
Entity Framework Documentation

7.0.0

Microsoft

9/19/2017

Contents

1	Available Tutorials	3
1.1	Getting Started on Full .NET (Console, WinForms, WPF, etc.)	3
1.2	Getting Started on ASP.NET 5	12
1.3	Getting Started on Universal Windows Platform	29
1.4	Getting Started on OSX	36
1.5	Getting Started on Linux	41
2	Database Providers	49
2.1	EntityFramework.MicrosoftSqlServer	49
2.2	EntityFramework.SQLite	50
2.3	EntityFramework.InMemory	50
2.4	EntityFramework.SqlServerCompact40	50
2.5	EntityFramework.SqlServerCompact35	50
2.6	EntityFramework.Npgsql	51
3	Modeling	53
3.1	Including & Excluding Types	53
3.2	Including & Excluding Properties	56
3.3	Keys (primary)	57
3.4	Generated Properties	59
3.5	Required/optional properties	63
3.6	Maximum Length	64
3.7	Concurrency Tokens	66
3.8	Shadow Properties	68
3.9	Relationships	70
3.10	Indexes	83
3.11	Alternate Keys	84
3.12	Inheritance	87
3.13	Relational Database Modeling	88
3.14	Methods of configuration	103
4	Contribute	105

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

Entity Framework Documentation Entity Framework =====

: This documentation is a work in progress. Topics marked with a are placeholders that have not been written yet. You can track the status of these topics through our public documentation [issue tracker](#). Learn how you can [contribute](#) on GitHub.

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

These 101 tutorials require no previous knowledge of Entity Framework (EF). 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 in a variety of different .NET applications. The following tutorials will get you started on the platform where you want to use EF.

CHAPTER 1

Available Tutorials

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

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.

Available Tutorials

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Prerequisites*
 - *Latest version of NuGet Package Manager*
- *Create a new project*
- *Install Entity Framework*
- *Create your model*
- *Create your database*
- *Use your model*

: You can view this article's [sample](#) on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2013 or Visual Studio 2015
- [Latest version of NuGet Package Manager](#)
- Latest version of Windows PowerShell

Latest version of NuGet Package Manager

Installing EF7 requires an up-to-date version of NuGet Package Manager. You can install the latest version from Visual Studio Gallery. Make sure you restart Visual Studio after installing the update.

- [NuGet Package Manager for Visual Studio 2015](#)
- [NuGet Package Manager for Visual Studio 2013](#)

Create a new project

- Open Visual Studio (this walkthrough uses 2015 but you can use any version from 2013 onwards)
- *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 EF7 you 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 EntityFramework.MicrosoftSqlServer -Pre`

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 EntityFramework.Commands -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

: Notice the `OnConfiguring` method (new in EF7) that is used to specify the provider to use and, optionally, other configuration too.

```

1 using Microsoft.Data.Entity;
2 using System.Collections.Generic;
3
4 namespace EFGetStarted.ConsoleApp
5 {
6     public class BloggingContext : DbContext
7     {
8         public DbSet<Blog> Blogs { get; set; }
9         public DbSet<Post> Posts { get; set; }
10
11         protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
12         {
13             // Visual Studio 2015 | Use the LocalDb 12 instance created by Visual_
14             ↩Studio
15             optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;
16             ↩Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;");
17
18             // Visual Studio 2013 | Use the LocalDb 11 instance created by Visual_
19             ↩Studio
20             // optionsBuilder.UseSqlServer(@"Server=(localdb)\v11.0;
21             ↩Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;");
22         }
23
24         protected override void OnModelCreating(ModelBuilder modelBuilder)
25         {
26             // Make Blog.Url required
27             modelBuilder.Entity<Blog>()
28                 .Property(b => b.Url)
29                 .IsRequired();
30         }
31     }
32
33     public class Blog
34     {
35         public int BlogId { get; set; }
36     }
37 }

```

```
32     public string Url { get; set; }
33
34     public List<Post> Posts { get; set; }
35 }
36
37 public class Post
38 {
39     public int PostId { get; set; }
40     public string Title { get; set; }
41     public string Content { get; set; }
42
43     public int BlogId { get; set; }
44     public Blog Blog { get; set; }
45 }
46 }
```

: 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.

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.

: 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

```
1 using System;
2
3 namespace EFGetStarted.ConsoleApp
4 {
5     class Program
6     {
7         static void Main(string[] args)
```

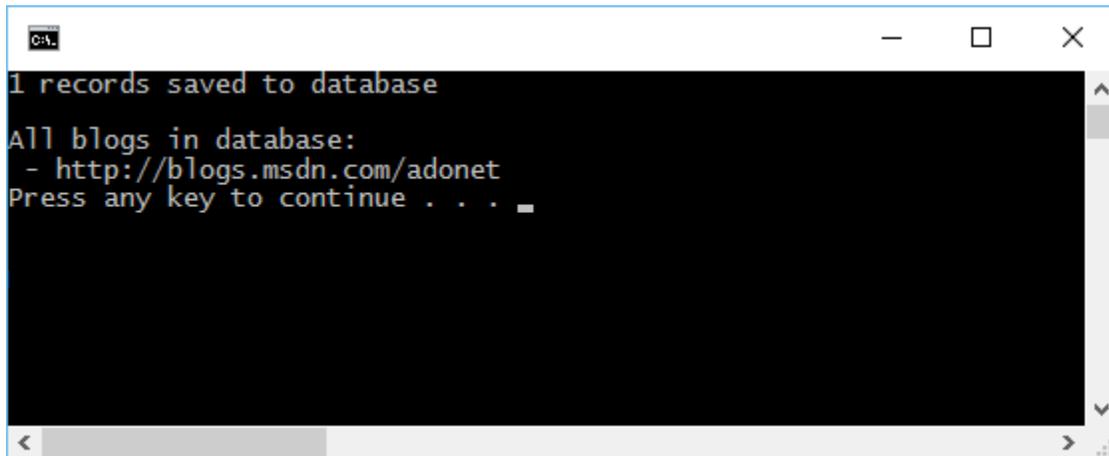
```

8      {
9          using (var db = new BloggingContext())
10         {
11             db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
12             var count = db.SaveChanges();
13             Console.WriteLine("{0} records saved to database", count);
14
15             Console.WriteLine();
16             Console.WriteLine("All blogs in database:");
17             foreach (var blog in db.Blogs)
18             {
19                 Console.WriteLine(" - {0}", blog.Url);
20             }
21         }
22     }
23 }
24

```

- *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.



```

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

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Prerequisites*
 - *Latest version of NuGet Package Manager*
 - *Blogging database*
- *Create a new project*
- *Install Entity Framework*
- *Reverse engineer your model*
 - *Entity Classes*
 - *Derived Context*
- *Use your model*

: You can view this article's [sample](#) on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2013 or Visual Studio 2015
- *Latest version of NuGet Package Manager*
- Latest version of Windows PowerShell
- *Blogging database*

Latest version of NuGet Package Manager

Installing EF7 requires an up-to-date version of NuGet Package Manager. You can install the latest version from Visual Studio Gallery. Make sure you restart Visual Studio after installing the update.

- NuGet Package Manager for Visual Studio 2015
- NuGet Package Manager for Visual Studio 2013

Blogging database

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

: If you have already created the **Blogging** database as part of another tutorial, you can skip these steps.

- *Tools* → *Connect to Database...*
- Select **Microsoft SQL Server** and click **Continue**
- Enter **(localdb)\mssqllocaldb** as the **Server Name**
- Enter **master** as the **Database Name**

- 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**

```

1 CREATE DATABASE [Bloggng]
2 GO
3
4 USE [Bloggng]
5 GO
6
7 CREATE TABLE [Blog] (
8     [BlogId] int NOT NULL IDENTITY,
9     [Url] nvarchar(max) NOT NULL,
10    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
11 );
12 GO
13
14 CREATE TABLE [Post] (
15     [PostId] int NOT NULL IDENTITY,
16     [BlogId] int NOT NULL,
17     [Content] nvarchar(max),
18     [Title] nvarchar(max),
19     CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
20     CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog]
21     ([BlogId]) ON DELETE CASCADE
22 );
23 GO
24 INSERT INTO [Blog] ([Url]) VALUES
25 ('http://blogs.msdn.com/dotnet'),
26 ('http://blogs.msdn.com/webdev'),
27 ('http://blogs.msdn.com/visualstudio')
28 GO

```

Create a new project

- Open Visual Studio (this walkthrough uses 2015 but you can use any version from 2013 onwards)
- *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 EF7 you 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 EntityFramework.MicrosoftSqlServer -Pre`

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

- Run `Install-Package EntityFramework.Commands -Pre`
- Run `Install-Package EntityFramework.MicrosoftSqlServer.Design -Pre`

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 -provider EntityFramework.MicrosoftSqlServer -connection  
↔"Server=(localdb)\mssqllocaldb;Database=Bloggging;Trusted_Connection=True;"
```

The reverse engineer process created entity classes and a derived context based on the schema of the existing database.

