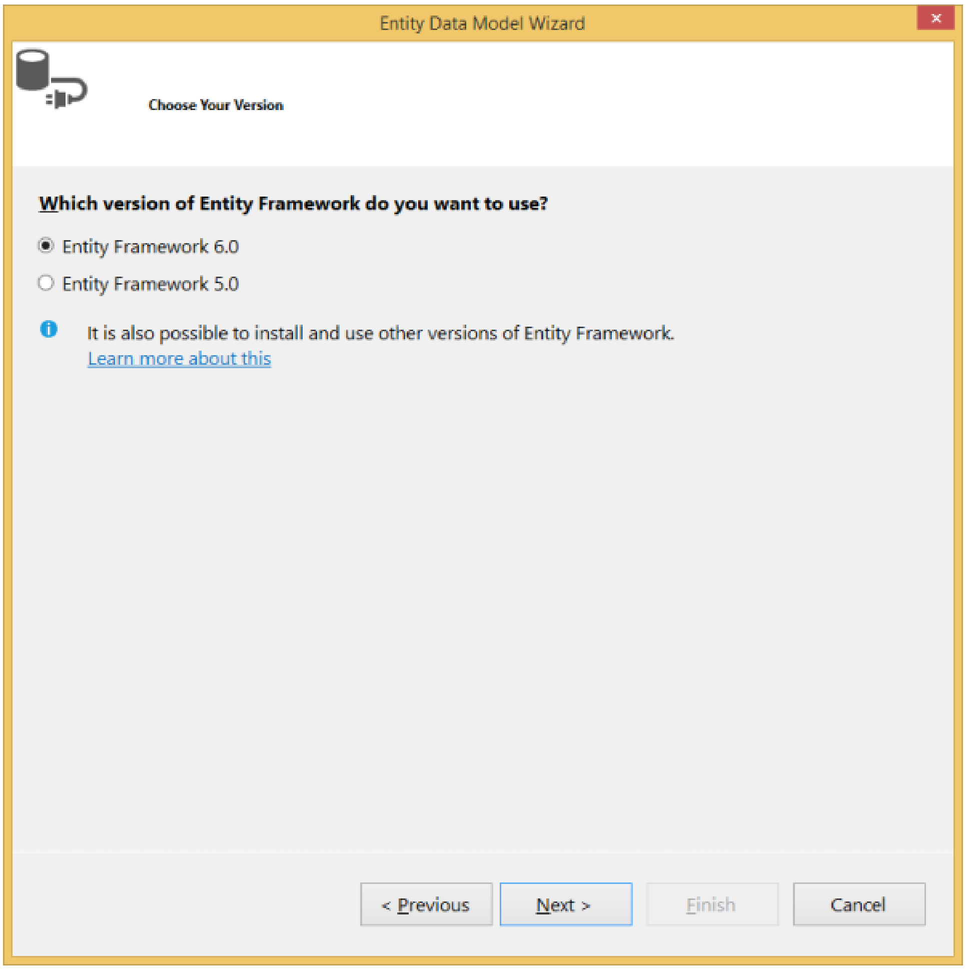
SHORTCUT	ACTION	NOTES
Alt+n	Move to next screen	Not available for all selections of model contents.
Alt+f	Finish wizard	Not available for all selections of model contents.
Alt+w	Switch focus to the “What should the model contain?” pane.	

Step Two: Choose Your Connection



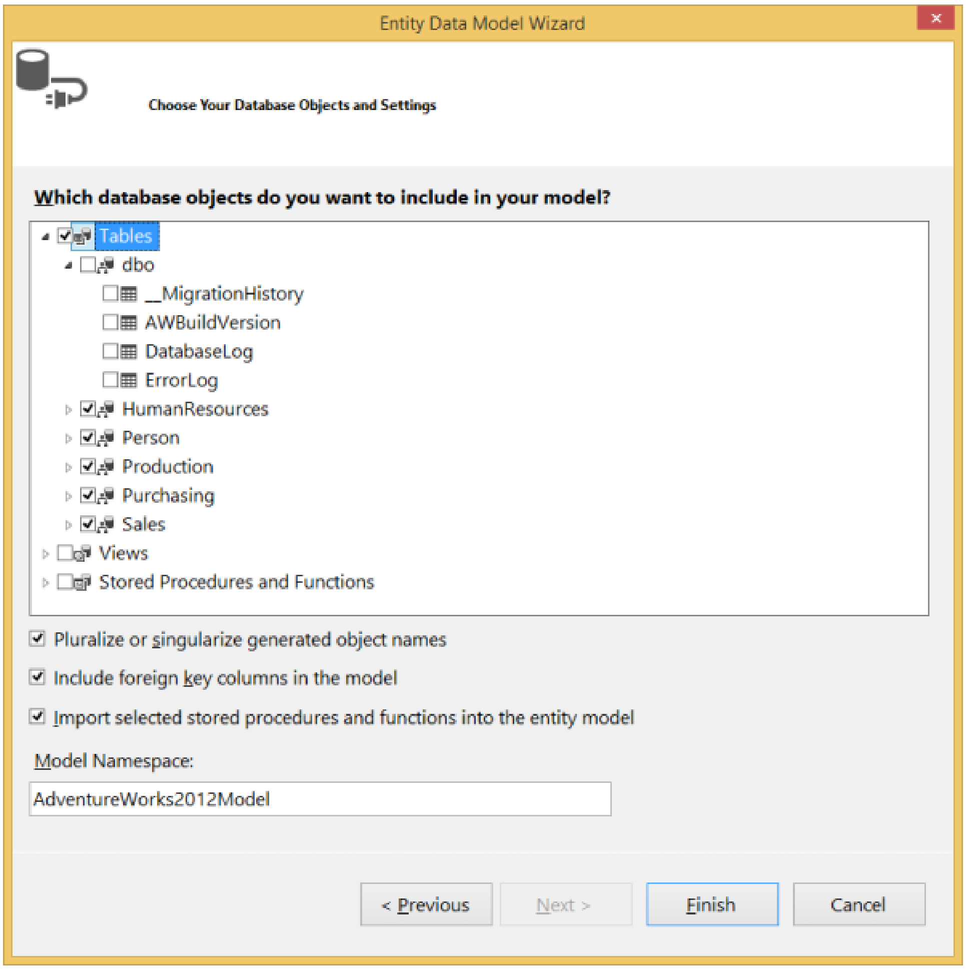
SHORTCUT	ACTION	NOTES
Alt+n	Move to next screen	
Alt+p	Move to previous screen	
Alt+w	Switch focus to the “What should the model contain?” pane.	
Alt+c	Open the “Connection Properties” window	Allows for the definition of a new database connection.
Alt+e	Exclude sensitive data from the connection string	
Alt+i	Include sensitive data in the connection string	
Alt+s	Toggle the “Save connection settings in App.Config” option	

Step Three: Choose Your Version



SHORTCUT	ACTION	NOTES
Alt+n	Move to next screen	
Alt+p	Move to previous screen	
Alt+w	Switch focus to Entity Framework version selection	Allows for specifying a different version of Entity Framework for use in the project.

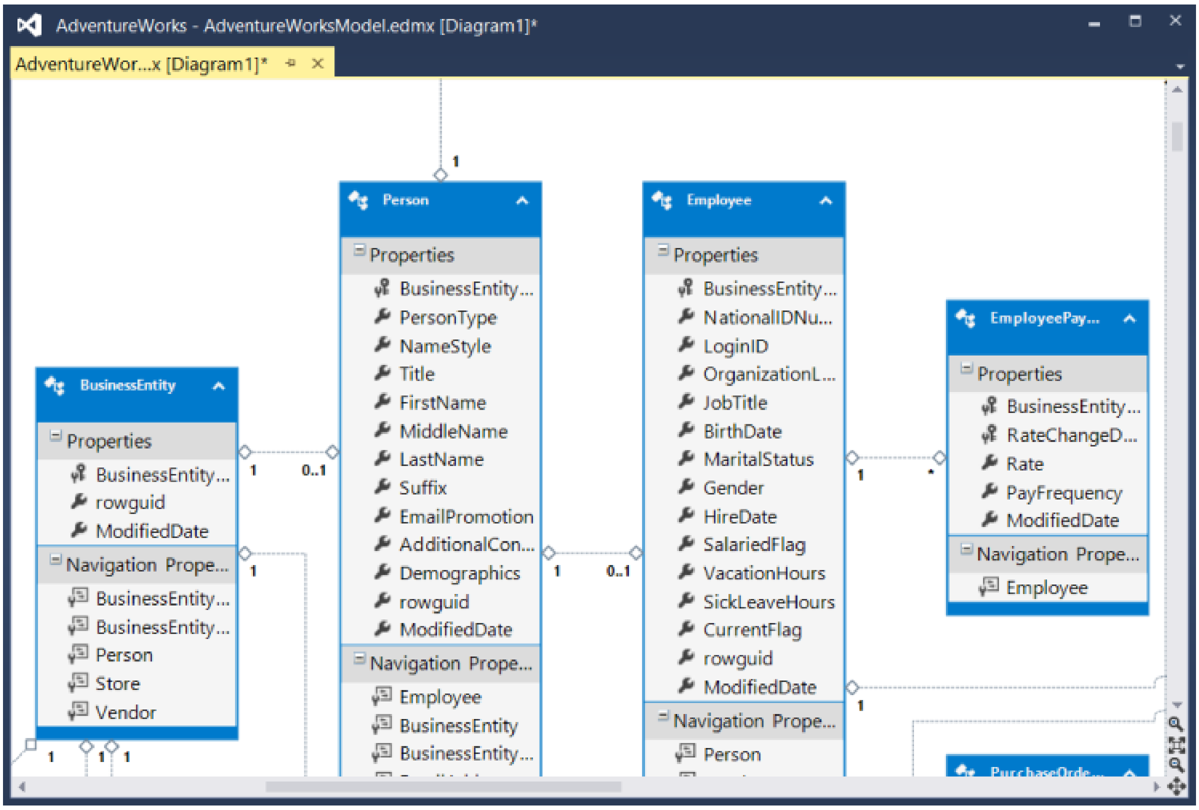
Step Four: Choose Your Database Objects and Settings



SHORTCUT	ACTION	NOTES
Alt+f	Finish wizard	
Alt+p	Move to previous screen	
Alt+w	Switch focus to database object selection pane	Allows for specifying database objects to be reverse engineered.
Alt+s	Toggle the “Pluralize or singularize generated object names” option	
Alt+k	Toggle the “Include foreign key columns in the model” option	Not available for all selections of model contents.
Alt+i	Toggle the “Import selected stored procedures and functions into the entity model” option	Not available for all selections of model contents.
Alt+m	Switches focus to the “Model Namespace” text field	Not available for all selections of model contents.
Space	Toggle selection on element	If element has children, all child elements will be toggled as well
Left	Collapse child tree	
Right	Expand child tree	

SHORTCUT	ACTION	NOTES
Up	Navigate to previous element in tree	
Down	Navigate to next element in tree	

EF Designer Surface



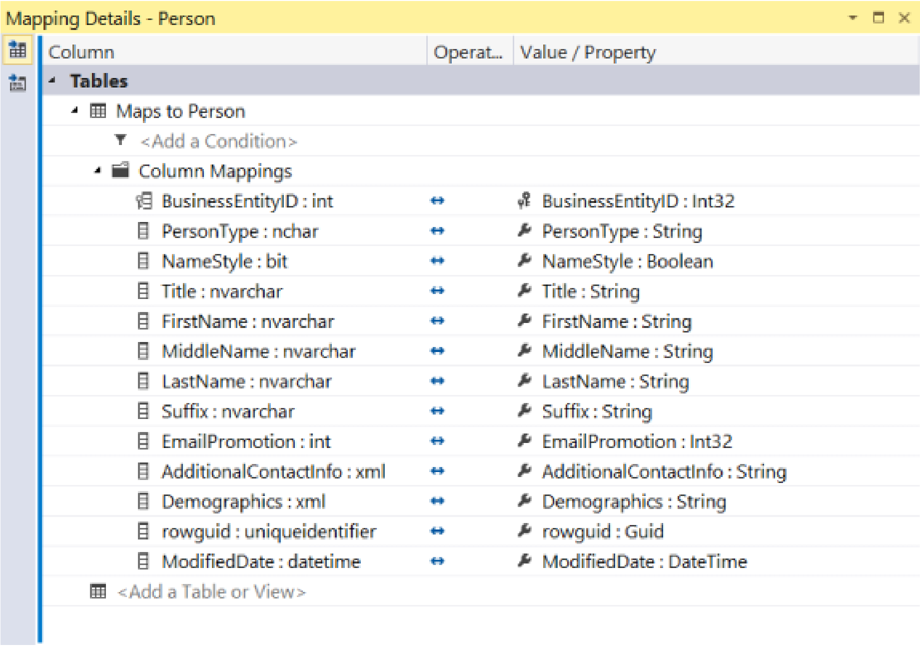
SHORTCUT	ACTION	NOTES
Space/Enter	Toggle Selection	Toggles selection on the object with focus.
Esc	Cancel Selection	Cancels the current selection.
Ctrl + A	Select All	Selects all the shapes on the design surface.
Up arrow	Move up	Moves selected entity up one grid increment. If in a list, moves to the previous sibling subfield.
Down arrow	Move down	Moves selected entity down one grid increment. If in a list, moves to the next sibling subfield.
Left arrow	Move left	Moves selected entity left one grid increment. If in a list, moves to the previous sibling subfield.

SHORTCUT	ACTION	NOTES
Right arrow	Move right	Moves selected entity right one grid increment. If in a list, moves to the next sibling subfield.
Shift + left arrow	Size shape left	Reduces the width of the selected entity by one grid increment.
Shift + right arrow	Size shape right	Increases the width of the selected entity by one grid increment.
Home	First Peer	Moves focus and selection to the first object on the design surface at the same peer level.
End	Last Peer	Moves focus and selection to the last object on the design surface at the same peer level.
Ctrl + Home	First Peer (focus)	Same as first peer, but moves focus instead of moving focus and selection.
Ctrl + End	Last Peer (focus)	Same as last peer, but moves focus instead of moving focus and selection.
Tab	Next Peer	Moves focus and selection to the next object on the design surface at the same peer level.
Shift+Tab	Previous Peer	Moves focus and selection to the previous object on the design surface at the same peer level.
Alt+Ctrl+Tab	Next Peer (focus)	Same as next peer, but moves focus instead of moving focus and selection.
Alt+Ctrl+Shift+Tab	Previous Peer (focus)	Same as previous peer, but moves focus instead of moving focus and selection.
<	Ascend	Moves to the next object on the design surface one level higher in the hierarchy. If there are no shapes above this shape in the hierarchy (that is, the object is placed directly on the design surface), the diagram is selected.
>	Descend	Moves to the next contained object on the design surface one level below this one in the hierarchy. If there are no contained object, this is a no-op.
Ctrl + <	Ascend (focus)	Same as ascend command, but moves focus without selection.

SHORTCUT	ACTION	NOTES
Ctrl + >	Descend (focus)	Same as descend command, but moves focus without selection.
Shift + End	Follow to connected	From an entity, moves to an entity which this entity is connected to.
Del	Delete	Delete an object or connector from the diagram.
Ins	Insert	Adds a new property to an entity when either the “Scalar Properties” compartment header or a property itself is selected.
Pg Up	Scroll diagram up	Scrolls the design surface up, in increments equal to 75% of the height of the currently visible design surface.
Pg Down	Scroll diagram down	Scrolls the design surface down.
Shift + Pg Down	Scroll diagram right	Scrolls the design surface to the right.
Shift + Pg Up	Scroll diagram left	Scrolls the design surface to the left.
F2	Enter edit mode	Standard keyboard shortcut for entering edit mode for a text control.
Shift + F10	Display shortcut menu	Standard keyboard shortcut for displaying a selected item’s shortcut menu.
Control + Shift + Mouse Left Click **Control + Shift + MouseWheel forward**	Semantic Zoom In	Zooms in on the area of the Diagram View beneath the mouse pointer.
Control + Shift + Mouse Right Click **Control + Shift + MouseWheel backward**	Semantic Zoom Out	Zooms out from the area of the Diagram View beneath the mouse pointer. It re-centers the diagram when you zoom out too far for the current diagram center.
Control + Shift + '+' **Control + MouseWheel forward**	Zoom In	Zooms in on the center of the Diagram View.
Control + Shift + '-' **Control + MouseWheel backward**	Zoom Out	Zooms out from the clicked area of the Diagram View. It re-centers the diagram when you zoom out too far for the current diagram center.
Control + Shift + Draw a rectangle with the left mouse button down	Zoom Area	Zooms in centered on the area that you've selected. When you hold down the Control + Shift keys, you'll see that the cursor changes to a magnifying glass, which allows you to define the area to zoom into.

SHORTCUT	ACTION	NOTES
Context Menu Key + 'M'	Open Mapping Details Window	Opens the Mapping Details window to edit mappings for selected entity

Mapping Details Window



SHORTCUT	ACTION	NOTES
Tab	Switch Context	Switches between the main window area and the toolbar on the left
Arrow keys	Navigation	Move up and down rows, or right and left across columns in the main window area. Move between the buttons in the toolbar on the left.
Enter **Space**	Select	Selects a button in the toolbar on the left.
Alt + Down Arrow	Open List	Drop down a list if a cell is selected that has a drop down list.
Enter	List Select	Selects an element in a drop down list.
Esc	List Close	Closes a drop down list.

Visual Studio Navigation

Entity Framework also supplies a number of actions that can have custom keyboard shortcuts mapped (no shortcuts are mapped by default). To create these custom shortcuts, click on the Tools menu, then Options. Under Environment, choose Keyboard. Scroll down the list in the middle until you can select the desired command, enter the shortcut in the "Press shortcut keys" text box, and click Assign. The possible shortcuts are as follows:

SHORTCUT
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
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

SHORTCUT
OtherContextMenus.MicrosoftDataEntityDesignContext.GenerateDatabasefromModel
OtherContextMenus.MicrosoftDataEntityDesignContext.GoToDefinition
OtherContextMenus.MicrosoftDataEntityDesignContext.Grid.ShowGrid
OtherContextMenus.MicrosoftDataEntityDesignContext.Grid.SnaptoGrid
OtherContextMenus.MicrosoftDataEntityDesignContext.IncludeRelated
OtherContextMenus.MicrosoftDataEntityDesignContext.MappingDetails
OtherContextMenus.MicrosoftDataEntityDesignContext.ModelBrowser
OtherContextMenus.MicrosoftDataEntityDesignContext.MoveDiagramstoSeparateFile
OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Down
OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Down5
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.BaseType
OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Entity
OtherContextMenus.MicrosoftDataEntityDesignContext.Select.Property

SHORTCUT
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
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

SHORTCUT
View.EntityDataModelBrowser
View.EntityDataModelMappingDetails

Querying and Finding Entities

9/13/2018 • 3 minutes to read • Edit Online

This topic covers the various ways you can query for data using Entity Framework, including LINQ and the Find method. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Finding entities using a query

DbSet and IDbSet implement IQueryable and so can be used as the starting point for writing a LINQ query against the database. This is not the appropriate place for an in-depth discussion of LINQ, but here are a couple of simple examples:

```
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();
}
```

Note that DbSet and IDbSet always create queries against the database and will always involve a round trip to the database even if the entities returned already exist in the context. A query is executed against the database when:

- It is enumerated by a **foreach** (C#) or **For Each** (Visual Basic) statement.
- It is enumerated by a collection operation such as ToArray, ToDictionary, or ToList.
- LINQ operators such as First or Any are specified in the outermost part of the query.
- The following methods are called: the Load extension method on a DbSet, DbEntityEntry.Reload, and Database.ExecuteSqlCommand.

When results are returned from the database, objects that do not exist in the context are attached to the context. If an object is already in the context, the existing object is returned (the current and original values of the object's properties in the entry are **not** overwritten with database values).

When you perform a query, entities that have been added to the context but have not yet been saved to the database are not returned as part of the result set. To get the data that is in the context, see Local Data.

If a query returns no rows from the database, the result will be an empty collection, rather than **null**.

Finding entities using primary keys

The Find method on DbSet uses the primary key value to attempt to find an entity tracked by the context. If the entity is not found in the context then a query will be sent to the database to find the entity there. Null is returned if the entity is not found in the context or in the database.

Find is different from using a query in two significant ways:

- A round-trip to the database will only be made if the entity with the given key is not found in the context.
- Find will return entities that are in the Added state. That is, Find will return entities that have been added to the

context but have not yet been saved to the database.

Finding an entity by primary key

The following code shows some uses of 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");
}
```

Finding an entity by composite primary key

Entity Framework allows your entities to have composite keys - that's a key that is made up of more than one property. For example, you could have a BlogSettings entity that represents a users settings for a particular blog. Because a user would only ever have one BlogSettings for each blog you could chose to make the primary key of BlogSettings a combination of BlogId and Username. The following code attempts to find the BlogSettings with BlogId = 3 and Username = "johndoe1987":

```
using (var context = new BloggingContext())
{
    var settings = context.BlogSettings.Find(3, "johndoe1987");
}
```

Note that when you have composite keys you need to use ColumnAttribute or the fluent API to specify an ordering for the properties of the composite key. The call to Find must use this order when specifying the values that form the key.

The Load Method

10/25/2018 • 2 minutes to read • Edit Online

There are several scenarios where you may want to load entities from the database into the context without immediately doing anything with those entities. A good example of this is loading entities for data binding as described in Local Data. One common way to do this is to write a LINQ query and then call ToList on it, only to immediately discard the created list. The Load extension method works just like ToList except that it avoids the creation of the list altogether.

The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Here are two examples of using Load. The first is taken from a Windows Forms data binding application where Load is used to query for entities before binding to the local collection, as described in Local Data:

```
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);

    _context = new ProductContext();

    _context.Categories.Load();
    categoryBindingSource.DataSource = _context.Categories.Local.ToBindingList();
}
```

The second example shows using Load to load a filtered collection of related entities, as described in Loading Related Entities:

```
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();
}
```

Local Data

9/13/2018 • 10 minutes to read • Edit Online

Running a LINQ query directly against a DbSet will always send a query to the database, but you can access the data that is currently in-memory using the DbSet.Local property. You can also access the extra information EF is tracking about your entities using the DbContext.Entry and DbContext.ChangeTracker.Entries methods. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Using Local to look at local data

The Local property of DbSet provides simple access to the entities of the set that are currently being tracked by the context and have not been marked as Deleted. Accessing the Local property never causes a query to be sent to the database. This means that it is usually used after a query has already been performed. The Load extension method can be used to execute a query so that the context tracks the results. For example:

```
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);
    }
}
```

If we had two blogs in the database - 'ADO.NET Blog' with a BlogId of 1 and 'The Visual Studio Blog' with a BlogId of 2 - we could expect the following output:

```
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
```

This illustrates three points:

- The new blog 'My New Blog' is included in the Local collection even though it has not yet been saved to the database. This blog has a primary key of zero because the database has not yet generated a real key for the entity.
- The 'ADO.NET Blog' is not included in the local collection even though it is still being tracked by the context. This is because we removed it from the DbSet thereby marking it as deleted.
- When DbSet is used to perform a query the blog marked for deletion (ADO.NET Blog) is included in the results and the new blog (My New Blog) that has not yet been saved to the database is not included in the results. This is because DbSet is performing a query against the database and the results returned always reflect what is in the database.

Using Local to add and remove entities from the context

The Local property on DbSet returns an ObservableCollection with events hooked up such that it stays in sync with the contents of the context. This means that entities can be added or removed from either the Local collection or the DbSet. It also means that queries that bring new entities into the context will result in the Local collection being updated with those entities. For example:

```
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);
    }
}
```

Assuming we had a few posts tagged with 'entity-framework' and 'asp.net' the output may look something like this:

```
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
```

This illustrates three points:

- The new post 'What's New in EF' that was added to the Local collection becomes tracked by the context in the Added state. It will therefore be inserted into the database when SaveChanges is called.
- The post that was removed from the Local collection (EF Beginners Guide) is now marked as deleted in the context. It will therefore be deleted from the database when SaveChanges is called.

- The additional post (ASP.NET Beginners Guide) loaded into the context with the second query is automatically added to the Local collection.

One final thing to note about Local is that because it is an ObservableCollection performance is not great for large numbers of entities. Therefore if you are dealing with thousands of entities in your context it may not be advisable to use Local.

Using Local for WPF data binding

The Local property on DbSet can be used directly for data binding in a WPF application because it is an instance of ObservableCollection. As described in the previous sections this means that it will automatically stay in sync with the contents of the context and the contents of the context will automatically stay in sync with it. Note that you do need to pre-populate the Local collection with data for there to be anything to bind to since Local never causes a database query.

This is not an appropriate place for a full WPF data binding sample but the key elements are:

- Setup a binding source
- Bind it to the Local property of your set
- Populate Local using a query to the database.

WPF binding to navigation properties

If you are doing master/detail data binding you may want to bind the detail view to a navigation property of one of your entities. An easy way to make this work is to use an ObservableCollection for the navigation property. For example:

```
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; }
    }
}
```

Using Local to clean up entities in SaveChanges

In most cases entities removed from a navigation property will not be automatically marked as deleted in the context. For example, if you remove a Post object from the Blog.Posts collection then that post will not be automatically deleted when SaveChanges is called. If you need it to be deleted then you may need to find these dangling entities and mark them as deleted before calling SaveChanges or as part of an overridden SaveChanges. For example:

```
public override int SaveChanges()
{
    foreach (var post in this.Posts.Local.ToList())
    {
        if (post.Blog == null)
        {
            this.Posts.Remove(post);
        }
    }

    return base.SaveChanges();
}
```

The code above uses the Local collection to find all posts and marks any that do not have a blog 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 query directly against the Local property without using ToList first.

Using Local and ToBindingList for Windows Forms data binding

Windows Forms does not support full fidelity data binding using ObservableCollection directly. However, you can still use the DbSet Local property for data binding to get all the benefits described in the previous sections. This is achieved through the ToBindingList extension method which creates an IBindingList implementation backed by the Local ObservableCollection.

This is not an appropriate place for a full Windows Forms data binding sample but the key elements are:

- Setup an object binding source
- Bind it to the Local property of your set using Local.ToBindingList()
- Populate Local using a query to the database

Getting detailed information about tracked entities

Many of the examples in this series use the Entry method to return a DbEntityEntry instance for an entity. This entry object then acts as the starting point for gathering information about the entity such as its current state, as well as for performing operations on the entity such as explicitly loading a related entity.

The Entries methods return DbEntityEntry objects for many or all entities being tracked by the context. This allows you to gather information or perform operations on many entities rather than just a single entry. For example:

```
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);
    }
}
```

You'll notice we are introducing a Author and Reader class into the example - both of these classes implement the IPerson interface.

