Entity Framework Documentation

Release 1.0.0

Microsoft

Oct 14, 2016

Contents

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Entity Framework Core

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

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

If you like to learn by writing code, we'd recommend one of our *Getting Started* guides to get you started with EF Core.

1.1 Get Entity Framework Core

Install the NuGet package for the database provider you want to use. See *Database Providers* for information.

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

1.2 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_
        }
    }

    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; }
    }
}
```

1.3 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();
}
```

1.4 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();
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 2

Getting Started

The following articles provide documentation for using EF on different platforms.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

2.1 Getting Started on Full .NET (Console, WinForms, WPF, etc.)

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

Entity Framework can create a model based on an existing database, or create a database for you based on your model. The following tutorials will demonstrate both of these approaches using a Console Application. You can use the techniques learned in these tutorials in any application that targets Full .NET, including WPF and WinForms.

2.1.1 Available Tutorials

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Console Application to New Database

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

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015 Update 3
- Latest version of NuGet Package Manager
- Latest version of Windows PowerShell

Create a new project

- Open Visual Studio 2015
- *File → New → Project...*
- From the left menu select *Templates → Visual C# → Windows*
- Select the **Console Application** project template
- Ensure you are targeting **.NET Framework 4.5.1** or later
- Give the project a name and click **OK**

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This walkthrough 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 walkthrough we will also be using some Entity Framework commands to maintain the database. So we will install the commands package as well.

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

Create your model

Now it's time to define a context and entity classes that make up your 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 EFGetStarted.ConsoleApp
{
    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.Console
        }
    }

    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 the `App.Config` file and read it out using `ConfigurationManager`. For the sake of simplicity, we are putting everything in a single code file for this tutorial.

Create your database

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

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

Tip: If you make future changes to your 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 your model

You can now use your model to perform data access.

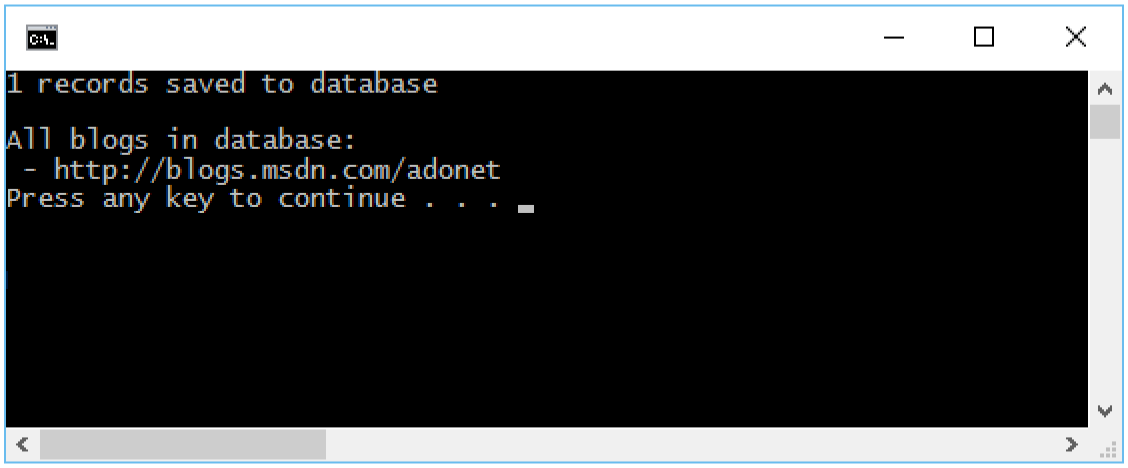
- Open *Program.cs*

- Replace the contents of the file with the following code

```
using System;

namespace EFGetStarted.ConsoleApp
{
    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 will see that one blog is saved to the database and then the details of all blogs are printed to the console.



```
1 records saved to database

All blogs in database:
 - http://blogs.msdn.com/adonet
Press any key to continue . . .
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Console Application to Existing Database (Database First)

In this walkthrough, you will build a console application that performs basic data access against a Microsoft SQL Server database using Entity Framework. You will use reverse engineering to create an Entity Framework model based on an existing database.

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015 Update 3
- Latest version of NuGet Package Manager
- Latest version of Windows PowerShell
- *Blogging database*

Blogging database This tutorial uses a **Blogging** database on your LocalDb instance as the existing database.

Note: If you have already created the **Blogging** database as part of another tutorial, you can 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 CA
);
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 2015
- *File → New → Project...*
- From the left menu select *Templates → Visual C# → Windows*
- Select the **Console Application** project template
- Ensure you are targeting **.NET Framework 4.5.1** or later
- Give the project a name and click **OK**

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This walkthrough 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`

To enable reverse engineering from an existing database we need to install a couple of other packages too.

- Run `Install-Package Microsoft.EntityFrameworkCore.Tools -Pre`
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design`

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;" Microso
```

The reverse engineer process created entity classes and a derived context 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.

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

namespace EFGetStarted.ConsoleApp.ExistingDb
{
    public partial class Blog
    {
        public Blog()
        {
            Post = new HashSet<Post>();
        }

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

        public virtual ICollection<Post> Post { get; set; }
    }
}
```

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

```csharp
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

namespace EFGetStarted.ConsoleApp.ExistingDb
{
    public partial class BloggingContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            #warning To protect potentially sensitive information in your connection string, you shou
            optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Con
        }

        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);
            });
        }

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

Use your model

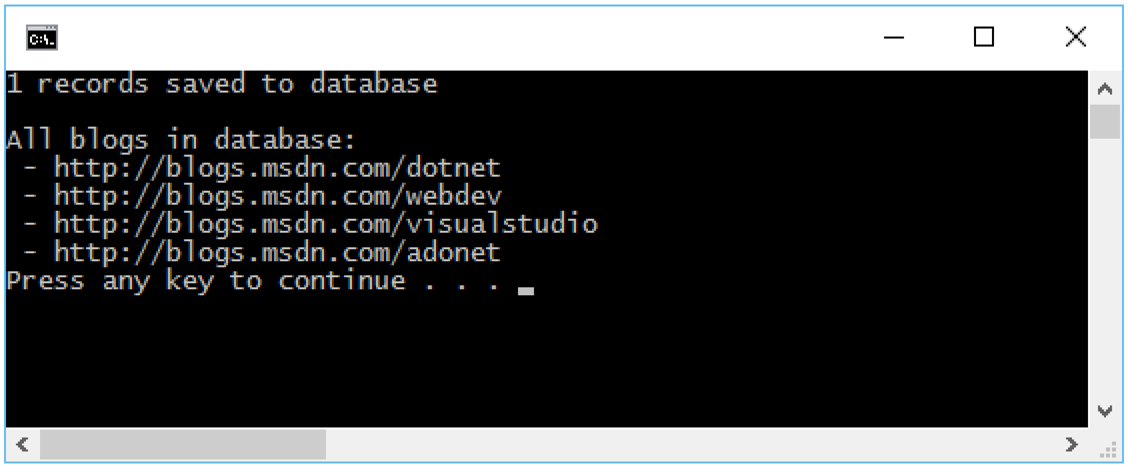
You can now use your model to perform data access.

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

```
using System;

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



```
1 records saved to database

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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

2.2 Getting Started on .NET Core (Windows, OSX, Linux, etc.)

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

EF can be used on all platforms (Windows, OSX, Linux, etc.) that support .NET Core.

2.2.1 Available Tutorials

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

.NET Core Application to New SQLite Database

In this walkthrough, you will build a .NET Core console application that performs basic data access using Entity Framework. You will use migrations to create the database from your model.

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

Minimum system requirements

- An operating system that supports .NET Core
- .NET Core SDK Preview 2
- A text editor

Caution: Known Issues

- Migrations on SQLite do not support more complex schema changes due to limitations in SQLite itself. See *SQLite Limitations*

Install the .NET Core SDK The .NET Core SDK provides the command-line tool `dotnet` which will be used to build and run our sample application.

See the .NET Core website for instructions on installing the SDK on your operating system.

Create a new project

- Create a new folder `ConsoleApp/` for your project. All files for the project should be in this folder.

```
mkdir ConsoleApp
cd ConsoleApp/
```

- Execute the following .NET Core CLI commands to create a new console application, download dependencies, and run the .NET Core app.

```
dotnet new
dotnet restore
dotnet run
```

Install Entity Framework

- To add EF to your project, modify `project.json` so it matches the following sample.

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "debugType": "portable",
    "emitEntryPoint": true
  },
  "dependencies": {
    "Microsoft.EntityFrameworkCore.Sqlite": "1.0.0",
    "Microsoft.EntityFrameworkCore.Design": {
      "version": "1.0.0-preview2-final",
      "type": "build"
    }
  },
  "frameworks": {
    "netcoreapp1.0": {
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.0.0"
        }
      },
      "imports": "dnxcore50"
    }
  },
  "tools": {
    "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
  }
}
```

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

```
dotnet restore
```

- Verify that Entity Framework is installed by running `dotnet ef --help`.

```
dotnet ef --help
```

Create your model

With this new project, you are ready to begin using Entity Framework. The next steps will add code to configure and access a SQLite database file.

- **Create a new file called `Model.cs`** All classes in the following steps will be added to this file.

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

namespace ConsoleApp.SQLite
{
```

- **Add a new class to represent the SQLite database.** We will call this `BloggingContext`. The call to `UseSqlite()` configures EF to point to a *.db file.

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

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

- **Add classes to represent tables.** Note that we will be using foreign keys to associate many posts to one blog.

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

- To make sure the files are correct, you can compile the project on the command line by running `dotnet build`

```
dotnet build
```

Create your database

We can now use Entity Framework command line tools to create and manage the schema of the database.

- **Create the first migration.** Execute the command below to generate your first migration. This will find our context and models, and generate a migration for us in a folder named `Migrations/`

```
dotnet ef migrations add MyFirstMigration
```

- **Apply the migrations.** You can now begin using the existing migration to create the database file and creates the tables.

```
dotnet ef database update
```

  This should create a new file `blog.db` in the output path. This SQLite file should now contain two empty tables.

Note: When using relative paths with SQLite, the path will be relative to the application's main assembly. In this sample, the main binary is `bin/Debug/netcoreapp1.0/ConsoleApp.dll`, so the SQLite database will be in `bin/Debug/netcoreapp1.0/blog.db`

Use your model

Now that we have configured our model and created the database schema, we can use BloggingContext to create, update, and delete objects.

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

Start your app

Run the application from the command line.

```
dotnet run
```

After adding the new post, you can verify the data has been added by inspecting the SQLite database file, `bin/Debug/netcoreapp1.0/blog.db`.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

2.3 Getting Started on ASP.NET Core

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

Entity Framework can create a model based on an existing database, or create a database for you based on your model. The following tutorials will demonstrate both of these approaches using an ASP.NET Core application.

2.3.1 Available Tutorials

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

ASP.NET Core Application to New Database

In this walkthrough, you will build an ASP.NET Core MVC application that performs basic data access using Entity Framework. You will use migrations to create the database from your model.

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015 Update 3
- .NET Core for Visual Studio

Create a new project

- Open Visual Studio 2015

- *File → New → Project...*
- From the left menu select *Templates → Visual C# → Web*
- Select the **ASP.NET Core Web Application (.NET Core)** project template
- Enter **EFGetStarted.AspNetCore.NewDb** as the name and click **OK**
- Wait for the **New ASP.NET Core Web Application** dialog to appear
- Select the **Web Application** template and ensure that **Authentication** is set to **No Authentication**
- Click **OK**

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

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This walkthrough 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`

Note: In ASP.NET Core projects the `Install-Package` command will complete quickly and the package installation will occur in the background. You will see **(Restoring...)** appear next to **References** in **Solution Explorer** while the install occurs.

Later in this walkthrough we will also be using some Entity Framework commands to maintain the database. So we will install the commands package as well.

- Run `Install-Package Microsoft.EntityFrameworkCore.Tools -Pre`
- Open **project.json**
- Locate the `tools` section and add the `ef` command as shown below

```
"tools": {
  "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
},
```

Create your model

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

- Right-click on the project in **Solution Explorer** and select *Add → New Folder*
- Enter **Models** as the name of the folder
- Right-click on the **Models** folder and select *Add → New Item...*
- From the left menu select *Installed → Code*
- Select the **Class** item template

- 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 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; }
    }
}
```

Note: In a real application you would typically put each class from your model in a separate file. For the sake of simplicity, we are putting all the classes in one file for this tutorial.

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.

In order for our MVC controllers to make use of `BloggingContext` we are going to register it as a service.

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

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

Now we can use the `AddDbContext` method to register it as a service.

- Locate the `ConfigureServices` method
- Add the lines that are highlighted in the following code

```
        public void ConfigureServices(IServiceCollection services)
        {
            var connection = @"Server=(localdb)\mssqllocaldb;Database=EFGetStarted.AspNetCore.NewDb;T
            services.AddDbContext<BloggingContext>(options => options.UseSqlServer(connection));
```

Create your database

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

- *Tools –> NuGet Package Manager –> Package Manager Console*
- Run `Add-Migration MyFirstMigration` to scaffold a migration to create the initial set of tables for your model. If you receive an error stating the term 'add-migration' is not recognized as the name of a cmdlet, then close and reopen Visual Studio
- Run `Update-Database` to apply the new migration to the database. Because your database doesn't exist yet, it will be created for you before the migration is applied.

Tip: If you make future changes to your 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.

Create a controller

Next, we'll add an MVC controller that will use EF to query and save data.

- Right-click on the **Controllers** folder in **Solution Explorer** and select *Add* → *New Item...*
- From the left menu select *Installed* → *Server-side*
- Select the **Class** item template
- Enter **BlogsController.cs** as the name and click **OK**
- Replace the contents of the file with the following code

```
using EFGetStarted.AspNetCore.NewDb.Models;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace EFGetStarted.AspNetCore.NewDb.Controllers
{
    public class BlogsController : Controller
    {
        private BloggingContext _context;

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

        public IActionResult Index()
        {
            return View(_context.Blogs.ToList());
        }

        public IActionResult Create()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(Blog blog)
        {
            if (ModelState.IsValid)
            {
                _context.Blogs.Add(blog);
                _context.SaveChanges();
                return RedirectToAction("Index");
            }

            return View(blog);
        }

    }
}
```

You'll notice that the controller takes a `BloggingContext` as a constructor parameter. ASP.NET dependency injection will take care of passing an instance of `BloggingContext` into your controller.

The controller contains an `Index` action, which displays all blogs in the database, and a `Create` action, which inserts a new blogs into the database.

Create views

Now that we have a controller it's time to add the views that will make up the user interface.

We'll start with the view for our `Index` action, that displays all blogs.

- Right-click on the **Views** folder in **Solution Explorer** and select *Add → New Folder*
- Enter **Blogs** as the name of the folder
- Right-click on the **Blogs** folder and select *Add → New Item...*
- From the left menu select *Installed → ASP.NET*
- Select the **MVC View Page** item template
- Enter **Index.cshtml** as the name and click **Add**
- Replace the contents of the file with the following code

```
@model IEnumerable<EFGetStarted.AspNetCore.NewDb.Models.Blog>

@{
    ViewBag.Title = "Blogs";
}
```

```

<h2>Blogs</h2>

<p>
    <a asp-controller="Blogs" asp-action="Create">Create New</a>
</p>

<table class="table">
    <tr>
        <th>Id</th>
        <th>Url</th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.BlogId)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Url)
            </td>
        </tr>
    }
</table>
```

We'll also add a view for the `Create` action, which allows the user to enter details for a new blog.