Entity Classes

The entity classes are simple C# objects that represent the data you will be querying and saving.

```
1 using System;  
2 using System.Collections.Generic;  
3  
4 namespace EFGetStarted.ConsoleApp.ExistingDb  
5 {  
6     public partial class Blog  
7     {  
8         public Blog()  
9         {  
10            Post = new HashSet<Post>();  
11        }  
12  
13        public int BlogId { get; set; }  
14        public string Url { get; set; }  
15  
16        public virtual ICollection<Post> Post { get; set; }  
17    }  
18 }
```

Derived Context

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

: Notice the `OnConfiguring` method (new in EF7) that is used to specify the provider to use and, optionally, other configuration too.

```
1 using Microsoft.Data.Entity;  
2 using Microsoft.Data.Entity.Metadata;  
3
```

```

4 namespace EFGetStarted.ConsoleApp.ExistingDb
5 {
6     public partial class BloggingContext : DbContext
7     {
8         protected override void OnConfiguring(DbContextOptionsBuilder options)
9         {
10            options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;
↵Trusted_Connection=True;");
11        }
12
13        protected override void OnModelCreating(ModelBuilder modelBuilder)
14        {
15            modelBuilder.Entity<Blog>(entity =>
16            {
17                entity.Property(e => e.Url).IsRequired();
18            });
19
20            modelBuilder.Entity<Post>(entity =>
21            {
22                entity.HasOne(d => d.Blog).WithMany(p => p.Post).HasForeignKey(d => d.
↵BlogId);
23            });
24        }
25
26        public virtual DbSet<Blog> Blog { get; set; }
27        public virtual DbSet<Post> Post { get; set; }
28    }
29 }

```

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

```

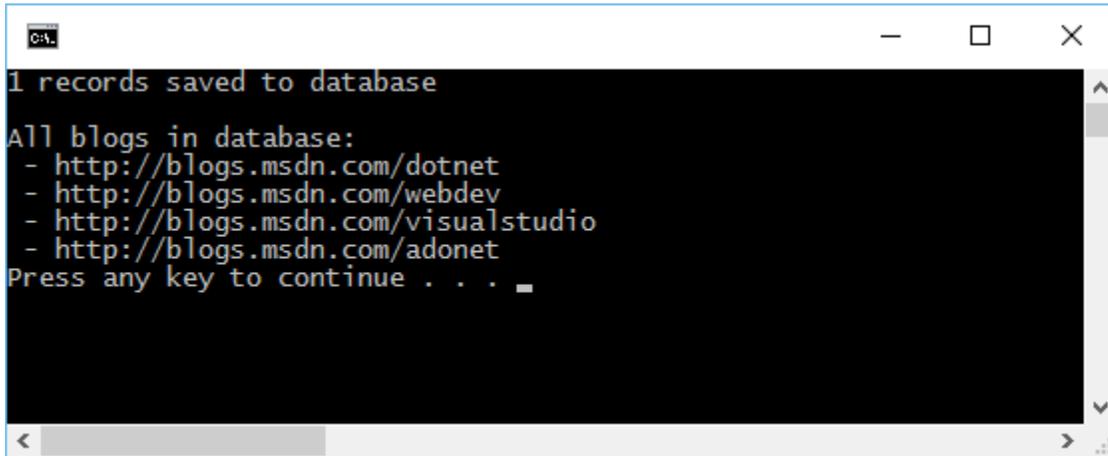
1 using System;
2
3 namespace EFGetStarted.ConsoleApp.ExistingDb
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             using (var db = new BloggingContext())
10            {
11                db.Blog.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
12                var count = db.SaveChanges();
13                Console.WriteLine("{0} records saved to database", count);
14
15                Console.WriteLine();
16                Console.WriteLine("All blogs in database:");
17                foreach (var blog in db.Blog)
18                {
19                    Console.WriteLine(" - {0}", blog.Url);
20                }

```

```
21     }
22   }
23 }
24 }
```

- *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 . . . .
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

Getting Started on ASP.NET 5

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 5 application.

Available Tutorials

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

ASP.NET 5 Application to New Database

In this walkthrough, you will build an ASP.NET 5 MVC application that performs basic data access using Entity Framework. You will use migrations to create the database from your model.

In this article:

- *Prerequisites*
- *Create a new project*
- *Install Entity Framework*
- *Create your model*
- *Register your context with dependency injection*
- *Create your database*
- *Create a controller*
- *Create views*
- *Run the application*

: You can view this article's [sample](#) on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015
- [ASP.NET 5 RC1 Tools 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 Web Application** project template
- Ensure you are targeting .NET 4.5.1 or later
- Enter **EFGetStarted.AspNet5.NewDb** as the name and click **OK**
- Wait for the **New ASP.NET Project** dialog to appear
- Under **ASP.NET 5 Preview Templates** select **Web Application**
- Ensure that **Authentication** is set to **No Authentication**
- Click **OK**

: 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 EF7 you 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 EntityFramework.MicrosoftSqlServer -Pre`

: In ASP.NET 5 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.

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 EntityFramework.Commands -Pre`
- Open **project.json**
- Locate the `commands` section and add the `ef` command as shown below

```
1 "commands": {
2   "web": "Microsoft.AspNet.Server.Kestrel",
3   "ef": "EntityFramework.Commands"
4 },
5
```

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

```
1 using Microsoft.Data.Entity;
2 using System.Collections.Generic;
3
4 namespace EFGetStarted.AspNet5.NewDb.Models
5 {
6     public class BloggingContext : DbContext
7     {
```

```

8     public DbSet<Blog> Blogs { get; set; }
9     public DbSet<Post> Posts { get; set; }
10
11    protected override void OnModelCreating(ModelBuilder modelBuilder)
12    {
13        // Make Blog.Url required
14        modelBuilder.Entity<Blog>()
15            .Property(b => b.Url)
16            .IsRequired();
17    }
18 }
19
20 public class Blog
21 {
22     public int BlogId { get; set; }
23     public string Url { get; set; }
24
25     public List<Post> Posts { get; set; }
26 }
27
28 public class Post
29 {
30     public int PostId { get; set; }
31     public string Title { get; set; }
32     public string Content { get; set; }
33
34     public int BlogId { get; set; }
35     public Blog Blog { get; set; }
36 }
37 }

```

: 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 5. 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

```

1 using EFGetStarted.AspNet5.NewDb.Models;
2 using Microsoft.Data.Entity;

```

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

```
1 // This method gets called by the runtime. Use this method to add services to
2 ↪ the container.
3     public void ConfigureServices(IServiceCollection services)
4     {
5         var connection = @"Server=(localdb)\mssqllocaldb;Database=EFGetStarted.
6 ↪ AspNet5.NewDb;Trusted_Connection=True;";
7
8         services.AddEntityFramework()
9             .AddSqlServer()
10            .AddDbContext<BloggingContext>(options => options.
11 ↪ UseSqlServer(connection));
```

Create your database

: The migrations experience in ASP.NET 5 is still a work-in-progress. The following steps are overly complex and will be simplified by the time we reach a stable release.

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

- Open a command prompt (**Windows Key + R**, type **cmd**, click **OK**)
- Use the `cd` command to navigate to the project directory
- Run `dnvm use 1.0.0-rc1-final`
- Run `dnx ef migrations add MyFirstMigration` to scaffold a migration to create the initial set of tables for your model.
- Run `dnx ef database update` 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.