```
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; }
}
```

Let's assume we have the following data in the database:

Blog with BlogId = 1 and Name = 'ADO.NET Blog'
Blog with BlogId = 2 and Name = 'The Visual Studio Blog'
Blog with BlogId = 3 and Name = '.NET Framework Blog'
Author with AuthorId = 1 and Name = 'Joe Bloggs'
Reader with ReaderId = 1 and Name = 'John Doe'

The output from running the code would be:

```
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
```

These examples illustrate several points:

- The Entries methods return entries for entities in all states, including Deleted. Compare this to Local which excludes Deleted entities.
- Entries for all entity types are returned when the non-generic Entries method is used. When the generic entries method is used entries are only returned for entities that are instances of the generic type. This was used above to get entries for all blogs. It was also used to get entries for all entities that implement IPerson. This demonstrates that the generic type does not have to be an actual entity type.
- LINQ to Objects can be used to filter the results returned. This was used above to find entities of any type as long as they are modified.

Note that DbEntityEntry instances always contain a non-null Entity. Relationship entries and stub entries are not represented as DbEntityEntry instances so there is no need to filter for these.

No-Tracking Queries

9/13/2018 • 2 minutes to read • Edit Online

Sometimes you may want to get entities back from a query but not have those entities be tracked by the context. This may result in better performance when querying for large numbers of entities in read-only scenarios. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

A new extension method AsNoTracking allows any query to be run in this way. For example:

```
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();
}
```

Raw SQL Queries

9/18/2018 • 2 minutes to read • Edit Online

Entity Framework allows you to query using LINQ with your entity classes. However, there may be times that you want to run queries using raw SQL directly against the database. This includes calling stored procedures, which can be helpful for Code First models that currently do not support mapping to stored procedures. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Writing SQL queries for entities

The SqlQuery method on DbSet allows a raw SQL query to be written that will return entity instances. The returned objects will be tracked by the context just as they would be if they were returned by a LINQ query. For example:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();
}
```

Note that, just as for LINQ queries, the query is not executed until the results are enumerated—in the example above this is done with the call to ToList.

Care should be taken whenever raw SQL queries are written for two reasons. First, the query should be written to ensure that it only returns entities that are really of the requested type. For example, when using features such as inheritance it is easy to write a query that will create entities that are of the wrong CLR type.

Second, some types of raw SQL query expose potential security risks, especially around SQL injection attacks. Make sure that you use parameters in your query in the correct way to guard against such attacks.

Loading entities from stored procedures

You can use DbSet.SqlQuery to load entities from the results of a stored procedure. For example, the following code calls the dbo.GetBlogs procedure in the database:

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("dbo.GetBlogs").ToList();
}
```

You can also pass parameters to a stored procedure using the following syntax:

```
using (var context = new BloggingContext())
{
    var blogId = 1;

    var blogs = context.Blogs.SqlQuery("dbo.GetBlogById @p0", blogId).Single();
}
```

Writing SQL queries for non-entity types

A SQL query returning instances of any type, including primitive types, can be created using the SqlQuery method on the Database class. For example:

```
using (var context = new BloggingContext())
{
    var blogNames = context.Database.SqlQuery<string>(
                       "SELECT Name FROM dbo.Blogs").ToList();
}
```

The results returned from SqlQuery on Database will never be tracked by the context even if the objects are instances of an entity type.

Sending raw commands to the database

Non-query commands can be sent to the database using the ExecuteSqlCommand method on Database. For example:

```
using (var context = new BloggingContext())
{
    context.Database.ExecuteSqlCommand(
        "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
}
```

Note that any changes made to data in the database using ExecuteSqlCommand are opaque to the context until entities are loaded or reloaded from the database.

Output Parameters

If output parameters are used, their values will not be available until the results have been read completely. This is due to the underlying behavior of DbDataReader, see Retrieving Data Using a DataReader for more details.

Loading Related Entities

9/27/2018 • 5 minutes to read • Edit Online

Entity Framework supports three ways to load related data - eager loading, lazy loading and explicit loading. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Eagerly Loading

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved by use of the Include method. For example, the queries below will load blogs and all the posts related to each blog.

```
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();
}
```

Note that Include is an extension method in the System.Data.Entity namespace so make sure you are using that namespace.

Eagerly loading multiple levels

It is also possible to eagerly load multiple levels of related entities. The queries below show examples of how to do this for both collection and reference navigation properties.

```
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();
}
```

Note that it is not currently possible to filter which related entities are loaded. Include will always bring in all related entities.

Lazy Loading

Lazy loading is the process whereby an entity or collection of entities is automatically loaded from the database the first time that a property referring to the entity/entities is accessed. When using POCO entity types, lazy loading is achieved by creating instances of derived proxy types and then overriding virtual properties to add the loading hook. For example, when using the Blog entity class defined below, the related Posts will be loaded the first time the Posts navigation property is accessed:

```
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; }
}
```

Turn lazy loading off for serialization

Lazy loading and serialization don’t mix well, and if you aren’t careful you can end up querying for your entire database just because lazy loading is enabled. Most serializers work by accessing each property on an instance of a type. Property access triggers lazy loading, so more entities get serialized. On those entities properties are accessed, and even more entities are loaded. It’s a good practice to turn lazy loading off before you serialize an entity. The following sections show how to do this.

Turning off lazy loading for specific navigation properties

Lazy loading of the Posts collection can be turned off by making the Posts property non-virtual:

```
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; }
}
```

Loading of the Posts collection can still be achieved using eager loading (see *Eagerly Loading* above) or the Load method (see *Explicitly Loading* below).

Turn off lazy loading for all entities

Lazy loading can be turned off for all entities in the context by setting a flag on the Configuration property. For example:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Loading of related entities can still be achieved using eager loading (see *Eagerly Loading* above) or the Load method (see *Explicitly Loading* below).

Explicitly Loading

Even with lazy loading disabled it is still possible to lazily load related entities, but it must be done with an explicit call. To do so you use the Load method on the related entity's entry. For example:

```
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();
}
```

Note that the Reference method should be used when an entity has a navigation property to another single entity. On the other hand, the Collection method should be used when an entity has a navigation property to a collection of other entities.

Applying filters when explicitly loading related entities

The Query method provides access to the underlying query that Entity Framework will use when loading related entities. You can then use LINQ to apply filters to the query before executing it with a call to a LINQ extension method such as ToList, Load, etc. The Query method can be used with both reference and collection navigation properties but is most useful for collections where it can be used to load only part of the collection. For example:

```
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();
}
```

When using the Query method it is usually best to turn off lazy loading for the navigation property. This is because otherwise the entire collection may get loaded automatically by the lazy loading mechanism either before or after the filtered query has been executed.

Note that while the relationship can be specified as a string instead of a lambda expression, the returned IQueryable is not generic when a string is used and so the Cast method is usually needed before anything useful can be done with it.

Using Query to count related entities without loading them

Sometimes it is useful to know how many entities are related to another entity in the database without actually incurring the cost of loading all those entities. The Query method with the LINQ Count method can be used to do this. For example:

```
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();
}
```

Saving Data with Entity Framework 6

9/13/2018 • 2 minutes to read • Edit Online

In this section you can find information about EF's change tracking capabilities and what happens when you call SaveChanges to persist any changes to objects into the database.

Automatic detect changes

9/13/2018 • 2 minutes to read • Edit Online

When using most POCO entities the determination of how an entity has changed (and therefore which updates need to be sent to the database) is handled by the Detect Changes algorithm. Detect Changes works by detecting the differences between the current property values of the entity and the original property values that are stored in a snapshot when the entity was queried or attached. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

By default, Entity Framework performs Detect Changes automatically when the following methods are called:

- DbSet.Find
- DbSet.Local
- DbSet.Add
- DbSet.AddRange
- DbSet.Remove
- DbSet.RemoveRange
- DbSet.Attach
- DbContext.SaveChanges
- DbContext.GetValidationErrors
- DbContext.Entry
- DbChangeTracker.Entries

Disabling automatic detection of changes

If you are tracking a lot of entities in your context and you call one of these methods many times in a loop, then you may get significant performance improvements by turning off detection of changes for the duration of the loop. For example:

```
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;
    }
}
```

Don't forget to re-enable detection of changes after the loop — We've used a try/finally to ensure it is always re-enabled even if code in the loop throws an exception.

An alternative to disabling and re-enabling is to leave automatic detection of changes turned off at all times and either call context.ChangeTracker.DetectChanges explicitly or use change tracking proxies diligently. Both of these options are advanced and can easily introduce subtle bugs into your application so use them with care.

If you need to add or remove many objects from a context, consider using DbSet.AddRange and DbSet.RemoveRange. This methods automatically detect changes only once after the add or remove operations are completed.

Working with entity states

12/4/2018 • 6 minutes to read • Edit Online

This topic will cover how to add and attach entities to a context and how Entity Framework processes these during SaveChanges. Entity Framework takes care of tracking the state of entities while they are connected to a context, but in disconnected or N-Tier scenarios you can let EF know what state your entities should be in. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Entity states and SaveChanges

An entity can be in one of five states as defined by the EntityState enumeration. These states are:

- Added: the entity is being tracked by the context but does not yet exist in the database
- Unchanged: the entity is being tracked by the context and exists in the database, and its property values have not changed from the values in the database
- Modified: the entity is being tracked by the context and exists in the database, and some or all of its property values have been modified
- Deleted: the entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called
- Detached: the entity is not being tracked by the context

SaveChanges does different things for entities in different states:

- Unchanged entities are not touched by SaveChanges. Updates are not sent to the database for entities in the Unchanged state.
- Added entities are inserted into the database and then become Unchanged when SaveChanges returns.
- Modified entities are updated in the database and then become Unchanged when SaveChanges returns.
- Deleted entities are deleted from the database and are then detached from the context.

The following examples show ways in which the state of an entity or an entity graph can be changed.

Adding a new entity to the context

A new entity can be added to the context by calling the Add method on DbSet. This puts the entity into the Added state, meaning that it will be inserted into the database the next time that SaveChanges is called. For example:

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

Another way to add a new entity to the context is to change its state to Added. For example:

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Entry(blog).State = EntityState.Added;
    context.SaveChanges();
}
```

Finally, you can add a new entity to the context by hooking it up to another entity that is already being tracked. This could be by adding the new entity to the collection navigation property of another entity or by setting a reference navigation property of another entity to point to the new entity. For example:

```
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
    blog.Posts.Add(new Post { Name = "How to Add Entities" });

    context.SaveChanges();
}
```

Note that for all of these examples if the entity being added has references to other entities that are not yet tracked then these new entities will also be added to the context and will be inserted into the database the next time that SaveChanges is called.

Attaching an existing entity to the context

If you have an entity that you know already exists in the database but which is not currently being tracked by the context then you can tell the context to track the entity using the Attach method on DbSet. The entity will be in the Unchanged state in the context. For example:

```
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();
}
```

Note that no changes will be made to the database if SaveChanges is called without doing any other manipulation of the attached entity. This is because the entity is in the Unchanged state.

Another way to attach an existing entity to the context is to change its state to Unchanged. For example:

```
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();
}
```

Note that for both of these examples if the entity being attached has references to other entities that are not yet tracked then these new entities will also attached to the context in the Unchanged state.

Attaching an existing but modified entity to the context

If you have an entity that you know already exists in the database but to which changes may have been made then you can tell the context to attach the entity and set its state to Modified. For example:

```
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();
}
```

When you change the state to Modified all the properties of the entity will be marked as modified and all the property values will be sent to the database when SaveChanges is called.

Note that if the entity being attached has references to other entities that are not yet tracked, then these new entities will attached to the context in the Unchanged state—they will not automatically be made Modified. If you have multiple entities that need to be marked Modified you should set the state for each of these entities individually.

Changing the state of a tracked entity

You can change the state of an entity that is already being tracked by setting the State property on its entry. For example:

```
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();
}
```

Note that calling Add or Attach for an entity that is already tracked can also be used to change the entity state. For example, calling Attach for an entity that is currently in the Added state will change its state to Unchanged.

Insert or update pattern

A common pattern for some applications is to either Add an entity as new (resulting in a database insert) or Attach an entity as existing and mark it as modified (resulting in a database update) depending on the value of the primary key. For example, when using database generated integer primary keys it is common to treat an entity with a zero key as new and an entity with a non-zero key as existing. This pattern can be achieved by setting the entity state based on a check of the primary key value. For example:

```
public void InsertOrUpdate(Blog blog)
{
    using (var context = new BloggingContext())
    {
        context.Entry(blog).State = blog.BlogId == 0 ?
                                    EntityState.Added :
                                    EntityState.Modified;

        context.SaveChanges();
    }
}
```

Note that when you change the state to Modified all the properties of the entity will be marked as modified and all the property values will be sent to the database when SaveChanges is called.

Working with property values

9/13/2018 • 10 minutes to read • Edit Online

For the most part Entity Framework will take care of tracking the state, original values, and current values of the properties of your entity instances. However, there may be some cases - such as disconnected scenarios - where you want to view or manipulate the information EF has about the properties. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

Entity Framework keeps track of two values for each property of a tracked entity. The current value is, as the name indicates, the current value of the property in the entity. The original value is the value that the property had when the entity was queried from the database or attached to the context.

There are two general mechanisms for working with property values:

- The value of a single property can be obtained in a strongly typed way using the Property method.
- Values for all properties of an entity can be read into a DbPropertyValues object. DbPropertyValues then acts as a dictionary-like object to allow property values to be read and set. The values in a DbPropertyValues object can be set from values in another DbPropertyValues object or from values in some other object, such as another copy of the entity or a simple data transfer object (DTO).

The sections below show examples of using both of the above mechanisms.

Getting and setting the current or original value of an individual property

The example below shows how the current value of a property can be read and then set to a new value:

```
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";
}
```

Use the OriginalValue property instead of the CurrentValue property to read or set the original value.

Note that the returned value is typed as “object” when a string is used to specify the property name. On the other hand, the returned value is strongly typed if a lambda expression is used.

Setting the property value like this will only mark the property as modified if the new value is different from the old value.

When a property value is set in this way the change is automatically detected even if AutoDetectChanges is turned off.

Getting and setting the current value of an unmapped property

The current value of a property that is not mapped to the database can also be read. An example of an unmapped property could be an RssLink property on Blog. This value may be calculated based on the BlogId, and therefore doesn't need to be stored in the database. For example:

```
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;
}
```

The current value can also be set if the property exposes a setter.

Reading the values of unmapped properties is useful when performing Entity Framework validation of unmapped properties. For the same reason current values can be read and set for properties of entities that are not currently being tracked by the context. For example:

```
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";
}
```

Note that original values are not available for unmapped properties or for properties of entities that are not being tracked by the context.

Checking whether a property is marked as modified

The example below shows how to check whether or not an individual property is marked as modified:

```
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;
}
```

The values of modified properties are sent as updates to the database when SaveChanges is called.

Marking a property as modified

The example below shows how to force an individual property to be marked as modified:

```
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;
}
```

Marking a property as modified forces an update to be send to the database for the property when SaveChanges is called even if the current value of the property is the same as its original value.

It is not currently possible to reset an individual property to be not modified after it has been marked as modified. This is something we plan to support in a future release.

Reading current, original, and database values for all properties of an entity

The example below shows how to read the current values, the original values, and the values actually in the database for all mapped properties of an entity.

```
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]);
    }
}
```

The current values are the values that the properties of the entity currently contain. The original values are the values that were read from the database when the entity was queried. The database values are the values as they are currently stored in the database. Getting the database values is useful when the values in the database may have changed since the entity was queried such as when a concurrent edit to the database has been made by another user.

Setting current or original values from another object

The current or original values of a tracked entity can be updated by copying values from another object. For example:

```
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; }
}
```

Running the code above will print out:

```
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
```

This technique is sometimes used when updating an entity with values obtained from a service call or a client in an n-tier application. Note that the object used does not have to be of the same type as the entity so long as it has properties whose names match those of the entity. In the example above, an instance of BlogDTO is used to update the original values.

Note that only properties that are set to different values when copied from the other object will be marked as modified.

Setting current or original values from a dictionary

The current or original values of a tracked entity can be updated by copying values from a dictionary or some other data structure. For example:

```
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);
}
```

Use the OriginalValues property instead of the CurrentValues property to set original values.

Setting current or original values from a dictionary using Property

An alternative to using CurrentValues or OriginalValues as shown above is to use the Property method to set the value of each property. This can be preferable when you need to set the values of complex properties. For example:

```
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];
    }
}
```

In the example above complex properties are accessed using dotted names. For other ways to access complex properties see the two sections later in this topic specifically about complex properties.

Creating a cloned object containing current, original, or database values

The DbPropertyValues object returned from CurrentValues, OriginalValues, or GetDatabaseValues can be used to create a clone of the entity. This clone will contain the property values from the DbPropertyValues object used to create it. For example:

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var clonedBlog = context.Entry(blog).GetDatabaseValues().ToObject();
}
```

Note that the object returned is not the entity and is not being tracked by the context. The returned object also does not have any relationships set to other objects.

The cloned object can be useful for resolving issues related to concurrent updates to the database, especially where a UI that involves data binding to objects of a certain type is being used.

Getting and setting the current or original values of complex properties

The value of an entire complex object can be read and set using the Property method just as it can be for a primitive property. In addition you can drill down into the complex object and read or set properties of that object, or even a nested object. Here are some examples:

```
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;
}
```

Use the OriginalValue property instead of the CurrentValue property to get or set an original value.

Note that either the Property or the ComplexProperty method can be used to access a complex property. However, the ComplexProperty method must be used if you wish to drill down into the complex object with additional Property or ComplexProperty calls.

Using DbPropertyValues to access complex properties

When you use CurrentValues, OriginalValues, or GetDatabaseValues to get all the current, original, or database values for an entity, the values of any complex properties are returned as nested DbPropertyValues objects. These nested objects can then be used to get values of the complex object. For example, the following method will print out the values of all properties, including values of any complex properties and nested complex properties.

```
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]);
        }
    }
}
```

To print out all current property values the method would be called like this:

```
using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    WritePropertyValues("", context.Entry(user).CurrentValues);
}
```

Handling Concurrency Conflicts

9/13/2018 • 5 minutes to read • Edit Online

Optimistic concurrency involves optimistically attempting to save your entity to the database in the hope that the data there has not changed since the entity was loaded. If it turns out that the data has changed then an exception is thrown and you must resolve the conflict before attempting to save again. This topic covers how to handle such exceptions in Entity Framework. The techniques shown in this topic apply equally to models created with Code First and the EF Designer.

This post is not the appropriate place for a full discussion of optimistic concurrency. The sections below assume some knowledge of concurrency resolution and show patterns for common tasks.

Many of these patterns make use of the topics discussed in Working with Property Values.

Resolving concurrency issues when you are using independent associations (where the foreign key is not mapped to a property in your entity) is much more difficult than when you are using foreign key associations. Therefore if you are going to do concurrency resolution in your application it is advised that you always map foreign keys into your entities. All the examples below assume that you are using foreign key associations.

A DbUpdateConcurrencyException is thrown by SaveChanges when an optimistic concurrency exception is detected while attempting to save an entity that uses foreign key associations.

Resolving optimistic concurrency exceptions with Reload (database wins)

The Reload method can be used to overwrite the current values of the entity with the values now in the database. The entity is then typically given back to the user in some form and they must try to make their changes again and re-save. For example:

```
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);
}
```

A good way to simulate a concurrency exception is to set a breakpoint on the SaveChanges call and then modify an entity that is being saved in the database using another tool such as SQL Management Studio. You could also insert a line before SaveChanges to update the database directly using SqlCommand. For example:

```
context.Database.SqlCommand(
    "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
```

The Entries method on DbUpdateConcurrencyException returns the DbEntityEntry instances for the entities that failed to update. (This property currently always returns a single value for concurrency issues. It may return multiple values for general update exceptions.) An alternative for some situations might be to get entries for all entities that may need to be reloaded from the database and call reload for each of these.

Resolving optimistic concurrency exceptions as client wins

The example above that uses Reload is sometimes called database wins or store wins because the values in the entity are overwritten by values from the database. Sometimes you may wish to do the opposite and overwrite the values in the database with the values currently in the entity. This is sometimes called client wins and can be done by getting the current database values and setting them as the original values for the entity. (See Working with Property Values for information on current and original values.) For example:

```
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);
}
```

Custom resolution of optimistic concurrency exceptions

Sometimes you may want to combine the values currently in the database with the values currently in the entity. This usually requires some custom logic or user interaction. For example, you might present a form to the user containing the current values, the values in the database, and a default set of resolved values. The user would then edit the resolved values as necessary and it would be these resolved values that get saved to the database. This can be done using the DbPropertyValues objects returned from CurrentValues and GetDatabaseValues on the entity's entry. For example:

```
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 currentValues = 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(currentValues, 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 currentValues,
                                       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.
}
```

Custom resolution of optimistic concurrency exceptions using objects

The code above uses DbPropertyValues instances for passing around current, database, and resolved values. Sometimes it may be easier to use instances of your entity type for this. This can be done using the ToObject and SetValues methods of DbPropertyValues. For example:

```
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.
}
```

Working with Transactions

9/13/2018 • 8 minutes to read • Edit Online

> **NOTE**
> **EF6 Onwards Only** - The features, APIs, etc. discussed in this page were introduced in Entity Framework 6. If you are using an earlier version, some or all of the information does not apply.

This document will describe using transactions in EF6 including the enhancements we have added since EF5 to make working with transactions easy.

What EF does by default

In all versions of Entity Framework, whenever you execute **SaveChanges()** to insert, update or delete on the database the framework will wrap that operation in a transaction. This transaction lasts only long enough to execute the operation and then completes. When you execute another such operation a new transaction is started.

Starting with EF6 **Database.ExecuteSqlCommand()** by default will wrap the command in a transaction if one was not already present. There are overloads of this method that allow you to override this behavior if you wish. Also in EF6 execution of stored procedures included in the model through APIs such as **ObjectContext.ExecuteFunction()** does the same (except that the default behavior cannot at the moment be overridden).

In either case, the isolation level of the transaction is whatever isolation level the database provider considers its default setting. By default, for instance, on SQL Server this is READ COMMITTED.

Entity Framework does not wrap queries in a transaction.

This default functionality is suitable for a lot of users and if so there is no need to do anything different in EF6; just write the code as you always did.

However some users require greater control over their transactions – this is covered in the following sections.

How the APIs work

Prior to EF6 Entity Framework insisted on opening the database connection itself (it threw an exception if it was passed a connection that was already open). Since a transaction can only be started on an open connection, this meant that the only way a user could wrap several operations into one transaction was either to use a TransactionScope or use the **ObjectContext.Connection** property and start calling **Open()** and **BeginTransaction()** directly on the returned **EntityConnection** object. In addition, API calls which contacted the database would fail if you had started a transaction on the underlying database connection on your own.

> **NOTE**
> The limitation of only accepting closed connections was removed in Entity Framework 6. For details, see Connection Management.

Starting with EF6 the framework now provides:

1. **Database.BeginTransaction()** : An easier method for a user to start and complete transactions themselves within an existing DbContext – allowing several operations to be combined within the same transaction and

hence either all committed or all rolled back as one. It also allows the user to more easily specify the isolation level for the transaction.

2. **Database.UseTransaction()** : which allows the DbContext to use a transaction which was started outside of the Entity Framework.

Combining several operations into one transaction within the same context

Database.BeginTransaction() has two overrides – one which takes an explicit IsolationLevel and one which takes no arguments and uses the default IsolationLevel from the underlying database provider. Both overrides return a **DbContextTransaction** object which provides **Commit()** and **Rollback()** methods which perform commit and rollback on the underlying store transaction.

The **DbContextTransaction** is meant to be disposed once it has been committed or rolled back. One easy way to accomplish this is the **using(…) {…}** syntax which will automatically call **Dispose()** when the using block completes:

```
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**
>
> Beginning a transaction requires that the underlying store connection is open. So calling Database.BeginTransaction() will open the connection if it is not already opened. If DbContextTransaction opened the connection then it will close it when Dispose() is called.