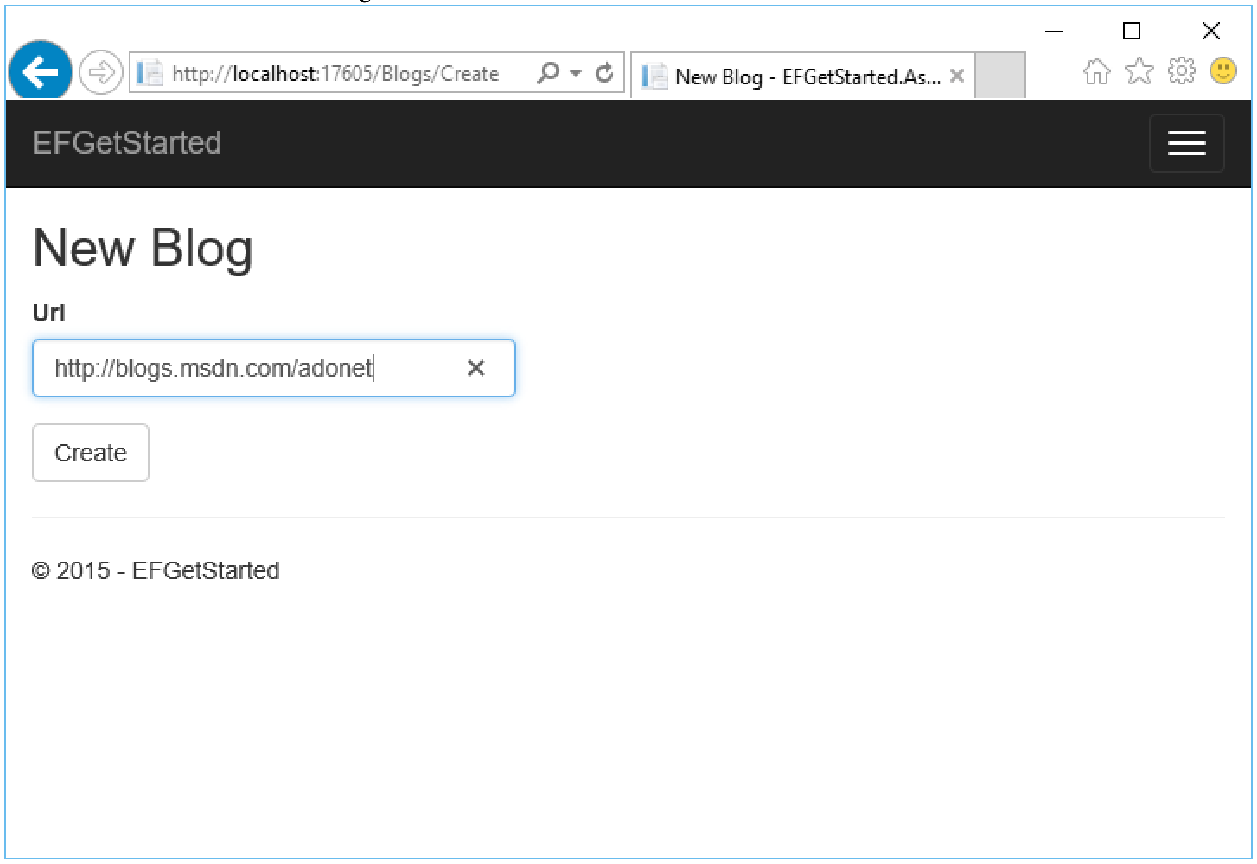
- Right-click on the **Blogs** folder and select *Add* → *New Item...*
- From the left menu select *Installed* → *ASP.NET Core*
- Select the **MVC View Page** item template
- Enter **Create.cshtml** as the name and click **Add**
- Replace the contents of the file with the following code

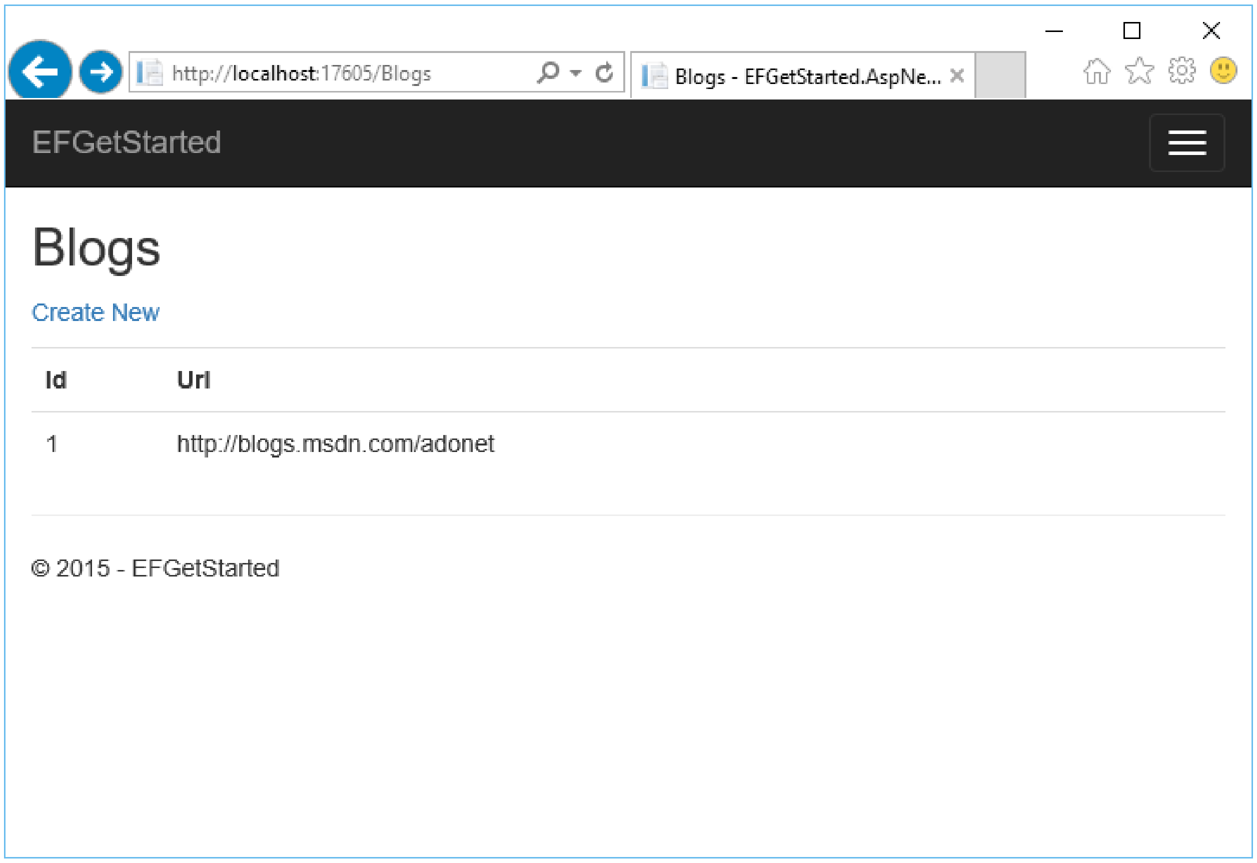
```
@model EFGetStarted.AspNetCore.NewDb.Models.Blog

@{
    ViewBag.Title = "New Blog";
}

<h2>@ViewData["Title"]</h2>

<form asp-controller="Blogs" asp-action="Create" method="post" class="form-horizontal" role="form">
    <div class="form-horizontal">
        <div asp-validation-summary="All" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Url" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Url" class="form-control" />
                <span asp-validation-for="Url" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>
```

Run the application

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

- *Debug → Start Without Debugging*
- The application will build and open in a web browser
- Navigate to **/Blogs**
- Click **Create New**
- Enter a **Url** for the new blog and click **Create**


http://localhost:17605/Blogs/Create
New Blog - EFGetStarted.As...
EFGetStarted
New Blog
Url
http://blogs.msdn.com/adonet
Create
© 2015 - EFGetStarted



Blogs

Create New

Id	Url
1	http://blogs.msdn.com/adonet

© 2015 - EFGetStarted

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

ASP.NET Core Application to Existing Database (Database First)

In this walkthrough, you will build an ASP.NET Core MVC application that performs basic data access using Entity Framework. You will use reverse engineering to create an Entity Framework model based on an existing database.

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015 Update 3
- .NET Core for Visual Studio
- *Blogging database*

Blogging database This tutorial uses a **Blogging** database on your LocalDb instance as the existing database.

Note: If you have already created the **Blogging** database as part of another tutorial, you can 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 CA
);
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 2015
- *File → New → Project...*
- From the left menu select *Templates → Visual C# → Web*
- Select the **ASP.NET Core Web Application (.NET Core)** project template
- Enter **EFGetStarted.AspNetCore.ExistingDb** as the name and click **OK**
- Wait for the **New ASP.NET Core Web Application** dialog to appear
- Select the **Web Application** template and ensure that **Authentication** is set to **No Authentication**
- Click **OK**

Install Entity Framework

To use EF Core, install the package for the database provider(s) you want to target. This walkthrough 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`

Note: In ASP.NET Core projects the `Install-Package` will complete quickly and the package installation will occur in the background. You will see **(Restoring...)** appear next to **References** in **Solution Explorer** while the install occurs.

To enable reverse engineering from an existing database we need to install a couple of other packages too.

- Run `Install-Package Microsoft.EntityFrameworkCore.Tools -Pre`
- Run `Install-Package Microsoft.EntityFrameworkCore.SqlServer.Design`
- Open **project.json**
- Locate the `tools` section and add the highlighted lines as shown below

```
"tools": {
  "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
},
```

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. 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.

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;" Micros
```

The reverse engineer process created entity classes and a derived context 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.

```
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 virtual ICollection<Post> Post { get; set; }
    }
}
```

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

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

namespace EFGetStarted.AspNetCore.ExistingDb.Models
{
    public partial class BloggingContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            #warning To protect potentially sensitive information in your connection string, you shou
            optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Co
        }

        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);
            });
        }

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

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.

Remove inline context configuration In ASP.NET Core, configuration is generally performed in **Startup.cs**. To conform to this pattern, we will move configuration of the database provider to **Startup.cs**.

- Open **Models\BloggingContext.cs**
- Delete the lines of code highlighted below

```
    public partial class BloggingContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            #warning To protect potentially sensitive information in your connection string, you shou
            optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Co
        }
```

- Add the lines of code highlighted below

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

Register and configure your context in Startup.cs In order for our MVC controllers to make use of `BloggingContext` we are going to 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 we can use the `AddDbContext` method to register it as a service.

- Locate the `ConfigureServices` method
- Add the lines that are highlighted in the following code

```
        public void ConfigureServices(IServiceCollection services)
        {
            var connection = @"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=Tru
            services.AddDbContext<BloggingContext>(options => options.UseSqlServer(connection));
```

Create a controller

Next, we'll add an MVC controller that will use EF to query and save data.

- Right-click on the **Controllers** folder in **Solution Explorer** and select *Add* → *New Item...*
- From the left menu select *Installed* → *Code*

- Select the **Class** item template
- Enter **BlogsController.cs** as the name and click **OK**
- Replace the contents of the file with the following code

```
using EFGetStarted.AspNetCore.ExistingDb.Models;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace EFGetStarted.AspNetCore.ExistingDb.Controllers
{
    public class BlogsController : Controller
    {
        private BloggingContext _context;

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

        public IActionResult Index()
        {
            return View(_context.Blog.ToList());
        }

        public IActionResult Create()
        {
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public IActionResult Create(Blog blog)
        {
            if (ModelState.IsValid)
            {
                _context.Blog.Add(blog);
                _context.SaveChanges();
                return RedirectToAction("Index");
            }

            return View(blog);
        }

    }
}
```

You'll notice that the controller takes a `BloggingContext` as a constructor parameter. ASP.NET dependency injection will take care of passing an instance of `BloggingContext` into your controller.

The controller contains an `Index` action, which displays all blogs in the database, and a `Create` action, which inserts a new blogs into the database.

Create views

Now that we have a controller it's time to add the views that will make up the user interface.

We'll start with the view for our `Index` action, that displays all blogs.

- Right-click on the **Views** folder in **Solution Explorer** and select *Add → New Folder*
- Enter **Blogs** as the name of the folder
- Right-click on the **Blogs** folder and select *Add → New Item...*
- From the left menu select *Installed → ASP.NET*
- Select the **MVC View Page** item template
- Enter **Index.cshtml** as the name and click **Add**
- Replace the contents of the file with the following code

```
@model IEnumerable<EFGetStarted.AspNetCore.ExistingDb.Models.Blog>

@{
    ViewBag.Title = "Blogs";
}

<h2>Blogs</h2>

<p>
    <a asp-controller="Blogs" asp-action="Create">Create New</a>
</p>

<table class="table">
    <tr>
        <th>Id</th>
        <th>Url</th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.BlogId)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Url)
            </td>
        </tr>
    }
</table>
```

We'll also add a view for the `Create` action, which allows the user to enter details for a new blog.

- Right-click on the **Blogs** folder and select *Add → New Item...*
- From the left menu select *Installed → ASP.NET*
- Select the **MVC View Page** item template
- Enter **Create.cshtml** as the name and click **Add**
- Replace the contents of the file with the following code

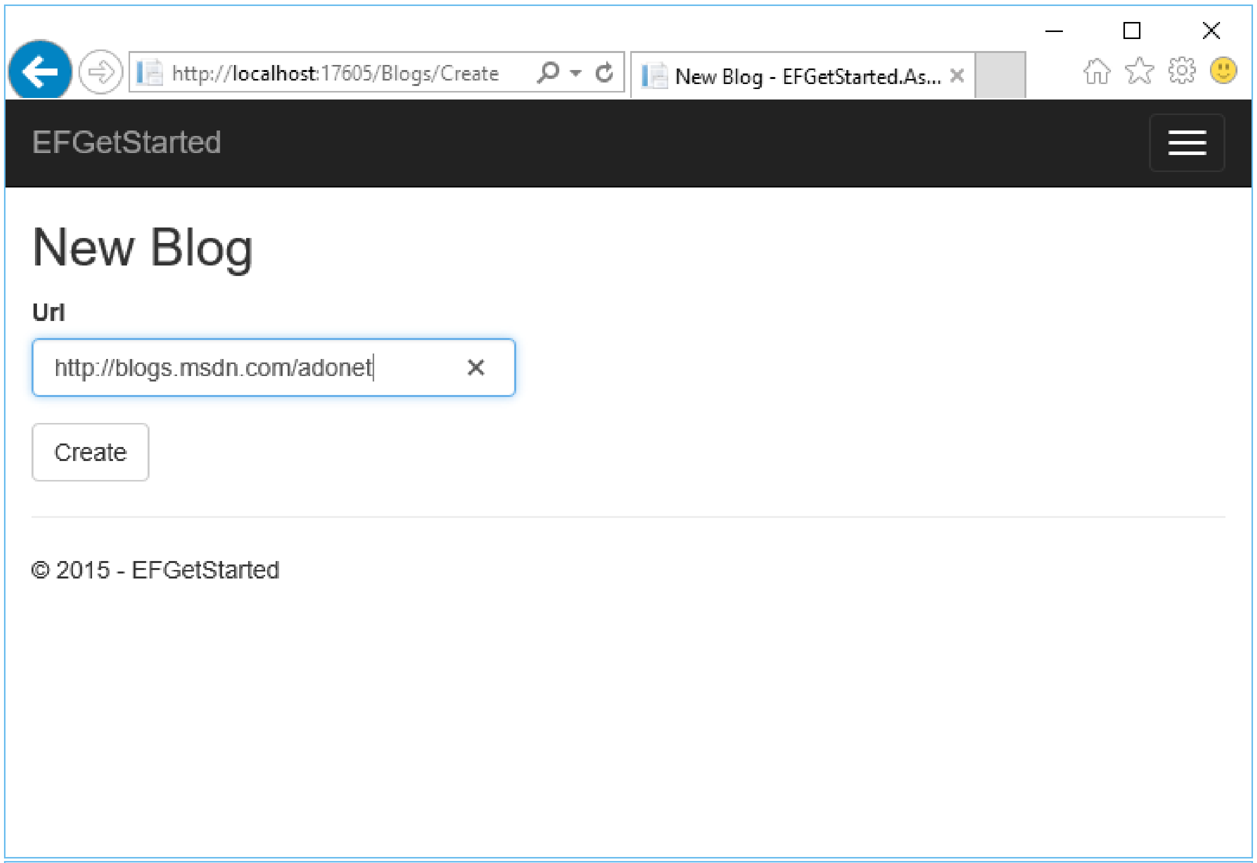
```
@model EFGetStarted.AspNetCore.ExistingDb.Models.Blog