: If you make future changes to your model, you can use the `dnx ef migrations add` command to scaffold a new migration to apply the corresponding changes to the database. Once you have checked the scaffolded code (and made any required changes), you can use the `dnx ef database update` command to apply the changes 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

```
1 using EFGetStarted.AspNet5.NewDb.Models;
2 using Microsoft.AspNet.Mvc;
3 using System.Linq;
4
```

```

5 namespace EFGetStarted.AspNet5.NewDb.Controllers
6 {
7     public class BlogsController : Controller
8     {
9         private BloggingContext _context;
10
11        public BlogsController(BloggingContext context)
12        {
13            _context = context;
14        }
15
16        public IActionResult Index()
17        {
18            return View(_context.Blogs.ToList());
19        }
20
21        public IActionResult Create()
22        {
23            return View();
24        }
25
26        [HttpPost]
27        [ValidateAntiForgeryToken]
28        public IActionResult Create(Blog blog)
29        {
30            if (ModelState.IsValid)
31            {
32                _context.Blogs.Add(blog);
33                _context.SaveChanges();
34                return RedirectToAction("Index");
35            }
36
37            return View(blog);
38        }
39    }
40 }
41

```

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 → Server-side*
- Select the **MVC View Page** item template

- Enter **Index.cshtml** as the name and click **OK**
- Replace the contents of the file with the following code

```
1 @model IEnumerable<EFGetStarted.AspNet5.NewDb.Models.Blog>
2
3 @{
4     ViewBag.Title = "Blogs";
5 }
6
7 <h2>Blogs</h2>
8
9 <p>
10     <a asp-controller="Blogs" asp-action="Create">Create New</a>
11 </p>
12
13 <table class="table">
14     <tr>
15         <th>Id</th>
16         <th>Url</th>
17     </tr>
18
19     @foreach (var item in Model)
20     {
21         <tr>
22             <td>
23                 @Html.DisplayFor(modelItem => item.BlogId)
24             </td>
25             <td>
26                 @Html.DisplayFor(modelItem => item.Url)
27             </td>
28         </tr>
29     }
30 </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 5*
- Select the **MVC View Page** item template
- Enter **Create.cshtml** as the name and click **OK**
- Replace the contents of the file with the following code

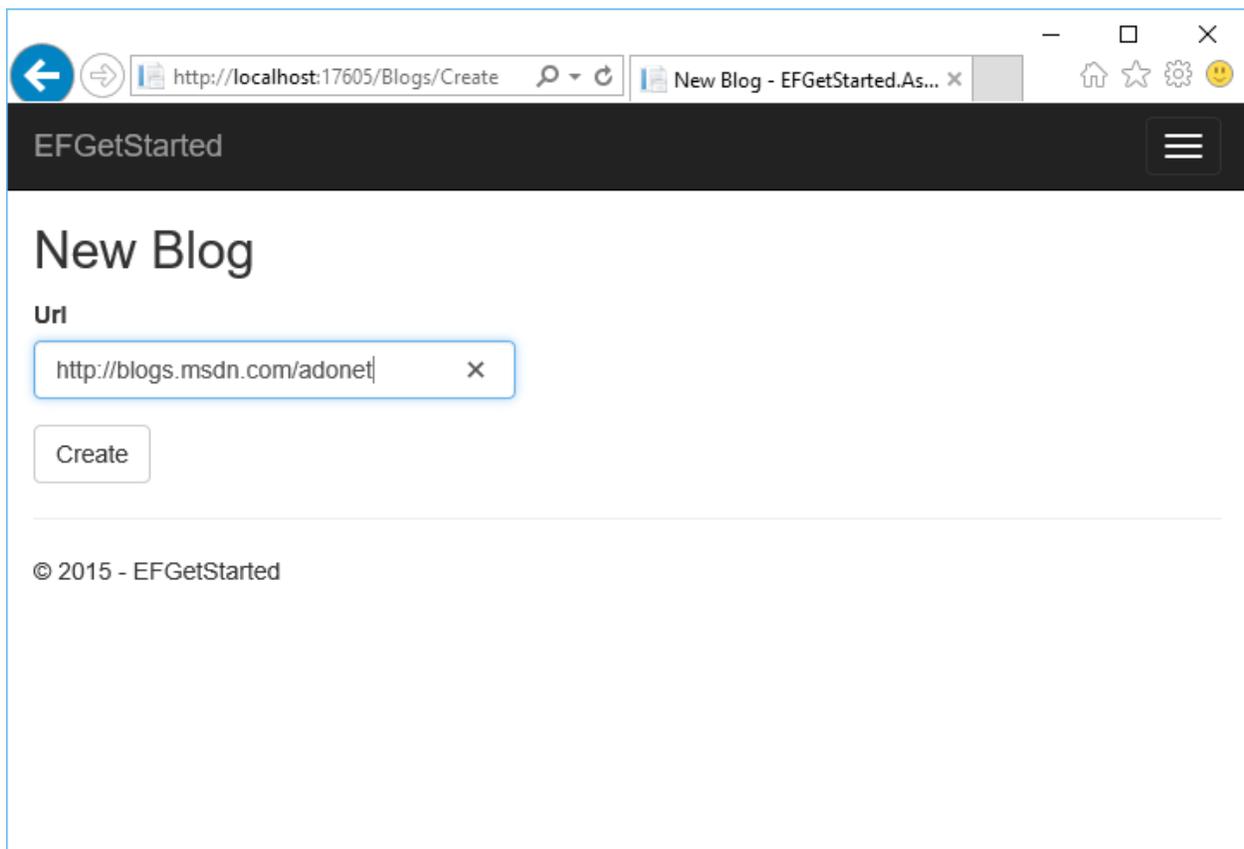
```
1 @model EFGetStarted.AspNet5.NewDb.Models.Blog
2
3 @{
4     ViewBag.Title = "New Blog";
5 }
6
7 <h2>@ViewData["Title"]</h2>
8
9 <form asp-controller="Blogs" asp-action="Create" method="post" class="form-horizontal
10     ↪" role="form">
11     <div class="form-horizontal">
12         <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
13         <div class="form-group">
14             <label asp-for="Url" class="col-md-2 control-label"></label>
```

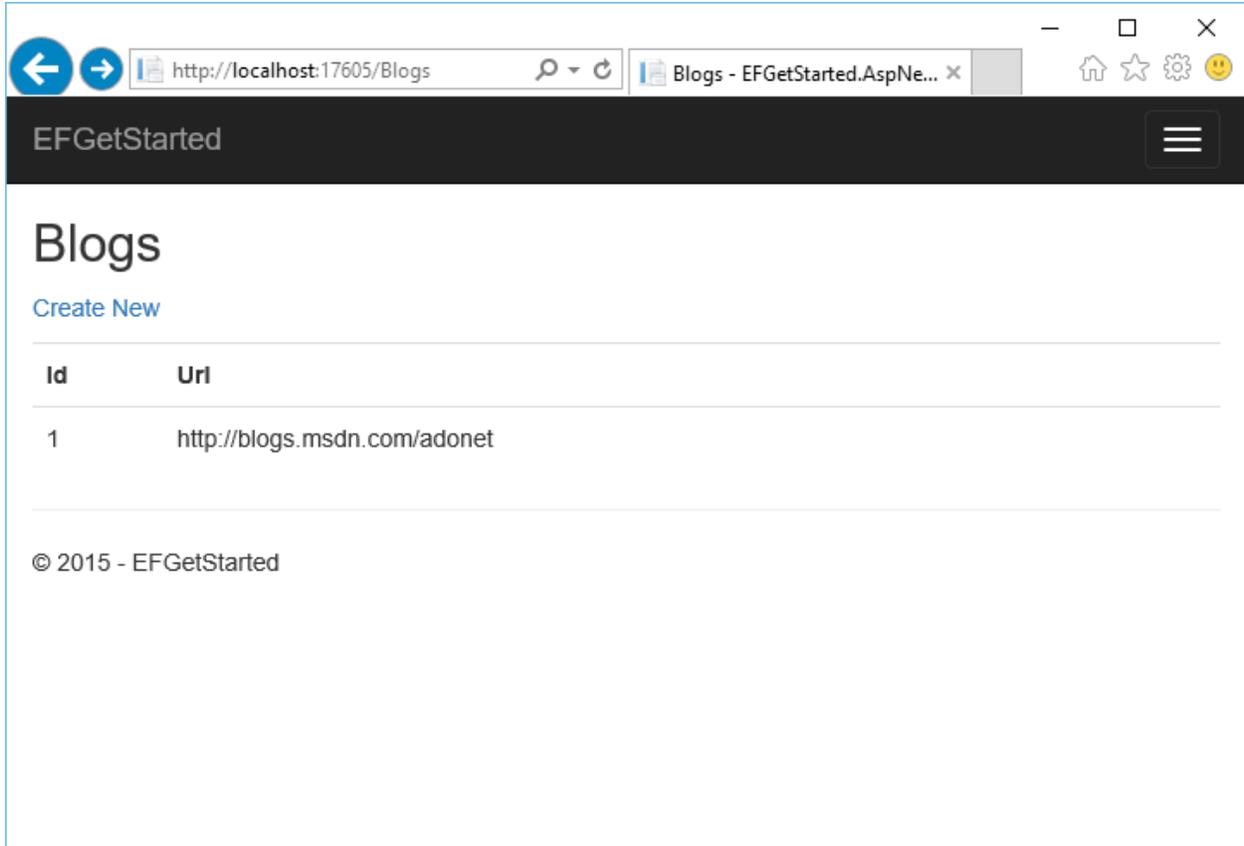
```
14     <div class="col-md-10">
15         <input asp-for="Url" class="form-control" />
16         <span asp-validation-for="Url" class="text-danger"></span>
17     </div>
18 </div>
19 <div class="form-group">
20     <div class="col-md-offset-2 col-md-10">
21         <input type="submit" value="Create" class="btn btn-default" />
22     </div>
23 </div>
24 </div>
25 </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**