Passing an existing transaction to the context

Sometimes you would like a transaction which is even broader in scope and which includes operations on the same database but outside of EF completely. To accomplish this you must open the connection and start the transaction yourself and then tell EF a) to use the already-opened database connection, and b) to use the existing transaction on that connection.

To do this you must define and use a constructor on your context class which inherits from one of the DbContext constructors which take i) an existing connection parameter and ii) the contextOwnsConnection boolean.

> **NOTE**
>
> The contextOwnsConnection flag must be set to false when called in this scenario. This is important as it informs Entity Framework that it should not close the connection when it is done with it (for example, see line 4 below):

```
using (var conn = new SqlConnection("..."))
{
    conn.Open();
    using (var context = new BloggingContext(conn, contextOwnsConnection: false))
    {
    }
}
```

Furthermore, you must start the transaction yourself (including the IsolationLevel if you want to avoid the default setting) and let Entity Framework know that there is an existing transaction already started on the connection (see line 33 below).

Then you are free to execute database operations either directly on the SqlConnection itself, or on the DbContext. All such operations are executed within one transaction. You take responsibility for committing or rolling back the transaction and for calling Dispose() on it, as well as for closing and disposing the database connection. For example:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
sing 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();
                    }
                }
            }
        }
    }
}
```

Clearing up the transaction

You can pass null to Database.UseTransaction() to clear Entity Framework's knowledge of the current transaction. Entity Framework will neither commit nor rollback the existing transaction when you do this, so use with care and only if you're sure this is what you want to do.

Errors in UseTransaction

You will see an exception from Database.UseTransaction() if you pass a transaction when:

- Entity Framework already has an existing transaction
- Entity Framework is already operating within a TransactionScope

- The connection object in the transaction passed is null. That is, the transaction is not associated with a connection – usually this is a sign that that transaction has already completed
- The connection object in the transaction passed does not match the Entity Framework's connection.

Using transactions with other features

This section details how the above transactions interact with:

- Connection resiliency
- Asynchronous methods
- TransactionScope transactions

Connection Resiliency

The new Connection Resiliency feature does not work with user-initiated transactions. For details, see Retrying Execution Strategies.

Asynchronous Programming

The approach outlined in the previous sections needs no further options or settings to work with the asynchronous query and save methods. But be aware that, depending on what you do within the asynchronous methods, this may result in long-running transactions – which can in turn cause deadlocks or blocking which is bad for the performance of the overall application.

TransactionScope Transactions

Prior to EF6 the recommended way of providing larger scope transactions was to use a TransactionScope object:

```
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();
            }
        }
    }
}
```

The SqlConnection and Entity Framework would both use the ambient TransactionScope transaction and hence be committed together.

Starting with .NET 4.5.1 TransactionScope has been updated to also work with asynchronous methods via the use of the TransactionScopeAsyncFlowOption enumeration:

```
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();
                    }
                }
            }
        }
    }
}
```

There are still some limitations to the TransactionScope approach:

- Requires .NET 4.5.1 or greater to work with asynchronous methods.
- It cannot be used in cloud scenarios unless you are sure you have one and only one connection (cloud scenarios do not support distributed transactions).
- It cannot be combined with the Database.UseTransaction() approach of the previous sections.
- It will throw exceptions if you issue any DDL and have not enabled distributed transactions through the MSDTC Service.

Advantages of the TransactionScope approach:

- It will automatically upgrade a local transaction to a distributed transaction if you make more than one connection to a given database or combine a connection to one database with a connection to a different database within the same transaction (note: you must have the MSDTC service configured to allow distributed transactions for this to work).
- Ease of coding. If you prefer the transaction to be ambient and dealt with implicitly in the background rather than explicitly under you control then the TransactionScope approach may suit you better.

In summary, with the new Database.BeginTransaction() and Database.UseTransaction() APIs above, the TransactionScope approach is no longer necessary for most users. If you do continue to use TransactionScope then be aware of the above limitations. We recommend using the approach outlined in the previous sections instead where possible.

Data Validation

1/15/2019 • 8 minutes to read • Edit Online

NOTE

EF4.1 Onwards Only - The features, APIs, etc. discussed in this page were introduced in Entity Framework 4.1. If you are using an earlier version, some or all of the information does not apply

The content on this page is adapted from an article originally written by Julie Lerman (http://thedatafarm.com).

Entity Framework provides a great variety of validation features that can feed through to a user interface for client-side validation or be used for server-side validation. When using code first, you can specify validations using annotation or fluent API configurations. Additional validations, and more complex, can be specified in code and will work whether your model hails from code first, model first or database first.

The model

I'll demonstrate the validations with a simple pair of classes: Blog and 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; }
  }
```

Data Annotations

Code First uses annotations from the System.ComponentModel.DataAnnotations assembly as one means of configuring code first classes. Among these annotations are those which provide rules such as the Required, MaxLength and MinLength. A number of .NET client applications also recognize these annotations, for example, ASP.NET MVC. You can achieve both client side and server side validation with these annotations. For example, you can force the Blog Title property to be a required property.

```
[Required]
public string Title { get; set; }
```

With no additional code or markup changes in the application, an existing MVC application will perform client side validation, even dynamically building a message using the property and annotation names.

In the post back method of this Create view, Entity Framework is used to save the new blog to the database, but MVC’s client-side validation is triggered before the application reaches that code.

Client side validation is not bullet-proof however. Users can impact features of their browser or worse yet, a hacker might use some trickery to avoid the UI validations. But Entity Framework will also recognize the Required annotation and validate it.

A simple way to test this is to disable MVC’s client-side validation feature. You can do this in the MVC application’s web.config file. The appSettings section has a key for ClientValidationEnabled. Setting this key to false will prevent the UI from performing validations.

```
<appSettings>
    <add key="ClientValidationEnabled"value="false"/>
    ...
</appSettings>
```

Even with the client-side validation disabled, you will get the same response in your application. The error message “The Title field is required” will be displayed as. Except now it will be a result of server-side validation. Entity Framework will perform the validation on the Required annotation (before it even bothers to build and INSERT command to send to the database) and return the error to MVC which will display the message.

Fluent API

You can use code first’s fluent API instead of annotations to get the same client side & server side validation. Rather than use Required, I’ll show you this using a MaxLength validation.

Fluent API configurations are applied as code first is building the model from the classes. You can inject the configurations by overriding the DbContext class’ OnModelCreating method. Here is a configuration specifying that the BloggerName property can be no longer than 10 characters.

```
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);
      }
    }
```

Validation errors thrown based on the Fluent API configurations will not automatically reach the UI, but you can capture it in code and then respond to it accordingly.

Here’s some exception handling error code in the application’s BlogController class that captures that validation error when Entity Framework attempts to save a blog with a BloggerName that exceeds the 10 character maximum.

```
[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();
    }
}
```

The validation doesn’t automatically get passed back into the view which is why the additional code that uses ModelState.AddModelError is being used. This ensures that the error details make it to the view which will then use the ValidationMessageFor Htmlhelper to display the error.

```
@Html.ValidationMessageFor(model => model.BloggerName)
```

IValidatableObject

IValidatableObject is an interface that lives in System.ComponentModel.DataAnnotations. While it is not part of the Entity Framework API, you can still leverage it for server-side validation in your Entity Framework classes. IValidatableObject provides a Validate method that Entity Framework will call during SaveChanges or you can call yourself any time you want to validate the classes.

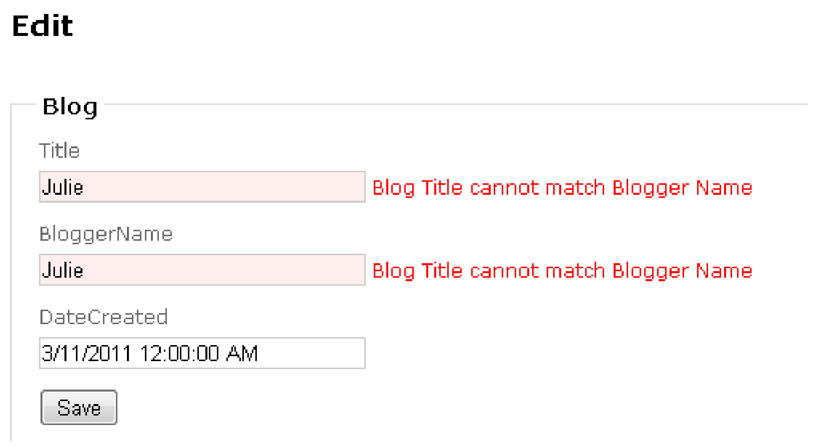
Configurations such as Required and MaxLength perform validaton on a single field. In the Validate method you can have even more complex logic, for example, comparing two fields.

In the following example, the Blog class has been extended to implement IValidatableObject and then provide a rule that the Title and BloggerName cannot match.

```
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” });
         }
     }
 }
```

The ValidationResult constructor takes a string that represents the error message and an array of strings that represent the member names that are associated with the validation. Since this validation checks both the Title and the BloggerName, both property names are returned.

Unlike the validation provided by the Fluent API, this validation result will be recognized by the View and the exception handler that I used earlier to add the error into ModelState is unnecessary. Because I set both property names in the ValidationResult, the MVC HtmlHelpers display the error message for both of those properties.



DbContext.ValidateEntity

DbContext has an Overridable method called ValidateEntity. When you call SaveChanges, Entity Framework will call this method for each entity in its cache whose state is not Unchanged. You can put validation logic directly in here or even use this method to call, for example, the Blog.Validate method added in the previous section.

Here's an example of a ValidateEntity override that validates new Posts to ensure that the post title hasn't been used already. It first checks to see if the entity is a post and that its state is Added. If that's the case, then it looks in the database to see if there is already a post with the same title. If there is an existing post already, then a new DbEntityValidationResult is created.

DbEntityValidationResult houses a DbEntityEntry and an ICollection of DbValidationErrors for a single entity. At the start of this method, a DbEntityValidationResult is instantiated and then any errors that are discovered are added into its ValidationErrors collection.

```
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.DbValidationError("Title",
                    "Post title must be unique."));
        }
    }

    if (result.ValidationErrors.Count > 0)
    {
        return result;
    }
    else
    {
     return base.ValidateEntity(entityEntry, items);
    }
}
```

Explicitly triggering validation

A call to SaveChanges triggers all of the validations covered in this article. But you don’t need to rely on SaveChanges. You may prefer to validate elsewhere in your application.

DbContext.GetValidationErrors will trigger all of the validations, those defined by annotations or the Fluent API, the validation created in IValidatableObject (for example, Blog.Validate), and the validations performed in the DbContext.ValidateEntity method.

The following code will call GetValidationErrors on the current instance of a DbContext. ValidationErrors are grouped by entity type into DbValidationResults. The code iterates first through the DbValidationResults returned by the method and then through each ValidationError inside.

```
foreach (var validationResults in db.GetValidationErrors())
    {
        foreach (var error in validationResults.ValidationErrors)
        {
            Debug.WriteLine(
                              "Entity Property: {0}, Error {1}",
                              error.PropertyName,
                              error.ErrorMessage);
        }
    }
```

Other considerations when using validation

Here are a few other points to consider when using Entity Framework validation:

- Lazy loading is disabled during validation.
- EF will validate data annotations on non-mapped properties (properties that are not mapped to a column in the database).
- Validation is performed after changes are detected during SaveChanges. If you make changes during validation

it is your responsibility to notify the change tracker.
- DbUnexpectedValidationException is thrown if errors occur during validation.
- Facets that Entity Framework includes in the model (maximum length, required, etc.) will cause validation, even if there are not data annotations on your classes and/or you used the EF Designer to create your model.
- Precedence rules:
  - Fluent API calls override the corresponding data annotations
- Execution order:
  - Property validation occurs before type validation
  - Type validation only occurs if property validation succeeds
- If a property is complex its validation will also include:
  - Property-level validation on the complex type properties
  - Type level validation on the complex type, including IValidatableObject validation on the complex type

Summary

The validation API in Entity Framework plays very nicely with client side validation in MVC but you don't have to rely on client-side validation. Entity Framework will take care of the validation on the server side for DataAnnotations or configurations you've applied with the code first Fluent API.

You also saw a number of extensibility points for customizing the behavior whether you use the IValidatableObject interface or tap into the DbContet.ValidateEntity method. And these last two means of validation are available through the DbContext, whether you use the Code First, Model First or Database First workflow to describe your conceptual model.

Entity Framework Resources

9/13/2018 • 2 minutes to read • Edit Online

Here you will find links and references to additional information related to EF, like blogs, third-party providers, tools and extensions, case studies, etc.

Entity Framework Blogs

9/13/2018 • 2 minutes to read • Edit Online

Besides the product documentation, these blogs can be a source of useful information on Entity Framework:

EF Team blogs

- .NET Blog - Tag: Entity Framework
- ADO.NET Blog (no longer in use)
- EF Design Blog (no longer in use)

Current and former EF team bloggers

- Arthur Vickers
- Brice Lambson
- Diego Vega
- Rowan Miller
- Pawel Kadluczka
- Alex James
- Zlatko Michailov

EF Community Bloggers

- Julie Lerman
- Shawn Wildermuth

Microsoft Case Studies for Entity Framework

9/13/2018 • 3 minutes to read • Edit Online

The case studies on this page highlight a few real-world production projects that have employed Entity Framework.

> **NOTE**
>
> The detailed versions of these case studies are no longer available on the Microsoft website. Therefore the links have been removed.

Epicor

Epicor is a large global software company (with over 400 developers) that develops Enterprise Resource Planning (ERP) solutions for companies in more than 150 countries. Their flagship product, Epicor 9, is based on a Service-Oriented Architecture (SOA) using the .NET Framework. Faced with numerous customer requests to provide support for Language Integrated Query (LINQ), and also wanting to reduce the load on their back-end SQL Servers, the team decided to upgrade to Visual Studio 2010 and the .NET Framework 4.0. Using the Entity Framework 4.0, they were able to achieve these goals and also greatly simplify development and maintenance. In particular, the Entity Framework's rich T4 support allowed them to take full control of their generated code and automatically build in performance-saving features such as pre-compiled queries and caching.

> "We conducted some performance tests recently with existing code, and we were able to reduce the requests to SQL Server by 90 percent. That is because of the ADO.NET Entity Framework 4." – Erik Johnson, Vice President, Product Research

Veracity Solutions

Having acquired an event-planning software system that was going to be difficult to maintain and extend over the long-term, Veracity Solutions used Visual Studio 2010 to re-write it as a powerful and easy-to-use Rich Internet Application built on Silverlight 4. Using .NET RIA Services, they were able to quickly build a service layer on top of the Entity Framework that avoided code duplication and allowed for common validation and authentication logic across tiers.

> "We were sold on the Entity Framework when it was first introduced, and the Entity Framework 4 has proven to be even better. Tooling is improved, and it's easier to manipulate the .edmx files that define the conceptual model, storage model, and mapping between those models... With the Entity Framework, I can get that data access layer working in a day—and build it out as I go along. The Entity Framework is our de facto data access layer; I don't know why anyone wouldn't use it." – Joe McBride, Senior Developer

NEC Display Solutions of America

NEC wanted to enter the market for digital place-based advertising with a solution to benefit advertisers and network owners and increase its own revenues. In order to do that, it launched a pair of web applications that automate the manual processes required in a traditional ad campaign. The sites were built using ASP.NET, Silverlight 3, AJAX and WCF, along with the Entity Framework in the data access layer to talk to SQL Server 2008.

> "With SQL Server, we felt we could get the throughput we needed to serve advertisers and networks with information in real time and the reliability to help ensure that the information in our mission-critical applications would always be available"- Mike Corcoran, Director of IT

Darwin Dimensions

Using a wide range of Microsoft technologies, the team at Darwin set out to create Evolver - an online avatar portal that consumers could use to create stunning, lifelike avatars for use in games, animations, and social networking pages. With the productivity benefits of the Entity Framework, and pulling in components like Windows Workflow Foundation (WF) and Windows Server AppFabric (a highly-scalable in-memory application cache), the team was able to deliver an amazing product in 35% less development time. Despite having team members split across multiple countries, the team following an agile development process with weekly releases.

> “We try not to create technology for technology’s sake. As a startup, it is crucial that we leverage technology that saves time and money. .NET was the choice for fast, cost-effective development.” – Zachary Olsen, Architect

Silverware

With more than 15 years of experience in developing point-of-sale (POS) solutions for small and midsize restaurant groups, the development team at Silverware set out to enhance their product with more enterprise-level features in order to attract larger restaurant chains. Using the latest version of Microsoft’s development tools, they were able to build the new solution four times faster than before. Key new features like LINQ and the Entity Framework made it easier to move from Crystal Reports to SQL Server 2008 and SQL Server Reporting Services (SSRS) for their data storage and reporting needs.

> “Effective data management is key to the success of SilverWare – and this is why we decided to adopt SQL Reporting.” - Nicholas Romanidis, Director of IT/Software Engineering

Contribute to Entity Framework 6

9/13/2018 • 2 minutes to read • Edit Online

Entity Framework 6 is developed using an open source model on GitHub. Although the main focus of the Entity Framework Team at Microsoft is on adding new features to Entity Framework Core, and we don't expect any major features to be added to Entity Framework 6, we still accept contributions.

For product contributions, please start at the Contributing wiki page in our GitHub repository.

For documentation contributions, please start read the contribution guidance in our documentation repository.

Get Help Using Entity Framework

9/13/2018 • 2 minutes to read • Edit Online

Questions About Using EF

The best way to get help using Entity Framework is to post a question on Stack Overflow using the **entity-framework** tag.

If you are not familiar with Stack Overflow, be sure to read the guidelines on asking questions. In particular, do not use Stack Overflow to report bugs, ask roadmap questions, or suggest new features.

Bug Reports and Feature Requests

If you have found a bug that you think should be fixed, have a feature you would like to see implemented, or a question you couldn't find an answer to, create an issue on the EF6 GitHub repository.

Entity Framework Glossary

10/9/2018 • 3 minutes to read • Edit Online

Code First

Creating an Entity Framework model using code. The model can target and existing database or a new database.

Context

A class that represents a session with the database, allowing you to query and save data. A context derives from the DbContext or ObjectContext class.

Convention (Code First)

A rule that Entity Framework uses to infer the shape of you model from your classes.

Database First

Creating an Entity Framework model, using the EF Designer, that targets an existing database.

Eager loading

A pattern of loading related data where a query for one type of entity also loads related entities as part of the query.

EF Designer

A visual designer in Visual Studio that allows you to create an Entity Framework model using boxes and lines.

Entity

A class or object that represents application data such as customers, products, and orders.

Entity Data Model

A model that describes entities and the relationships between them. EF uses EDM to describe the conceptual model against which the developer programs. EDM builds on the Entity Relationship model introduced by Dr. Peter Chen. The EDM was originally developed with the primary goal of becoming the common data model across a suite of developer and server technologies from Microsoft. EDM is also used as part of the OData protocol.

Explicit loading

A pattern of loading related data where related objects are loaded by calling an API.

Fluent API

An API that can be used to configure a Code First model.

Foreign key association

An association between entities where a property that represents the foreign key is included in the class of the dependent entity. For example, Product contains a CategoryId property.

Identifying relationship

A relationship where the primary key of the principal entity is part of the primary key of the dependent entity. In this kind of relationship, the dependent entity cannot exist without the principal entity.

Independent association

An association between entities where there is no property representing the foreign key in the class of the dependent entity. For example, a Product class contains a relationship to Category but no CategoryId property. Entity Framework tracks the state of the association independently of the state of the entities at the two association ends.

Lazy loading

A pattern of loading related data where related objects are automatically loaded when a navigation property is accessed.

Model First

Creating an Entity Framework model, using the EF Designer, that is then used to create a new database.

Navigation property

A property of an entity that references another entity. For example, Product contains a Category navigation property and Category contains a Products navigation property.

POCO

Acronym for Plain-Old CLR Object. A simple user class that has no dependencies with any framework. In the context of EF, an entity class that does not derive from EntityObject, implements any interfaces or carries any attributes defined in EF. Such entity classes that are decoupled from the persistence framework are also said to be "persistence ignorant".

Relationship inverse

The opposite end of a relationship, for example, product.Category and category.Product.

Self-tracking entity

An entity built from a code generation template that helps with N-Tier development.

Table-per-concrete type (TPC)

A method of mapping the inheritance where each non-abstract type in the hierarchy is mapped to separate table in the database.

Table-per-hierarchy (TPH)

A method of mapping the inheritance where all types in the hierarchy are mapped to the same table in the database. A discriminator column(s) is used to identify what type each row is associated with.

Table-per-type (TPT)

A method of mapping the inheritance where the common properties of all types in the hierarchy are mapped to the same table in the database, but properties unique to each type are mapped to a separate table.

Type discovery

The process of identifying the types that should be part of an Entity Framework model.

School Sample Database

9/13/2018 • 14 minutes to read • Edit Online

This topic contains the schema and data for the School database. The sample School database is used in various places throughout the Entity Framework documentation.

> **NOTE**
>
> The database server that is installed with Visual Studio is different depending on the version of Visual Studio you use. See Visual Studio Releases for details on what to use.