@{
    ViewBag.Title = "New Blog";
}

```

```
<h2>@ViewData["Title"]</h2>

<form asp-controller="Blogs" asp-action="Create" method="post" class="form-horizontal" role="form">
    <div class="form-horizontal">
        <div asp-validation-summary="All" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Url" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Url" class="form-control" />
                <span asp-validation-for="Url" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
</form>
```

Run the application

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

- *Debug* → *Start Without Debugging*
- The application will build and open in a web browser
- Navigate to **/Blogs**
- Click **Create New**
- Enter a **Url** for the new blog and click **Create**


http://localhost:17605/Blogs/Create
New Blog - EFGetStarted.As...
EFGetStarted

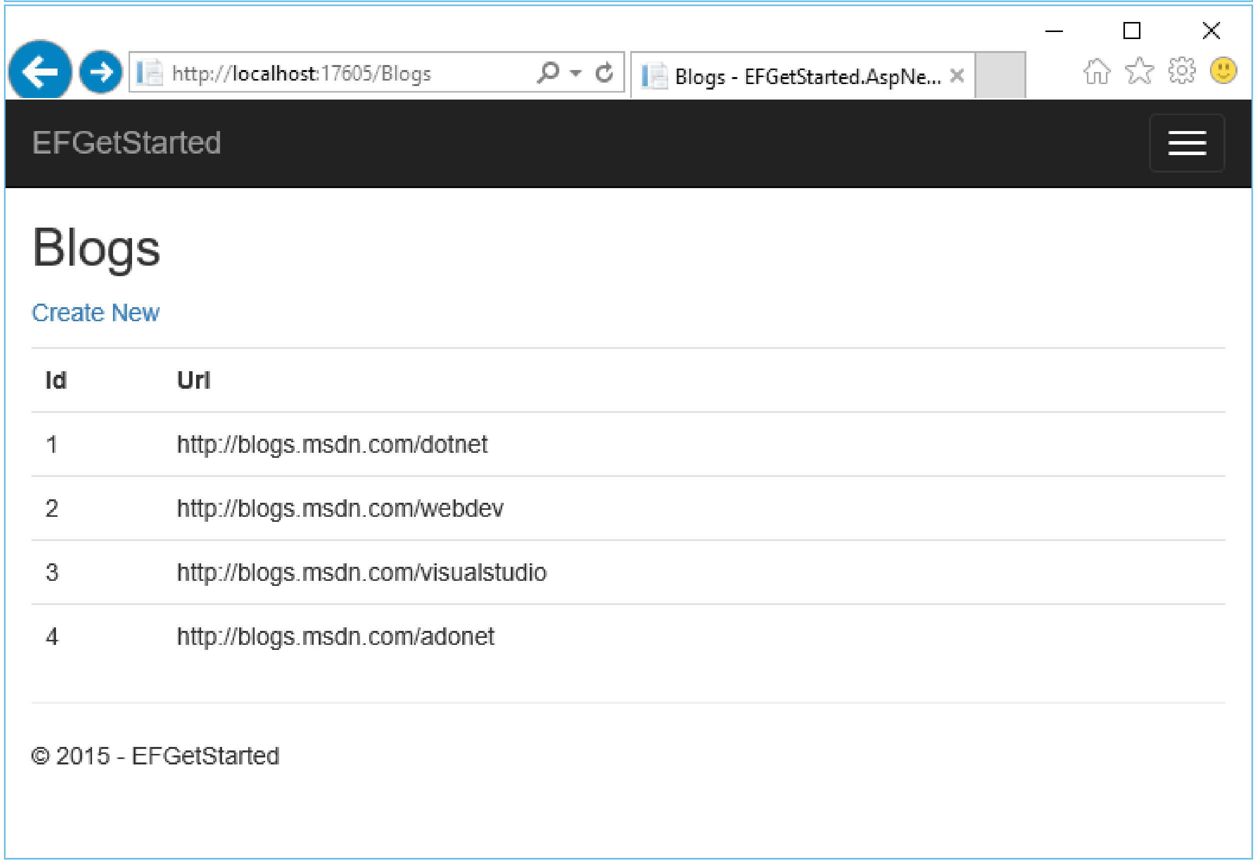
New Blog

Url

http://blogs.msdn.com/adonet

Create

© 2015 - EFGetStarted



Blogs

Create New

Id	Url
1	http://blogs.msdn.com/dotnet
2	http://blogs.msdn.com/webdev
3	http://blogs.msdn.com/visualstudio
4	http://blogs.msdn.com/adonet

© 2015 - EFGetStarted

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

2.4 Getting Started on Universal Windows Platform (UWP)

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

Entity Framework can be used to access a local SQLite database in Universal Windows Platform applications.

2.4.1 Available Tutorials

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Local SQLite on UWP

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

Caution: **Avoid using anonymous types in LINQ queries on UWP**. Deploying a UWP application to the app store requires your application to be compiled with .NET Native. Queries with anonymous types have poor performance on .NET Native or may crash the application.

In this article:

Tip: You can view this article's sample on GitHub.

Prerequisites

The following items are required to complete this walkthrough:

- Windows 10
- Visual Studio 2015 Update 3
- The latest version of Windows 10 Developer Tools

Create a new project

- Open Visual Studio 2015
- *File → New → Project...*
- From the left menu select *Templates → Visual C# → Windows → Universal*
- Select the **Blank App (Universal Windows)** project template
- Give the project a name and click **OK**

Upgrade Microsoft.NETCore.UniversalWindowsPlatform

Depending on your version of Visual Studio, the template may have generated your project with an old version of .NET Core for UWP. EF Core requires `Microsoft.NETCore.UniversalWindowsPlatform` version **5.2.2** or greater.

- *Tools → NuGet Package Manager → Package Manager Console*
- Run `Update-Package Microsoft.NETCore.UniversalWindowsPlatform`

Install Entity Framework

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*.

- *Tools → NuGet Package Manager → Package Manager Console*
- Run `Install-Package Microsoft.EntityFrameworkCore.Sqlite`

Later in this walkthrough we will also be using some Entity Framework commands to maintain the database. So we will install the commands package as well.

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

Create your model

Now it's time to define a context and entity classes that make up your 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 EFGetStarted.UWP
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Filename=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; }
    }
}
```

Tip: In a real application you would put each class in a separate file and put the connection string in the `App.Config` file and read it out using `ConfigurationManager`. For the sake of simplicity, we are putting everything in a single code file for this tutorial.

Create your database

Warning: Known Issue in Preview 2

Using EF Tools on UWP projects does not work without manually adding binding redirects.

- *File –> New –> File...*
- From the left menu select *Visual C# -> General -> Text File*
- Give the file the name "App.config"
- Add the following contents to the file

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="System.IO.FileSystem.Primitives" publicKeyToken="b03f5f7f11d50a3a"
        <bindingRedirect oldVersion="4.0.0.0" newVersion="4.0.1.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Threading.Overlapped" publicKeyToken="b03f5f7f11d50a3a" cult
        <bindingRedirect oldVersion="4.0.0.0" newVersion="4.0.1.0"/>
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.ComponentModel.Annotations" publicKeyToken="b03f5f7f11d50a3a
        <bindingRedirect oldVersion="4.1.0.0" newVersion="4.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

See Issue #5471 for more details.

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

- *Tools –> NuGet Package Manager –> Package Manager Console*
- Run `Add-Migration MyFirstMigration` to scaffold a migration to create the initial set of tables for your model.

Since we want the database to be created on the device that the app runs on, we will add some 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 for us.

- Right-click on **App.xaml** in **Solution Explorer** and select **View Code**
- Add the highlighted using to the start of the file

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
```

- Add the highlighted code to apply any pending migrations

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

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

Tip: If you make future changes to your model, you can 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 uses a `__EFMigrationsHistory` table in the database to keep track of which migrations have already been applied to the database.

Use your model

You can now use your model to perform data access.

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

```
<Page
    x:Class="EFGetStarted.UWP.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:EFGetStarted.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>
```

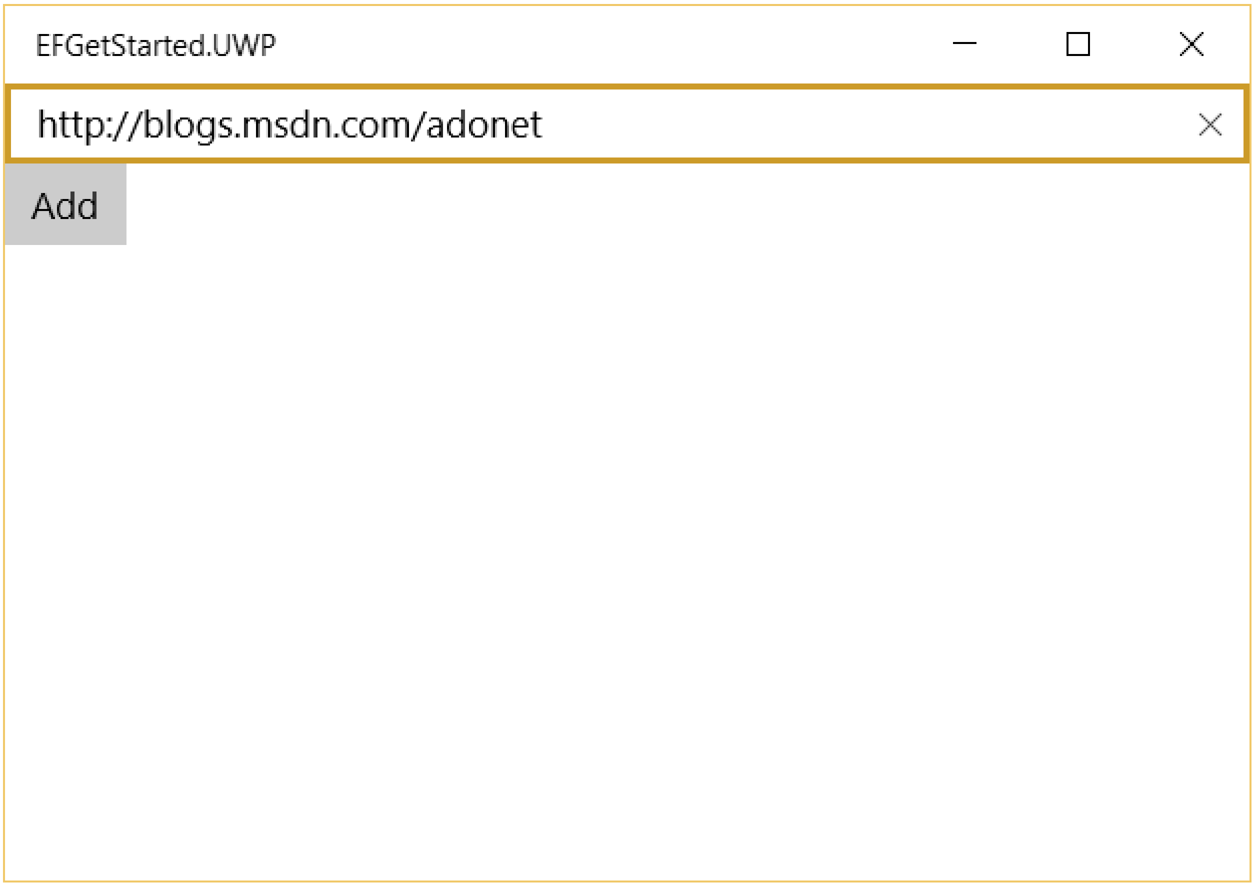
Now we'll add code to wire up the UI with the database

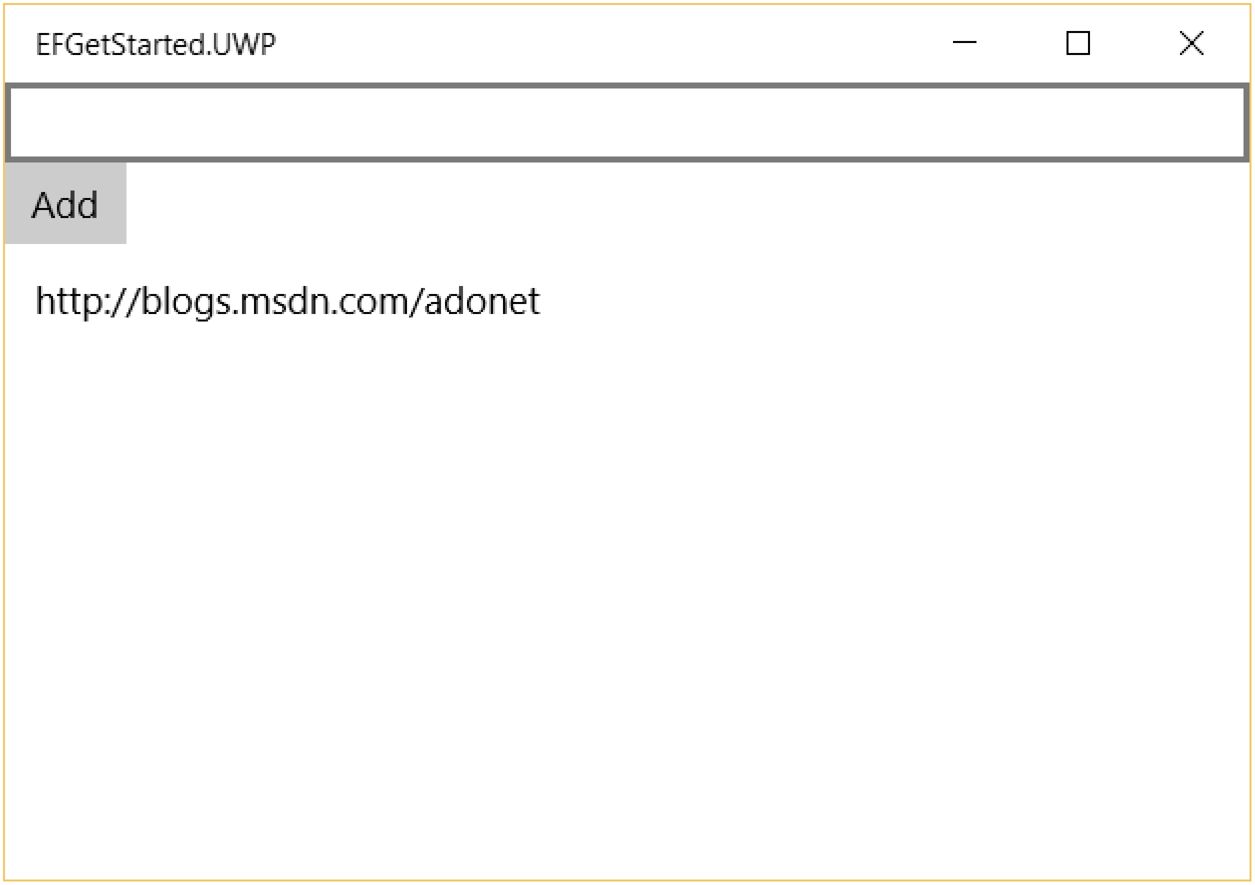
- Right-click **MainPage.xaml** in **Solution Explorer** and select **View Code**
- Add the highlighted code from the following listing

```
    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.