: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

ASP.NET 5 Application to Existing Database (Database First)

In this walkthrough, you will build an ASP.NET 5 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:

- *Prerequisites*
 - *Blogging database*
- *Create a new project*
- *Install Entity Framework*
- *Reverse engineer your model*

- *Entity Classes*
- *Derived Context*
- *Register your context with dependency injection*
 - *Remove inline context configuration*
 - *Register and configure your context in Startup.cs*
- *Create a controller*
- *Create views*
- *Run the application*

: You can view this article's [sample](#) on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

- Visual Studio 2015
- ASP.NET 5 RC1 Tools for Visual Studio
- *Blogging database*

Blogging database

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

: If you have already created the **Blogging** database as part of another tutorial, you can skip these steps.

- *Tools* → *Connect to Database...*
- Select **Microsoft SQL Server** and click **Continue**
- Enter **(localdb)\mssqllocaldb** as the **Server Name**
- Enter **master** as the **Database Name**
- 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**

```

1 CREATE DATABASE [Blogging]
2 GO
3
4 USE [Blogging]
5 GO
6
7 CREATE TABLE [Blog] (
```

```
8      [BlogId] int NOT NULL IDENTITY,
9      [Url] nvarchar(max) NOT NULL,
10     CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
11 );
12 GO
13
14 CREATE TABLE [Post] (
15     [PostId] int NOT NULL IDENTITY,
16     [BlogId] int NOT NULL,
17     [Content] nvarchar(max),
18     [Title] nvarchar(max),
19     CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
20     CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog]_
↵ ([BlogId]) ON DELETE CASCADE
21 );
22 GO
23
24 INSERT INTO [Blog] (Url) VALUES
25 ('http://blogs.msdn.com/dotnet'),
26 ('http://blogs.msdn.com/webdev'),
27 ('http://blogs.msdn.com/visualstudio')
28 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 Web Application** project template
- Ensure you are targeting .NET 4.5.1 or later
- Enter **EFGetStarted.AspNet5.ExistingDb** as the name and click **OK**
- Wait for the **New ASP.NET Project** dialog to appear
- Under **ASP.NET 5 Preview Templates** select **Web Application**
- Ensure that **Authentication** is set to **No Authentication**
- Click **OK**

Install Entity Framework

To use EF7 you 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 EntityFramework.MicrosoftSqlServer -Pre`

: In ASP.NET 5 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 EntityFramework.Commands -Pre`
- Run `Install-Package EntityFramework.MicrosoftSqlServer.Design -Pre`
- Open **project.json**
- Locate the `commands` section and add the `ef` command as shown below

```
1 "commands": {
2   "web": "Microsoft.AspNet.Server.Kestrel",
3   "ef": "EntityFramework.Commands"
4 },
```

Reverse engineer your model

: The reverse engineer experience in ASP.NET 5 is still a work-in-progress. The following steps are overly complex and will be simplified by the time we reach a stable release.

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

- Open a command prompt (**Windows Key + R**, type `cmd`, click **OK**)
- Use the `cd` command to navigate to the project directory
- Run `dnvm use 1.0.0-rc1-update1`
- Run the following command to create a model from the existing database

```
dnx ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Bloggging;Trusted_
↪Connection=True;" EntityFramework.MicrosoftSqlServer --outputDir Models
```

The reverse engineer process created entity classes and a derived context based on the schema of the existing database. These classes were created in a **Models** folder in your project.

Entity Classes

The entity classes are simple C# objects that represent the data you will be querying and saving.

```
1 using System;
2 using System.Collections.Generic;
3
4 namespace EFGetStarted.AspNet5.ExistingDb.Models
5 {
6   public partial class Blog
7   {
8     public Blog()
9     {
10      Post = new HashSet<Post>();
11    }
12
13    public int BlogId { get; set; }
14    public string Url { get; set; }
15
16    public virtual ICollection<Post> Post { get; set; }
```

```
17     }
18 }
```

Derived Context

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

```
1 using Microsoft.Data.Entity;
2 using Microsoft.Data.Entity.Metadata;
3
4 namespace EFGetStarted.AspNet5.ExistingDb.Models
5 {
6     public partial class BloggingContext : DbContext
7     {
8         protected override void OnConfiguring(DbContextOptionsBuilder options)
9         {
10             options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;
↪Trusted_Connection=True;");
11         }
12
13         protected override void OnModelCreating(ModelBuilder modelBuilder)
14         {
15             modelBuilder.Entity<Blog>(entity =>
16             {
17                 entity.Property(e => e.Url).IsRequired();
18             });
19
20             modelBuilder.Entity<Post>(entity =>
21             {
22                 entity.HasOne(d => d.Blog).WithMany(p => p.Post).HasForeignKey(d => d.
↪BlogId);
23             });
24         }
25
26         public virtual DbSet<Blog> Blog { get; set; }
27         public virtual DbSet<Post> Post { get; set; }
28     }
29 }
```

Register your context with dependency injection

The concept of dependency injection is central to ASP.NET 5. 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 5, configuration is generally performed in `Startup.cs`. To conform to this pattern, we will move configuration of the database provider to `Startup.cs`.

- Open `ModelsBlogginContext.cs`

- Delete the lines of code highlighted below

```

1 public partial class BloggingContext : DbContext
2 {
3     protected override void OnConfiguring(DbContextOptionsBuilder options)
4     {
5         options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;
↪Trusted_Connection=True;");
6     }
7
8     protected override void OnModelCreating(ModelBuilder modelBuilder)

```

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

```

1 using EFGetStarted.AspNet5.ExistingDb.Models;
2 using Microsoft.Data.Entity;

```

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

```

1 // This method gets called by the runtime. Use this method to add services to
↪the container.
2 public void ConfigureServices(IServiceCollection services)
3 {
4     var connection = @"Server=(localdb)\mssqllocaldb;Database=Blogging;
↪Trusted_Connection=True;";
5
6     services.AddEntityFramework()
7         .AddSqlServer()
8         .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 → 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

```

1 using EFGetStarted.AspNet5.ExistingDb.Models;
2 using Microsoft.AspNet.Mvc;
3 using System.Linq;