Here are the steps to create the database:

- Open Visual Studio
- **View** -> **Server Explorer**
- Right click on **Data Connections** -> **Add Connection...**
- If you haven't connected to a database from Server Explorer before you'll need to select **Microsoft SQL Server** as the data source
- Connect to either LocalDB or SQL Express, depending on which one you have installed
- Enter **School** as the database name
- Select **OK** and you will be asked if you want to create a new database, select **Yes**
- The new database will now appear in Server Explorer
- If you are using Visual Studio 2012 or newer
  - Right-click on the database in Server Explorer and select **New Query**
  - Copy the following SQL into the new query, then right-click on the query and select **Execute**
- If you are using Visual Studio 2010
  - Select **Data** -> **Transact SQL Editor** -> **New Query Connection...**
  - Enter **.\SQLEXPRESS** as the server name and click **OK**
  - Select the **STESample** database from the drop down at the top of the query editor
  - Copy the following SQL into the new query, then right-click on the query and select **Execute 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_Department] 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]
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]')
```

```
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
```

```
-- 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
```

```
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', '2001-7-23', 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 (2030, '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, 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)
```

```
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);
```

```
VALUES (1061, 30, 4);
GO
```

Entity Framework Tools & Extensions

9/13/2018 • 2 minutes to read • Edit Online

IMPORTANT

Extensions are built by a variety of sources and not maintained as part of Entity Framework. When considering a third party extension, be sure to evaluate quality, licensing, compatibility, support, etc. to ensure they meet your requirements.

Entity Framework has been a popular O/RM for many years. Here are some examples of free and paid tools and extensions developed for it:

- EF Power Tools Community Edition
- EF Profiler
- ORM Profiler
- LINQPad
- LLBLGen Pro
- Huagati DBML/EDMX Tools
- Entity Developer

Entity Framework 5 License (CHS)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT 软件补充程序许可条款

ENTITY FRAMEWORK 5.0（适用于 MICROSOFT WINDOWS OPERATING SYSTEM）

Microsoft Corporation（或 Microsoft Corporation 在您所在地的关联公司）现授予您本补充程序的许可证。如果您获得了使用 Microsoft Windows Operating System 软件（"软件"）的许可证，您可以使用本补充程序。如果您没有该软件的许可证，则不得使用。您可以将本补充程序用于获得有效许可的每份软件副本。

下列许可条款说明了本补充程序的使用条款。这些条款和软件的许可条款在您使用本补充程序时适用。如果发生冲突，则以这些补充程序许可条款为准。

使用本补充程序即表示您接受这些条款。如果您不接受这些条款，请不要使用本补充程序。

如果您遵守这些许可条款，您将具有下列权利。

1. **可分发代码。**本补充程序中包含"可分发代码"。"可分发代码"是指，如果您遵守下述条款，则可以在您开发的程序中分发这些代码。

a. **使用和分发权。**

- 您可以复制和分发对象代码形式的补充程序。
- *第三方分发*。您可以允许您的程序分销商作为这些程序的一部分复制和分发"可分发代码"。

b. **分发要求。对于您分发的任何可分发代码，您必须**

- 在您的程序中对其增加重要的主要功能；
- 对于任何文件扩展名为 .lib 的可分发代码，仅分发通过链接器与您的程序一起运行上述可分发代码的结果；
- 仅分发作为安装程序的一部分包含在安装程序中的未经修改的可分发代码；
- 要求分销商及外部最终用户同意至少能够像本协议一样保护"可分发代码"的条款；
- 显示您的程序的有效版权声明；以及
- 对于与分发或使用您的程序有关的任何索赔，为 Microsoft 提供辩护、赔偿，包括支付律师费，并使 Microsoft 免受损失。

c. **分发限制。您不可以**

- 更改"可分发代码"中的任何版权、商标或专利声明；
- 在您的程序名称中使用 Microsoft 的商标，或者以其他方式暗示您的程序来自 Microsoft 或经 Microsoft 认可；
- 分发可分发代码，以便在 Windows 平台以外的任何平台上运行；
- 在恶意的、欺骗性的或非法的程序中包括可分发代码；或者
- 修改或分发任何可分发代码的源代码，致使其任何部分受到"排除许可"的制约。"排除许可"指符合以下使用、修改或分发条件的许可：
  - 以源代码形式披露或分发代码；或
  - 其他人有权对其进行修改。

2. **对补充程序的支持服务。**Microsoft 为 www.support.microsoft.com/common/international.aspx 中指明的本软件提供支持服务。

Entity Framework 5 License (CHT)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT 軟體增補程式授權條款

ENTITY FRAMEWORK 5.0 之 MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft 公司(或其關係企業，視 貴用戶所居住的地點而定)授權 貴用戶使用本增補程式。若 貴用戶取得 Microsoft Windows Operating System 軟體(「軟體」)之使用授權，即可使用本增補程式。 貴用戶若未取得軟體授權，即不得使用本增補程式。 貴用戶擁有之每份有效授權軟體拷貝，均得使用本增補程式。

以下授權款條說明本增補程式之其他使用條款。 貴用戶使用本增補程式時，請遵守上述條款與軟體授權條款。若發生使用爭議，亦適用這些補充的授權條款。

增補程式一經使用，即表示 貴用戶同意接受這些條款。若 貴用戶不同意這些授權條款，請不要使用本增補程式。

若 貴用戶遵守本授權條款，則 貴用戶得享有以下各項權利。

1. 可散布程式碼。增補程式由「可散佈程式碼」構成。「可散布程式碼」係若 貴用戶遵守以下條款，則可於 貴用戶開發的程式中散布之程式碼。

a. 使用及散布的權利。

- 貴用戶得以目的碼形式複製與散布增補程式。
- *第三者廠商散布。* 貴用戶得同意程式經銷商將「可散布程式碼」視為 貴用戶那些程式之一部分，進行複製與散布。

b. 散布要件。針對 貴用戶散布的任何「可散布程式碼」， 貴用戶必須

- 在程式中，為「可散布程式碼」加入重要的新功能；
- 針對任何副檔名為 .lib 的「可散布程式碼」， 貴用戶僅得散布透過 貴用戶程式之連結器執行這類「可散布程式碼」所產生的結果；
- 散布包含於某一安裝程式中的「可散布程式碼」時，僅能做為該安裝程式之一部分進行散布，且不得經過任何修改；
- 要求散布者及外部終端使用者，需同意「保護『可散布程式碼』的程度不得低於本合約」之相關條款；
- 在程式中顯示有效的著作權聲明；以及
- 若因 貴用戶散布或使用程式而使 Microsoft 遭他人提出損害賠償請求權時， 貴用戶應賠償 Microsoft 之損失 (包括律師費)，為之辯護，並使其不受損害。

c. 散布限制。 貴用戶不得

- 變更「可散布程式碼」中之任何著作權、商標或專利聲明；
- 於 貴用戶的程式名稱使用 Microsoft 的商標，或暗示該程式來自 Microsoft 或由 Microsoft 背書；
- 散布並於非 Windows 的平台上執行；
- 將「可散布程式碼」置於惡意、欺騙或違法的程式碼中；或
- 修改或散布任何「可散布程式碼」的原始碼，如此將會使其任何部分受到「排除性授權」之限制。「排除性授權」係指在使用、修改或散布時，應遵守下列條件：
  - 程式碼必須以原始碼形式揭露或散布，或
  - 他人有修改的權利。

2. 增補程式支援服務。Microsoft 為軟體提供支援服務之相關說明，請參閱

www.support.microsoft.com/common/international.aspx。

Entity Framework 5 License (DEU)

9/13/2018 • 2 minutes to read • Edit Online

LIZENZBESTIMMUNGEN FÜR MICROSOFT-SOFTWAREERGÄNZUNG

ENTITY FRAMEWORK 5.0 FÜR MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (oder eine andere Microsoft-Konzerngesellschaft, wenn diese an dem Ort, an dem Sie die Software erwerben, die Software lizenziert) lizenziert diese Softwareergänzung an Sie. Wenn Sie über eine Lizenz zur Verwendung der Software Microsoft Windows Operating System (die „Software") verfügen, sind Sie berechtigt, diese Softwareergänzung zu verwenden. Sie sind nicht berechtigt, sie zu verwenden, wenn Sie keine Lizenz für die Software haben. Sie sind berechtigt, diese Softwareergänzung mit jeder ordnungsgemäß lizenzierten Kopie der Software zu verwenden.

In den folgenden Lizenzbestimmungen werden zusätzliche Nutzungsbedingungen für diese Softwareergänzung beschrieben. Diese Bestimmungen und die Lizenzbestimmungen für die Software gelten für Ihre Verwendung der Softwareergänzung. Im Falle eines Widerspruchs gelten diese ergänzenden Lizenzbestimmungen.

Durch die Verwendung dieser Softwareergänzung erkennen Sie diese Bestimmungen an. Falls Sie die Bestimmungen nicht akzeptieren, sind Sie nicht berechtigt, diese Softwareergänzung zu verwenden.

Wenn Sie diese Lizenzbestimmungen einhalten, haben Sie die nachfolgend aufgeführten Rechte.

1. VERTREIBBARER CODE. Die Softwareergänzung besteht aus Vertreibbarem Code. „Vertreibbarer Code" ist Code, den Sie in von Ihnen entwickelten Programmen vertreiben dürfen, wenn Sie die nachfolgenden Bestimmungen einhalten.

a. Recht zur Nutzung und zum Vertrieb.

- Sie sind berechtigt, die Objektcodeform der Softwareergänzung zu kopieren und zu vertreiben.
- *Vertrieb durch Dritte*. Sie sind berechtigt, Distributoren Ihrer Programme zu erlauben, den vertreibbaren Code als Teil dieser Programme zu kopieren und zu vertreiben.

b. Vertriebsbedingungen. Für vertreibbaren Code, den Sie vertreiben, sind Sie verpflichtet:

- diesem in Ihren Programmen wesentliche primäre Funktionalität hinzuzufügen
- für Vertreibbaren Code mit der Dateinamenerweiterung LIB nur die Ergebnisse des Durchlaufs dieses Vertreibbaren Codes durch einen Linker mit Ihrem Programm zu vertreiben
- in einem Setupprogramm enthaltenen Vertreibbaren Code nur als Teil dieses Setupprogramms ohne Änderung zu vertreiben
- von Distributoren und externen Endbenutzern die Zustimmung zu Bestimmungen zu verlangen, die einen mindestens gleichwertigen Schutz für ihn bieten wie dieser Vertrag
- Ihren gültigen Urheberrechtshinweis auf Ihren Programmen anzubringen
- Microsoft freizustellen und gegen alle Ansprüche zu verteidigen, inklusive Anwaltsgebühren, die mit dem Vertrieb oder der Verwendung Ihrer Programme in Zusammenhang stehen.

c. Vertriebseinschränkungen. Sie sind nicht dazu berechtigt:

- Urheberrechts-, Markenrechts- oder Patenthinweise im vertreibbaren Code zu ändern
- die Marken von Microsoft in den Namen Ihrer Programme oder auf eine Weise zu verwenden, die nahe legt, dass Ihre Programme von Microsoft stammen oder von Microsoft empfohlen werden
- vertreibbaren Code zur Ausführung auf einer anderen Plattform als der Windows-Plattform zu vertreiben

- vertreibbaren Code in böswillige, täuschende oder rechtswidrige Programme aufzunehmen
- den Quellcode von vertreibbarem Code so zu ändern oder zu vertreiben, dass irgendein Teil von ihm einer Ausschlusslizenz unterliegt. Eine Ausschlusslizenz ist eine Lizenz, die als Bedingung für eine Verwendung, Änderung oder einen Vertrieb erfordert, dass:
  - der Code in Quellcodeform offen gelegt oder vertrieben wird
  - andere das Recht haben, ihn zu ändern.

2. SUPPORTLEISTUNGEN FÜR SOFTWAREERGÄNZUNG. Microsoft stellt Supportleistungen für diese Software bereit, die unter www.support.microsoft.com/common/international.aspx beschrieben werden.

Entity Framework 5 License (ENU)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT SOFTWARE SUPPLEMENTAL LICENSE TERMS

ENTITY FRAMEWORK 5.0 FOR MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (or based on where you live, one of its affiliates) licenses this supplement to you. If you are licensed to use Microsoft Windows Operating System software (the "software"), you may use this supplement. You may not use it if you do not have a license for the software. You may use this supplement with each validly licensed copy of the software.

The following license terms describe additional use terms for this supplement. These terms and the license terms for the software apply to your use of the supplement. If there is a conflict, these supplemental license terms apply.

By using this supplement, you accept these terms. If you do not accept them, do not use this supplement.

If you comply with these license terms, you have the rights below.

1. DISTRIBUTABLE CODE. The supplement is comprised of Distributable Code. "Distributable Code" is code that you are permitted to distribute in programs you develop if you comply with the terms below.

a. Right to Use and Distribute.

- You may copy and distribute the object code form of the supplement.
- *Third Party Distribution*. You may permit distributors of your programs to copy and distribute the Distributable Code as part of those programs.

b. Distribution Requirements. For any Distributable Code you distribute, you must

- add significant primary functionality to it in your programs;
- for any Distributable Code having a filename extension of .lib, distribute only the results of running such Distributable Code through a linker with your program;
- distribute Distributable Code included in a setup program only as part of that setup program without modification;
- require distributors and external end users to agree to terms that protect it at least as much as this agreement;
- display your valid copyright notice on your programs; and
- indemnify, defend, and hold harmless Microsoft from any claims, including attorneys' fees, related to the distribution or use of your programs.

c. Distribution Restrictions. You may not

- alter any copyright, trademark or patent notice in the Distributable Code;
- use Microsoft's trademarks in your programs' names or in a way that suggests your programs come from or are endorsed by Microsoft;
- distribute Distributable Code to run on a platform other than the Windows platform;
- include Distributable Code in malicious, deceptive or unlawful programs; or
- modify or distribute the source code of any Distributable Code so that any part of it becomes subject to an Excluded License. An Excluded License is one that requires, as a condition of use, modification or distribution, that
    - the code be disclosed or distributed in source code form; or

- others have the right to modify it.

2. SUPPORT SERVICES FOR SUPPLEMENT. Microsoft provides support services for this software as described at www.support.microsoft.com/common/international.aspx.

Entity Framework 5 License (ESN)

9/13/2018 • 2 minutes to read • Edit Online

TÉRMINOS SUPLEMENTARIOS A LA LICENCIA DE USO DE SOFTWARE DE MICROSOFT

ENTITY FRAMEWORK 5.0 PARA MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (o, en función del lugar en el que usted resida, alguna de las afiliadas de su grupo) le concede a usted la licencia de este suplemento. Si dispone de licencia de uso para el software de Microsoft Windows Operating System (el "software"), podrá utilizar este suplemento. No podrá utilizarlo si no dispone de una licencia del software. Podrá utilizar este complemento con cada copia licenciada válida del software.

Los siguientes términos de licencia describen términos de uso adicionales para este suplemento. Dichos términos y los términos de licencia para el software se aplicarán al uso que haga del suplemento. En caso de conflicto, prevalecerán los presentes términos de la licencia suplementaria.

Al hacer uso del suplemento, se entiende que acepta estos términos. Si usted no los acepta, no use el suplemento.

Si cumple usted los presentes términos de esta licencia, dispondrá de los siguientes derechos.

1. CÓDIGO DISTRIBUIBLE. El suplemento se compone de Código Distribuible. El "Código Distribuible" es código que usted está autorizado a distribuir en los programas que desarrolle, siempre y cuando cumpla los términos que se especifican a continuación,

a. Derecho de utilización y distribución.

- Podrá copiar y distribuir el código objeto del complemento.
- *Distribución a terceros*. Podrá permitir a los distribuidores de sus programas copiar y distribuir el Código Distribuible como parte de dichos programas.

b. Requisitos de distribución. Para cualquier Código distribuible, deberá:

- añadir al mismo una funcionalidad principal significativa en sus programas;
- Para cualquier Código Distribuible con extensión de archivo .lib, distribuir únicamente los resultados de la ejecución e dicho Código Distribuible a través de un vinculador con su programa.
- Distribuir el Código Distribuible que se incluya en un programa de instalación exclusivamente como parte de dicho programa sin modificación alguna.
- exigir a los distribuidores y usuarios finales externos que acepten proteger el software como mínimo tal y como lo especifica el presente contrato;
- mostrar un aviso válido de sus derechos de autor en los programas, así como
- indemnizar, proteger y defender a Microsoft frente a toda reclamación, incluidos los honorarios de abogados, relacionados con el uso o la distribución de sus programas.

c. Restricciones de distribución. Queda prohibido:

- modificar cualquier aviso de derechos de autor, marca comercial o patente del Código Distribuible;
- utilizar las marcas comerciales de Microsoft en los nombres de sus programas o sugerir de alguna manera que sus programas son de la familia de Microsoft o están promocionados por Microsoft;
- distribuir Código Distribuible para ejecutarlo en una plataforma distinta de Windows;
- incluir Código distribuible en programas maliciosos, engañosos o ilegales; o
- modificar o distribuir el código fuente de cualquier Código Distribuible de modo que alguna parte del mismo

pase a estar sujeta a una Licencia de Exclusión. Una Licencia de Exclusión es aquella que requiere, como condición de uso, modificación o distribución, que:

- el código sea divulgado o distribuido en forma de código fuente, o que
- otros tengan derecho a modificarlo.

2. SERVICIOS DE SOPORTE TÉCNICO PARA SUPLEMENTOS. Microsoft presta servicios de soporte técnico para este software, tal y como se describe en www.support.microsoft.com/common/international.aspx.

Entity Framework 5 License (FRA)

9/13/2018 • 2 minutes to read • Edit Online

TERMES DU CONTRAT DE LICENCE D'UN SUPPLÉMENT MICROSOFT

ENTITY FRAMEWORK 5.0 POUR MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (ou en fonction du lieu où vous vivez, l'un de ses affiliés) vous accorde une licence pour ce supplément. Si vous êtes titulaire d'une licence d'utilisation du logiciel Microsoft Windows Operating System (le « logiciel »), vous pouvez utiliser ce supplément. Vous n'êtes pas autorisé à utiliser ce supplément si vous n'êtes pas titulaire d'une licence pour le logiciel. Vous pouvez utiliser une copie de ce supplément avec chaque copie concédée sous licence du logiciel.

Les conditions de licence suivantes décrivent les conditions d'utilisation supplémentaires applicables pour ce supplément. Les présentes conditions et les conditions de licence pour le logiciel s'appliquent à l'utilisation du supplément. En cas de conflit, les présentes conditions de licence supplémentaires s'appliquent.

En utilisant ce supplément, vous acceptez ces termes. Si vous ne les acceptez pas, n'utilisez pas ce supplément.

Dans le cadre du présent accord de licence, vous disposez des droits ci-dessous.

1. CODE DISTRIBUABLE. Le supplément constitue du Code Distribuable. Le « Code Distribuable » est le code que vous êtes autorisé à distribuer dans les programmes que vous développez, sous réserve de vous conformer aux termes ci-après.

a. Droit d'utilisation et de distribution.

- Vous êtes autorisé à copier et à distribuer la version en code objet du supplément.
- *Distribution par des tierces parties.* Vous pouvez autoriser les distributeurs de vos programmes à copier et à distribuer le code distribuable en tant que partie intégrante de ces programmes.

b. Conditions de distribution. Pour pouvoir distribuer du code distribuable, vous devez :

- y ajouter des fonctionnalités importantes au sein de vos programmes,
- pour tout Code distribuable dont l'extension de nom de fichier est .lib, distribuer seulement les résultats de l'exécution de ce Code distribuable à l'aide d'un éditeur de liens avec votre programme ;
- distribuer le Code distribuable inclus dans un programme d'installation seulement en tant que partie intégrante de ce programme sans modification ;
- lier les distributeurs et les utilisateurs externes par un contrat dont les termes les protègent autant que le présent contrat,
- afficher votre propre mention de droits d'auteur valable sur vos programmes et
- garantir et défendre Microsoft contre toute réclamation, y compris pour les honoraires d'avocats, qui résulterait de la distribution ou l'utilisation de vos programmes.

c. Restrictions de distribution. Vous n'êtes pas autorisé à :

- modifier toute mention de droits d'auteur, de marques ou de droits de propriété industrielle pouvant figurer dans le code distribuable,
- utiliser les marques de Microsoft dans les noms de vos programmes ou d'une façon qui suggère que vos programmes sont fournis par Microsoft ou sous la responsabilité de Microsoft,
- distribuer le Code distribuable en vue de son exécution sur une plate-forme autre que la plate-forme Windows,

- inclure le Code distribuable dans des programmes malveillants, trompeurs ou interdits par la loi, ou
- modifier ou distribuer le code source de code distribuable de manière à ce qu'il fasse l'objet, en partie ou dans son intégralité, d'une Licence Exclue. Une Licence Exclue implique comme condition d'utilisation, de modification ou de distribution, que :
  - le code soit dévoilé ou distribué dans sa forme de code source, ou
  - d'autres aient le droit de le modifier.

2. SERVICES D'ASSISTANCE TECHNIQUE POUR LE SUPPLÉMENT. Microsoft fournit des services d'assistance technique pour ce logiciel disponibles sur le site www.support.microsoft.com/common/international.aspx.

Entity Framework 5 License (ITA)

9/13/2018 • 2 minutes to read • Edit Online

CONTRATTO DI LICENZA SUPPLEMENTARE PER IL SOFTWARE MICROSOFT

ENTITY FRAMEWORK 5.0 PER MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (o, in base al luogo di residenza del licenziatario, una delle sue consociate) concede in licenza al licenziatario il presente supplemento. Qualora il licenziatario sia autorizzato a utilizzare Microsoft Windows Operating System il software (il "software"), potrà usare il presente supplemento. Il licenziatario non potrà utilizzarlo qualora non disponga di una licenza per il software. Il licenziatario potrà utilizzare il presente supplemento con ciascuna copia validamente concessa in licenza del software.

Nelle condizioni di licenza che seguono sono descritte condizioni di utilizzo aggiuntive per il presente supplemento. Tali condizioni e le condizioni di licenza per il software si applicano all'utilizzo del supplemento da parte del licenziatario. Qualora esista un conflitto tra le predette condizioni di licenza, prevarranno le condizioni di licenza supplementari.