- *Debug* → *Start Without Debugging*
- The application will build and launch
- Enter a URL and click the **Add** button


EFGetStarted.UWP
http://blogs.msdn.com/adonet
Add



http://blogs.msdn.com/adonet

Next steps

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

Check out the numerous articles in this documentation to learn more about Entity Framework's features.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 3

Creating a Model

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.

In this section you can find information about conventions and configuration for the following:

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.1 Including & Excluding Types

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.

In this article:

3.1.1 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; }
}
```

3.1.2 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; }
}
```

3.1.3 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; }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.2 Including & Excluding Properties

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.

In this article:

3.2.1 Conventions

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

3.2.2 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; }
    }
```

3.2.3 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.3 Keys (primary)

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).

In this article:

3.3.1 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; }
    }
```

3.3.2 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; }
    }
```

3.3.3 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.4 Generated Properties

In this article:

3.4.1 Value Generation Patterns

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

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.

Caution: 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 (such as setting default value SQL of *GETDATE()*, see *Default Values*).

If you add an entity to the context that has a value assigned to the primary key 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.).

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`.

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).

Caution: 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 (such as a database trigger).

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. Also, if you explicitly change the value assigned to the property (thus marking it as modified) then that new value will be set in the database rather than a value being generated.

3.4.2 Conventions

By convention, primary keys that are of an integer or GUID data type will be setup to have values generated on add. All other properties will be setup with no value generation.

3.4.3 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; }
    }
```

Caution: 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; }
    }
```

Caution: 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.

3.4.4 Fluent API

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

No value generation (Fluent API)

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

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

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

Value generated on add (Fluent API)

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

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

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

Caution: 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 (Fluent API)

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

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

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

Caution: 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.5 Required/optional properties

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.

In this article:

3.5.1 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.

3.5.2 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; }
    }
```

3.5.3 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.6 Maximum Length

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.

In this article:

3.6.1 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).

3.6.2 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; }
    }
```

3.6.3 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; }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.7 Concurrency Tokens

If a property is configured as a concurrency token then EF will check that no other user has modified that value in the database when saving changes to that record. EF uses an optimistic concurrency pattern, meaning it will assume the value has not changed and try to save the data, but throw if it finds the value has been changed.

For example we may want to configure `LastName` on `Person` to be a concurrency token. This means that if one user tries to save some changes to a `Person`, but another user has changed the `LastName` then an exception will be thrown. This may be desirable so that your application can prompt the user to ensure this record still represents the same actual person before saving their changes.

In this article:

3.7.1 How concurrency tokens work in EF

Data stores can enforce concurrency tokens by checking that any record being updated or deleted still has the same value for the concurrency token that was assigned when the context originally loaded the data from the database.

For example, relational database achieve this by including the concurrency token in the `WHERE` clause of any `UPDATE` or `DELETE` commands and checking the number of rows that were affected. If the concurrency token still matches then one row will be updated. If the value in the database has changed, then no rows are updated.

```
UPDATE [Person] SET [FirstName] = @p1
WHERE [PersonId] = @p0 AND [LastName] = @p2;
```

3.7.2 Conventions

By convention, properties are never configured as concurrency tokens.

3.7.3 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; }
    }
```

3.7.4 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; }
    }
```

3.7.5 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)
                .ValueGeneratedOnAddOrUpdate()
                .IsConcurrencyToken();
        }
    }

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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.8 Shadow Properties

Shadow properties are properties that do not exist in your entity class. The value and state of these properties is maintained purely in the Change Tracker.

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"));
```

In this article:

3.8.1 Conventions

By convention, shadow properties are only created 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; }
    }
```

3.8.2 Data Annotations

Shadow properties can not be created with data annotations.

3.8.3 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.9 Relationships

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.

In this article:

3.9.1 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; }
    }
```

3.9.2 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 foreig

- 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; }
    }
```

Caution: If there are multiple navigation properties defined between two types (i.e. 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 *SetNull* for optional relationships. *Cascade* means dependent entities are also deleted. *SetNull* means that foreign key properties in dependent entities are set to null.

Note: This cascading behavior is only applied to entities that are being tracked by the context. A corresponding cascade behavior should be setup in the database to ensure data that is not being tracked by the context has the same action applied. If you use EF to create the database, this cascade behavior will be setup for you.

3.9.3 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; }
    }
```

Note: 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; }
    }
```

3.9.4 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; }
    }
```

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; }
    }
```

Caution: The order that you specify principal key properties must match the order they are specified for the foreign key.

Required

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/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.

There are three behaviors that control how a delete operation is applied to dependent entities in a relationship when the princip

- **Cascade:** Dependent entities are also deleted.
- **SetNull:** The foreign key properties in dependent entities are set to null.
- **Restrict:** The delete operation is not applied to dependent entities. The dependent entities remain unchanged.

Note: This cascading behavior is only applied to entities that are being tracked by the context. A corresponding cascade behavior should be setup in the database to ensure data that is not being tracked by the context has the same action applied. If you use EF to create the database, this cascade behavior will be setup for you.

```
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; }
}
```

3.9.5 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.10 Indexes

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.

In this article:

3.10.1 Conventions

By convention, an index is created in each property (or set of properties) that are used as a foreign key.

3.10.2 Data Annotations

Indexes can not be created using data annotations.

3.10.3 Fluent API

You can use the Fluent API 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; }
    }
```

Note: 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.11 Alternate Keys

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).

Note: If you just want to enforce uniqeness 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.

In this article:

3.11.1 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; }
    }
```

3.11.2 Data Annotations

Alternate keys can not be configured using Data Annotations.

3.11.3 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; }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.12 Inheritance

Inheritance in the EF model is used to control how inheritance in the entity classes is represented in the database.

In this article:

3.12.1 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<TEntity>* 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<TEntity>* for one or more entities in the hierarchy, you can use the Fluent API to ensure they are included in the model.

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

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

3.12.2 Data Annotations

You cannot use Data Annotations to configure inheritance.

3.12.3 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

3.13 Backing Fields

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). This is useful when there is no property setter, or the setter contains logic that should not be executed when setting initial property values for existing entities being loaded from the database.

Caution: The `ChangeTracker` has not yet been enabled to use backing fields when it needs to set the value of a property. This is only an issue for foreign key properties and generated properties - as the change tracker needs to propagate values into these properties. For these properties, a property setter must still be exposed.
Issue #4461 is tracking enabling the `ChangeTracker` to write to backing fields for properties with no setter.

3.13.1 Conventions

By convention, the following fields will be discovered as backing fields for a given property (listed in precedence order):

- <propertyName> differing only by case
- _<propertyName>
- m_<propertyName>

```
public class Blog
{
    private string _url;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```

3.13.2 Data Annotations

Backing fields cannot be configured with data annotations.

3.13.3 Fluent API

There is no top level API for configuring backing fields, but you can use the Fluent API to set annotations that are used to store backing field information.

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasAnnotation("BackingField", "_blogUrl");
    }
}

public class Blog
{
    private string _blogUrl;

    public int BlogId { get; set; }
    public string Url => _blogUrl;
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14 Relational Database Modeling

This section covers aspects of modeling that are specific to relational databases.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.1 Table Mapping

Table mapping identifies which table data should be queried from and saved to in the database.

In this article:

Conventions

By convention, each entity will be setup 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.

```
    [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.

```
    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");
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.2 Column Mapping

Column mapping identifies which column data should be queried from and saved to in the database.

In this article:

Conventions

By convention, each property will be setup 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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.3 Data Types

Data type refers to the database specific type of the column to which a property is mapped.

In this article:

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 the column.

```
    public class Blog
    {
        public int BlogId { get; set; }
        [Column(TypeName = "varchar(200)")]
        public string Url { get; set; }
    }
```

Fluent API

You can use the Fluent API to specify an exact data type for the column.

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .HasColumnType("varchar(200)");
        }
    }

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

If you are targeting more than one relational provider with the same model then you probably want to specify a data type for each provider rather than a global one to be used for all relational providers.

```
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .ForSqlServerHasColumnType("varchar(200)");
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.4 Primary Keys

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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.5 Default Schema

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");
        }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.6 Computed Columns

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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.7 Sequences

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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.8 Default Values

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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.9 Indexes

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; }
    }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.10 Foreign Key Constraints

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; }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.11 Alternate Keys (Unique Constraints)

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; }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

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).

3.14.12 Inheritance (Relational Database)

Inheritance in the EF model is used to control how inheritance in the entity classes is represented in the database.

In this article:

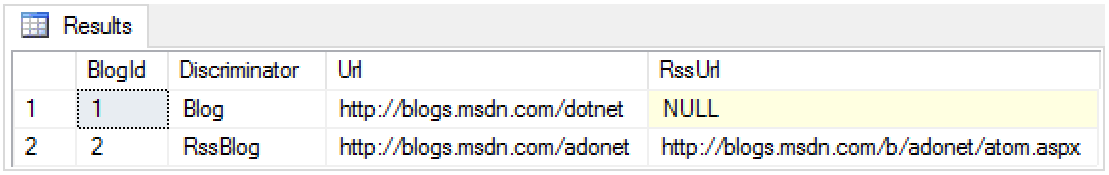
Conventions

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 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; }
    }
```



	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; }
    }
```

Tip: You can view this article's sample on GitHub.

3.15 Methods of configuration

3.15.1 Fluent API

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();
    }
}
```

3.15.2 Data Annotations

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

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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 4

Querying Data

The following articles are available

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.1 Basic Query

Entity Framework Core uses Language Integrate 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.

In this article:

Tip: You can view this article's sample on GitHub.

4.1.1 101 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.

4.1.2 Loading all data

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

4.1.3 Loading a single entity

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

4.1.4 Filtering

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.2 Loading Related Data

Entity Framework Core allows you to use the navigation properties in your model to load related entities. There are three comm

- **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. Lazy loading is not yet possible with EF Core.

In this article:

Tip: You can view this article's sample on GitHub.

4.2.1 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.

```
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.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
    .Include(blog => blog.Owner)
    .ToList();
```

Including multiple levels

You can drill down thru 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.

```
var blogs = context.Blogs
    .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
    .ToList();
```

You can chain multiple calls to `ThenInclude` to continue including further levels of related data.

```
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.

```
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();
```

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.

```
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;")
        .ConfigureWarnings(warnings => warnings.Throw(CoreEventId.IncludeIgnoredWarning));
}
```

4.2.2 Explicit loading

Explicit loading does not yet have a first class API in EF Core. You can view the explicit loading item on our backlog to track this feature.

However, you can use a LINQ query to load the data related to an existing entity instance, by filtering to entities related to the entity in question. Because EF Core will automatically fix-up navigation properties to any other entities that were previously loaded into the context instance, the loaded data will be populated into the desired navigation property.

In the following example, a query is used to load a blog, and then a later query is used to load the posts related to the blog. The loaded posts will be present in the `Posts` property of the previously loaded blog.

```
var blog = context.Blogs
    .Single(b => b.BlogId == 1);

context.Posts
    .Where(p => p.BlogId == blog.BlogId)
    .Load();
```

4.2.3 Lazy loading

Lazy loading is not yet supported by EF Core. You can view the lazy loading item on our backlog to track this feature.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.3 Client vs. Server Evaluation

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.

In this article:

Tip: You can view this article's sample on GitHub.

4.3.1 Client eval

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;
}
```

4.3.2 Disabling client evaluation

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();
```

By default, EF Core will log a warning when client evaluation is performed. See *Logging* for more information on viewing logging output. 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)
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.4 Tracking vs. No-Tracking

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.

In this article:

Tip: You can view this article's sample on GitHub.

4.4.1 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();
}
```

4.4.2 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();
}
```

4.4.3 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
            });
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.5 Raw SQL Queries

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 being sent to the database.

In this article:

Tip: You can view this article's sample on GitHub.

4.5.1 Limitations

There are a couple of limitations to be aware of when using raw SQL queries:

- SQL queries can only be used to return entity types that are part of your model. There is an enhancement on our backlog to enable returning ad-hoc types from raw SQL queries.
- The SQL query must return data for all properties of the entity type.
- The column names in the result set must match the column names that properties are mapped to. Note this is different from EF6.x 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*).

4.5.2 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();
```

4.5.3 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();
```

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();
```

4.5.4 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 being 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 {0}", 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 {0}", searchTerm)
    .Include(b => b.Posts)
    .ToList();
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

4.6 How Query Works

Entity Framework Core uses Language Integrate 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.

In this article:

4.6.1 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

1. 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

1. The database provider identifies which parts of the query can be evaluated in the database

2. These parts of the query are translated to database specific query language (e.g. SQL for a relational database)
3. 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
    1. 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
    2. 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
    - If not, a new entity is created and returned

4.6.2 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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 5

Saving Data

The following articles are available

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.1 Basic Save

Learn how to add, modify, and remove data using your context and entity classes.

In this article:

Tip: You can view this article's sample on GitHub.

5.1.1 ChangeTracker & SaveChanges

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*.

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

                Console.WriteLine(blog.BlogId + ": " +  blog.Url);
            }
```

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

5.1.4 Deleting Data

Use the *DbSet.Remove* method to delete instances of you 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 (i.e. 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 db = new BloggingContext())
        {
            var blog = db.Blogs.First();
            db.Blogs.Remove(blog);
            db.SaveChanges();
        }
```

5.1.5 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 db = new BloggingContext())
        {
            db.Blogs.Add(new Blog { Url = "http://sample.com/blog_one" });
            db.Blogs.Add(new Blog { Url = "http://sample.com/blog_two" });

            var firstBlog = db.Blogs.First();
            firstBlog.Url = "";

            var lastBlog = db.Blogs.Last();
            db.Blogs.Remove(lastBlog);

```

```
                db.SaveChanges();
            }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.2 Related Data

In addition to isolated entities, you can also make use of the relationships defined in your model.

In this article:

Tip: You can view this article's sample on GitHub.

5.2.1 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();
            }
```

5.2.2 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.First();
                var post = new Post { Title = "Intro to EF Core" };

                blog.Posts.Add(post);
                context.SaveChanges();
            }
```

5.2.3 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();

                blog.Posts.Add(post);
                context.SaveChanges();
            }
```

5.2.4 Removing relationships

You can remove a relationship by setting a reference navigation to `null`, or removing the related entity from a collection navigation.

If a cascade delete is configured, the child/dependent entity will be deleted from the database, see *Cascade Delete* for more information. If no cascade delete is configured, the foreign key column in the database will be set to null (if the column does not accept nulls, an exception will be thrown).

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();
            }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.3 Cascade Delete

Cascade delete allows deletion of a principal/parent entity to have a side effect on dependent/child entities it is related to.

There are three cascade delete behaviors:

- **Cascade:** Dependent entities are also deleted.
- **SetNull:** The foreign key properties in dependent entities are set to null.
- **Restrict:** The delete operation is not applied to dependent entities. The dependent entities remain unchanged.

See *Relationships* for more information about conventions and configuration for cascade delete.

In this article:

Tip: You can view this article's sample on GitHub.

5.3.1 Cascading to tracked entities

When you call *SaveChanges*, the cascade delete rules will be applied to any entities that are being tracked by the context.

Consider a simple *Blog* and *Post* model where the relationship between the two entities is required. By convention. the cascade behavior for this relationship is set to *Cascade*.

The following code loads a Blog and all its related Posts from the database (using the *Include* method). The code then deletes the Blog.