```

```
4
5 namespace EFGetStarted.AspNet5.ExistingDb.Controllers
6 {
7     public class BlogsController : Controller
8     {
9         private BloggingContext _context;
10
11        public BlogsController(BloggingContext context)
12        {
13            _context = context;
14        }
15
16        public IActionResult Index()
17        {
18            return View(_context.Blog.ToList());
19        }
20
21        public IActionResult Create()
22        {
23            return View();
24        }
25
26        [HttpPost]
27        [ValidateAntiForgeryToken]
28        public IActionResult Create(Blog blog)
29        {
30            if (ModelState.IsValid)
31            {
32                _context.Blog.Add(blog);
33                _context.SaveChanges();
34                return RedirectToAction("Index");
35            }
36
37            return View(blog);
38        }
39    }
40 }
41 }
```

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 → Server-side*

- Select the **MVC View Page** item template
- Enter **Index.cshtml** as the name and click **OK**
- Replace the contents of the file with the following code

```

1 @model IEnumerable<EFGetStarted.AspNet5.ExistingDb.Models.Blog>
2
3 @{
4     ViewBag.Title = "Blogs";
5 }
6
7 <h2>Blogs</h2>
8
9 <p>
10     <a asp-controller="Blogs" asp-action="Create">Create New</a>
11 </p>
12
13 <table class="table">
14     <tr>
15         <th>Id</th>
16         <th>Url</th>
17     </tr>
18
19     @foreach (var item in Model)
20     {
21         <tr>
22             <td>
23                 @Html.DisplayFor(modelItem => item.BlogId)
24             </td>
25             <td>
26                 @Html.DisplayFor(modelItem => item.Url)
27             </td>
28         </tr>
29     }
30 </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 5*
- Select the **MVC View Page** item template
- Enter **Create.cshtml** as the name and click **OK**
- Replace the contents of the file with the following code

```

1 @model EFGetStarted.AspNet5.ExistingDb.Models.Blog
2
3 @{
4     ViewBag.Title = "New Blog";
5 }
6
7 <h2>@ViewData["Title"]</h2>
8
9 <form asp-controller="Blogs" asp-action="Create" method="post" class="form-horizontal
10     ↪" role="form">
11     <div class="form-horizontal">
12         <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>

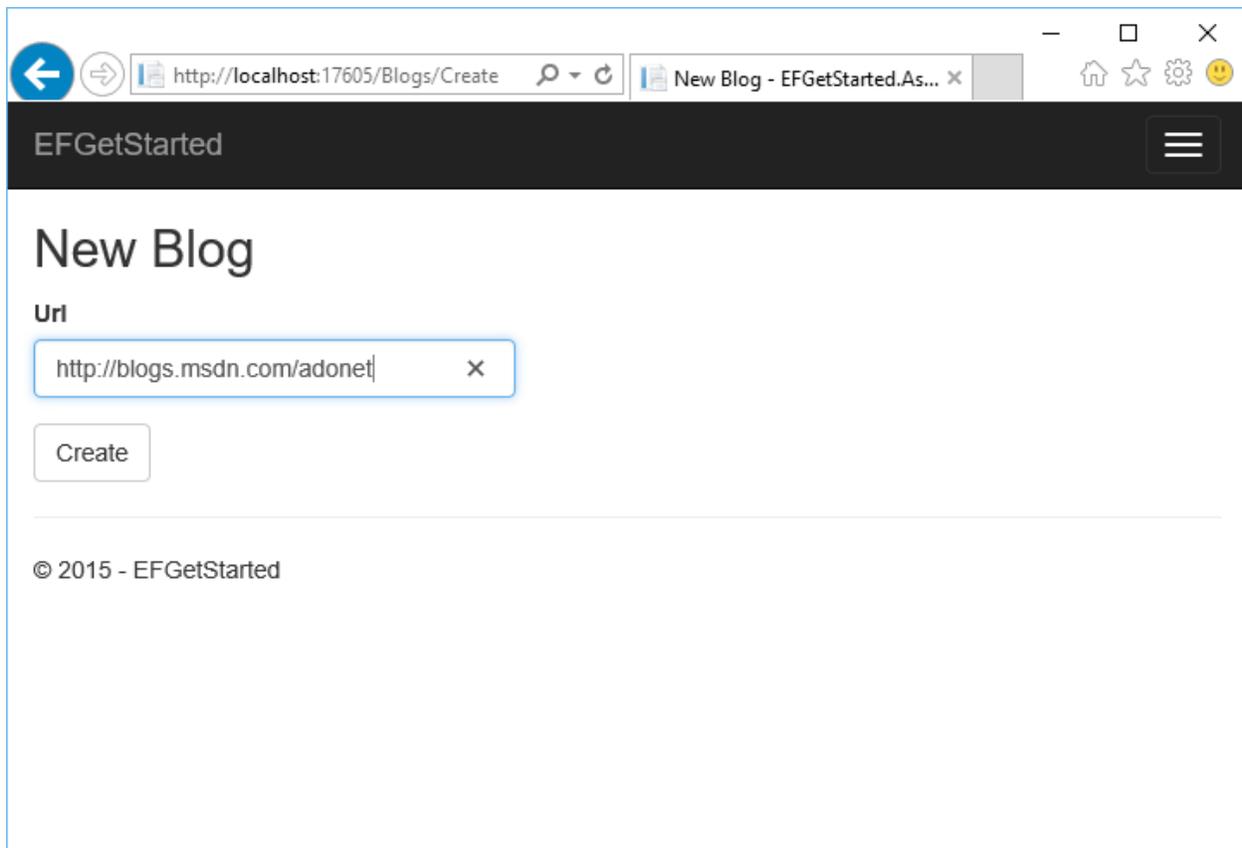
```

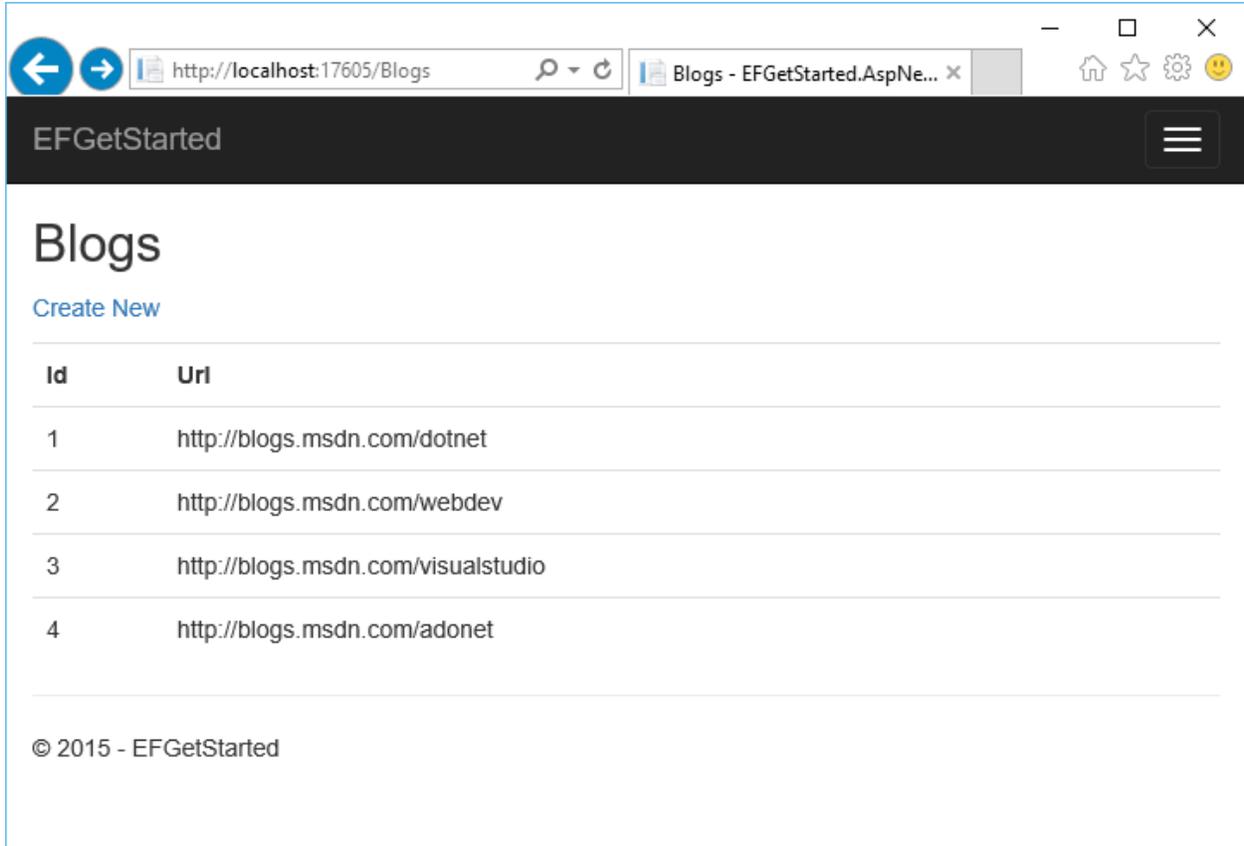
```
12     <div class="form-group">
13         <label asp-for="Url" class="col-md-2 control-label"></label>
14         <div class="col-md-10">
15             <input asp-for="Url" class="form-control" />
16             <span asp-validation-for="Url" class="text-danger"></span>
17         </div>
18     </div>
19     <div class="form-group">
20         <div class="col-md-offset-2 col-md-10">
21             <input type="submit" value="Create" class="btn btn-default" />
22         </div>
23     </div>
24 </div>
25 </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**