Utilizzando il presente supplemento, il licenziatario accetta le presenti condizioni. Qualora il licenziatario non le accetti, non potrà utilizzare il supplemento.

Qualora il licenziatario si attenga alle presenti condizioni di licenza, disporrà dei diritti di seguito indicati.

1. CODICE DISTRIBUIBILE. Il supplemento è costituito dal "Codice Distribuibile". Il "Codice Distribuibile" è codice che il licenziatario potrà distribuire nei programmi che svilupperà, a condizione che si attenga alle condizioni di seguito riportate.

a. Diritto di Utilizzo e Distribuzione.

- Il licenziatario potrà duplicare e distribuire il formato in codice oggetto del supplemento.
- *Distribuzione a Terzi*. Il licenziatario potrà autorizzare i distributori dei programmi del licenziatario stesso a duplicare e distribuire il Codice Distribuibile come parte di tali programmi.

b. Requisiti per la Distribuzione. Per distribuire il Codice Distribuibile, il licenziatario deve:

- aggiungere rilevanti e significative funzionalità al Codice Distribuibile nei programmi del licenziatario;
- distribuire, per ogni file del Codice distribuibile con estensione .lib, solo i risultati dell'esecuzione di tale Codice Distribuibile attraverso un linker al programma;
- distribuire il Codice Distribuibile incluso in un programma di installazione solo nell'ambito di tale programma e senza alcuna modifica;
- far accettare ai distributori e agli utenti finali esterni un contratto con condizioni che garantiscano almeno lo stesso livello di tutela definito nel presente contratto;
- visualizzare una valida comunicazione di copyright del licenziatario stesso nei suoi programmi;
- indennizzare, manlevare e difendere Microsoft da ogni e qualsiasi domanda o azione giudiziaria, ivi incluse le spese legali, relative all'utilizzo o alla distribuzione dei programmi del licenziatario.

c. Restrizioni per la Distribuzione. Il licenziatario non potrà

- modificare le eventuali comunicazioni relative ai copyright, ai marchi o ai brevetti riportati nel Codice Distribuibile;
- utilizzare i marchi di Microsoft nei nomi dei programmi del licenziatario stesso o in modo tale da suggerire che tali programmi provengano o siano riconosciuti in qualsiasi modo da Microsoft;

- distribuire il Codice Distribuibile al fine di essere eseguito su una piattaforma diversa dalla piattaforma Windows;
- includere Codice Distribuibile in programmi dannosi, ingannevoli o illegali;
- modificare o distribuire il codice sorgente di qualsiasi Codice Distribuibile in modo da assoggettare una qualsiasi parte di tale codice alle condizioni di una Licenza Esclusa. Per Licenza Esclusa si intende qualsiasi licenza che richieda, come condizione per l'utilizzo, la modifica o la distribuzione, che
  - il codice sia divulgato o distribuito nel formato in codice sorgente;
  - altri abbiano il diritto di modificarlo.

2. SERVIZI DI SUPPORTO TECNICO PER IL SUPPLEMENTO. Microsoft fornisce servizi di supporto tecnico per questo software come descritto all'indirizzo www.support.microsoft.com/common/international.aspx.

Entity Framework 5 License (JPN)

9/13/2018 • 2 minutes to read • Edit Online

マイクロソフト ソフトウェア 追加ライセンス条項

ENTITY FRAMEWORK 5.0 FOR MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation (以下「マイクロソフト」といいます) は、本追加ソフトウェアのライセンスをお客様に供与します。Microsoft Windows Operating System ソフトウェア(以下「本ソフトウェア」といいます) を使用するためのラインセンスを取得している場合は、本追加ソフトウェアを使用できます。本ソフトウェアのライセンスを取得していない場合は、本追加ソフトウェアを使用することはできません。お客様は、本ソフトウェアの有効なライセンス取得済みの複製 1 部ごとに本追加ソフトウェアを使用できます。

以下のライセンス条項は、について説明しています。これらの条項と本ソフトウェアのライセンス条項が本追加ソフトウェアの使用に適用されます。両者の間に矛盾がある場合は、本追加ライセンス条項が適用されます。

本追加ソフトウェアを使用することにより、お客様はこれらの条項に同意されたものとします。これらの条項に同意されない場合、本追加ソフトウェアを使用することはできません。

お客様がこれらのライセンス条項を遵守することを条件として、お客様は以下が許諾されます。

1. 再頒布可能コード　本追加ソフトウェアは再頒布可能コードで構成されています。「再頒布可能コード」とは、お客様が開発されたプログラムに含めて再頒布することができるコードです。ただし、お客様は以下の条件に従うものとします。

**a. 使用および再頒布の権利　**

- お客様は、本追加ソフトウェアをオブジェクト コード形式で複製し、再頒布することができます。
- *第三者による再頒布*　お客様は、お客様のプログラムの頒布者に対して、お客様のプログラムの一部として再頒布可能コードの複製および頒布を許可することができます。

b. 再頒布の条件　お客様は、お客様が頒布するすべての再頒布可能コードにつき、以下に従わなければなりません。

- お客様のプログラムにおいて再頒布可能コードに重要な新しい機能を追加すること
- .lib というファイル名拡張子が付いた再頒布可能コードの場合は、リンカーによってその再頒布可能コードを実行した結果だけをお客様のプログラムと共に再頒布すること。
- セットアップ プログラムに含まれる再頒布可能コードを、改変されていないセットアップ プログラムの一部としてのみ頒布すること。
- お客様のアプリケーションの頒布者およびエンド ユーザーに、本ライセンス条項と同等以上に再頒布可能コードを保護する条項に同意させること
- お客様のアプリケーションにお客様名義の有効な著作権表示を行うこと
- お客様のプログラムの頒布または使用に関するクレームについて、マイクロソフトを免責、保護、補償すること (弁護士費用についての免責、保護、補償も含む)

c. 再頒布の制限　以下の行為は一切禁止されています。

- 再頒布可能コードの著作権、商標または特許の表示を改変すること
- お客様のプログラムの名称の一部にマイクロソフトの商標を使用したり、お客様の製品がマイクロソフトから由来したり、マイクロソフトが推奨するように見せかけること
- Windows プラットフォーム以外のプラット フォームで実行ために再頒布可能コードを再頒布すること
- 再頒布可能コードを悪質、詐欺的または違法なプログラムに組み込むこと
- 除外ライセンスのいずれかの条項が適用されることとなるような方法で再頒布可能コードのソース コードを改変または再頒布すること。「除外ライセンス」とは、使用、改変または再頒布の条件として以下の条件を満たすことを要求するライセンスです。
  - コードをソース コード形式で公表または頒布すること
  - 他者が改変を行う権利を有すること

2. 本追加ソフトウェアのサポート サービス マイクロソフトは、本ソフトウェアに対し www.support.microsoft.com/common/international.aspx で説明されるサポート サービスを提供します。

Entity Framework 5 License (KOR)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT 소프트웨어 사용권 조항

ENTITY FRAMEWORK 5.0 FOR MICROSOFT WINDOWS OPERATING SYSTEM

Microsoft Corporation(또는 거주 지역에 따라 계열사 중 하나)은 귀하에게 본 추가 구성 요소의 사용권을 부여합니다. 소프트웨어 ("소프트웨어")에 대한 사용권이 있는 경우 Microsoft Windows Operating System 의 추가 구성 요소를 사용할 수 있습니다. 해당 소프트웨어 사용권이 없는 경우에는 추가 구성 요소도 사용할 수 없습니다. 적법하게 사용권이 허여된 소프트웨어의 각 사본에 본 추가 구성 요소를 사용할 수 있습니다.

다음 사용권 조항은 본 추가 구성 요소에 대한 추가적인 사용 조건입니다. 소프트웨어에 대한 이러한 조건 및 사용권 조항은 추가 구성 요소의 사용에 적용됩니다. 상충되는 경우에는 본 추가 구성 요소의 사용권 조항을 적용합니다.

이 추가 구성 요소를 사용하는 것으로 귀하는 아래의 조건들에 동의하게 됩니다. 동의하지 않을 경우에는 추가 구성 요소를 사용하지 마십시오.

본 사용권 조항을 준수하는 경우 아래와 같은 권한을 행사할 수 있습니다.

1. 배포 가능 코드. 본 추가 구성 요소에는 배포 가능 코드가 포함되어 있습니다. "배포 가능 코드"란 귀하가 아래 조항을 준수하는 경우 귀하가 개발하는 프로그램에 배포할 수 있는 코드입니다.

a. 사용 및 배포 권한.

- 귀하는 본 추가 구성 요소를 개체 코드 형태로 복사 및 배포할 수 있습니다.
- *제3자에 의한 배포.* 배포 가능 코드를 프로그램의 일부로 복사 및 배포할 수 있도록 프로그램 배포자에게 허용할 수 있습니다.

b. 배포 조건. 배포 가능 코드를 배포하려면

- 귀하의 프로그램 내에서 배포 가능 코드에 중요한 기능을 추가해야 합니다.
- .lib라는 파일 이름 확장자를 가진 모든 배포 가능 코드의 경우 귀하의 프로그램과 함께 링커를 통한 배포 가능 코드 실행 결과만 배포해야 합니다.
- 설치 프로그램에 포함된 배포 가능 코드를 수정하지 않은 상태로 설치 프로그램의 일부로만 배포해야 합니다.
- 최소한 본 계약에 준하는 배포 가능 코드를 보호할 수 있는 조항에 배포자와 외부의 최종 사용자가 동의하도록 해야 합니다.
- 귀하의 프로그램에 유효한 저작권 표시를 해야 합니다.
- 귀하의 프로그램 배포 또는 사용과 관련된 모든 청구(변호사 비용 포함)로부터 Microsoft를 면책하고 해를 입히지 않으며 방어해야 합니다.

c. 배포 제한. 다음과 같은 행위는 허용되지 않습니다.

- 배포 가능 코드의 저작권, 상표 또는 특허 표시를 변경하는 행위
- Microsoft의 상표를 프로그램 이름에 사용하거나 Microsoft에서 만들거나 보증한다고 광고하는 행위
- 배포 가능 코드를 배포하여 Windows가 아닌 플랫폼에서 실행하는 행위
- 배포 가능 코드를 악의적이고 기만적이거나 불법적인 프로그램에 포함하는 행위
- 배포 가능 코드의 일부분이 예외적 사용권에 적용되도록 배포 가능 코드의 소스 코드를 수정하거나 배포하는 행위. 예외적 사용권이란 사용, 수정 또는 배포를 위해 다음과 같은 조건을 필요로 하는 사용권입니다.
  - 코드를 소스 코드 형태로 공개하거나 배포합니다.
  - 다른 사람이 배포 가능 코드를 수정할 수 있는 권한을 갖습니다.

2. 추가 구성 요소에 대한 지원 서비스. Microsoft는www.support.microsoft.com/common/international.aspx에 기술된 대로 본 소프트웨어에 대한 지원 서비스를 제공합니다.

Entity Framework 5 License (RUS)

9/13/2018 • 2 minutes to read • Edit Online

УСЛОВИЯ ДОПОЛНЕНИЯ К ЛИЦЕНЗИИ КОРПОРАЦИИ МАЙКРОСОФТ НА ИСПОЛЬЗОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ENTITY FRAMEWORK 5.0 ДЛЯ MICROSOFT WINDOWS OPERATING SYSTEM

Корпорация Майкрософт (или одно из ее аффилированных лиц, в зависимости от места вашего проживания) предоставляет вам данное дополнение к лицензии. Если у вас есть лицензия на использование программного обеспечения Microsoft Windows Operating System («программное обеспечение»), вы можете использовать данное дополнение. Вы не имеете права использовать дополнение, если у вас нет лицензии на программное обеспечение. Вы имеете право использовать данное дополнение с каждой лицензионной копией программного обеспечения.

Следующие условия лицензии описывают дополнительные условия использования для данного дополнения. Использование вами этого дополнения регулируется данными условиями и условиями лицензии на использование программного обеспечения. В случае расхождения между этими условиями и условиями лицензии на использование программного обеспечения применяются данные дополнительные условия лицензии.

Используя это дополнение, вы выражаете свое согласие соблюдать эти условия. Если вы не согласны, не используйте это дополнение.

При соблюдении вами этих условияй лицензии вам предоставляются следующие права.

1. ВТОРИЧНО РАСПРОСТРАНЯЕМЫЙ КОД. Данный дополнительный компонент представляет собой вторично распространяемый код. «Вторично распространяемый код» — это код, который разрешается распространять в составе разрабатываемых вами программ при соблюдении изложенных ниже условий.

a. Право на использование и распространение.

- Вы имеете право копировать и распространять дополнительный компонент в виде объектного кода.
- *Распространение третьими лицами.* Вы можете разрешить дистрибьюторам ваших программ копировать и распространять Вторично распространяемый код как часть этих программ.

b. Условия распространения. Для распространения любого Вторично распространяемого кода вы должны:

- существенно расширить основные функциональные возможности кода в своих программах;
- для любого Вторично распространяемого кода с расширением LIB распространять только результаты запуска такого кода посредством компоновщика с помощью вашей программы;
- распространять Вторично распространяемый код, включенный в программу установки, только как часть этой программы установки без изменений;
- потребовать от дистрибуторов и внешних конечных пользователей принять на себя обязательств по соблюдению условий, которые будут защищать Вторично распространяемый код не меньше, чем данное соглашение;
- отображать действительное уведомление об авторских правах в ваших программах;
- освободить от ответственности, оградить и защитить корпорацию Майкрософт от любых претензий и исков, связанных с использованием или распространением ваших программ, включая расходы на оплату услуг адвокатов.

c. Ограничения на распространение. Вы не имеете права:

- изменять уведомления об авторских или патентных правах, а также товарные знаки, присутствующие во Вторично распространяемом коде;
- использовать товарные знаки корпорации Майкрософт в названиях своих программ, а также иным способом, который позволяет предположить, что ваша продукция произведена или рекомендована корпорацией Майкрософт;
- распространять Вторично распространяемый код для использования на платформе, отличной от Windows;
- включать Вторично распространяемый код в программы, созданные со злым умыслом или с целью обмана, а также в незаконные;
- модифицировать или распространять исходный код любого Вторично распространяемого кода таким образом, что какая-либо его часть попадает под действие условий Исключенной лицензии. Исключенная лицензия - это любая лицензия, согласно которой использование, изменение или распространение Вторично распространяемого кода возможно только при соблюдении следующих условий:
  - код раскрывается и распространяется в виде исходного кода;
  - другие имеют право его модифицировать.

2. ТЕХНИЧЕСКАЯ ПОДДЕРЖКА ДОПОЛНЕНИЯ. Корпорация Майкрософт предоставляет техническую поддержку дополнения, как указано на веб-узле www.support.microsoft.com/common/international.aspx.

Entity Framework 6 Runtime Alpha License (ENU)

9/13/2018 • 6 minutes to read • Edit Online

MICROSOFT PRE-RELEASE SOFTWARE LICENSE TERMS

MICROSOFT Entity Framework

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the pre-release software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this software, unless other terms accompany those items. If so, those terms apply.

By using the software, you accept these terms. If you do not accept them, do not use the software.

If you comply with these license terms, you have the rights below.

**1) INSTALLATION AND USE RIGHTS. ** You may install and use any number of copies of the software on your devices.

2) TERM. This agreement will automatically expire on August 1, 2013 or the commercial release of the software, whichever comes first.

3) PRE-RELEASE SOFTWARE. This software is a pre-release version. It may not work the way a final version of the software will. We may change it for the final, commercial version. We also may not release a commercial version.

4) FEEDBACK. If you give feedback about the software to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

5) THIRD PARTY NOTICES. The software may include third party code that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code are included for your information only.

6) Scope of License. The software is licensed, not sold. This agreement only gives you some rights to use the software. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the software only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the software that only allow you to use it in certain ways. You may not

- disclose the results of any benchmark tests of the software to any third party without Microsoft’s prior written approval;
- work around any technical limitations in the software;
- reverse engineer, decompile or disassemble the software, except and only to the extent that applicable law expressly permits, despite this limitation;

- make more copies of the software than specified in this agreement or allowed by applicable law, despite this limitation;
- publish the software for others to copy;
- rent, lease or lend the software; or
- transfer the software or this agreement to any third party.

7) Export Restrictions. The software is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the software. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.

8) SUPPORT SERVICES. Because this software is "as is," we may not provide support services for it.

9) Entire Agreement. This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the software and support services.

10) Applicable Law.

a) United States. If you acquired the software in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.

b) Outside the United States. If you acquired the software in any other country, the laws of that country apply.

11) Legal Effect. This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the software. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

12) Disclaimer of Warranty.The software is licensed "as-is." You bear the risk of using it. Microsoft gives no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this agreement cannot change. To the extent permitted under your local laws, Microsoft excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

13) Limitation on and Exclusion of Remedies and Damages.You can recover from Microsoft and its suppliers only direct damages up to U.S. $5.00. You cannot recover any other damages, including consequential, lost profits, special, indirect or incidental damages.

This limitation applies to

- anything related to the software, services, content (including code) on third party Internet sites, or third party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this software is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce logiciel étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le logiciel visé par une licence est offert « tel quel ». Toute utilisation de ce logiciel est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection des consommateurs, que ce contrat ne peut modifier.

La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 $ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne :

- tout ce qui est relié au logiciel, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Entity Framework 6 Runtime Beta/RC License (ENU)

9/13/2018 • 4 minutes to read • Edit Online

MICROSOFT PRE-RELEASE SOFTWARE LICENSE TERMS

MICROSOFT ENTITY FRAMEWORK

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the pre-release software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this software, unless other terms accompany those items. If so, those terms apply.

BY USING THE SOFTWARE, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT USE THE SOFTWARE.

IF YOU COMPLY WITH THESE LICENSE TERMS, YOU HAVE THE RIGHTS BELOW.

1. INSTALLATION AND USE RIGHTS.

a. Installation and Use.

- You may install and use any number of copies of the software on your premises to design, develop and test your programs for use with the software.
- You may not test the software in a live operating environment unless Microsoft permits you to do so under another agreement.

2. TERM. The term of this agreement is until 31/12/2013 (day/month/year), or commercial release of the software, whichever is first.

3. PRE-RELEASE SOFTWARE. This software is a pre-release version. It may not work the way a final version of the software will. We may change it for the final, commercial version. We also may not release a commercial version.

4. FEEDBACK. If you give feedback about the software to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

5. SCOPE OF LICENSE. The software is licensed, not sold. This agreement only gives you some rights to use the software. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the software only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the software that only allow you to use it in certain ways. You may not

- work around any technical limitations in the software;
- reverse engineer, decompile or disassemble the software, except and only to the extent that applicable law expressly permits, despite this limitation;

- make more copies of the software than specified in this agreement or allowed by applicable law, despite this limitation;
- publish the software for others to copy;
- rent, lease or lend the software;
- transfer the software or this agreement to any third party; or
- use the software for commercial software hosting services.

6. **EXPORT RESTRICTIONS.** The software is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the software. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.

7. **SUPPORT SERVICES.** Because this software is "as is," we may not provide support services for it.

8. **ENTIRE AGREEMENT.** This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the software and support services.

9. **APPLICABLE LAW.**

a. **United States.** If you acquired the software in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.

b. **Outside the United States.** If you acquired the software in any other country, the laws of that country apply.

10. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the software. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

11. DISCLAIMER OF WARRANTY. THE SOFTWARE IS LICENSED "AS-IS." YOU BEAR THE RISK OF USING IT. MICROSOFT GIVES NO EXPRESS WARRANTIES, GUARANTEES OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS OR STATUTORY GUARANTEES UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT EXCLUDES THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.

FOR AUSTRALIA – YOU HAVE STATUTORY GUARANTEES UNDER THE AUSTRALIAN CONSUMER LAW AND NOTHING IN THESE TERMS IS INTENDED TO AFFECT THOSE RIGHTS.

12. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO U.S. $5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.

This limitation applies to

- anything related to the software, services, content (including code) on third party Internet sites, or third party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Entity Framework 6 Runtime License (CHS)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT 软件许可条款

MICROSOFT ENTITY FRAMEWORK

这些许可条款是Microsoft Corporation(或您所在地的 Microsoft 关联公司)与您之间达成的协议。请阅读条款内容。这些条款适用于上述软件，包括您用来接收该软件的介质(如果有)。这些条款也适用于 Microsoft 为本软件提供的任何

- 更新
- 补充程序
- 基于 Internet 的服务，以及
- 支持服务

(除非这些项目附带有其他条款)。如果确实附带有其他条款，应遵守那些条款。

使用该软件，即表示您接受这些条款。如果您不接受这些条款，请不要使用该软件。

如果您遵守这些许可条款，将拥有以下永久权利。

1. 安装和使用权利。您可以在您的设备上安装和使用本软件任意数量的副本。

2. 其他许可要求和/或使用权利。

- **a. 可分发代码。**该软件包含可分发代码。如果您遵守下述条款，则可以在您开发的程序中分发这些代码。
  - **i. 使用权利和分发权利。下列代码和文件为“可分发代码”。**
    - 您可以复制和分发对象代码形式的软件文件。
    - *第三方分发*。您可以允许您的程序分销商将可分发代码作为这些程序的一部分进行复制和分发。
  - **ii. 分发要求。对于您分发的任何可分发代码，您必须**
    - 在您的程序中为其添加重要的主要功能；
    - 要求分销商及外部最终用户同意遵守保护条款且保护范围不得小于本协议；
    - 在您的程序上显示有效的版权声明；以及
    - 对于与分发或使用您的程序有关的任何索赔，为 Microsoft 提供辩护、补偿，包括支付律师费，并使 Microsoft 免受损害。
  - **iii. 分发限制。您不得**
    - 改变可分发代码中的任何版权、商标或专利声明；
    - 在您的程序名称中使用 Microsoft 的商标，或者以其他方式暗示您的程序来自 Microsoft 或经 Microsoft 认可；
    - 分发“可分发代码”以在 Windows 平台以外的任何平台上运行；
    - 在恶意的、欺骗性的或非法的程序中添加可分发代码；或者
    - 修改或分发任何可分发代码的源代码，致使其任何部分受到“排除许可”的制约。排除许可指要求以如下规定为使用、修改或分发条件的许可：
      - 以源代码形式公布或分发代码；或者
      - 其他人有权对其进行修改。