```
            using (var db = new BloggingContext())
            {
                var blog = db.Blogs.Include(b => b.Posts).First();
                db.Remove(blog);
                db.SaveChanges();
            }
```

Because all the Posts are tracked by the context, the cascade behavior is applied to them before saving to the database. EF therefore issues a *DELETE* statement for each entity.

```
DELETE FROM [Post]
WHERE [PostId] = @p0;
DELETE FROM [Post]
WHERE [PostId] = @p1;
DELETE FROM [Blog]
WHERE [BlogId] = @p2;
```

5.3.2 Cascading to untracked entities

The following code is almost the same as our previous example, except it does not load the related Posts from the database.

```
            using (var db = new BloggingContext())
            {
                var blog = db.Blogs.First();
                db.Remove(blog);
                db.SaveChanges();
            }
```

Because the Posts are not tracked by the context, a *DELETE* statement is only issued for the *Blog*. This relies on a corresponding cascade behavior being present in the database to ensure data that is not tracked by the context is also deleted. If you use EF to create the database, this cascade behavior will be setup for you.

```
DELETE FROM [Blog]
WHERE [BlogId] = @p0;
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.4 Concurrency Conflicts

If a property is configured as a concurrency token then EF will check that no other user has modified that value in the database when saving changes to that record.

In this article:

Tip: You can view this article's sample on GitHub.

5.4.1 How concurrency works in EF

For a detailed description of how concurrency works in Entity Framework Core, see *Concurrency Tokens*.

5.4.2 Resolving concurrency conflicts

Resolving a concurrency conflict involves using an algorithm to merge the pending changes from the current user with the changes made in the database. The exact approach will vary based on your application, but a common approach is to display the values to the user and have them decide the correct values to be stored in the database.

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.

To handle a concurrency conflict, catch a `DbUpdateConcurrencyException` during `SaveChanges()`, use `DbUpdateConcurrencyException.Entries` to prepare a new set of changes for the affected entities, and then retry the `SaveChanges()` operation.

In the following example, `Person.FirstName` and `Person.LastName` are setup as concurrency token. There is a `// TODO:` comment in the location where you would include application specific logic to choose the value to be saved to the database.

```
using Microsoft.EntityFrameworkCore;
using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;

namespace EFSaving.Concurrency
{
    public class Sample
    {
        public static void Run()
        {
            // Ensure database is created and has a person in it
            using (var context = new PersonContext())
            {
                context.Database.EnsureDeleted();
                context.Database.EnsureCreated();

                context.People.Add(new Person { FirstName = "John", LastName = "Doe" });
                context.SaveChanges();
            }

            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 persons name in the database (will cause a concurrency conflict)
                context.Database.ExecuteSqlCommand("UPDATE dbo.People SET FirstName = 'Jane' WHERE Pe

                try
                {
                    // Attempt to save changes to the database
                    context.SaveChanges();
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    foreach (var entry in ex.Entries)
                    {
                        if (entry.Entity is Person)
                        {
                            // Using a NoTracking query means we get the entity but it is not tracke
                            // and will not be merged with existing entities in the context.
                            var databaseEntity = context.People.AsNoTracking().Single(p => p.PersonI
                            var databaseEntry = context.Entry(databaseEntity);

                            foreach (var property in entry.Metadata.GetProperties())
                            {
                                var proposedValue = entry.Property(property.Name).CurrentValue;
                                var originalValue = entry.Property(property.Name).OriginalValue;
                                var databaseValue = databaseEntry.Property(property.Name).CurrentValu

                                // TODO: Logic to decide which value should be written to database
                                // entry.Property(property.Name).CurrentValue = <value to be saved>;

                                // Update original values to
                                entry.Property(property.Name).OriginalValue = databaseEntry.Property
                            }
                        }
                        else
                        {
                            throw new NotSupportedException("Don't know how to handle concurrency cor
                        }
                    }

                    // Retry the save operation
                    context.SaveChanges();
                }
            }
        }

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

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

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

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

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

            public string PhoneNumber { get; set; }
        }

    }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.5 Transactions

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.

In this article:

Tip: You can view this article's sample on GitHub.

5.5.1 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.

5.5.2 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-rollbac
                        // when disposed if either commands fails
                        transaction.Commit();
                    }
                    catch (Exception)
                    {
                        // TODO: Handle failure
                    }
                }
            }
```

5.5.3 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-rollbac
                        // when disposed if either commands fails
                        transaction.Commit();
                    }
                    catch (Exception)
                    {
                        // TODO: Handle failure
                    }
                }
            }
```

5.5.4 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.

```
            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
                }
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.6 Disconnected Entities

Note: This topic hasn't been written yet! You can track the status of this issue through our public GitHub issue tracker. Learn how you can contribute on GitHub.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

5.7 Setting explicit values for generated properties

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.

In this article:

Tip: You can view this article's sample on GitHub.

5.7.1 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; }
    }
```

The context is setup to target SQL Server:

- By convention the `Employee.EmployeeId` property will be a store generated `IDENTITY` column
- The `Employee.EmploymentStarted` property has also been setup to have values generated by the database for new entities

```
    public class EmployeeContext : DbContext
    {
        public DbSet<Employee> Employees { get; set; }

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Employee>()
                .Property(b => b.EmploymentStarted)
                .HasDefaultValueSql("CONVERT(date, GETDATE())");
        }
    }
```

5.7.2 Saving an explicit value during add

In the following code, two employees are being inserted 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 db = new EmployeeContext())
            {
                db.Employees.Add(new Employee { Name = "John Doe" });
                db.Employees.Add(new Employee { Name = "Jane Doe", EmploymentStarted = new DateTime(2
                db.SaveChanges();

                foreach (var employee in db.Employees)
                {
                    Console.WriteLine(employee.EmployeeId + ": " + employee.Name + ", " + employee.Em
            }
        }
```

The code results in the following output, showing that the database generated a value for the first employee and our explicit value was used for the second:

```
1: John Doe, 1/28/2016 12:00:00 AM
2: Jane Doe, 1/1/2000 12:00:00 AM
```

Explicit values into SQL Server IDENTITY columns

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 db = new EmployeeContext())
            {
                db.Employees.Add(new Employee { EmployeeId = 100, Name = "John Doe" });
                db.Employees.Add(new Employee { EmployeeId = 101, Name = "Jane Doe" });

                db.Database.OpenConnection();
                try
                {
                    db.Database.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.Employee ON");
                    db.SaveChanges();
                    db.Database.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.Employee OFF");
                }
                finally
                {
                    db.Database.CloseConnection();
                }


                foreach (var employee in db.Employees)
                {
                    Console.WriteLine(employee.EmployeeId + ": " + employee.Name);
                }
            }
```

5.7.3 Setting an explicit values during update

Caution: Due to various bugs, this scenario is not properly supported in the current pre-release of EF Core. See documentation issue #122 for more details.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 6

EF Core vs. EF6.x

The following articles are available

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.1 Which One Is Right for You

The following information will help you choose between Entity Framework Core and Entity Framework 6.x.

In this article:

6.1.1 What is EF6.x

Entity Framework 6.x (EF6.x) is a tried and tested data access technology with many years of features and stabilization. It first 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 the most popular package on NuGet.org.

EF6.x continues to be a supported product, and will continue to see bug fixes and minor improvements for some time to come.

6.1.2 What is EF Core

Entity Framework Core (EF Core) is a lightweight, extensible, and cross-platform version of Entity Framework. EF Core introduces many improvements and new features when compared with EF6.x. At the same time, EF Core is a new code base and very much a v1 product.

EF Core keeps the developer experience from EF6.x, and most of the top-level APIs remain the same too, so EF Core will feel very familiar to folks who have used EF6.x. At the same time, EF Core is built over a completely new set of core components. This means EF Core doesn't automatically inherit all the features from EF6.x. Some of these features will show up in future releases (such as lazy loading and connection resiliency), other less commonly used features will not be implemented in EF Core.

The new, extensible, and lightweight core has also allowed us to add some features to EF Core that will not be implemented in EF6.x (such as alternate keys and mixed client/database evaluation in LINQ queries).

See *Feature Comparison* for a detailed comparison of how the feature set in EF Core compares to EF6.x.

6.1.3 Guidance for new applications

Because EF Core is a new product, and still lacks some critical O/RM features, EF6.x will still be the most suitable choice for many applications.

These are the types of applications we would recommend using EF Core for.

- New applications that do not require features that are not yet implemented in EF Core. Review *Feature Comparison* to see if EF Core may be a suitable choice for your application.
- Applications that target .NET Core, such as Universal Windows Platform (UWP) and ASP.NET Core applications. These applications can not use EF6.x as it requires the Full .NET Framework (i.e. .NET Framework 4.5).

6.1.4 Guidance for existing EF6.x applications

Because of the fundamental changes in EF Core we do not recommend attempting to move an EF6.x application to EF Core unless you have a compelling reason to make the change. If you want to move to EF Core to make use of new features, then make sure you are aware of its limitations before you start. Review *Feature Comparison* to see if EF Core may be a suitable choice for your application.

You should view the move from EF6.x to EF Core as a port rather than an upgrade. See *Porting from EF6.x to EF Core* for more information.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.2 Feature Comparison

The following information will help you choose between Entity Framework Core and Entity Framework 6.x.

In this article:

- *Features not in EF Core*
- *Side-by-side comparison*

6.2.1 Features not in EF Core

This is a list of features not currently implemented in EF Core that are likely to impact your ability to use it in a given application. This is by no means an exhaustive list of possible O/RM features, but the features that we feel have the highest impact on developers.

- **Creating a Model**
  - **Complex/value types** are types that do not have a primary key and are used to represent a set of properties on an entity type.
  - **Visualizing a model** to see a graphical representation of the code-based model.

    - **Simple type conversions** such as string => xml.
    - **Spatial data types** such as SQL Server's *geography* & *geometry*.
    - **Many-to-many relationships** without join entity. You can already model a many-to-many relationship with a join entity, see *Relationships* for details.
    - **Alternate inheritance mapping patterns** for relational databases, such as table per type (TPT) and table per concrete type (TPC). Table per hierarchy (TPH) is already supported.

- **Querying Data**
    - **Improved translation** to enable more queries to successfully execute, with more logic being evaluated in the database (rather than in-memory).
    - **GroupBy translation** in particular will move translation of the LINQ GroupBy operator to the database, rather than in-memory.
    - **Lazy loading** enables navigation properties to be automatically populated from the database when they are accessed.
    - **Explicit Loading** allows you to trigger population of a navigation property on an entity that was previously loaded from the database.
    - **Raw SQL queries for non-model types** allows a raw SQL query to be used to populate types that are not part of the model (typically for denormalized view-model data).
- **Saving Data**
    - **Simple command interception** provides an easy way to read/write commands before/after they are sent to the database.
    - **Missing EntityEntry APIs from EF6.x** such as `Reload`, `GetModifiedProperties`, `GetDatabaseValues` etc.
    - **Stored procedure mapping** allows EF to use stored procedures to persist changes to the database (`FromSql` already provides good support for using a stored procedure to query, see *Raw SQL Queries* for details).
    - **Connection resiliency** automatically retries failed database commands. This is especially useful when connection to SQL Azure, where transient failures are common.
- **Database Schema Management**
    - **Visual Studio wizard for reverse engineer** that allows you to visually configure connection, select tables, etc. when creating a model from an existing database.
    - **Update model from database** allows a model that was previously reverse engineered from the database to be refreshed with changes made to the schema.
    - **Seed data** allows a set of data to be easily upserted to the database.

6.2.2 Side-by-side comparison

The following table compares the features available in EF Core and EF6.x. It is intended to give a high level comparison and does not list every feature, or attempt to give details on possible differences between how the same feature works.

Creating a Model	EF6.x	EF Core 1.0.0
Basic modelling (classes, properties, etc.)	Yes	Yes
Conventions	Yes	Yes

Continued on next page

Table 6.1 – continued from previous page

Custom conventions	Yes	Partial
Data annotations	Yes	Yes
Fluent API	Yes	Yes
Inheritance: Table per hierarchy (TPH)	Yes	Yes
Inheritance: Table per type (TPT)	Yes	
Inheritance: Table per concrete class (TPC)	Yes	
Shadow state properties		Yes
Alternate keys		Yes
Many-to-many: With join entity	Yes	Yes
Many-to-many: Without join entity	Yes	
Key generation: Database	Yes	Yes
Key generation: Client		Yes
Complex/value types	Yes	
Spatial data	Yes	
Graphical visualization of model	Yes	
Graphical drag/drop editor	Yes	
Model format: Code	Yes	Yes
Model format: EDMX (XML)	Yes	
Reverse engineer model from database: Command line		Yes
Reverse engineer model from database: VS wizard	Yes	
Incremental update model from database	Yes	
Querying Data	**EF6.x**	**EF Core 1.0.0**
LINQ: Simple queries	Stable	Stable
LINQ: Moderate queries	Stable	Stabilizing
LINQ: Complex queries	Stable	In-Progress
LINQ: Queries using navigation properties	Stable	In-Progress
“Pretty” SQL generation	Poor	Yes
Mixed client/server evaluation		Yes
Loading related data: Eager	Yes	Yes
Loading related data: Lazy	Yes	
Loading related data: Explicit	Yes	
Raw SQL queries: Model types	Yes	Yes
Raw SQL queries: Un-mapped types	Yes	
Raw SQL queries: Composing with LINQ		Yes
Saving Data	**EF6.x**	**EF Core 1.0.0**
SaveChanges	Yes	Yes
Change tracking: Snapshot	Yes	Yes
Change tracking: Notification	Yes	Yes
Accessing tracked state	Yes	Partial
Optimistic concurrency	Yes	Yes
Transactions	Yes	Yes
Batching of statements		Yes
Stored procedure	Yes	
Detached graph support (N-Tier): Low level APIs	Poor	Yes
Detached graph support (N-Tier): End-to-end		Poor
Other Features	**EF6.x**	**EF Core 1.0.0**
Migrations	Yes	Yes

Continued on next page

Table 6.1 – continued from previous page

Database creation/deletion APIs	Yes	Yes
Seed data	Yes	
Connection resiliency	Yes	
Lifecycle hooks (events, command interception, ...)	Yes	
Database Providers	**EF6.x**	**EF Core 1.0.0**
SQL Server	Yes	Yes
MySQL	Yes	Paid only, unpaid coming soon [1]
PostgreSQL	Yes	Yes
Oracle	Yes	Paid only, unpaid coming soon [1]
SQLite	Yes	Yes
SQL Compact	Yes	Yes
DB2	Yes	Yes
InMemory (for testing)		Yes
Azure Table Storage		Prototype
Redis		Prototype
Application Models	**EF6.x**	**EF Core 1.0.0**
WinForms	Yes	Yes
WPF	Yes	Yes
Console	Yes	Yes
ASP.NET	Yes	Yes
ASP.NET Core		Yes
Xamarin		Coming soon [2]
UWP		Yes

Footnotes:

- [1] Paid providers are available, unpaid providers are being worked on. The teams working on the unpaid providers have not shared public details of timeline etc.
- [2] EF Core is built to work on Xamarin when support for .NET Standard is enabled in Xamarin.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.3 Porting from EF6.x to EF Core

The following articles cover porting from EF6.x to EF Core

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.3.1 Ensure EF Core Will Work for Your Application

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

In this article:

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.x. 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.x 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.x, 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 (i.e. `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.x has a significant amount of magic it performs around selecting the database connection and initializing the database. Son

- If no configuration is performed, EF6.x 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.x 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.x 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.3.2 Porting an EDMX-Based Model (Model First & Database First)

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.

In this article:

Install EF Core NuGet packages

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

You also need to install the **design time** NuGet package for the database provider you want to use. This is typically the runtime database provider package name with `.Design` post-fixed. For example, when targeting SQL Server, you would install `Microsoft.EntityFrameworkCore.SqlServer.Design`. See *Database Providers* for details.

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;" Microso
```

Remove EF6.x model

You would now remove the EF6.x model from your application.