: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

Getting Started on Universal Windows Platform

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.

: Deploying a UWP application to the app store requires your application to be compiled with .NET Native. When using Entity Framework there are some query APIs you should avoid to ensure your application is compatible with .NET Native.

You can query using simple LINQ operators that do not change the type of results returned by the query

- Where
- OrderBy

- Distinct
- Skip/Take
- ToList/ToArray
- etc.

You cannot use operators that would change the type of results returned by the query. We are working to support these operators

- Select (you can select a single property, but not an anonymous type)
- GroupBy
- Include/ThenInclude
- Join
- etc.

In this article:

- *Prerequisites*
- *Create a new project*
- *Install Entity Framework*
- *Create your model*
- *Create your database*
- *Use your model*

: You can view this article's [sample](#) on GitHub.

Prerequisites

The following items are required to complete this walkthrough:

- Windows 10
- Visual Studio 2015
- 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**

: To work around an issue with EF7 and .NET Native, you need to add a runtime directive to your application. This issue will be fixed for future releases.

- Open **Properties/Default.rd.xml**
- Add the highlighted line shown below

```
<!-- Add your application specific runtime directives here. -->
<Type Name="System.Collections.ArrayList" Dynamic="Required All" />
```

Install Entity Framework

To use EF7 you 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 EntityFramework.SQLite -Pre`

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 EntityFramework.Commands -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

: The `try/catch` code to set `databaseFilePath` is a temporary workaround to enable migrations to be added at design time. When the application runs, `databaseFilePath` will always be under `ApplicationData.Current.LocalFolder.Path`. However, that API can not be called when migrations creates the context at design time in Visual Studio. The database is never accessed when adding migrations, so we just return a relative file path that will never be used.

: Notice the `OnConfiguring` method (new in EF7) that is used to specify the provider to use and, optionally, other configuration too.

```
1 using Microsoft.Data.Entity;
2 using System.Collections.Generic;
3
4 namespace EFGetStarted.UWP
5 {
6     public class BloggingContext : DbContext
7     {
8         public DbSet<Blog> Blogs { get; set; }
9         public DbSet<Post> Posts { get; set; }
10
```

```
11     protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
12     {
13         optionsBuilder.UseSqlite($"Filename=Bloging.db");
14     }
15
16     protected override void OnModelCreating(ModelBuilder modelBuilder)
17     {
18         // Make Blog.Url required
19         modelBuilder.Entity<Blog>()
20             .Property(b => b.Url)
21             .IsRequired();
22     }
23 }
24
25 public class Blog
26 {
27     public int BlogId { get; set; }
28     public string Url { get; set; }
29
30     public List<Post> Posts { get; set; }
31 }
32
33 public class Post
34 {
35     public int PostId { get; set; }
36     public string Title { get; set; }
37     public string Content { get; set; }
38
39     public int BlogId { get; set; }
40     public Blog Blog { get; set; }
41 }
42 }
```

: 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.

Create your database

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

- *Build -> Build Solution*
- *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.

: Notice that you need to manually build the solution before running the `Add-Migration` command. The command does invoke the build operation on the project, but we are currently investigating why this does not result in the correct assemblies being outputted.

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

```

1 using System;
2 using Microsoft.Data.Entity;
3 using System.Collections.Generic;
4 using System.IO;
5 using System.Linq;
6 using System.Runtime.InteropServices.WindowsRuntime;

```

- Add the highlighted code to apply any pending migrations

```

1     public App ()
2     {
3         Microsoft.ApplicationInsights.WindowsAppInitializer.InitializeAsync (
4             Microsoft.ApplicationInsights.WindowsCollectors.Metadata |
5             Microsoft.ApplicationInsights.WindowsCollectors.Session);
6         this.InitializeComponent ();
7         this.Suspending += OnSuspending;
8
9         using (var db = new BloggingContext ())
10        {
11            db.Database.Migrate ();
12        }
13    }

```

: 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

```

1 <Page
2     x:Class="EFGetStarted.UWP.MainPage"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:EFGetStarted.UWP"
6     xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7     xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8     mc:Ignorable="d"
9     Loaded="Page_Loaded">
10
11     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
12         <StackPanel>
13             <TextBox Name="NewBlogUrl"></TextBox>

```

```
14         <Button Click="Add_Click">Add</Button>
15         <ListView Name="Blogs">
16             <ListView.ItemTemplate>
17                 <DataTemplate>
18                     <TextBlock Text="{Binding Url}" />
19                 </DataTemplate>
20             </ListView.ItemTemplate>
21         </ListView>
22     </StackPanel>
23 </Grid>
24 </Page>
```

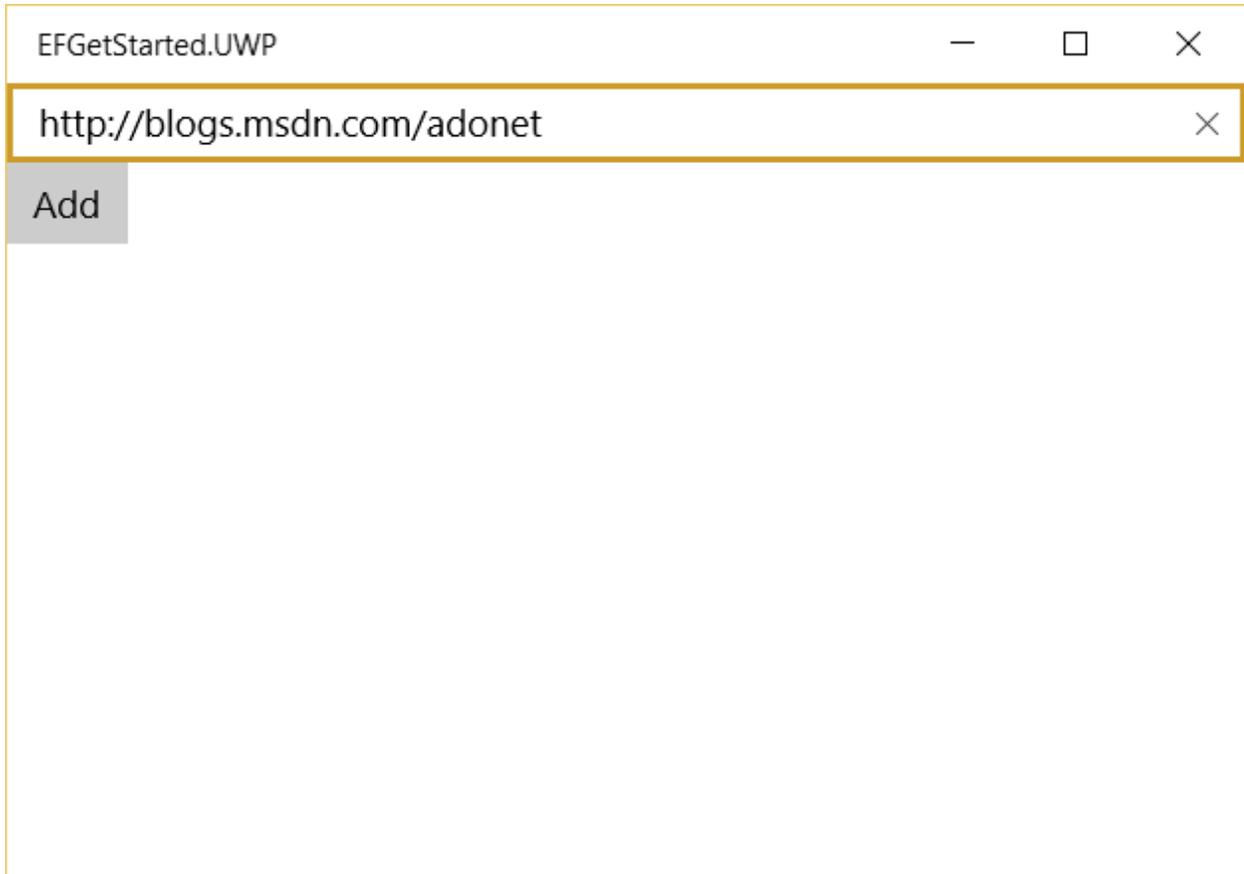
Now we'll add code to wire up the UI with the database