3. 许可范围。软件只授予使用许可，而非出售。本协议只授予您使用软件的某些权利。Microsoft 保留所有其他权利。除非适用的法律赋予您此项限制之外的权利，否则您只能在本协议明示允许的范围内使用软件。为此，您必须遵守该软件中的任何技术限制，这些限制只允许您以特定的方式使用软件。您不得

- 绕过软件中的任何技术限制；
- 对软件进行反向工程、反向编译或反汇编；尽管有此项限制，但如果适用的法律明示允许上述活动，并仅在适用的法律明示允许的范围内从事上述活动则不在此限；
- 发布软件供他人复制；
- 出租、租赁或出借该软件；
- 将该软件或本协议转让给任何第三方；
- 使用软件提供商业软件托管服务。

4. 文档。能够合法访问您的计算机或内部网络的所有用户都可以复制该文档，但仅供内部参考之用。

5. 出口限制。该软件受美国出口法律和法规的约束。您必须遵守适用于该软件的所有国内和国际出口法律和法规。这些法律包括对目的地、最终用户和最终用途的各种限制。有关详细信息，请参阅 www.microsoft.com/exporting。

6. 支持服务。该软件是按"现状"提供的，所以我们可能不为其提供支持服务。

7. 完整协议。本协议以及您使用的补充程序、更新、基于 Internet 的服务和支持服务的有关条款，共同构成了该软件和支持服务的完整协议。

8. 适用的法律。

- **a. 美国。**如果您在美国购买该软件，则对本协议的解释以及由于违反本协议而引起的索赔均以华盛顿州法律为准并受其管辖，而不考虑冲突法原则。您所居住的州的法律管辖其他所有索赔项目，包括根据州消费者保护法、不正当竞争法以及侵权行为提出的相关索赔。
- **b. 美国以外。**如果您在其他任何国家/地区购买该软件，则应遵守该国家/地区的法律。

9. 法律效力。本协议规定了某些合法权利。根据您所在国家/地区的法律规定，您可能享有其他权利。您还可能享有与您的软件卖方相关的权利。如果您所在国家/地区的法律不允许本协议改变您所在国家/地区法律赋予您的权利，则本协议将不改变您按照所在国家/地区的法律应享有的权利。

10. 免责声明。该软件按"现状"授予许可。您须自行承担使用该软件的风险。Microsoft 不提供任何明示的保证、保障或条件。根据您当地的法律，您可能享有本协议无法改变的其他消费者权利或法定保障。在您当地法律允许的范围内，Microsoft 排除有关适销性、针对特定目的的适用性和不侵权的默示保证。

以下内容适用于澳大利亚 - 您享有《澳大利亚消费者法》规定的法定保障，这些条款中的任何规定均无意影响这些权利。

11. 损害补偿责任的限制和排除条款。您只能因直接损害从 Microsoft 及其供应商处获得退款，且退款金额上限为 5.00 美元。您不能因其他任何损害获得退款，包括后果性损害、利润损失、特别的损害、间接损害或附带性损害。

此限制适用于：

- 与第三方 Internet 站点上的软件、服务、内容（包括代码）或第三方程序相关的任何情况；以及
- 在适用的法律允许的范围内，因违约、违反保证、保障或条件、严格责任、过失或其他侵权行为引起的索赔。

即使 Microsoft 知道或应该知道可能会出现损害，此项限制也同样适用。由于您所在国家/地区可能不允许排除或限制附带损害、后果性损害或其他损害的赔偿责任，因此上述限制或排除条款可能对您不适用。

Entity Framework 6 Runtime License (CHT)

9/13/2018 • 2 minutes to read • Edit Online

MICROSOFT 軟體授權條款

MICROSOFT ENTITY FRAMEWORK

本授權條款係一份由 貴用戶與Microsoft Corporation (或其關係企業，視 貴用戶所居住的地點而定) 之間所成立之協議。請仔細閱讀這些授權條款。這些授權條款適用於上述軟體，包括 貴用戶所收受的媒體(如果有的話)。這些條款亦適用於任何Microsoft 之

- 更新程式、
- 增補程式、
- 網際網路服務與
- 支援服務

但若上述項目另附有其他條款，如遇此情形，則其他條款優先適用。

軟體一經使用，即表示 貴用戶接受這些授權條款。若 貴用戶不接受這些授權條款，請不要使用軟體。

若 貴用戶遵守本授權條款，即可永久享有以下權利。

1. 安裝與使用權利。 貴用戶得於裝置上安裝和使用任何數量之軟體拷貝。

2. 其他授權要件及/或使用權利。

- **a. 可散布程式碼。**若 貴用戶遵守以下條款，則 貴用戶得於自己開發的程式中散布軟體包含的部分程式碼。
  - **i. 使用及散布權利。下列程式碼與檔案為「可散布程式碼」。**
    - 貴用戶得以軟體檔案的目的碼形式複製與散布。
    - *第三人散布。* 貴用戶得同意程式經銷商將「可散布程式碼」視為 貴用戶之程式的一部分，進行複製與散布。
  - **ii. 散布要件。針對 貴用戶散布的任何「可散布程式碼」， 貴用戶必須**
    - 在 貴用戶的程式中，為「可散布程式碼」加入重要的主要功能；
    - 要求散布者及外部終端使用者同意「保護『可散布程式碼』的程度不得低於本合約」之相關條款；
    - 在程式中顯示有效的著作權標示；和
    - 若因散布或使用 貴用戶之程式而使 Microsoft 遭他人提出索賠時， 貴用戶應賠償 Microsoft 之損失 (包括律師費)，使之免遭損害，並出面代為辯護。
  - **iii. 散布限制。 貴用戶不得**
    - 變更「可散布程式碼」中之任何著作權、商標或專利聲明；
    - 於 貴用戶的程式名稱使用 Microsoft 的商標，或暗示程式來自 Microsoft 或經由 Microsoft 背書；
    - 散佈「可散佈程式碼」並於非 Windows 的平台上執行；
    - 將「可散布程式碼」置於惡意、欺騙或違法的程式中；或
    - 修改或散布任何可散布程式碼的原始碼，使其任何部分受到除外授權之約束。「除外授權」係指在使用、修改或散布時，應遵守下列條件：
      - 程式碼必須以原始碼形式揭露或散布，或
      - 提供他人修改的權利。

3. 授權範圍。軟體係授權使用，而非出售賣斷。本合約僅提供 貴用戶使用軟體的部分權利。Microsoft 保留所有其他權利。除非相關法律賦予 貴用戶超出本合約限制的其他權利，否則 貴用戶僅得在本合約明示許可之範圍內

使用軟體。因此， 貴用戶必須遵守只允許以特定方式使用軟體的科技保護措施。 貴用戶不得

- 規避軟體中所包含的科技保護措施；
- 對軟體進行還原工程、解編或反向組譯，但儘管有此限制相關法律仍明文允許者，不在此限；
- 將軟體發佈給其他人進行複製；
- 出租、租賃或出借軟體；
- 將軟體或本合約移轉給任何第三人；或者
- 利用軟體提供商業軟體主機服務。

4. 說明文件。任何有權存取 貴用戶之電腦或內部網路的人，皆得基於 貴用戶內部參考之目的，複製及使用該說明文件。

5. 出口限制。軟體受到美國出口法令規定之規範。 貴用戶必須遵守適用於軟體之一切本國及國際出口法令規定之規範。這些法規包括目的地限制、使用者限制和使用用途限制。如需詳細資訊，請參閱 www.microsoft.com/exporting。

6. 支援服務。本軟體係依「現況」提供，因此本公司得不提供支援服務。

7. 整份合約。本合約以及 貴用戶所使用的增補程式、更新程式、網際網路服務和支援服務之條款構成關於軟體和支援服務之整份合約。

8. 準據法。

- **a. 美國。**若 貴用戶在美國境內取得軟體，本合約之解釋或任何違反本合約所衍生的訴訟，無論是否有法規衝突產生，均應以美國華盛頓州之法律做為準據法。所有其他訴訟將以 貴用戶居住之州法律為準據法，包含違反州消費者保護法、不當競爭法和侵權行為的訴訟。
- **b. 美國境外。**若 貴用戶在美國以外的國家/地區取得軟體，則本合約應以 貴用戶所居住之國家/地區的法律為準據法。

9. 法律效力。本合約敘述了特定的法律權利。 貴用戶所在國家的法律可能會提供 貴用戶其他權利。此外， 貴用戶取得軟體的單位可能也會提供相關的權利。若 貴用戶所在之國家/地區法律不允許，則本合約無法改變 貴用戶所在之國家/地區法律提供給 貴用戶的權利。

10. 不為瑕疵擔保之聲明。軟體係依「現況」授權。 貴用戶須自行承擔使用風險。Microsoft 不提供明示擔保、保證或條件。 貴用戶所在地區的法律可能會提供本合約無法改變的額外消費者權利或法律保證。在 貴用戶所屬當地法律允許之範圍內，Microsoft 可排除適售性、符合特定目的或未侵權之默示擔保。

僅適用於澳大利亞 – 貴用戶依據澳大利亞消費者法律 (Australian Consumer Law) 享有法定保證，本合約條款並不意圖影響這些權利。

11. 救濟權與損害賠償責任之限制與排除。 貴用戶僅得就直接損害，要求 Microsoft 及其供應商負擔損害賠償責任，且其金額不得超過 $5.00 美元。 貴用戶無法就其他損害 (包括衍生性損害、利潤損失、特殊損害、間接損害或附隨性損害) 請求損害賠償。

這項限制適用於

- 與軟體、服務、第三方廠商網站上的內容 (包括程式碼) 或第三方廠商程式相關的任何事項；和
- 在相關法律許可的範圍之內，因為違反合約、瑕疵擔保、保證或條件、無過失責任、過失或其他侵權行為所主張之訴訟案件。

即使 Microsoft 已知悉或應知悉該等損害發生之可能性，此項限制仍然適用。此外， 貴用戶所在之國家/地區也可能不允許對附隨性損害、衍生性損害或其他損害加以排除或限制，這種情況也可能造成上述限制或排除規定並不適用於 貴用戶。

Entity Framework 6 Runtime License (DEU)

9/13/2018 • 6 minutes to read • Edit Online

MICROSOFT-SOFTWARELIZENZBESTIMMUNGEN

MICROSOFT ENTITY FRAMEWORK

Diese Lizenzbestimmungen sind ein Vertrag zwischen Ihnen und der Microsoft Corporation (oder einer anderen Microsoft-Konzerngesellschaft, wenn diese an dem Ort, an dem Sie leben, die Software lizenziert). Bitte lesen Sie die Bestimmungen aufmerksam durch. Sie gelten für die oben genannte Software und gegebenenfalls für die Medien, auf denen Sie diese erhalten haben. Diese Bestimmungen gelten auch für alle von Microsoft diesbezüglich angebotenen

- Updates
- Ergänzungen
- internetbasierten Dienste und
- Supportservices.

Liegen letztgenannten Elementen eigene Bestimmungen bei, gelten diese eigenen Bestimmungen.

Durch die Verwendung der Software erkennen Sie diese Bestimmungen an. Falls Sie die Bestimmungen nicht akzeptieren, sind Sie nicht berechtigt, die Software zu verwenden.

Wenn Sie diese Lizenzbestimmungen einhalten, verfügen Sie über die nachfolgend aufgeführten zeitlich unbeschränkten Rechte.

1. RECHTE ZUR INSTALLATION UND NUTZUNG. Sie sind berechtigt, eine beliebige Anzahl von Kopien der Software auf Ihren Geräten zu installieren und zu verwenden.

2. ZUSÄTZLICHE LIZENZANFORDERUNGEN UND/ODER NUTZUNGSRECHTE.

- **a. Vertreibbarer Code.** Die Software enthält Code, den Sie in von Ihnen entwickelten Programmen vertreiben dürfen, wenn Sie die nachfolgenden Bestimmungen einhalten.
  - **i. Recht zur Nutzung und zum Vertrieb. Bei dem nachfolgend aufgelisteten Code und den nachfolgend aufgelisteten Dateien handelt es sich um „Vertreibbaren Code".**
    - Sie sind berechtigt, die Objektcodeform der Softwaredateien zu kopieren und zu vertreiben.
    - *Vertrieb durch Dritte*. Sie sind berechtigt, Distributoren Ihrer Programme zu erlauben, den Vertreibbaren Code als Teil dieser Programme zu kopieren und zu vertreiben.
  - **ii. Vertriebsbedingungen. Für Vertreibbaren Code, den Sie vertreiben, sind Sie verpflichtet:**
    - diesem in Ihren Programmen wesentliche primäre Funktionalität hinzuzufügen
    - von Distributoren und externen Endbenutzern die Zustimmung zu Bestimmungen zu verlangen, die einen mindestens gleichwertigen Schutz für ihn bieten wie dieser Vertrag
    - Ihren gültigen Urheberrechtshinweis auf Ihren Programmen anzubringen
    - Microsoft von allen Ansprüchen freizustellen und gegen alle Ansprüche zu verteidigen, einschließlich Anwaltsgebühren, die mit dem Vertrieb oder der Verwendung Ihrer Programme in Zusammenhang stehen.
  - **iii. Vertriebsbeschränkungen. Sie sind nicht dazu berechtigt:**
    - Urheberrechts-, Markenrechts- oder Patenthinweise im Vertreibbaren Code zu ändern
    - die Marken von Microsoft in den Namen Ihrer Programme oder auf eine Weise zu verwenden, die nahe legt, dass Ihre Programme von Microsoft stammen oder von Microsoft empfohlen werden

  - Vertreibbaren Code zur Ausführung auf einer anderen Plattform als der Windows-Plattform zu vertreiben
  - Vertreibbaren Code in bösartige, täuschende oder rechtswidrige Programme aufzunehmen
  - den Quellcode von Vertreibbarem Code so zu ändern oder zu vertreiben, dass irgendein Teil von ihm einer Ausgeschlossenen Lizenz unterliegt. Eine Ausgeschlossene Lizenz ist eine Lizenz, die als Bedingung für eine Verwendung, Änderung oder einen Vertrieb erfordert, dass:
    - der Code in Quellcodeform offengelegt oder vertrieben wird oder
    - andere das Recht haben, ihn zu ändern.

3. **GÜLTIGKEITSBEREICH DER LIZENZ.** Die Software wird lizenziert, nicht verkauft. Dieser Vertrag gibt Ihnen nur einige Rechte zur Verwendung der Software. Microsoft behält sich alle anderen Rechte vor. Sie dürfen die Software nur wie in diesem Vertrag ausdrücklich gestattet verwenden, es sei denn, das anwendbare Recht gibt Ihnen ungeachtet dieser Einschränkung umfassendere Rechte. Dabei sind Sie verpflichtet, alle technischen Beschränkungen der Software einzuhalten, die Ihnen nur spezielle Verwendungen gestatten. Sie sind nicht dazu berechtigt:

- technische Beschränkungen der Software zu umgehen
- die Software zurückzuentwickeln (Reverse Engineering), zu dekompilieren oder zu disassemblieren, es sei denn, dass (und nur insoweit) es das anwendbare Recht ungeachtet dieser Einschränkung ausdrücklich gestattet
- die Software zu veröffentlichen, damit andere sie kopieren können
- die Software zu vermieten, zu verleasen oder zu verleihen
- die Software oder diesen Vertrag an Dritte zu übertragen oder
- die Software für kommerzielle Software-Hostingdienste zu verwenden.

4. **DOKUMENTATION.** Jede Person, die über einen gültigen Zugriff auf Ihren Computer oder Ihr internes Netzwerk verfügt, ist berechtigt, die Dokumentation zu Ihren internen Referenzzwecken zu kopieren und zu verwenden.

5. **AUSFUHRBESCHRÄNKUNGEN.** Die Software unterliegt den Exportgesetzen und -regelungen der USA sowie des Landes, aus dem sie ausgeführt wird. Sie sind verpflichtet, alle nationalen und internationalen Exportgesetze und -regelungen einzuhalten, die für die Software gelten. Diese Gesetze enthalten auch Beschränkungen in Bezug auf die Endnutzer und Endnutzung. Weitere Informationen finden Sie unter www.microsoft.com/exporting.

6. **SUPPORTSERVICES.** Da diese Software „wie besehen" bereitgestellt wird, stellen wir möglicherweise keine Supportservices für sie bereit.

7. **GESAMTER VERTRAG.** Dieser Vertrag sowie die Bestimmungen für von Ihnen verwendete Ergänzungen, Updates, internetbasierte Dienste und Supportservices stellen den gesamten Vertrag für die Software und die Supportservices dar.

8. **ANWENDBARES RECHT.**

- a. **Vereinigte Staaten.** Wenn Sie die Software in den Vereinigten Staaten erworben haben, regelt das Gesetz des Staates Washington die Auslegung dieses Vertrages und gilt für Ansprüche, die aus einer Vertragsverletzung entstehen, ungeachtet der Bestimmungen des internationalen Privatrechts. Die Gesetze des Staates Ihres Wohnorts regeln alle anderen Ansprüche, einschließlich Ansprüche aus den Verbraucherschutzgesetzen des Staates, aus Gesetzen gegen unlauteren Wettbewerb und aus Deliktsrecht.
- b. **Außerhalb der Vereinigten Staaten.** Wenn Sie die Software in einem anderen Land erworben haben, gelten die Gesetze dieses Landes.

9. **RECHTLICHE WIRKUNG.** Dieser Vertrag beschreibt bestimmte Rechte. Möglicherweise haben Sie unter den Gesetzen Ihres Landes weitergehende Rechte. Möglicherweise verfügen Sie außerdem über Rechte im Hinblick auf die Partei, von der Sie die Software erworben haben. Dieser Vertrag ändert nicht Ihre Rechte, die sich aus den

Gesetzen Ihres Landes ergeben, sofern die Gesetze Ihres Landes dies nicht zulassen.

10. AUSSCHLUSS VON GARANTIEN. Die Software wird „wie besehen“ lizenziert. Sie tragen das mit der Verwendung verbundene Risiko. Microsoft gewährt keine ausdrücklichen Gewährleistungen oder Garantien. Möglicherweise gelten unter den örtlich anwendbaren Gesetzen zusätzliche Verbraucherrechte oder gesetzliche Garantien, die durch diesen Vertrag nicht abgeändert werden können. Im durch das örtlich anwendbare Recht zugelassenen Umfang schließt Microsoft konkludente Garantien der Handelsüblichkeit, Eignung für einen bestimmten Zweck und Nichtverletzung von Rechten Dritter aus.

FÜR AUSTRALIEN – Nach dem Australian Consumer Law gelten gesetzliche Garantien, und es besteht an keiner Stelle in diesen Bestimmungen die Absicht, diese Rechte einzuschränken.

11. BESCHRÄNKUNG UND AUSSCHLUSS DES SCHADENERSATZES. Sie können von Microsoft und deren Lieferanten nur einen Ersatz für direkte Schäden bis zu einem Betrag von 5,00 US-Dollar erhalten. Sie können keinen Ersatz für andere Schäden erhalten, einschließlich Folgeschäden, Schäden aus entgangenem Gewinn, spezielle, indirekte oder zufällige Schäden.

Diese Beschränkung gilt für

- jeden Gegenstand im Zusammenhang mit der Software, Diensten, Inhalten (einschließlich Code) auf Internetseiten von Drittanbietern oder Programmen von Drittanbietern und
- Ansprüche aus Vertragsverletzungen, Verletzungen der Garantie oder der Gewährleistung, verschuldensunabhängiger Haftung, Fahrlässigkeit oder anderen unerlaubten Handlungen im durch das anwendbare Recht zugelassenen Umfang.

Sie hat auch dann Gültigkeit, wenn Microsoft von der Möglichkeit der Schäden gewusst hat oder hätte wissen müssen. Obige Beschränkung und obiger Ausschluss gelten möglicherweise nicht für Sie, weil Ihr Land den Ausschluss oder die Beschränkung von zufälligen Schäden, Folgeschäden oder sonstigen Schäden nicht gestattet. Wenn Sie die Software in DEUTSCHLAND oder in ÖSTERREICH erworben haben, findet die Beschränkung im vorstehenden Absatz „Beschränkung und Ausschluss des Schadenersatzes“ auf Sie keine Anwendung. Stattdessen gelten für Schadenersatz oder Ersatz vergeblicher Aufwendungen, gleich aus welchem Rechtsgrund einschließlich unerlaubter Handlung, die folgenden Regelungen: Microsoft haftet bei Vorsatz, grober Fahrlässigkeit, bei Ansprüchen nach dem Produkthaftungsgesetz sowie bei Verletzung von Leben, Körper oder der Gesundheit nach den gesetzlichen Vorschriften. Microsoft haftet nicht für leichte Fahrlässigkeit. Wenn Sie die Software jedoch in Deutschland erworben haben, haftet Microsoft auch für leichte Fahrlässigkeit, wenn Microsoft eine Vertragspflicht verletzt, deren Erfüllung die ordnungsgemäße Durchführung des Vertrages überhaupt erst ermöglicht, deren Verletzung die Erreichung des Vertragszwecks gefährdet und auf deren Einhaltung Sie regelmäßig vertrauen dürfen (sog. „Kardinalpflichten“). In diesen Fällen ist die Haftung von Microsoft auf typische und vorhersehbare Schäden beschränkt. In allen anderen Fällen haftet Microsoft auch in Deutschland nicht für leichte Fahrlässigkeit.

Entity Framework 6 Runtime License (ENU)

9/13/2018 • 5 minutes to read • Edit Online

MICROSOFT SOFTWARE LICENSE TERMS

MICROSOFT ENTITY FRAMEWORK

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this software, unless other terms accompany those items. If so, those terms apply.

By using the software, you accept these terms. If you do not accept them, do not use the software.

If you comply with these license terms, you have the perpetual rights below.

1. INSTALLATION AND USE RIGHTS. You may install and use any number of copies of the software on your devices.

2. ADDITIONAL LICENSING REQUIREMENTS AND/OR USE RIGHTS.

- **a. Distributable Code.** The software contains code that you are permitted to distribute in programs you develop if you comply with the terms below.
  - **i. Right to Use and Distribute. The code and files listed below are “Distributable Code.”**
    - You may copy and distribute the object code form of the software files.
    - *Third Party Distribution*. You may permit distributors of your programs to copy and distribute the Distributable Code as part of those programs.
  - **ii. Distribution Requirements. For any Distributable Code you distribute, you must**
    - add significant primary functionality to it in your programs;
    - require distributors and external end users to agree to terms that protect it at least as much as this agreement;
    - display your valid copyright notice on your programs; and
    - indemnify, defend, and hold harmless Microsoft from any claims, including attorneys’ fees, related to the distribution or use of your programs.
  - **iii. Distribution Restrictions. You may not**
    - alter any copyright, trademark or patent notice in the Distributable Code;
    - use Microsoft’s trademarks in your programs’ names or in a way that suggests your programs come from or are endorsed by Microsoft;
    - distribute Distributable Code, to run on a platform other than the Windows platform;
    - include Distributable Code in malicious, deceptive or unlawful programs; or
    - modify or distribute the source code of any Distributable Code so that any part of it becomes subject to an Excluded License. An Excluded License is one that requires, as a condition of use, modification or distribution, that
      - the code be disclosed or distributed in source code form; or

- others have the right to modify it.

3. **SCOPE OF LICENSE.** The software is licensed, not sold. This agreement only gives you some rights to use the software. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the software only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the software that only allow you to use it in certain ways. You may not

- work around any technical limitations in the software;
- reverse engineer, decompile or disassemble the software, except and only to the extent that applicable law expressly permits, despite this limitation;
- publish the software for others to copy;
- rent, lease or lend the software;
- transfer the software or this agreement to any third party; or
- use the software for commercial software hosting services.

4. **DOCUMENTATION.** Any person that has valid access to your computer or internal network may copy and use the documentation for your internal, reference purposes.

5. **EXPORT RESTRICTIONS.** The software is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the software. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.

6. **SUPPORT SERVICES.** Because this software is "as is," we may not provide support services for it.

7. **ENTIRE AGREEMENT.** This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the software and support services.

8. **APPLICABLE LAW.**

- a. **United States**. If you acquired the software in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
- b. **Outside the United States.** If you acquired the software in any other country, the laws of that country apply.

9. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the software. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

10. **DISCLAIMER OF WARRANTY. The software is licensed "as-is." You bear the risk of using it. Microsoft gives no express warranties, guarantees or conditions. You may have additional consumer rights or statutory guarantees under your local laws which this agreement cannot change. To the extent permitted under your local laws, Microsoft excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.**

FOR AUSTRALIA – You have statutory guarantees under the Australian Consumer Law and nothing in these terms is intended to affect those rights.

11. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. You can recover from Microsoft and its suppliers only direct damages up to U.S. $5.00. You cannot recover any other damages, including consequential, lost profits, special, indirect or incidental damages.**

This limitation applies to

- anything related to the software, services, content (including code) on third party Internet sites, or third party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Entity Framework 6 Runtime License (ESN)

9/13/2018 • 5 minutes to read • Edit Online

TÉRMINOS DE LICENCIA DEL SOFTWARE DE MICROSOFT

MICROSOFT ENTITY FRAMEWORK

Los presentes términos de licencia constituyen un contrato entre Microsoft Corporation (o, en función de donde resida, una de sus filiales) y usted. Le rogamos que los lea atentamente. Se aplican al software antes mencionado, el cual incluye los soportes físicos en los que lo haya recibido, si los hubiera. Estos términos también se aplicarán a los siguientes elementos de Microsoft:

- actualizaciones,
- complementos,
- servicios basados en Internet, y
- servicios de soporte técnico.

Todos ellos deben corresponder a este software, salvo que existan otros términos aplicables a dichos elementos. En tal caso, se aplicarán esos términos.

Al hacer uso del software, estará aceptando estos términos. Si no los acepta, no utilice el software.

Si cumple con estos términos de licencia, tendrá los derechos perpetuos que se describen a continuación.

1. **DERECHOS DE INSTALACIÓN Y USO.** Podrá instalar y utilizar cualquier número de copias del software en sus dispositivos.

2. **REQUISITOS DE LICENCIA Y/O DERECHOS DE USO ADICIONALES.**

- **a. Código Distribuible.** El software contiene código que puede distribuir en los programas que desarrolle siempre que cumpla con los términos siguientes.
  - **i. Derecho a utilizar y distribuir. El código y los archivos que se indican a continuación son "Código Distribuible".**