It is fine to leave the EF6.x NuGet package (EntityFramework) installed, as EF Core and EF6.x can be used side-by-side in the same application. However, if you aren't intending to use EF6.x 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.x 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.3.3 Porting a Code-Based Model (Code First & Code First to Existing Database)

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

In this article:

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.x NuGet package (EntityFramework) installed, as EF Core and EF6.x can be used side-by-side in the same application. However, if you aren't intending to use EF6.x 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.x 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.x 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"].Connect
    }
}
```

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.x migrations to EF Core.

If possible, it is best to assume that all previous migrations from EF6.x 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

6.4 EF6.x and EF Core in the Same Application

It is possible to use EF Core and EF6.x in the same application. EF Core and EF6.x have the same type names that differ only by namespace, so this may complicate code that attempts to use both EF Core and EF6.x in the same code file.

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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

CHAPTER 7

Database Providers

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.1 Microsoft SQL Server

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 EntityFramework GitHub project.

In this article:

7.1.1 Install

Install the Microsoft.EntityFrameworkCore.SqlServer NuGet package.

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

7.1.2 Get Started

The following resources will help you get started with this provider.

- *Getting Started on Full .NET (Console, WinForms, WPF, etc.)*
- *Getting Started on ASP.NET Core*
- UnicornStore Sample Application

7.1.3 Supported Database Engines

- Microsoft SQL Server (2008 onwards)

7.1.4 Supported Platforms

- Full .NET (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).

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.2 SQLite

This database provider allows Entity Framework Core to be used with SQLite. The provider is maintained as part of the EntityFramework GitHub project.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.2.1 SQLite Limitations

When using the SQLite provider, there are a number of limitations you should be aware of. Most of these 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 number 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?
AddColumn	
AddForeignKey	
AddPrimaryKey	
AddUniqueConstraint	
AlterColumn	
AlterSequence	
CreateIndex	
CreateSchema	
CreateSequence	
CreateTable	
DropColumn	
DropForiegnKey	
DropIndex	
DropPrimaryKey	
DropSchema	
DropSequence	
DropTable	
DropUniqueConstraint	
RenameColumn	
RenameIndex	
RenameSequence	
RenameTable	
RestartSequence	

Tip: 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.

In this article:

7.2.2 Install

Install the Microsoft.EntityFrameworkCore.SQLite NuGet package.

```
PM>  Install-Package Microsoft.EntityFrameworkCore.SQLite
```

7.2.3 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

7.2.4 Supported Database Engines

- SQLite (3.7 onwards)

7.2.5 Supported Platforms

- Full .NET (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)
- Universal Windows Platform

7.2.6 Limitations

See *SQLite Limitations* for some important limitations of the SQLite provider.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.3 Npgsql (PostgreSQL)

This database provider allows Entity Framework Core to be used with PostgreSQL. The provider is maintained as part of the Npgsql project.

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

In this article:

7.3.1 Install

Install the Npgsql.EntityFrameworkCore.PostgreSQL NuGet package.

```
PM>  Install-Package Npgsql.EntityFrameworkCore.PostgreSQL
```

7.3.2 Get Started

See the Npgsql documentation to get started.

7.3.3 Supported Database Engines

- PostgreSQL

7.3.4 Supported Platforms

- Full .NET (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.4 MySQL (Official)

This database provider allows Entity Framework Core to be used with MySQL. The provider is maintained as part of the MySQL project.

Caution: This provider is pre-release.

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

In this article:

7.4.1 Install

Install the MySql.Data.EntityFrameworkCore NuGet package.

```
PM>  Install-Package MySql.Data.EntityFrameworkCore -Pre
```

7.4.2 Get Started

See Starting with MySQL EF Core provider and Connector/Net 7.0.4.

7.4.3 Supported Database Engines

- MySQL

7.4.4 Supported Platforms

- Full .NET (4.5.1 onwards)
- .NET Core

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.5 Pomelo (MySQL)

This database provider allows Entity Framework Core to be used with MySQL. The provider is maintained as part of the Pomelo Foundation Project.

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

In this article:

7.5.1 Install

Install the Pomelo.EntityFrameworkCore.MySQL NuGet package.

```
PM>  Install-Package Pomelo.EntityFrameworkCore.MySQL
```

7.5.2 Get Started

The following resources will help you get started with this provider.

- Getting started documentation
- Yuuko Blog sample application

7.5.3 Supported Database Engines

- MySQL

7.5.4 Supported Platforms

- Full .NET (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.6 Microsoft SQL Server Compact Edition

This database provider allows Entity Framework Core to be used with SQL Server Compact Edition. The provider is maintained as part of the ErikEJ/EntityFramework.SqlServerCompact GitHub Project.

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

In this article:

7.6.1 Install

To work with SQL Server Compact Edition 4.0, install the EntityFrameworkCore.SqlServerCompact40 NuGet package.

```
PM>  Install-Package EntityFrameworkCore.SqlServerCompact40
```

To work with SQL Server Compact Edition 3.5, install the EntityFrameworkCore.SqlServerCompact35.

```
PM>  Install-Package EntityFrameworkCore.SqlServerCompact35
```

7.6.2 Get Started

See the getting started documentation on the project site

7.6.3 Supported Database Engines

- SQL Server Compact Edition 3.5
- SQL Server Compact Edition 4.0

7.6.4 Supported Platforms

- Full .NET (4.5.1 onwards)

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.7 IBM Data Servers

This database provider allows Entity Framework Core to be used with IBM Data Servers. Issues, questions, etc. can be posted in the .Net Development with DB2 and IDS forum

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

Caution: This provider currently only supports the Entity Framework Core RC1 pre-release.

In this article:

7.7.1 Install

Install the EntityFramework.IBMDataServer NuGet package.

```
PM>  Install-Package EntityFramework.IBMDataServer -Pre
```

7.7.2 Get Started

The following resources will help you get started with this provider.

- Sample application
- Updates & Limitations

7.7.3 Supported Database Engines

- IBM Data Servers

7.7.4 Supported Platforms

- Full .NET (4.5.1 onwards)

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.8 InMemory (for Testing)

This database provider allows Entity Framework Core to be used with an in-memory database. This is useful when testing code that uses Entity Framework Core. The provider is maintained as part of the EntityFramework GitHub project.

In this article:

7.8.1 Install

Install the Microsoft.EntityFrameworkCore.InMemory NuGet package.

```
PM>  Install-Package Microsoft.EntityFrameworkCore.InMemory
```

7.8.2 Get Started

The following resources will help you get started with this provider.

- *Testing with InMemory*
- UnicornStore Sample Application Tests

7.8.3 Supported Database Engines

- Built-in in-memory database (designed for testing purposes only)

7.8.4 Supported Platforms

- Full .NET (4.5.1 onwards)
- .NET Core
- Mono (4.2.0 onwards)
- Universal Windows Platform

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.9 Devart (MySQL, Oracle, PostgreSQL, SQLite, DB2, SQL Server, and more)

Devart is a third party provider writer that offers database providers for a wide range of databases. Find out more at devart.com/dotconnect.

Note: This provider is not maintained as part of the Entity Framework Core project. When considering a third party provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements.

In this article:

7.9.1 Paid Versions Only

Devart dotConnect is a commercial third party provider. Entity Framework support is only available in paid versions of dotConnect.

7.9.2 Install

See the Devart dotConnect documentation for installation instructions.

7.9.3 Get Started

See the Devart dotConnect Entity Framework documentation and blog article about Entity Framework Core 1 Support to get started.

7.9.4 Supported Database Engines

- MySQL
- Oracle
- PostgreSQL
- SQLite
- DB2
- SQL Server
- Cloud apps

7.9.5 Supported Platforms

- Full .NET (4.5.1 onwards)

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

7.10 Oracle (Coming Soon)

The Oracle .NET team is evaluating EF Core support, but have not announced any timing. You can vote on the Oracle EF Core Provider feature request.

Please direct any questions about this provider, including the release timeline, to the Oracle Community Site.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

Miscellaneous

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.1 Connection Strings

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.

In this article:

8.1.1 Full .NET Applications

Full .NET 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
<?xml version="1.0" encoding="utf-8"?>
<configuration>

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

Note: 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"].Connect
    }
}
```

8.1.2 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("Filename=Blogging.db");
    }
}
```

8.1.3 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;Truste
  },
}
```

The context is typically configured in `Startup.cs` with the connection string being read from configuration. Note the `GetConnectionString()` method simply looks for a configuration value whose key is `ConnectionStrings:<connection string name>`.

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

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.2 Logging

In this article:

Tip: You can view this article's sample on GitHub.

8.2.1 Create a logger

The first step is to create an implementation of `ILoggerProvider` and `ILogger`.

- `ILoggerProvider` is the component that decides when to create instances of your logger(s). The provider may choose to create different loggers in different situations.
- `ILogger` is the component that does the actual logging. It will be passed information from the framework when certain events occur.

Here is a simple implementation that logs a human readable representation of every event to a text file and the Console.

```
using Microsoft.Extensions.Logging;
using System;
using System.IO;

namespace EFLogging
{
    public class MyLoggerProvider : ILoggerProvider
    {
        public ILogger CreateLogger(string categoryName)
        {
            return new MyLogger();
        }

        public void Dispose()
        { }

        private class MyLogger : ILogger
        {
            public bool IsEnabled(LogLevel logLevel)
            {
                return true;
            }

            public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception excep
            {
                File.AppendAllText(@"C:\temp\log.txt", formatter(state, exception));
                Console.WriteLine(formatter(state, exception));
            }

            public IDisposable BeginScope<TState>(TState state)
            {
                return null;
            }
        }
    }
}
```

Tip:

The arguments passed to the Log method are:

- `logLevel` is the level (e.g. Warning, Info, Verbose, etc.) of the event being logged
- `eventId` is a library/assembly specific id that represents the type of event being logged
- `state` can be any object that holds state relevant to what is being logged
- `exception` gives you the exception that occurred if an error is being logged
- `formatter` uses state and exception to create a human readable string to be logged

8.2.2 Register your logger

ASP.NET Core

In an ASP.NET Core application, you register your logger in the Configure method of Startup.cs:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddProvider(new MyLoggerProvider());

    ...
}
```

Other applications

In your application startup code, create and instance of you context and register your logger.

Note: You only need to register the logger with a single context instance. Once you have registered it, it will be used for all other instances of the context in the same AppDomain.

```
            using (var db = new BloggingContext())
            {
                var serviceProvider = db.GetInfrastructure<IServiceProvider>();
                var loggerFactory = serviceProvider.GetService<ILoggerFactory>();
                loggerFactory.AddProvider(new MyLoggerProvider());
            }
```

8.2.3 Filtering what is logged

The easiest way to filter what is logged, is to adjust your logger provider to only return your logger for certain categories of events. For EF, the category passed to your logger provider will be the type name of the component that is logging the event.

For example, here is a logger provider that returns the logger only for events related to executing SQL against a relational database. For all other categories of events, a null logger (which does nothing) is returned.

```
using Microsoft.Extensions.Logging;
using System;
using System.Linq;

namespace EFLogging
{
    public class MyFilteredLoggerProvider : ILoggerProvider
    {
        private static string[] _categories =
        {
            typeof(Microsoft.EntityFrameworkCore.Storage.Internal.RelationalCommandBuilderFactory).Fu
            typeof(Microsoft.EntityFrameworkCore.Storage.Internal.SqlServerConnection).FullName
        };

        public ILogger CreateLogger(string categoryName)
        {
            if( _categories.Contains(categoryName))
            {
                return new MyLogger();
            }

            return new NullLogger();
        }

        public void Dispose()
        { }

        private class MyLogger : ILogger
        {
            public bool IsEnabled(LogLevel logLevel)
            {
                return true;
            }

            public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception excep
            {
                Console.WriteLine(formatter(state, exception));
            }

            public IDisposable BeginScope<TState>(TState state)
            {
                return null;
            }
        }

        private class NullLogger : ILogger
        {
            public bool IsEnabled(LogLevel logLevel)
            {
                return false;
            }

            public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception excep
            { }

            public IDisposable BeginScope<TState>(TState state)
            {
                return null;
            }
        }
    }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.3 Testing with InMemory

This article covers how to use the InMemory provider to write efficient tests with minimal impact to the code being tested.

Caution: Currently you need to use `ServiceCollection` and `IServiceProvider` to control the scope of the InMemory database, which adds complexity to your tests. In the next release after RC2, there will be improvements to make this easier, see issue #3253 for more details.

In this article:

Tip: You can view this article's sample on GitHub.

8.3.1 When to use InMemory for testing

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.

For example, 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.

```
    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();
        }
    }
```

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.

Tip: For many test purposes these difference 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.

8.3.2 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.

Note: 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.InMe
            }
```

Add a constructor for testing

The simplest way to enable testing with the InMemory provider 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)
        { }
```

Note: `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.

8.3.3 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.

`DbContextOptions<TContext>` exposes a `UseInternalServiceProvider` method that allows us to control the `IServiceProvider` the context will use. `IServiceProvider` is the container that EF will resolve all its services from (including the InMemory database instance). Typically, EF creates a single `IServiceProvider` for all contexts of a given type in an AppDomain - meaning all context instances share the same InMemory database instance. By allowing one to be passed in, you can control the scope of the InMemory database.