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

```
1 public sealed partial class MainPage : Page
2 {
3     public MainPage()
4     {
5         this.InitializeComponent();
6     }
7
8     private void Page_Loaded(object sender, RoutedEventArgs e)
9     {
10        using (var db = new BloggingContext())
11        {
12            Blogs.ItemsSource = db.Blogs.ToList();
13        }
14    }
15
16    private void Add_Click(object sender, RoutedEventArgs e)
17    {
18        using (var db = new BloggingContext())
19        {
20            var blog = new Blog { Url = NewBlogUrl.Text };
21            db.Blogs.Add(blog);
22            db.SaveChanges();
23
24            Blogs.ItemsSource = db.Blogs.ToList();
25        }
26    }
27 }
```

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





: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

Getting Started on OSX

This walkthrough will create a simple console application using ASP.NET 5 and the SQLite provider.

In this article:

- *Prerequisites*
- *Install ASP.NET 5*
- *Create a new project*
- *Create your model*

- *Create your database*
- *Use your model*
- *Start your app*

: You can view this article's [sample](#) on GitHub.

Prerequisites

Minimum system requirements

- OS X Yosemite or newer

: Known Issues

- DNX-coreclr will die silently if you are missing the ICU library. Make sure to install all dependencies listed in the install guide. (See [Issue dnx#2875](#))
- Migrations on SQLite do not support more complex schema changes due to limitations in SQLite itself.
- Package names are case-sensitive on Linux and OS X due to the case-sensitive filesystem. Make sure to use `EntityFramework.Sqlite` not `EntityFramework.SQLite` in your `project.json` file. (See [Issue dotnet/cli#236](#))

Install ASP.NET 5

A summary of steps to install ASP.NET 5 are included below. For a more up-to-date guide, follow the steps for [Installing ASP.NET 5 on Mac OS X](#). This will ensure you meet the following requirements.

The following steps will install `dnvm`, a command-line tool for installing the .NET Execution environment.

- Install Homebrew
- Use brew to install ICU

```
~ $ brew install icu4c
```

- Run the dnvm

```
~ $ curl -sSL https://raw.githubusercontent.com/aspnet/Home/dev/  
↪dnvminstall.sh | DNX_BRANCH=dev sh && source ~/.dnx/dnvm/dnvm.sh
```

- Install the latest version of DNX. The `-r` selects the CoreCLR runtime. OSX also supports the Mono runtime, but it is not used in this tutorial.

```
~ $ dnvm upgrade -r coreclr
```

If you have trouble installing `dnvm`, consult [Installing ASP.NET 5 on Mac OS X](#).

Create a new project

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


```

  | _|| _| | )  \\
  | _| | _| \_/ | //|\\
  |__||_| /  \\/\
Entity Framework Commands 7.0.0-rc1-16297

Usage: dnx ef [options] [command]

Options:
  --version      Show version information
  -?|-h|--help  Show help information

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

Use "dnx ef [command] --help" for more information about a command.

```

Create your model

With this new project, you are ready to begin using Entity Framework. We will create a simple console application that allows us to write a blog post from the command line.

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

```

1 using System.Collections.Generic;
2 using System.IO;
3 using Microsoft.Data.Entity;
4 using Microsoft.Extensions.PlatformAbstractions;
5
6 namespace ConsoleApp
7 {

```

- **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 in the same folder as the `project.json` file for our project.

```

1     public class BloggingContext : DbContext
2     {
3         public DbSet<Blog> Blogs { get; set; }
4         public DbSet<Post> Posts { get; set; }
5
6         protected override void OnConfiguring(DbContextOptionsBuilder
↵optionsBuilder)
7         {
8             var path = PlatformServices.Default.Application.
↵ApplicationBasePath;
9             optionsBuilder.UseSqlite("Filename=" + Path.Combine(path,
↵"blog.db"));
10        }
11    }

```

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

```
1 public class Blog
2 {
3     public int BlogId { get; set; }
4     public string Url { get; set; }
5     public string Name { get; set; }
6
7     public List<Post> Posts { get; set; }
8 }
9
10 public class Post
11 {
12     public int PostId { get; set; }
13     public string Title { get; set; }
14     public string Content { get; set; }
15
16     public int BlogId { get; set; }
17     public Blog Blog { get; set; }
18 }
```

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

```
~/ConsoleApp/ $ dnu build --quiet
Microsoft .NET Development Utility CoreClr-x64-1.0.0-rc1-16137

Building ConsoleApp for DNXCORE,Version=v5.0

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:03.7102187
Total build time elapsed: 00:00:03.8096587
Total projects built: 1
```

Create your database

We can now use Entity Framework commands to create and manage the schema of our 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/`

```
~/ConsoleApp/ $ dnx ef migrations add MyFirstMigration
```

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

```
~/ConsoleApp/ $ dnx ef database update
```

This should create a new file, `blog.db` which contains two empty tables.

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.

```
1 using System;
2
3 namespace ConsoleApp
4 {
5     public class Program
6     {
7         public static void Main()
8         {
9             using (var db = new BloggingContext())
10            {
11                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
12                var count = db.SaveChanges();
13                Console.WriteLine("{0} records saved to database", count);
14
15                Console.WriteLine();
16                Console.WriteLine("All blogs in database:");
17                foreach (var blog in db.Blogs)
18                {
19                    Console.WriteLine(" - {0}", blog.Url);
20                }
21            }
22        }
23    }
24 }
```

Start your app

Run the application from the command line.

```
~/ConsoleApp/ $ dnx run
1 records saved to database

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

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

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

Getting Started on Linux

This walkthrough will create a simple console application using ASP.NET 5 and the SQLite provider.

In this article:

- *Prerequisites*
- *Install ASP.NET 5*
- *Install SQLite*
- *Create a new project*
- *Create your model*
- *Create your database*
- *Use your model*
- *Start your app*

: You can view this article's [sample](#) on GitHub.

Prerequisites

Minimum system requirements

- Ubuntu, Debian or one of their derivatives

: Known Issues

- DNX-coreclr will die silently if you are missing the ICU library. Make sure to install all dependencies listed in the install guide. (See [Issue dnx#2875](#))
- Migrations on SQLite do not support more complex schema changes due to limitations in SQLite itself.
- Package names are case-sensitive on Linux and OS X due to the case-sensitive filesystem. Make sure to use `EntityFramework.Sqlite` not `EntityFramework.SQLite` in your `project.json` file. (See [Issue dotnet/cli#236](#))

Install ASP.NET 5

A summary of steps to install ASP.NET 5 are included below. For a more up-to-date guide, follow the steps for [Installing ASP.NET 5 on Linux](#). This will ensure you meet the following requirements.

The following steps will install `dnvm`, a command-line tool for installing the .NET Execution environment.

- Install the required libraries

```
~ $ sudo apt-get install unzip curl libunwind8 gettext libssl-dev  
↳ libcurl3-dev zlib1g libicu-dev
```

- Install `dnvm`

```
~ $ curl -sSL https://raw.githubusercontent.com/aspnet/Home/dev/  
↳ dnvminstall.sh | DNX_BRANCH=dev sh && source ~/.dnx/dnvm/dnvm.sh
```

- Install the latest version of DNX. The `-r` selects the CoreCLR runtime. Linux also supports the Mono runtime, but it is not used in this tutorial.