    - Puede copiar y distribuir el código objeto de los siguientes archivos de software.
    - *Distribución de terceros*. Puede permitir a los distribuidores de sus programas que copien y distribuyan el Código Distribuible como parte de esos programas.
  - **ii. Requisitos de distribución. Para cualquier Código Distribuible que distribuya debe:**
    - agregarle una funcionalidad primaria significativa en sus programas;
    - exigir a los distribuidores y usuarios finales externos que acepten términos que lo protejan en la misma medida que este contrato;
    - mostrar su aviso de propiedad intelectual válido en sus programas; e
    - indemnizar, defender y eximir de responsabilidad a Microsoft ante cualquier reclamación, incluidos honorarios de abogados, relacionada con la distribución o el uso de sus programas.
  - **iii. Restricciones de distribución. No podrá:**
    - alterar ningún aviso de propiedad intelectual, marca o patente en el Código Distribuible;
    - utilizar las marcas de Microsoft en los nombres de sus programas de una forma que sugiera que estos provienen de Microsoft o que esta los respalda;
    - distribuir el Código Distribuible para que se ejecute en una plataforma distinta de Windows;
    - incluir Código Distribuible en programas malintencionados, engañosos o ilícitos; o

- modificar o distribuir el código fuente de cualquier Código Distribuible para que cualquier parte de él esté sujeta a una Licencia Excluida. Una Licencia Excluida es una que exige, como condición de uso, modificación o distribución que
  - el código se revele o distribuya en código fuente; u
  - otros tengan el derecho de modificarlo.

3. ÁMBITO DE LA LICENCIA. El software se cede sujeto a licencia y no es objeto de venta. El presente contrato solo le otorga algunos derechos de uso del software. Microsoft se reserva todos los demás derechos. A menos que la legislación aplicable le otorgue más derechos a pesar de esta limitación, solo podrá utilizar el software tal como se permite expresamente en este contrato. Al hacerlo, deberá ajustarse a las limitaciones técnicas del software que solo permiten utilizarlo de determinadas formas. No podrá:

- eludir las limitaciones técnicas del software;
- utilizar técnicas de ingeniería inversa, descompilar o desensamblar el software, excepto y únicamente en la medida en que lo permita expresamente la legislación aplicable, a pesar de la presente limitación;
- hacer público el software para que otros lo copien;
- alquilar, arrendar o dar en préstamo el software;
- transmitir el software o este contrato a un tercero; o
- utilizar el software para prestar servicios de hosting de software comercial.

4. DOCUMENTACIÓN. Toda persona que tenga acceso válido a su equipo o a la red interna puede copiar y utilizar la documentación a efectos internos de consulta.

5. RESTRICCIONES EN MATERIA DE EXPORTACIÓN. El software está sujeto a las leyes y reglamentos en materia de exportación de los Estados Unidos. Debe cumplir con todas las leyes y reglamentos, nacionales e internacionales, en materia de exportación que sean de aplicación al software. Dichas leyes incluyen limitaciones en cuanto a destino, usuarios finales y uso final. Para obtener más información, visite www.microsoft.com/exporting.

6. SERVICIOS DE SOPORTE TÉCNICO. Debido a que este software se presenta “tal cual”, no podemos prestar servicios de soporte técnico para el mismo.

7. CONTRATO COMPLETO. El presente contrato y los términos aplicables a complementos, actualizaciones, servicios basados en Internet y servicios de soporte técnico que utilice constituyen el contrato completo respecto al software y a los servicios de soporte técnico.

8. LEGISLACIÓN APLICABLE.

- **a. Estados Unidos.** Si adquirió el software en los Estados Unidos, la interpretación del presente contrato se regirá por la legislación del Estado de Washington, que se aplicará a las reclamaciones por incumplimiento del mismo, con independencia de conflictos de principios legales. Para el resto de las reclamaciones, será aplicable la legislación de su estado de residencia, incluidas las reclamaciones en virtud de las leyes estatales en materia de protección al consumidor, competencia desleal y responsabilidad extracontractual.
- **b. Fuera de los Estados Unidos.** Si adquirió el software en otro país, se aplicará la legislación de dicho país.

9. EFECTOS LEGALES. El presente contrato describe determinados derechos legales. Es posible que disponga de otros derechos en virtud de la legislación de su país. Asimismo, pueden asistirle determinados derechos con respecto a la parte de la que adquirió el software. Este contrato no modifica los derechos de los que dispone en virtud de la legislación de su país si dicha legislación no permite tal cosa.

10. RENUNCIA DE GARANTÍA. El software se concede con licencia “tal cual”, por consiguiente, usted asume el riesgo de utilizarlo. Microsoft no otorga ninguna garantía ni condición expresa. Es posible que la legislación local le otorgue derechos del consumidor o garantías legales adicionales que el presente contrato no pueda modificar. En la medida en que así lo permita la legislación local, Microsoft excluye las garantías implícitas de comerciabilidad, idoneidad para un fin particular y ausencia de infracción de derechos.

PARA AUSTRALIA: Usted cuenta con garantías legales conforme a la ley australiana del consumidor y nada en estos términos pretende afectar dichos derechos.

11. LIMITACIÓN Y EXCLUSIÓN DE RECURSOS E INDEMNIZACIONES. La cantidad máxima que se podrá obtener de Microsoft y de sus proveedores en concepto de indemnización por daños directos será de $5,00 dólares estadounidenses. No podrá obtener indemnización alguna por daños de otra índole, incluidos los daños consecuenciales, por lucro cesante, especiales, indirectos o incidentales.

Esta limitación se aplica a:

- Cualquier cuestión relacionada con el software, los servicios, el contenido (incluido el código) que se hallen en sitios de Internet de terceros o programas de terceros.
- Reclamaciones por incumplimiento de contrato, incumplimiento de garantía o condición, responsabilidad objetiva, negligencia u otra responsabilidad extracontractual en la medida permitida por la legislación aplicable.

Asimismo, también será de aplicación incluso si Microsoft conocía o debería haber conocido la posibilidad de que se produjesen dichos daños. También pueden producirse situaciones en las que la limitación o exclusión precedente no pueda aplicarse a su caso porque su jurisdicción no admita la exclusión o limitación de daños incidentales consecuenciales o de otra índole.

Entity Framework 6 Runtime License (FRA)

9/13/2018 • 6 minutes to read • Edit Online

TERMES DU CONTRAT DE LICENCE LOGICIEL MICROSOFT

MICROSOFT ENTITY FRAMEWORK

Les présents termes du contrat de licence constituent un contrat entre Microsoft Corporation (ou en fonction du lieu où vous vivez, l'un de ses affiliés) et vous. Lisez-les attentivement. Ils portent sur le logiciel visé ci-dessus, y compris le support sur lequel vous l'avez reçu, le cas échéant. Ce contrat porte également sur les produits Microsoft suivants :

- les mises à jour,
- les suppléments,
- les services Internet, et
- les services d'assistance technique

de ce logiciel à moins que d'autres termes n'accompagnent ces produits, auquel cas ces derniers prévalent.

En utilisant le logiciel, vous acceptez ces termes. Si vous ne les acceptez pas, n'utilisez pas le logiciel.

Dans le cadre du présent contrat de licence, vous disposez des droits perpétuels présentés ci-dessous.

1. INSTALLATION ET DROITS D'UTILISATION. Vous êtes autorisé à installer et utiliser un nombre quelconque de copies du logiciel sur vos dispositifs.

2. CONDITIONS DE LICENCE ET/OU DROITS D'UTILISATION SUPPLÉMENTAIRES.

- **a. Code distribuable.** Le logiciel contient du code que vous êtes autorisé à distribuer dans les programmes que vous développez, sous réserve de vous conformer aux termes ci-après.
  - **i. Droit d'utilisation et de distribution. Le code et les fichiers répertoriés ci-après constituent le « Code distribuable ».**
    - Vous êtes autorisé à copier et à distribuer la version en code objet des fichiers du logiciel.
    - *Distribution par des tiers.* vous pouvez autoriser les distributeurs de vos programmes à copier et à distribuer le Code distribuable en tant que partie intégrante de ces programmes.
  - **ii. Conditions de distribution. Pour tout Code distribuable que vous distribuez, vous devez :**
    - y ajouter des fonctionnalités importantes et principales au sein de vos programmes,
    - exiger des distributeurs et des utilisateurs finaux externes qu'ils acceptent les termes qui protègent le Code Distribuable de manière au moins équivalente à ceux du présent contrat ;
    - afficher votre propre mention de droits d'auteur valable dans vos programmes, et
    - garantir et défendre Microsoft contre toute réclamation, y compris pour les honoraires d'avocats, qui résulterait de la distribution ou de l'utilisation de vos programmes.
  - **iii. Restrictions de distribution. Vous n'êtes pas autorisé à :**
    - modifier toute mention de droits d'auteur, marques commerciales ou mention de droits de brevet pouvant figurer dans le Code distribuable,
    - utiliser les marques commerciales de Microsoft dans les noms de vos programmes ou d'une façon qui suggère que vos programmes sont fournis ou recommandés par Microsoft,
    - distribuer le Code distribuable en vue de son exécution sur une plateforme autre que la plateforme Windows,
    - inclure le Code distribuable dans des programmes malveillants, trompeurs ou interdits par la loi ;

ou

  - modifier ou distribuer le code source de tout Code distribuable de manière à ce qu'il fasse l'objet, en tout ou partie, d'une Licence exclue. Une Licence exclue implique comme condition d'utilisation, de modification ou de distribution, que :
    - le code soit divulgué ou distribué sous forme de code source, ou
    - d'autres aient le droit de le modifier.

3. **CHAMP D'APPLICATION DE LA LICENCE.** Le logiciel n'est pas vendu mais concédé sous licence. Le présent contrat vous confère certains droits d'utilisation du logiciel. Microsoft se réserve tous les autres droits. Sauf si la réglementation applicable vous confère d'autres droits, nonobstant la présente limitation, vous n'êtes autorisé à utiliser le logiciel qu'en conformité avec les termes du présent contrat. À cette fin, vous devez vous conformer aux restrictions techniques contenues dans le logiciel qui vous permettent de l'utiliser d'une certaine façon. Vous n'êtes pas autorisé à :

- contourner les restrictions techniques contenues dans le logiciel ;
- reconstituer la logique du logiciel, le décompiler ou le désassembler, sauf dans la mesure où ces opérations seraient expressément permises par la réglementation applicable nonobstant la présente limitation ;
- publier le logiciel en vue d'une reproduction par autrui ;
- louer ou prêter le logiciel ;
- transférer le logiciel ou le présent contrat à un tiers ; ou
- utiliser le logiciel en association avec des services d'hébergement commercial.

4. **DOCUMENTATION.** Tout utilisateur disposant d'un accès valable à votre ordinateur ou à votre réseau interne est autorisé à copier et à utiliser la documentation à titre de référence et à des fins internes.

5. **RESTRICTIONS À L'EXPORTATION.** Le logiciel est soumis aux lois et réglementations américaines en matière d'exportation. Vous devez vous conformer à toutes les lois et réglementations nationales et internationales en matière d'exportation concernant le logiciel. Ces lois comportent des restrictions sur les utilisateurs finaux et les utilisations finales. Des informations supplémentaires sont disponibles sur le site www.microsoft.com/exporting.

6. **SERVICES D'ASSISTANCE TECHNIQUE.** Ce logiciel étant fourni « en l'état », nous ne fournirons aucun service d'assistance technique.

7. **INTÉGRALITÉ DES ACCORDS.** Le présent contrat ainsi que les termes concernant les suppléments, les mises à jour, les services Internet et d'assistance technique que vous utilisez constituent l'intégralité des accords en ce qui concerne le logiciel et les services d'assistance technique.

8. **RÉGLEMENTATION APPLICABLE.**

- **a. États-Unis.** Si vous avez acquis le logiciel aux États-Unis, les lois de l'État de Washington, États-Unis d'Amérique, régissent l'interprétation de ce contrat et s'appliquent en cas de réclamation pour manquement aux termes du contrat, sans donner d'effet aux dispositions régissant les conflits de lois. Les lois du pays dans lequel vous vivez régissent toutes les autres réclamations, notamment les réclamations fondées sur les lois fédérales en matière de protection des consommateurs, de concurrence déloyale et de délits.
- **b. En dehors des États-Unis.** Si vous avez acquis le logiciel dans un autre pays, les lois de ce pays s'appliquent.

9. **EFFET JURIDIQUE.** Le présent contrat décrit certains droits légaux. Vous pouvez bénéficier d'autres droits prévus par les lois de votre État ou pays. Vous pouvez également bénéficier de certains droits à l'égard de la partie auprès de laquelle vous avez acquis le logiciel. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre État ou pays si celles-ci ne le permettent pas.

10. **EXCLUSIONS DE GARANTIE. Le logiciel est concédé sous licence « en l'état ». Vous assumez tous les risques liés à son utilisation. Microsoft n'accorde aucune garantie ou condition expresse. Vous pouvez bénéficier de droits des consommateurs supplémentaires ou de garanties statutaires dans le**

cadre du droit local, que ce contrat ne peut modifier. Lorsque cela est autorisé par le droit local, Microsoft exclut les garanties implicites de qualité, d'adéquation à un usage particulier et d'absence de violation.

POUR L'AUSTRALIE – La loi australienne sur la consommation (Australian Consumer Law) vous accorde des garanties statutaires qu'aucun élément du présent accord ne peut affecter.

11. LIMITATION ET EXCLUSION DE RECOURS ET DE DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs limitée à 5,00 USD. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects, incidents ou accessoires et les pertes de bénéfices.

Cette limitation concerne :

- toute affaire liée au logiciel, aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations pour manquement aux termes du contrat ou violation de garantie, les réclamations en cas de responsabilité sans faute, de négligence ou autre délit dans la limite autorisée par la réglementation applicable.

Elle s'applique également même si Microsoft connaissait l'éventualité d'un tel dommage. La limitation ou l'exclusion cidessus peut également ne pas vous être applicable si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages incidents, indirects ou de quelque nature que ce soit.

Entity Framework 6 Runtime License (ITA)

9/13/2018 • 6 minutes to read • Edit Online

CONDIZIONI DI LICENZA SOFTWARE MICROSOFT

MICROSOFT ENTITY FRAMEWORK

Le presenti condizioni di licenza costituiscono il contratto tra Microsoft Corporation (o, in base al luogo di residenza del licenziatario, una delle sue consociate) e il licenziatario. Il licenziatario dovrà leggerle con attenzione. Le presenti condizioni si applicano al suddetto software Microsoft, inclusi gli eventuali supporti di memorizzazione sui quali è stato ricevuto. Le presenti condizioni si applicano inoltre a

- aggiornamenti,
- supplementi,
- servizi basati su Internet e
- servizi di supporto tecnico

forniti da Microsoft e relativi al predetto software, a meno che questi non siano accompagnati da condizioni specifiche. In tal caso, tali condizioni specifiche prevarranno su quelle del presente contratto.

Utilizzando il software, il licenziatario accetta le presenti condizioni. Qualora non le accetti, non potrà utilizzare il software.

Qualora il licenziatario si attenga alle presenti condizioni di licenza, disporrà dei diritti perpetui di seguito indicati.

1. DIRITTI DI INSTALLAZIONE E DI UTILIZZO. Il licenziatario potrà installare e utilizzare un numero qualsiasi di copie del software sui propri dispositivi.

2. REQUISITI AGGIUNTIVI PER LE LICENZE E/O DIRITTI SULL'UTILIZZO.

- **a. Codice Distribuibile.** Il software contiene codice che il licenziatario potrà distribuire nei programmi che svilupperà, purché si attenga alle condizioni di seguito riportate.
  - **i. Diritto di Utilizzo e Distribuzione. Il codice e i file elencati di seguito costituiscono il "Codice Distribuibile".**
    - Il licenziatario potrà duplicare e distribuire il formato in codice oggetto dei file del software.
    - *Distribuzione da Parte di Terzi*. Il licenziatario potrà autorizzare i distributori dei propri programmi a duplicare e distribuire il Codice Distribuibile nell'ambito di tali programmi.
  - **ii. Requisiti per la Distribuzione. Per distribuire il Codice Distribuibile, il licenziatario dovrà**
    - aggiungere rilevanti e significative funzionalità nei programmi;
    - far accettare ai distributori e agli utenti finali esterni un contratto con condizioni che garantiscano almeno lo stesso livello di tutela del Codice Distribuibile definito nel presente contratto;
    - visualizzare valide informazioni sul copyright nei propri programmi e
    - indennizzare, manlevare e difendere Microsoft da qualsiasi reclamo, ivi incluse le spese legali, relativo alla distribuzione o all'utilizzo dei propri programmi.
  - **iii. Limitazioni relative alla Distribuzione. Il licenziatario non potrà**
    - modificare le eventuali informazioni su copyright, marchi o brevetti presenti nel Codice Distribuibile;
    - utilizzare i marchi di Microsoft nei nomi dei programmi o in modo tale da far presumere che i programmi provengano o siano approvati da Microsoft;

- distribuire Codice Distribuibile da eseguire su una piattaforma diversa dalla piattaforma Windows;
- includere il Codice Distribuibile in programmi dannosi, ingannevoli o illegali né
- modificare o distribuire il codice sorgente di alcun Codice Distribuibile in modo che nessuna parte di tale codice diventi soggetta a una Licenza Esclusa. Per Licenza Esclusa si intende qualsiasi licenza che come condizione per l'utilizzo, la modifica o la distribuzione richieda che
  - il codice sia divulgato o distribuito nel formato in codice sorgente oppure
  - altri abbiano il diritto di modificarlo.

3. AMBITO DI VALIDITÀ DELLA LICENZA. Il software non viene venduto, ma è concesso in licenza. Il presente contratto concede al licenziatario solo alcuni diritti di utilizzo del software. Microsoft si riserva tutti gli altri diritti. Nel limite massimo consentito dalla legge applicabile, il licenziatario potrà utilizzare il software esclusivamente nei modi espressamente previsti dal presente contratto. Nel far ciò il licenziatario dovrà attenersi a qualsiasi limitazione tecnica presente nel software che gli consenta di utilizzarlo solo in determinati modi. Il licenziatario non potrà

- aggirare le limitazioni tecniche presenti nel software;
- decompilare o disassemblare il software, fatta eccezione per i casi in cui le suddette attività siano espressamente consentite dalla legge applicabile, nonostante questa limitazione;
- pubblicare il software per consentirne la duplicazione da parte di altri;
- noleggiare il software né concederlo in locazione o in prestito;
- trasferire il software o il presente contratto a terzi né
- utilizzare il software per fornire hosting di servizi commerciali.

4. DOCUMENTAZIONE. Qualsiasi persona che disponga di accesso valido al computer o alla rete interna del licenziatario potrà duplicare e utilizzare la documentazione per fini di riferimento interno.

5. LIMITAZIONI RELATIVE ALL'ESPORTAZIONE. Il software è soggetto alle leggi e ai regolamenti in vigore negli Stati Uniti in materia di controllo dell'esportazione. Il licenziatario dovrà attenersi a tutte le leggi e a tutti i regolamenti locali e internazionali applicabili al software in materia di controllo dell'esportazione. Queste leggi includono limitazioni circa le destinazioni, gli utenti finali e l'utilizzo finale. Per ulteriori informazioni, il licenziatario potrà visitare la pagina www.microsoft.com/exporting.

6. SERVIZI DI SUPPORTO TECNICO. Poiché il presente software viene fornito "com'è", non è prevista l'erogazione di servizi di supporto tecnico da parte di Microsoft.