Here is an example of a test class that uses the InMemory database. Each test method creates a new `DbContextOptions<TContext>` with a new `IServiceProvider`, meaning each method has its own In-Memory database.

```
using BusinessLogic;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestProject
{
    [TestClass]
    public class BlogServiceTests
    {
        private static DbContextOptions<BloggingContext> CreateNewContextOptions()
        {
            // Create a fresh service provider, and therefore a fresh
            // InMemory database instance.
            var serviceProvider = new ServiceCollection()
```

```
                .AddEntityFrameworkInMemoryDatabase()
                .BuildServiceProvider();

            // Create a new options instance telling the context to use an
            // InMemory database and the new service provider.
            var builder = new DbContextOptionsBuilder<BloggingContext>();
            builder.UseInMemoryDatabase()
                   .UseInternalServiceProvider(serviceProvider);

            return builder.Options;
        }

        [TestMethod]
        public void Add_writes_to_database()
        {
            // All contexts that share the same service provider will share the same InMemory databa
            var options = CreateNewContextOptions();

            // 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()
        {
            // All contexts that share the same service provider will share the same InMemory databa
            var options = CreateNewContextOptions();

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

8.3.4 Sharing a database instance for read-only tests

If a test class has read-only tests that share the same seed data, then you can share the InMemory database instance for the whole class (rather than a new one for each method). This means you have a single `DbContextOptions<TContext>` and `IServiceProvider` for the test class, rather than one for each test method.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Extensions.DependencyInjection;
using BusinessLogic;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System;

namespace TestProject
{
    [TestClass]
    public class BlogServiceTestsReadOnly
    {
        private DbContextOptions<BloggingContext> _contextOptions;

        public BlogServiceTestsReadOnly()
        {
            // Create a service provider to be shared by all test methods
            var serviceProvider = new ServiceCollection()
                 .AddEntityFrameworkInMemoryDatabase()
                 .BuildServiceProvider();

            // Create options telling the context to use an
            // InMemory database and the service provider.
            var builder = new DbContextOptionsBuilder<BloggingContext>();
            builder.UseInMemoryDatabase()
                   .UseInternalServiceProvider(serviceProvider);

            _contextOptions = builder.Options;

            // Insert the seed data that is expected by all test methods
            using (var context = new BloggingContext(_contextOptions))
            {
                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();
            }
        }

        [TestMethod]
        public void Find_with_empty_term()
        {
            using (var context = new BloggingContext(_contextOptions))
            {
                var service = new BlogService(context);
                var result = service.Find("");
                Assert.AreEqual(3, result.Count());
            }
        }

        [TestMethod]
        public void Find_with_unmatched_term()
        {
            using (var context = new BloggingContext(_contextOptions))
            {
                var service = new BlogService(context);
                var result = service.Find("horse");
                Assert.AreEqual(0, result.Count());
            }
        }

        [TestMethod]
        public void Find_with_some_matched()
        {
            using (var context = new BloggingContext(_contextOptions))
            {
                var service = new BlogService(context);
                var result = service.Find("cat");
                Assert.AreEqual(2, result.Count());
            }
        }
    }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.4 Configuring a DbContext

This article shows patterns for configuring a `DbContext` with `DbContextOptions`. Options are primarily used to select and configure the data store.

In this article

8.4.1 Configuring DbContextOptions

`DbContext` must have an instance of `DbContextOptions` in order to execute. This can be supplied to `DbContext` in one of two ways.

1. *Constructor argument*
2. *OnConfiguring*

If both are used, "OnConfiguring" takes higher priority, which means it can overwrite or change options supplied by the constructor argument.

Constructor argument

Listing 8.1: 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`. Using the non-generic version is not recommended for applications with multiple context types.

Listing 8.2: Application code to initialize from constructor argument

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

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

OnConfiguring

Caution: `OnConfiguring` occurs last and can overwrite options obtained from DI or the constructor. This approach does not lend itself to testing (unless you target the full database).

Listing 8.3: Context code with OnConfiguring

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

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

Listing 8.4: Application code to initialize with “OnConfiguring”

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

8.4.2 Using DbContext with dependency injection

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

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

See *more reading* below for information on dependency injection.

Listing 8.5: Adding dbcontext to dependency injection

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

This requires adding a *constructor argument* to you DbContext type that accepts `DbContextOptions`.

Listing 8.6: Context code

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

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

Listing 8.7: Application code (in ASP.NET Core)

```
public MyController(BloggingContext context)
```

Listing 8.8: Application code (using ServiceProvider directly, less common)

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

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

8.4.3 Using `IDbContextFactory<TContext>`

As an alternative to the options above, you may also provide an implementation of `IDbContextFactory<TContext>`. EF command line tools and dependency injection can use this factory to create an instance of your DbContext. This may be required in order to enable specific design-time experiences such as migrations.

Implement this interface to enable design-time services for context types that do not have a public default constructor. Design-time services will automatically discover implementations of this interface that are in the same assembly as the derived context.

Example:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace MyProject
{
    public class BloggingContextFactory : IDbContextFactory<BloggingContext>
    {
        public BloggingContext Create()
        {
            var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
            optionsBuilder.UseSqlite("Filename=./blog.db");

            return new BloggingContext(optionsBuilder.Options);
        }
    }
}
```

8.4.4 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 with InMemory* for more information.
- Read *Understanding EF Services* for more details on how EF uses dependency injection internally.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.5 Upgrading from RC1 to RC2

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

In this article:

8.5.1 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 <to be advised>
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

8.5.2 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`.

8.5.3 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.

8.5.4 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)
{
}
```

8.5.5 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 with InMemory* 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)); );
```

Caution: 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

8.5.6 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.

8.5.7 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).

8.5.8 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-As
  - 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
  - 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.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.6 Upgrading from RC2 to RTM

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

In this article:

8.6.1 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 (e.g. `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.

8.6.2 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.

8.6.3 .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"]
    }
}
```

8.6.4 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>
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.7 Command Line Reference

Entity Framework provides command line tooling to automate common tasks such as code generation and database migrations.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.7.1 Package Manager Console (Visual Studio)

EF command line tools for Visual Studio's Package Manager Console (PMC) window.

In this article:

Caution: The commands require the latest version of Windows PowerShell

Installation

Package Manager Console commands are installed with the *Microsoft.EntityFrameworkCore.Tools* package.

To open the console, follow these steps.

- Open Visual Studio 2015
- *Tools* → *Nuget Package Manager* → *Package Manager Console*
- Execute `Install-Package Microsoft.EntityFrameworkCore.Tools -Pre`

.NET Core and ASP.NET Core Projects

.NET Core and ASP.NET Core projects also require installing .NET Core CLI. See *.NET Core CLI* for more information about this installation.

Note: .NET Core CLI has known issues in Preview 1. Because PMC commands call .NET Core CLI commands, these known issues also apply to PMC commands. See *Preview 2 Known Issues*.

Tip: On .NET Core and ASP.NET Core projects, add `-Verbose` to any Package Manager Console command to see the equivalent .NET Core CLI command that was invoked.

Usage

Note: All commands support the common parameters: `-Verbose`, `-Debug`, `-ErrorAction`, `-ErrorVariable`, `-WarningAction`, `-WarningVariable`, `-OutBuffer`, `-PipelineVariable`, and `-OutVariable`. For more information, see about_CommonParameters.

Add-Migration

Adds a new migration.

```
SYNTAX
    Add-Migration [-Name] <String> [-OutputDir <String>] [-Context <String>] [-Project <String>]
     [-StartupProject <String>] [-Environment <String>] [<CommonParameters>]

PARAMETERS
    -Name <String>
        Specifies the name of the migration.

    -OutputDir <String>
        The directory (and sub-namespace) to use. If omitted, "Migrations" is used. Relative paths ar

    -Context <String>
        Specifies the DbContext to use. If omitted, the default DbContext is used.

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.
    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.
```

Remove-Migration

Removes the last migration.

```
SYNTAX
    Remove-Migration [-Context <String>] [-Project <String>] [-StartupProject <String>] [-Environment
     [-Force] [<CommonParameters>]

PARAMETERS
    -Context <String>
        Specifies the DbContext to use. If omitted, the default DbContext is used.

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.

    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.

    -Force [<SwitchParameter>]
        Removes the last migration without checking the database. If the last migration has been appl
```

Scaffold-DbContext

Scaffolds a DbContext and entity type classes for a specified database.

```
SYNTAX
    Scaffold-DbContext [-Connection] <String> [-Provider] <String> [-OutputDir <String>] [-Context <S
      [-Schemas <String>] [-Tables <String>] [-DataAnnotations] [-Force] [-Project <String>]
      [-StartupProject <String>] [-Environment <String>] [<CommonParameters>]

PARAMETERS
    -Connection <String>
        Specifies the connection string of the database.

    -Provider <String>
        Specifies the provider to use. For example, Microsoft.EntityFrameworkCore.SqlServer.

    -OutputDir <String>
        Specifies the directory to use to output the classes. If omitted, the top-level project direc

    -Context <String>
        Specifies the name of the generated DbContext class.

    -Schemas <String>
        Specifies the schemas for which to generate classes.

    -Tables <String>
        Specifies the tables for which to generate classes.
    -DataAnnotations [<SwitchParameter>]
        Use DataAnnotation attributes to configure the model where possible. If omitted, the output c

    -Force [<SwitchParameter>]
        Force scaffolding to overwrite existing files. Otherwise, the code will only proceed if no ou

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.

    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.
```

Script-Migration

Generates a SQL script from migrations.

```
SYNTAX
    Script-Migration -From <String> -To <String> [-Idempotent] [-Context <String>] [-Project <String>
      [-StartupProject <String>] [-Environment <String>] [<CommonParameters>]

    Script-Migration [-From <String>] [-Idempotent] [-Context <String>] [-Project <String>]
      [-StartupProject <String>] [-Environment <String>] [<CommonParameters>]

PARAMETERS
    -From <String>
        Specifies the starting migration. If omitted, '0' (the initial database) is used.

    -To <String>
        Specifies the ending migration. If omitted, the last migration is used.

    -Idempotent [<SwitchParameter>]
        Generates an idempotent script that can be used on a database at any migration.

    -Context <String>
        Specifies the DbContext to use. If omitted, the default DbContext is used.

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.

    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.
```

Update-Database

Updates the database to a specified migration.

```
SYNTAX
    Update-Database [[-Migration] <String>] [-Context <String>] [-Project <String>] [-StartupProject
      [-Environment <String>] [<CommonParameters>]
```

```
PARAMETERS
    -Migration <String>
        Specifies the target migration. If '0', all migrations will be reverted. If omitted, all penc

    -Context <String>
        Specifies the DbContext to use. If omitted, the default DbContext is used.

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.

    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.
```

Use-DbContext

Sets the default DbContext to use.

```
SYNTAX
    Use-DbContext [-Context] <String> [-Project <String>] [-StartupProject <String>] [-Environment <S
      [<CommonParameters>]

PARAMETERS
    -Context <String>
        Specifies the DbContext to use.

    -Project <String>
        Specifies the project to use. If omitted, the default project is used.

    -StartupProject <String>
        Specifies the startup project to use. If omitted, the solution's startup project is used.

    -Environment <String>
        Specifies the environment to use. If omitted, "Development" is used.
```

Using EF Core commands and EF 6 commands side-by-side

EF Core commands do not work on EF 6 or earlier version of EF. However, EF Core re-uses some of the same command names from these earlier versions. These commands can be installed side-by-side, however, EF does not automatically know which version of the command to use. This is solved by prefixing the command with the module name. The EF 6 commands PowerShell module is named “EntityFramework”, and the EF Core module is named “EntityFrameworkCore”. Without the prefix, PowerShell may call the wrong version of the command.

```
# Invokes the EF Core command
PS> EntityFrameworkCore\Add-Migration

# Invokes the EF 6 command
PS> EntityFramework\Add-Migration
```

Common Errors

Error: "No parameterless constructor was found"

Design-time tools attempt to automatically find how your application creates instances of your DbContext type. If EF cannot find a suitable way to initialize your DbContext, you may encounter this error.

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

As the error message suggests, one solution is to add an implementation of `IDbContextFactory<TContext>` to the current project. See *Using IDbContextFactory<TContext>* for an example of how to create this factory.

See also *Preview 2 Known Issues* for .NET Core CLI commands.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.7.2 .NET Core CLI

EF command-line tools for .NET Core Command Line Interface (CLI).

In this article:

Note: Command-line tools for .NET Core CLI has known issues. See *Preview 2 Known Issues* for more details.

Installation

Prerequisites

EF command-line tools requires .NET Core CLI Preview 2 or newer. See the .NET Core website for installation instructions.

Supported Frameworks

EF supports .NET Core CLI commands on these frameworks:

- .NET Framework 4.5.1 and newer. ("net451", "net452", "net46", etc.)
- .NET Core App 1.0. ("netcoreapp1.0")

Install by editing project.json

EF command-line tools for .NET Core CLI are installed by manually editing `project.json`.

1. Add `Microsoft.EntityFrameworkCore.Tools` as a "tool" and `Microsoft.EntityFrameworkCore.Design` as a build-only dependency under "dependencies". See sample project.json below.
2. Execute `dotnet restore`. If restore does not succeed, the command-line tools may not have installed correctly.

The resulting project.json should include these items (in addition to your other project dependencies).

```
{
    "dependencies": {
        "Microsoft.EntityFrameworkCore.Design": {
            "type": "build",
            "version": "1.0.0-preview2-final"
        }
    },

    "tools": {
        "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
    },

    "frameworks": {
        "netcoreapp1.0": { }
    }
}
```

Tip: A build-only dependency (`"type": "build"`) means this dependency is local to the current project. For example, if Project A has a build only dependency and Project B depends on A, `dotnet restore` will not add A's build-only dependencies into Project B.

Usage

Commands can be run from the command line by navigating to the project directory and executing `dotnet ef [subcommand]`. To see usage, add `--help` to any command to see more information about parameters and sub-commands.

dotnet-ef

```
Usage: dotnet ef [options] [command]

Options:
  -h|--help                            Show help information
  -p|--project <PROJECT>               The project to target (defaults to the project in the current d
  -s|--startup-project <PROJECT>       The path to the project containing Startup (defaults to the tar
  -c|--configuration <CONFIGURATION>  Configuration under which to load (defaults to Debug)
  -f|--framework <FRAMEWORK>           Target framework to load from the startup project (defaults to
  -b|--build-base-path <OUTPUT_DIR>   Directory in which to find temporary outputs.
  -o|--output <OUTPUT_DIR>             Directory in which to find outputs

Commands:
  database    Commands to manage your database
  dbcontext   Commands to manage your DbContext types
  migrations  Commands to manage your migrations
```

dotnet-ef-database

```
Usage: dotnet ef database [options] [command]

Options:
  -h|--help     Show help information
  -v|--verbose  Enable verbose output

Commands:
  drop    Drop the database for specific environment
  update  Updates the database to a specified migration
```

dotnet-ef-database-drop

```
Usage: dotnet ef database drop [options]

Options:
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  -c|--context <context>          The DbContext to use. If omitted, the default DbContext is used
  -f|--force                      Drop without confirmation
  -h|--help                       Show help information
  -v|--verbose                    Enable verbose output
```

dotnet-ef-database-update

```
Usage: dotnet ef database update [arguments] [options]

Arguments:
  [migration]  The target migration. If '0', all migrations will be reverted. If omitted, all pending

Options:
  -c|--context <context>          The DbContext to use. If omitted, the default DbContext is used
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  -h|--help                       Show help information
  -v|--verbose                    Enable verbose output
```

dotnet-ef-dbcontext

```
Usage: dotnet ef dbcontext [options] [command]

Options:
  -h|--help     Show help information
  -v|--verbose  Enable verbose output

Commands:
  list      List your DbContext types
  scaffold  Scaffolds a DbContext and entity type classes for a specified database
```

dotnet-ef-dbcontext-list

```
Usage: dotnet ef dbcontext list [options]

Options:
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  --json                          Use json output. JSON is wrapped by '//BEGIN' and '//END'
  -h|--help                       Show help information
  -v|--verbose                    Enable verbose output
```

dotnet-ef-dbcontext-scaffold

```
Usage: dotnet ef dbcontext scaffold [arguments] [options]

Arguments:
  [connection]  The connection string of the database
  [provider]    The provider to use. For example, Microsoft.EntityFrameworkCore.SqlServer