7. INTERO ACCORDO. Il presente contratto e le condizioni che disciplinano l'utilizzo dei supplementi, degli aggiornamenti, dei servizi basati su Internet e dei servizi di supporto tecnico usati dal licenziatario costituiscono l'intero accordo relativo al software e ai servizi di supporto tecnico.

8. LEGGE APPLICABILE.

- **a. Stati Uniti.** Qualora il software sia stato acquistato negli Stati Uniti, il presente contratto è disciplinato e interpretato esclusivamente in base alla legge dello Stato di Washington e tale legge si applica ai reclami aventi ad oggetto gli inadempimenti contrattuali, indipendentemente dai principi in materia di conflitto di leggi. Tutti gli altri reclami, inclusi quelli aventi ad oggetto inadempimenti della normativa a tutela dei consumatori, inadempimenti delle norme in materia di concorrenza sleale e l'illecito civile, saranno disciplinati dalle leggi dello Stato di residenza del licenziatario.
- **b. Al di fuori degli Stati Uniti.** Qualora il licenziatario abbia acquistato il software in qualsiasi altro Paese, il presente contratto è disciplinato dalle leggi di tale Paese.

9. EFFETTI GIURIDICI. Con il presente contratto vengono concessi determinati diritti. Al licenziatario potranno essere concessi altri diritti ai sensi delle leggi del Paese di residenza. Il licenziatario potrebbe, inoltre, vantare ulteriori diritti direttamente nei confronti della parte da cui ha acquistato il software. Il presente contratto non modifica i diritti del licenziatario che le leggi del Paese di residenza del licenziatario non consentono di modificare.

10. ESCLUSIONE DI GARANZIE. Il software viene concesso in licenza "com'è". Il licenziatario lo utilizza a proprio rischio. Non vengono fornite garanzie o condizioni espresse. Il presente contratto non modifica eventuali ulteriori diritti dei consumatori o garanzie di legge riconosciute al licenziatario dalle leggi locali. Nella misura massima consentita dalle leggi locali Microsoft esclude eventuali garanzie implicite di commerciabilità (qualità non inferiore alla media), adeguatezza per uno scopo specifico e non violazione di diritti di terzi.

PER L'AUSTRALIA: il licenziatario è soggetto alle garanzie di legge previste ai sensi della Legge Australiana a Tutela dei Consumatori (Australian Consumer Law) e nessuna disposizione contenuta nelle presenti condizioni influisce su tali diritti.

11. LIMITAZIONE DI RESPONSABILITÀ ED ESCLUSIONE DI RIMEDI E DANNI. Il licenziatario può richiedere a Microsoft e ai suoi fornitori il solo risarcimento per i danni diretti nel limite di cinque dollari (USD 5). Il licenziatario non ha diritto a ottenere il risarcimento per eventuali altri danni, inclusi i danni consequenziali, speciali, indiretti, incidentali o relativi alla perdita di profitti.

Questa limitazione si applica a

- qualsiasi questione relativa al software, ai servizi, al contenuto (incluso il codice) sui siti Internet o nei programmi di terzi e
- reclami relativi a inadempimento contrattuale, inadempimenti della garanzia o delle condizioni, responsabilità oggettiva, negligenza o altro illecito civile, nella misura massima consentita dalla legge applicabile.

Tale limitazione si applica anche nel caso in cui Microsoft sia stata informata o avrebbe dovuto essere informata della possibilità del verificarsi di tali danni. La limitazione o l**'**esclusione di cui sopra potrebbe non essere applicabile al licenziatario in quanto l'esclusione o la limitazione di danni incidentali, consequenziali o di altro tipo potrebbe non essere consentita nel Paese di residenza del licenziatario.

Entity Framework 6 Runtime License (JPN)

9/13/2018 • 2 minutes to read • Edit Online

マイクロソフト ソフトウェア ライセンス条項

MICROSOFT ENTITY FRAMEWORK

マイクロソフト ソフトウェア ライセンス条項 (以下、「本ライセンス条項」といいます) は、お客様と Microsoft Corporation (またはお客様の所在地に応じた関連会社。以下、「マイクロソフト」といいます) との契約を構成します。以下のライセンス条項を注意してお読みください。本ライセンス条項は、上記のソフトウェアおよびソフトウェアが記録されたメディア (以下総称して「本ソフトウェア」といいます) に適用されます。また、本ライセンス条項は本ソフトウェアに関連する下記マイクロソフト製品にも適用されるものとします。

- 更新プログラム
- 追加ソフトウェア
- インターネット ベースのサービス
- サポート サービス

これらの製品に別途ライセンス条項が付属している場合には、当該ライセンス条項が適用されるものとします。

本ソフトウェアを使用することにより、お客様は本ライセンス条項に同意されたものとします。本ライセンス条項に同意されない場合、本ソフトウェアを使用することはできません。

お客様が本ライセンス条項を遵守することを条件として、お客様には以下の永続的な権利が許諾されます。

1.　インストールおよび使用に関する権利。お客様は、本ソフトウェアの任意の部数の複製をお客様のデバイスにインストールして使用することができます。

2.　追加のライセンス条件および追加の使用権。

- **a.　再頒布可能コード。**本ソフトウェアには、お客様が開発されたプログラムに含めて再頒布可能なコードが含まれています。ただし、以下の条件に従うものとします。
  - **i.　使用および再頒布の権利。以下に記載するコードおよびファイルを「再頒布可能コード」と定義します。**
    - お客様は、ソフトウェア ファイルをオブジェクト コード形式で複製し、再頒布することができます。
    - *第三者による再頒布*。お客様は、お客様のプログラムの頒布者に対して、お客様のプログラムの一部として再頒布可能コードの複製および頒布を許可することができます。
  - **ii.　再頒布の条件。お客様は、お客様が頒布するすべての再頒布可能コードにつき、以下の条項に従わなければなりません。**
    - お客様のプログラムにおいて再頒布可能コードに重要かつ主要な機能を追加すること。
    - お客様のプログラムの頒布者および外部エンド ユーザーに、本ライセンス条項と同等以上に再頒布可能コードを保護する条項に同意させること。
    - お客様のプログラムにお客様名義の有効な著作権表示を行うこと。
    - お客様のプログラムの頒布または使用に関する請求 (弁護士報酬を含みます) について、マイクロソフトを免責、防御、および補償すること。
  - **iii.　再頒布の制限。お客様は、以下を行うことはできません。**
    - 再頒布可能コードの著作権、商標または特許の表示を改変すること。
    - お客様のプログラムの名称の一部にマイクロソフトの商標を使用したり、お客様の製品がマイクロソフトから由来したり、マイクロソフトが推奨するように見せかけること。
    - Windows プラットフォーム以外のプラットフォームで実行するプログラムにおいて再配布可能コードを配布すること

- 悪意のある、欺瞞的、または違法なプログラムに再頒布可能コードを含めること。
- 再頒布可能コードの一部に除外ライセンスが適用されることとなるような方法で再頒布可能コードのソース コードを改変または頒布すること。「除外ライセンス」とは、使用、改変または頒布の条件として以下を満たすことを要求するライセンスです。
  - コードをソース コード形式で公表または頒布すること。または
  - その他の者がコード改変の権利を有すること。

3.　ライセンスの適用範囲。本ソフトウェアは使用許諾されるものであり、販売されるものではありません。本ライセンス条項は、お客様に本ソフトウェアを使用する限定的な権利を付与します。マイクロソフトはその他の権利をすべて留保します。適用される法令により上記の制限を超える権利が与えられる場合を除き、お客様は本ライセンス条項で明示的に許可された方法でのみ本ソフトウェアを使用することができます。お客様は、使用方法を制限するために本ソフトウェアに組み込まれている技術的制限に従わなければなりません。お客様は、以下を行うことはできません。

- 本ソフトウェアの技術的な制限を回避して使用すること。
- 本ソフトウェアをリバース エンジニアリング、逆コンパイル、または逆アセンブルすること。ただし、適用される法令により明示的に認められている場合を除きます。
- 第三者が複製できるように本ソフトウェアを公開すること。
- 本ソフトウェアをレンタル、リース、または貸与すること。
- 本ソフトウェアまたは本ライセンス条項を第三者に譲渡すること。
- 本ソフトウェアを商用ソフトウェア ホスティング サービスで使用すること。

4.　ドキュメンテーション。お客様のコンピューターまたは内部ネットワークに有効なアクセス権を有する者は、お客様の内部使用目的に限り、ドキュメントを複製して使用することができます。

5.　輸出規制。本ソフトウェアは米国および日本国の輸出に関する規制の対象となります。お客様は、本ソフトウェアに適用されるすべての国内法および国際法 (輸出対象国、エンド ユーザーおよびエンド ユーザーによる使用に関する制限を含みます) を遵守しなければなりません。詳細については www.microsoft.com/japan/exporting をご参照ください。

6.　サポート サービス。本ソフトウェアは現状有姿で提供されます。そのため、マイクロソフトはサポート サービスを提供しない場合があります。

7.　完全合意。本ライセンス条項およびお客様が使用する追加ソフトウェア、更新プログラム、インターネット ベースのサービス、ならびにサポート サービスに関する条項は、本ソフトウェアおよびサポート サービスについてのお客様とマイクロソフトとの間の完全なる合意です。

8.　準拠法。

- **a.　日本。**お客様が本ソフトウェアを日本国内で入手された場合、本ライセンス条項は日本法に準拠するものとします。
- **b.　米国。**お客様が本ソフトウェアを米国内で入手された場合、抵触法に関わらず、本ライセンス条項の解釈および契約違反への主張は、米国ワシントン州法に準拠するものとします。消費者保護法、公正取引法、および違法行為を含みますがこれに限定されない他の主張については、お客様が所在する地域の法律に準拠します。
- **c.　日本および米国以外。**お客様が本ソフトウェアを日本国および米国以外の国で入手された場合、本ライセンス条項は適用される地域法に準拠するものとします。

9.　法的効力。本ライセンス条項は、特定の法的な権利を規定します。お客様は、地域や国によっては、本ライセンス条項の定めにかかわらず、本ライセンス条項と異なる権利を有する場合があります。また、お客様は本ソフトウェアの取得取引の相手方に対して権利を取得できる場合もあります。本ライセンス条項は、お客様の地域または国の法律により権利の拡大が認められない限り、それらの権利を変更しないものとします。

10. あらゆる保証の免責。本ソフトウェアは、現状有姿のまま瑕疵を問わない条件で提供されます。本ソフトウェアの使用に伴う危険は、お客様の負担とします。マイクロソフトは、明示的な瑕疵担保責任または保証責任を一切負いません。本ライセンス条項では変更できないお客様の地域の法律による追加の消費者の権利または法定保証が存在する場合があります。お客様の地域の国内法等によって認められる限り、マイクロソフトは、商品性、特定目的に対する適合性、および侵害の不存在に関する瑕疵担保責任または黙示の保証責任を負いません。

オーストラリア限定。お客様は、オーストラリア消費者法に基づく法定保証を有し、これらの条項は、それらの権利に影響を与えることを意図するものではありません。

11. 救済手段および責任の制限および除外。マイクロソフトおよびそのサプライヤーの責任は、5.00 米ドルを上限とする直接損害に限定されます。その他の損害 (派生的損害、逸失利益、特別損害、間接損害、および付随的損害を含みますがこれらに限定されません) に関しては、一切責任を負いません。

この制限は、以下に適用されるものとします。

- 本ソフトウェア、サービス、第三者のインターネットのサイト上のコンテンツ (コードを含みます)、または第三者のプログラムに関連した事項
- 契約違反、保証違反、厳格責任、過失、または不法行為等の請求 (適用される法令により認められている範囲において)

この制限は、マイクロソフトが損害の可能性を認識していたか、または認識し得た場合にも適用されます。また、一部の国では付随的損害および派生的損害の免責、または責任の制限が認められないため、上記の制限事項が適用されない場合があります。

Entity Framework 6 Runtime License (KOR)

9/13/2018 • 4 minutes to read • Edit Online

MICROSOFT 소프트웨어 사용권 계약서

MICROSOFT ENTITY FRAMEWORK

본 사용권 계약은 Microsoft Corporation(또는 거주 지역에 따라 계열사 중 하나)과 귀하 간에 체결되는 계약입니다. 본 사용권 계약을 읽어 주시기 바랍니다. 본 사용권 계약은 위에 명시된 소프트웨어 및 이 소프트웨어가 포함된 미디어가 있는 경우 해당 미디어에 적용됩니다. 본 계약은 해당 품목에 별도의 약정이 있지 않는 한, 이 소프트웨어에 대한

- 업데이트,
- 추가 구성 요소,
- 인터넷 기반 서비스 및
- 지원 서비스

와 같은Microsoft 구성 요소에도 적용됩니다. 별도의 약정이 있는 경우, 해당 약정이 적용됩니다.

이 소프트웨어를 사용함으로써 귀하는 아래의 조항들에 동의하게 됩니다. 동의하지 않을 경우에는 소프트웨어를 사용하지 마십시오.

본 사용권 계약을 준수하는 경우 아래와 같은 영구적 권리를 행사할 수 있습니다.

1. **설치 및 사용 권한.** 본 소프트웨어의 사본을 수량에 관계없이 장치에 설치하여 사용할 수 있습니다.
2. **추가 사용권 요구 사항 및/또는 사용 권한.**

- **a. 배포 가능 코드.** 소프트웨어에는 귀사가 아래 조항을 준수하는 경우 귀사가 개발하는 프로그램에서 배포 권한이 부여되는 코드가 포함되어 있습니다.
  - **i. 사용 및 배포 권한. 아래에 나열된 코드 및 파일은 "배포 가능 코드"입니다.**
    - 귀하는 소프트웨어 파일을 개체 코드 형태로 복사 및 배포할 수 있습니다.
    - *제3자에 의한 배포.* 프로그램 배포자가 배포 가능 코드를 프로그램의 일부로 복사 및 배포하도록 허용할 수 있습니다.
  - **ii. 배포 조건. 배포 가능 코드를 배포하려면**
    - 귀사의 프로그램 내에서 배포 가능 코드에 중요한 기능을 추가해야 합니다.
    - 최소한 본 계약에 준하는 배포 가능 코드를 보호할 수 있는 조항에 배포자와 외부의 최종 사용자가 동의하도록 해야 합니다.
    - 귀사의 프로그램에 유효한 저작권 표시를 해야 합니다.
    - 귀사의 프로그램 배포 또는 사용과 관련된 모든 청구(변호사 비용 포함)로부터 Microsoft를 면책하고 해를 입히지 않으며 방어해야 합니다.
  - **iii. 배포 제한. 다음과 같은 행위는 허용되지 않습니다.**
    - 배포 가능 코드의 저작권, 상표 또는 특허 표시를 변경하는 행위
    - Microsoft의 상표를 프로그램 이름에 사용하거나 귀사의 프로그램을Microsoft에서 만들거나 보증한다고 광고하는 행위
    - 배포 가능 코드를 배포하여 Windows가 아닌 플랫폼에서 실행하는 행위
    - 배포 가능 코드를 악성, 기만적 또는 불법적인 프로그램에 포함하는 행위
    - 배포 가능 코드의 일부분이 예외적 라이선스에 적용되도록 배포 가능 코드의 소스 코드를 수정하거나 배포하는 행위. 예외적 라이선스란 사용, 수정 또는 배포를 위해 다음과 같은 조건을 필요로 하는 라이선스입니다.

  - 코드가 소스 코드 형태로 공개되거나 배포됩니다.
  - 다른 사람에게 배포 가능 코드를 수정할 수 있는 권리가 있습니다.

3. 사용권의 범위. 본 소프트웨어는 판매되는 것이 아니라 그 사용이 허여되는 것입니다. 본 계약은 귀하에게 소프트웨어를 사용할 수 있는 권한을 허여합니다. 기타 모든 권한은Microsoft가 보유합니다. 이러한 제한과 관계없이 관련 법률에서 귀하에게 더 많은 권한을 부여하지 않는 한, 귀하는 본 계약에서 명시적으로 허용되는 조건에 한해서만 소프트웨어를 사용할 수 있습니다. 그렇게 하는 경우 귀하는 특정 방식으로만 사용할 수 있도록 하는 소프트웨어의 모든 기술적 제한 사항을 준수해야 합니다. 다음과 같은 행위는 허용되지 않습니다.

- 소프트웨어의 기술적 제한 사항을 위반하는 행위
- 이러한 제한에도 불구하고 관련 법률에서 명시적으로 허용하는 경우를 제외한 소프트웨어의 리버스 엔지니어링, 디컴파일 또는 디스어셈블 작업을 수행하는 행위
- 다른 사람이 복사할 수 있도록 소프트웨어를 게시하는 행위
- 소프트웨어를 임대, 대여 또는 대부하는 행위
- 소프트웨어나 본 계약서를 제3자에게 양도하는 행위
- 상업용 소프트웨어 호스팅 서비스에 소프트웨어를 사용하는 행위

4. 설명서. 귀하의 컴퓨터 또는 내부 네트워크에 유효한 액세스 권한이 있는 사용자는 내부적인 참고 목적으로 설명서를 복사 및 사용할 수 있습니다.

5. 수출 제한. 소프트웨어는 미국 수출 법률 및 규정의 적용을 받습니다. 귀하는 소프트웨어에 적용되는 모든 국내 및 국제 수출 법률 및 규정을 준수해야 합니다. 이러한 법률에는 목적지, 최종 사용자 및 최종 용도에 대한 제한이 포함됩니다. 자세한 내용은 www.microsoft.com/exporting을 참조하십시오.

6. 지원 서비스. 이 소프트웨어는 "있는 그대로" 제공되므로 이 소프트웨어에 대한 지원 서비스가 제공되지 않을 수 있습니다.

7. 전면 합의. 본 계약 및 귀하가 이용하는 추가 구성 요소, 업데이트, 인터넷 기반 서비스 및 지원 서비스에 대한 조항은 소프트웨어 및 지원 서비스에 대한 전면 합의입니다.

8. 관련 법률.

- **a. 미국.** 소프트웨어를 미국에서 구입한 경우, 국제사법 원칙에 관계없이 본 계약의 해석은 워싱턴 주법을 따르며 계약 위반에 대한 청구 발생 시에도 워싱턴 주법이 적용됩니다. 소비자 보호법, 불공정거래법 및 기타 불법 행위 관련 법규의 적용을 받는 청구가 발생한 경우 귀하가 거주하고 있는 주의 주법이 적용됩니다.
- **b. 미국 외 지역.** 본 사용권 계약에는 대한민국 법이 적용됩니다.

9. 법적 효력. 본 계약은 특정 법적 권리에 대해 기술하고 있습니다. 귀하는 귀하가 거주하고 있는 국가의 법규가 보장하는 다른 권리를 보유할 수 있습니다. 또한 귀하가 소프트웨어를 구입한 당사자와 관련된 권리를 보유할 수도 있습니다. 귀하가 거주하고 있는 국가의 법에서 권리 변경을 허용하지 않는 경우 본 계약은 해당 권리를 변경하지 않습니다.

10. 보증의 부인. 이 소프트웨어는 "있는 그대로" 사용권이 허여됩니다. 소프트웨어의 사용으로 발생하는 위험은 귀하의 책임입니다. Microsoft는 어떠한 명시적 보증, 보장 또는 조건도 제시하지 않습니다. 귀하는 귀하가 거주하는 지역의 법규에 따른 추가적인 소비자 권리 또는 법적 권리를 보유할 수 있으며, 이 권리는 본 계약을 통해 변경되지 않습니다. 귀하가 거주하는 지역의 법규가 허용하는 범위 내에서Microsoft는 상업성, 특정 목적에의 적합성 및 비침해성과 관련된 묵시적 보증을 배제합니다.

오스트레일리아의 경우 – 귀하는 오스트레일리아 소비자 보호법에 따라 법적 권리를 보유하며 본 계약서의 어떠한 내용도 해당 권리에 영향을 미칠 수 없습니다.

11. 손해 및 구제수단의 제한 및 배제. 귀하는 직접적인 손해에 한해Microsoft와 그 공급자로부터 최대 미화 $5.00까지 보상받을 수 있습니다. 결과적 손해, 이익 손실, 특별, 간접 또는 부수적 손해를 포함한 기타 모든 손해에 대해서는 보상을 받을 수 없습니다.

이 제한 사항은 다음에 적용됩니다.

- 제3자 인터넷 사이트상의 소프트웨어, 서비스, 콘텐츠(코드 포함) 또는 제3자 프로그램과 관련하여 발생하는 모든 문제
- 계약 위반, 보증, 보장 또는 조건의 불이행, 무과실 책임, 과실 또는 관련 법률에서 허용하는 범위 내의 기타 불법 행위 등으로 인한 청구

Microsoft가 그러한 손해의 가능성에 대해 사전에 알고 있었거나 알아야만 했던 경우에도 적용됩니다. 귀하가 거주하고 있는 국가에서 부수적, 결과적 또는 기타 손해의 배제나 제한을 허용하지 않는 경우에는 위의 제한이나 배제가 적용되지 않을 수 있습니다.

Entity Framework 6 Runtime License (RUS)

9/13/2018 • 5 minutes to read • Edit Online

УСЛОВИЯ ЛИЦЕНЗИИ НА ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ MICROSOFT

MICROSOFT ENTITY FRAMEWORK

Настоящие условия лицензии являются соглашением между корпорацией Microsoft (или, в зависимости от места вашего проживания, одним из ее аффилированных лиц) и вами. Прочтите их внимательно. Они применяются к вышеуказанному программному обеспечению, включая носители, на которых оно распространяется (если они есть). Эти условия распространяются также на все

- обновления,
- дополнительные компоненты,
- службы Интернета и
- службы технической поддержки

Microsoft для данного программного обеспечения, если эти элементы не сопровождаются другими условиями. В последнем случае применяются соответствующие условия.

Используя это программное обеспечение, вы выражаете согласие соблюдать данные условия. Если вы не согласны с условиями лицензии, не используйте это программное обеспечение.

При соблюдении вами условий данной лицензии вам предоставляются следующие бессрочные права.

1. УСТАНОВКА И ПРАВА ИСПОЛЬЗОВАНИЯ. Вы можете установить и использовать любое количество копий программного обеспечения на своих устройствах.

2. ДОПОЛНИТЕЛЬНЫЕ ТРЕБОВАНИЯ ЛИЦЕНЗИРОВАНИЯ И ПРАВА НА ИСПОЛЬЗОВАНИЕ.

- **a. Вторично распространяемый код.** Программное обеспечение содержит код, который разрешается распространять в составе разрабатываемых вами программ при соблюдении вами следующих условий.
  - **i. Право на использование и распространение. Программный код и файлы, перечисленные ниже, представляют собой «Вторично распространяемый код».**
    - Вы имеете право копировать и распространять в виде объектного кода файлы программного обеспечения.
    - *Распространение третьими лицами*. Вы можете разрешить дистрибьюторам ваших программ копировать и распространять Вторично распространяемый код как часть этих программ.
  - **ii. Условия распространения. Для распространения любого Вторично распространяемого кода вы должны:**
    - существенно расширить основные функциональные возможности кода в своих программах;
    - потребовать от дистрибьюторов и внешних конечных пользователей соблюдения условий, которые будут защищать Вторично распространяемый код не меньше, чем данное соглашение;
    - отображать действительное уведомление об авторских правах в ваших программах; и
    - освободить от ответственности, защитить и оградить Microsoft от любых претензий и исков, связанных с использованием и распространением ваших программ, включая расходы на оплату услуг адвокатов.
  - **iii. Ограничения распространения. Вы не имеете права:**

  - изменять уведомления об авторских правах, патентных правах и правах на товарные знаки, присутствующие во Вторично распространяемом коде;
  - использовать товарные знаки Microsoft в названиях своих программ или таким способом, который заставил бы пользователя предположить, что программа является продуктом Microsoft или одобрена ею;
  - распространять Вторично распространяемый код для использования его на платформе, отличной от Windows;
  - включать Вторично распространяемый код во вредоносные, незаконные или вводящие в заблуждение программы;
  - изменять или распространять исходный код любого Вторично распространяемого кода таким образом, чтобы любая его часть подпадала под действие исключенной лицензии. Исключенная лицензия — это любая лицензия, согласно которой использование, изменение или распространение возможны только при соблюдении следующих условий:
    - код раскрывается и распространяется в виде исходного кода;
    - другие лица имеют право его изменять.

3. ОБЪЕМ ЛИЦЕНЗИИ. Программное обеспечение не продается, а лицензируется. Это соглашение дает вам только некоторые права на использование программного обеспечения. Microsoft оставляет за собой все остальные права. За исключением случаев, когда, несмотря на данное ограничение, применимое право предоставляет вам больше прав, вы можете использовать программное обеспечение только теми способами, которые явно указаны в условиях настоящего соглашения. При этом вы должны соблюдать все технические ограничения в программном обеспечении, допускающие использование программного обеспечения только определенным образом. Вы не имеете права:

- пытаться обойти технические ограничения в программном обеспечении;
- изучать технологию, декомпилировать или деассемблировать программное обеспечение, если это прямо не разрешено применимым правом, несмотря на данное ограничение;
- публиковать программное обеспечение, предоставляя другим лицам возможность его копировать;
- предоставлять программное обеспечение в прокат, в аренду или во временное пользование;
- передавать программное обеспечение или это соглашение третьим лицам;
- использовать это программное обеспечение для предоставления услуг удаленного доступа на коммерческой основе.

4. ДОКУМЕНТАЦИЯ. Любое лицо, имеющее право на доступ к вашему компьютеру или внутренней сети, может копировать и использовать документацию для внутренних целей справочного характера.

5. ОГРАНИЧЕНИЯ НА ЭКСПОРТ. Данное программное обеспечение подпадает под действие экспортного законодательства Соединенных Штатов. Вы обязаны соблюдать все внутренние и международные нормы экспортного законодательства, применимые к программному обеспечению. К таким положениям экспортного законодательства относятся ограничения по конечным пользователям, порядку и регионам конечного использования. Дополнительные сведения см. на веб-сайте www.microsoft.com/exporting.

6. ТЕХНИЧЕСКАЯ ПОДДЕРЖКА. Так как это программное обеспечение предоставляется «как есть», Microsoft может не предоставлять услуг по его технической поддержке.

7. ПОЛНОТА СОГЛАШЕНИЯ. Это соглашение, а также условия, которые сопровождают используемые вами дополнения, обновления, службы Интернета и службы технической поддержки, составляют полное соглашение по программному обеспечению и службам поддержки.

8. ПРИМЕНИМОЕ ПРАВО.

- a. **На территории Соединенных Штатов.** Если вы приобрели программное обеспечение в Соединенных Штатах, это соглашение подлежит толкованию в соответствии с законодательством штата

Вашингтон, США. Любые претензии в связи с нарушением данного соглашения также рассматриваются в соответствии с этим законодательством независимо от принципов коллизионного права. Все остальные претензии, включая претензии на основании законов штата о защите потребителей и законов о недобросовестной конкуренции, а также в связи с гражданским правонарушением, регулируются законами штата, в котором вы проживаете.

- **b. За пределами Соединенных Штатов.** Если вы приобрели программное обеспечение в любой другой стране, применяются законы этой страны.

9. ЮРИДИЧЕСКАЯ СИЛА. Это соглашение описывает определенные юридические права. По законам своего штата или своей страны вы можете иметь дополнительные права. Вы также можете иметь права в отношении стороны, у которой вы приобрели программное обеспечение. Это соглашение не изменяет ваших прав, предусмотренных законами вашей страны, если это не допускается законами вашей страны.

10. ОТКАЗ ОТ ПРЕДОСТАВЛЕНИЯ ГАРАНТИЙ. Программное обеспечение лицензируется «как есть». Вы пользуетесь им на собственный риск. Microsoft не предоставляет никаких явных гарантий и не гарантирует соблюдение каких-либо условий. Вы можете иметь дополнительные права потребителя или гарантии, предусмотренные местным законодательством, которое это соглашение не может изменить. В степени, допускаемой местным законодательством, Microsoft исключает подразумеваемые гарантии товарной пригодности, пригодности для определенной цели и отсутствия нарушения прав иных правообладателей.

ДЛЯ АВСТРАЛИИ — вы имеете гарантии, предусмотренные Законом Австралии о правах потребителей, и ничто в настоящих условиях не подразумевает ущемление этих прав.

11. ОГРАНИЧЕНИЕ И ОТСУТСТВИЕ СРЕДСТВ ЗАЩИТЫ ПРАВ ПОТРЕБИТЕЛЕЙ В СВЯЗИ С УБЫТКАМИ И УЩЕРБОМ. Вы можете взыскать с Microsoft и поставщиков Microsoft только прямые убытки в размере не более 5,00 долларов США. Вы не можете взыскать никакие другие убытки, включая косвенные, специальные, опосредованные или случайные убытки, а также убытки в связи с упущенной выгодой.

Это ограничение распространяется:

- на все, что связано с программным обеспечением, службами и содержимым веб-сайтов третьих лиц (включая код), а также с программами сторонних разработчиков;
- на претензии, связанные с нарушением условий соглашения, гарантии или других условий, строгой ответственностью, неосторожностью или другим гражданским правонарушением, в максимально допустимой степени в соответствии с применимым правом.

Это ограничение действует даже в том случае, если в Microsoft было или должно было быть известно о возможности таких убытков. Вышеуказанные ограничения и исключения могут к вам не относиться, если законодательство вашей страны не допускает исключения или ограничения ответственности за случайные, косвенные или другие убытки.