Options:
  -a|--data-annotations           Use DataAnnotation attributes to configure the model where possible
  -c|--context <name>             Name of the generated DbContext class.
  -f|--force                      Force scaffolding to overwrite existing files. Otherwise, the code
  -o|--output-dir <path>          Directory of the project where the classes should be output. If omi
  --schema <schema>               Selects a schema for which to generate classes.
  -t|--table <schema.table>       Selects a table for which to generate classes.
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  -h|--help                       Show help information
  -v|--verbose                    Enable verbose output
```

dotnet-ef-migrations

```
Usage: dotnet ef migrations [options] [command]

Options:
  -h|--help     Show help information
  -v|--verbose  Enable verbose output

Commands:
  add     Add a new migration
  list    List the migrations
  remove   Remove the last migration
  script   Generate a SQL script from migrations
```

dotnet-ef-migrations-add

```
Usage: dotnet ef migrations add [arguments] [options]

Arguments:
  [name]  The name of the migration

Options:
  -o|--output-dir <path>           The directory (and sub-namespace) to use. If omitted, "Migrations"
  -c|--context <context>           The DbContext to use. If omitted, the default DbContext is used
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  --json                           Use json output. JSON is wrapped by '//BEGIN' and '//END'
  -h|--help                        Show help information
  -v|--verbose                     Enable verbose output
```

dotnet-ef-migrations-list

```
Usage: dotnet ef migrations list [options]

Options:
  -c|--context <context>           The DbContext to use. If omitted, the default DbContext is used
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  --json                           Use json output. JSON is wrapped by '//BEGIN' and '//END'
  -h|--help                        Show help information
  -v|--verbose                     Enable verbose output
```

dotnet-ef-migrations-remove

```
Usage: dotnet ef migrations remove [options]

Options:
  -c|--context <context>           The DbContext to use. If omitted, the default DbContext is used
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  -f|--force                       Removes the last migration without checking the database. If the la
  -h|--help                        Show help information
  -v|--verbose                     Enable verbose output
```

dotnet-ef-migrations-script

```
Usage: dotnet ef migrations script [arguments] [options]

Arguments:
  [from]  The starting migration. If omitted, '0' (the initial database) is used
  [to]    The ending migration. If omitted, the last migration is used

Options:
  -o|--output <file>               The file to write the script to instead of stdout
  -i|--idempotent                  Generates an idempotent script that can used on a database at any m
  -c|--context <context>          The DbContext to use. If omitted, the default DbContext is used
  -e|--environment <environment>  The environment to use. If omitted, "Development" is used.
  -h|--help                       Show help information
  -v|--verbose                    Enable verbose output
```

Common Errors

Error: "No parameterless constructor was found"

Design-time tools attempt to automatically find how your application creates instances of your DbContext type. If EF cannot find a suitable way to initialize your DbContext, you may encounter this error.

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

As the error message suggests, one solution is to add an implementation of `IDbContextFactory<TContext>` to the current project. See *Using IDbContextFactory<TContext>* for an example of how to create this factory.

Preview 2 Known Issues

Targeting class library projects is not supported

.NET Core CLI does not support running commands on class libraries as of Preview 2. Despite being able to install EF tools, executing commands may throw this error message.

```
Could not invoke this command on the startup project '(your project name)'. This preview of Entity Fr
```

See issue https://github.com/dotnet/cli/issues/2645.

Explanation If `dotnet run` does not work in the startup project, then `dotnet ef` cannot run either.

"dotnet ef" is invoked as an alternate entry point into an application. If the "startup" project is not an application, then it is not currently possible to run the project as an application with the "dotnet ef" alternate entry point.

The "startup" project defaults to the current project, unless specified differently with the parameter `--startup-project`.

Workaround 1 - Utilize a separate startup project Convert the class library project into an "app" project. This can either be a .NET Core app or a desktop .NET app.

Example:

```
{
    "frameworks": {
        "netcoreapp1.0": {
            "dependencies": {
                "Microsoft.NETCore.App": {
                    "type": "platform",
                    "version": "1.0.0-*"
                }
            }
        }
    }
}
```

Be sure to register the EntityFramework Tools as a project dependency and in the tools section of your project.json.

Example:

```
{
    "dependencies": {
        "Microsoft.EntityFrameworkCore.Tools": {
            "version": "1.0.0-preview2-final",
            "type": "build"
        }
    },
    "tools": {
        "Microsoft.EntityFrameworkCore.Tools": "1.0.0-preview2-final"
    }
}
```

Finally, specify a startup project that is a "runnable app."

Example:

```
dotnet ef --startup-project ../MyConsoleApplication/ migrations list
```

Workaround 2 - Modify your class library to be a startup application Convert the class library project into an "app" project. This can either be a .NET Core app or a desktop .NET app.

To make the project a .NET Core App, add the "netcoreapp1.0" framework to project.json along with the other settings in the sample below:

```
{
    "buildOptions": {
        "emitEntryPoint": true
    },
    "frameworks": {
        "netcoreapp1.0": {
            "dependencies": {
                "Microsoft.NETCore.App": {
                    "type": "platform",
                    "version": "1.0.0-*"
                }
            }
        }
    }
}
```

To make a desktop .NET app, ensure you project targets "net451" or newer (example "net461" also works) and ensure the build option `"emitEntryPoint"` is set to true.

```
{
    "buildOptions": {
        "emitEntryPoint": true
    },
    "frameworks": {
        "net451": { }
    }
}
```

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.8 Internals

Caution: These articles are advanced topics. The target audience is provider writers and EF contributors.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.8.1 Writing a Database Provider

In this article

EF Core is designed to be extensible. It provides general purpose building blocks that are intended for use in multiple providers. The purpose of this article is to provide basic guidance on creating a new provider that is compatible with EF Core.

Tip: EF Core source code is open-source. The best source of information is the code itself.

Tip: This article shows snippets from an empty EF provider. You can view the full stubbed-out provider on GitHub.

DbContext Initialization

A user's interaction with EF begins with the `DbContext` constructor. Before the context is available for use, it initializes **options** and **services**. We will example both of these to understand what they represent and how EF configures itself to use different providers.

Options

`Microsoft.EntityFrameworkCore.Infrastructure.DbContextOptions` is the API surface for **users** to configure `DbContext`. Provider writers are responsible for creating API to configure options and to make services responsive to these options. For example, most providers require a connection string. These options are typically created using `DbContextOptionsBuilder`.

Services

`System.IServiceProvider` is the main interface used for interaction with services. EF makes heavy use of dependency injection (DI). The `ServiceProvider` contains a collection of services available for injection. Initialization uses `DbContextOptions` to add additional services if needed and select a scoped set of services that all EF operations will use during execution.

See also *Understanding EF Services*.

Note: EF uses Microsoft.Extensions.DependencyInjection to implement dependency injection. Documentation for this project is available on docs.asp.net.

Plugging in a Provider

As explained above, EF uses options and services. Each provider must create API so users to add provider-specific options and services. This API is best created by using extension methods.

Tip: When defining an extension method, define it in the namespace of the object being extended so Visual Studio auto-complete will include the extension method as a possible completion.

The *Use* Method

By convention, providers define a `UseX()` extension on `DbContextOptionsBuilder`. This configures **options** which it typically takes as arguments to method.

```
optionsBuilder.UseMyProvider("Server=contoso.com")
```

The `UseX()` extension method creates a provider-specific implementation of `IDbContextOptionsExtension` which is added to the collection of extensions stored within `DbContextOptions`. This is done by a call to the API `IDbContextOptionsBuilderInfrastructure.AddOrUpdateExtension`.

Listing 8.9: An example implementation of the "Use" method

```
    public static class MyProviderDbContextOptionsExtensions
    {
        public static DbContextOptionsBuilder UseMyProvider(this DbContextOptionsBuilder optionsBuil
            string connectionString)
        {
            ((IDbContextOptionsBuilderInfrastructure) optionsBuilder).AddOrUpdateExtension(
                new MyProviderOptionsExtension
                {
                    ConnectionString = connectionString
                });

            return optionsBuilder;
        }
    }
```

Tip: The `UseX()` method can also be used to return a special wrapper around `DbContextOptionsBuilder` that allows users to configure multiple options with chained calls. See `SqlServerDbContextOptionsBuilder` as an example.

The *Add* Method

By convention, providers define a `AddX()` extension on `EntityFrameworkServicesBuilder`. This configures **services** and does not take arguments.

`EntityFrameworkServicesBuilder` is a wrapper around `ServiceCollection` which is accessible by calling `GetInfrastructure()`. The `AddX()` method should register services in this collection to be available for dependency injection.

In some cases, users may call the *Add* method directly. This is done when users are configuring a service provider manually and use this service provider to resolve an instance of `DbContext`. In other cases, the *Add* method is called by EF upon service initialization. For more details on service initialization, see *Understanding EF Services*.

A provider *must register* an implementation of `IDatabaseProvider`. Implementing this in-turn requires configuring several more required services. Read more about working with services in *Understanding EF Services*.

EF provides many complete or partial implementations of the required services to make it easier for provider-writers. For example, EF includes a class `DatabaseProvider<TProviderServices, TOptionsExtension>` which can be used in service registration to hook up a provider.

Listing 8.10: An example implementation of the "Add" method

```
    public static class MyProviderServiceCollectionExtensions
    {
        public static IServiceCollection AddEntityFrameworkMyProvider(this IServiceCollection service
        {
            services.AddEntityFramework();

            services.TryAddEnumerable(ServiceDescriptor
                .Singleton<IDatabaseProvider, DatabaseProvider<MyDatabaseProviderServices, MyProvide

            services.TryAdd(new ServiceCollection()
                // singleton services
                .AddSingleton<MyModelSource>()
                .AddSingleton<MyValueGeneratorCache>()
                // scoped services
                .AddScoped<MyDatabaseProviderServices>()
                .AddScoped<MyDatabaseCreator>()
                .AddScoped<MyDatabase>()
                .AddScoped<MyEntityQueryableExpressionVisitorFactory>()
                .AddScoped<MyEntityQueryModelVisitorFactory>()
                .AddScoped<MyQueryContextFactory>()
                .AddScoped<MyTransactionManager>());

            return services;
        }
    }
```

Next Steps

With these two extensibility APIs now defined, users can now configure their "DbContext" to use your provider. To make your provider functional, you will need to implement other services.

Reading the source code of other providers is an excellent way to learn how to create a new EF provider. See *Database Providers* for a list of current EF providers and to find links to their source code (if applicable).

`Microsoft.EntityFrameworkCore.Relational` includes an extensive library of services designed for relational providers. In many cases, these services need little or no modification to work for multiple relational databases.

For more information on other internal parts of EF, see *Internals*.

Caution: This documentation is for EF Core. For EF6.x and earlier release see http://msdn.com/data/ef.

8.8.2 Understanding EF Services

In this article

Entity Framework executes as a collection of services working together. A service is a reusable component. A service is typically an implementation of an interface. Services are available to other services via dependency injection (DI), which is implemented in EF using Microsoft.Extensions.DependencyInjection.

This article covers some fundamentals principles for understanding how EF uses services and DI.

Terms

Service A reusable component. In .NET, a service can be identified by a class or interface. By convention, Entity Framework only uses interfaces to identify services.

Service lifetime A description of the way in which a service is persisted and disposed across multiple uses of the same service type.

Service provider The mechanism for storing a collection of services. Also known as a service container.

Service collection The mechanism for constructing a service provider.

Categories of Services

Services fall into one or more categories.

Context services Services that are related to a specific instance of `DbContext`. They provide functionality for working with the user model and context options.

Provider services Provider-specific implementations of services. For example, SQLite uses "provider services" to customize the behavior of SQL generation, migrations, and file I/O.

Design-time services Services used when a developer is creating an application. For example, EF commands uses design-time services to execute migrations and code generation (aka scaffolding).

User services A user can define custom services to interact with EF. These are written in application code, not provider code. For example, users can provide an implementation of `IModelCustomizer` for controlling how a model is created.

Note: Service provider is not to be confused with a "provider's services".

Service Lifetime

EF services can be registered with different lifetime options. The suitable option depends on how the service is used and implemented.

Transient Transient lifetime services are created each time they are injected into other services. This isolates each instance of the service. For example, `MigrationsScaffolder` should not be reused, therefore it is registered as transient.

Scoped Scoped lifetime services are created once per `DbContext` instance. This is used to isolate instance of `DbContext`. For example, `StateManager` is added as scoped because it should only track entity states for one context.

Singleton Singleton lifetime services exists once per service provider and span all scopes. Each time the service is injected, the same instance is used. For example, `IModelCustomizer` is a singleton because it is idempotent, meaning each call to `IModelCustomizer.Customize()` does not change the customizer.

How AddDbContext works

EF provides an extension method `AddDbContext<TContext>()` for adding using EF into a service collection. This method adds the following into a service collection:

- `TContext` as "scoped"
- `DbContextOptions` as a "singleton"
- `DbContextOptionsFactory<T>` as a "singleton"

`AddDbContext` does not add any context services, provider services, or design-time services to the service collection (except for *special cases*). DbContext constructs its own internal service provider for this.

Special cases

`AddDbContext` adds `DbContextOptionsFactory<T>` to the service collection AddDbContext was called on (which is used to create the "external" service provider). `DbContextOptionsFactory<T>` acts as a bridge between the external service provider and DbContext's internal service provider. If the external provider has services for `ILoggerFactory` or `IMemoryCache`, these will be added to the internal service provider.

The bridging is done for these common scenarios so users can easily configure logging and memory caching without needing to provide a custom internal service provider.

DbContext's internal service provider

By default, `DbContext` uses an internal service provider that is **separate** from all other service providers in the application. This internal provider is constructed from an instance of `DbContextOptions`. Methods such as `UseSqlServer()` extend the construction step add specialized services for their database system.

Providing a custom internal service provider

`DbContextOptionsBuilder` provides an API for giving a custom service provider to DbContext for EF to use internally. This API is `DbContextOptions.UseInternalServiceProvider(IServiceProvider provider)`.

If a custom service provider is provided, DbContext will not use `DbContextOptions` to create its own internal service provider. The custom service provider must already have provider-specific services added.

Database provider writers should provided methods such as AddEntityFrameworkSqlServer" or "AddEntityFrameworkSqlite" to simplify the process of creating a custom service container.

```
var services = new ServiceCollection()
    .AddEntityFrameworkSqlServer()
    .AddSingleton<MyCustomService>()
    .BuildServiceProvider();

var options = new DbContextOptionsBuilder();

options
    .UseInternalServiceProvider(services)
    .UseSqlServer(connectionString);

using (var context = new DbContext(options))
{ }
```

Service provider caching

EF caches this internal service provider with `IDbContextOptions` as the key. This means the service provider is only created once per unique set of options. It is reused when a DbContext is instantiated using a set of options that have already been used during the application lifetime.

Required Provider Services

EF database providers must register a basic set of services. These required services are defined as properties on `IDatabaseProviderServices`. Provider writers may need to implement some services from scratch. Others have partial or complete implementations in EF's library that can be reused.

For more information on required provider services, see *Writing a Database Provider*.

Additional Information

EF uses ' the Microsoft.Extensions.DependencyInjection library <https://www.nuget.org/packages/Microsoft.Extensions.DependencyInjection/>'_ to implement DI. Documentation for this library is available on docs.asp.net.

"System.IServiceProvider" is defined in the .NET base class library.

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

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

If you like to learn by writing code, we'd recommend one of our *Getting Started* guides to get you started with EF Core.

CHAPTER 9

Get Entity Framework Core

Install the NuGet package for the database provider you want to use. See *Database Providers* for information.

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

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_
        }
    }

    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; }
    }
}
```

CHAPTER 11

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();
}
```