```
~ $ dnvm upgrade -r coreclr
```

If you have trouble installing dnvm, consult [Installing ASP.NET 5 on Linux](#).

Install SQLite

EntityFramework.SQLite requires `libsqlite3`. This may not be installed by default.

On Ubuntu 14, install the SQLite library.

```
~ $ sudo apt-get install libsqlite3-dev
```

Create a new project

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

```
~ $ mkdir ConsoleApp
~ $ cd ConsoleApp/
```

- Create a new file `project.json` with the following contents

```
1 {
2   "dependencies": {
3     "EntityFramework.SQLite": "7.0.0-rc1-final",
4     "EntityFramework.Commands": "7.0.0-rc1-final",
5     "Microsoft.Extensions.PlatformAbstractions": "1.0.0-rc1-final"
6   },
7   "commands": {
8     "run": "ConsoleApp",
9     "ef": "EntityFramework.Commands"
10  },
11  "frameworks": {
12    "dnxcore50": {
13      "dependencies": {
14        "System.Console": "4.0.0-beta-*"
15      }
16    }
17  }
18 }
```

- Execute the following command to install the packages required for this project, including EntityFramework 7 and all its dependencies.

```
~/ConsoleApp/ $ dnu restore
```

- Create a new file named `Program.cs`. Add the following contents to

```
1 using System;
2
3 namespace ConsoleApp
4 {
5     public class Program
6     {
```


for our project.

```

1     public class BloggingContext : DbContext
2     {
3         public DbSet<Blog> Blogs { get; set; }
4         public DbSet<Post> Posts { get; set; }
5
6         protected override void OnConfiguring(DbContextOptionsBuilder
↳optionsBuilder)
7         {
8             var path = PlatformServices.Default.Application.
↳ApplicationBasePath;
9             optionsBuilder.UseSqlite("Filename=" + Path.Combine(path,
↳"blog.db"));
10        }
11    }

```

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

```

1     public class Blog
2     {
3         public int BlogId { get; set; }
4         public string Url { get; set; }
5         public string Name { get; set; }
6
7         public List<Post> Posts { get; set; }
8     }
9
10    public class Post
11    {
12        public int PostId { get; set; }
13        public string Title { get; set; }
14        public string Content { get; set; }
15
16        public int BlogId { get; set; }
17        public Blog Blog { get; set; }
18    }

```

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

```

~/ConsoleApp/ $ dnu build --quiet
Microsoft .NET Development Utility CoreClr-x64-1.0.0-rc1-16137

Building ConsoleApp for DNXCORE,Version=v5.0

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time elapsed 00:00:03.7102187
Total build time elapsed: 00:00:03.8096587
Total projects built: 1

```

Create your database

We can now use Entity Framework commands to create and manage the schema of our 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/`

```
~/ConsoleApp/ $ dnx ef migrations add MyFirstMigration
```

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

```
~/ConsoleApp/ $ dnx ef database update
```

This should create a new file, `blog.db` which contains two empty tables.

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.

```
1 using System;
2
3 namespace ConsoleApp
4 {
5     public class Program
6     {
7         public static void Main()
8         {
9             using (var db = new BloggingContext())
10            {
11                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
12                var count = db.SaveChanges();
13                Console.WriteLine("{0} records saved to database", count);
14
15                Console.WriteLine();
16                Console.WriteLine("All blogs in database:");
17                foreach (var blog in db.Blogs)
18                {
19                    Console.WriteLine(" - {0}", blog.Url);
20                }
21            }
22        }
23    }
24 }
```

Start your app

Run the application from the command line.

```
~/ConsoleApp/ $ dnx run
1 records saved to database

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

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

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x
<http://msdn.com/data/ef>.

The following providers are either available or being developed:

- *EntityFramework.MicrosoftSqlServer*
- *EntityFramework.SQLite*
- *EntityFramework.InMemory*
- *EntityFramework.SqlServerCompact40*
- *EntityFramework.SqlServerCompact35*
- *EntityFramework.Npgsql*

EntityFramework.MicrosoftSqlServer

Database Engine: Microsoft SQL Server (2008 onwards)

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 and dnxcore50), Mono (4.2.0 onwards)

<p>: 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).</p>
--

Status: Pre-release EntityFramework.MicrosoftSqlServer package on NuGet.org that supports the latest EF7 pre-release

Project Site: EntityFramework GitHub project

Getting Started: See *Getting Started on Full .NET (Console, WinForms, WPF, etc.)* or *Getting Started on ASP.NET 5* for a walkthrough that uses this provider. The UnicornStore sample application also uses this provider.

EntityFramework.SQLite

Database Engine: SQLite (3.7 onwards)

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 and dnxcore50), Mono (4.2.0 onwards), Universal Windows Platform (local development only)

Status: Pre-release EntityFramework.SQLite package on NuGet.org that supports the latest EF7 pre-release

Project Site: EntityFramework GitHub project

Getting Started: See *Getting Started on Universal Windows Platform*, *Getting Started on Linux* or *Getting Started on OSX* for walkthroughs that use this provider

EntityFramework.InMemory

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

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 and dnxcore50), Mono (4.2.0 onwards), Universal Windows Platform (local development only)

Status: Pre-release EntityFramework.InMemory package on NuGet.org that supports the latest EF7 pre-release

Project Site: EntityFramework GitHub project

Getting Started: See the tests for the UnicornStore sample application for an example of using this provider.

EntityFramework.SqlServerCompact40

Database Engine: SQL Server Compact Edition 4.0

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 only)

Status: Pre-release EntityFramework.SqlServerCompact40 package on NuGet.org that supports the latest EF7 pre-release

Project Site: ErikEJ/EntityFramework7.SqlServerCompact GitHub Project

Getting Started: See the documentation for this project

EntityFramework.SqlServerCompact35

Database Engine: SQL Server Compact Edition 3.5

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 only)

Status: Pre-release EntityFramework.SqlServerCompact35 package on NuGet.org that supports the latest EF7 pre-release

Project Site: ErikEJ/EntityFramework7.SqlServerCompact GitHub Project

Getting Started: See the documentation for this project

EntityFramework.Npgsql

Database Engine: PostgreSQL

Platforms: Full .NET (4.5.1 onwards), DNX/ASP.NET 5 (dnx451 and dnxcore50), Mono (4.2.0 onwards)

Status: Pre-release EntityFramework7.Npgsql package on NuGet.org that supports the latest EF7 pre-release

Project Site: Npgsql.org

Getting Started: See the [getting started](#) documentation at the [Npgsql](#) site

<p>: This documentation is for EF7 onwards. For EF6.x and earlier release see http://msdn.com/data/ef. EF7.EF6.x http://msdn.com/data/ef.</p>

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 the *Database Providers* section.

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

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Including & Excluding Types*
 - *Conventions*
 - *Data Annotations*

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`

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<AuditEntry>()
8              .Property(a => a.Username)
9              .IsRequired();
10     }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17
18     public List<Post> Posts { get; set; }
19 }
20
21 public class Post
22 {
23     public int PostId { get; set; }
24     public string Title { get; set; }
25     public string Content { get; set; }
26
27     public Blog Blog { get; set; }
28 }
29
30 public class AuditEntry
31 {
32     public int AuditEntryId { get; set; }
33     public string Username { get; set; }
34     public string Action { get; set; }
35 }
```

Data Annotations

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

```

1  public class Blog
2  {
3      public int BlogId { get; set; }
4      public string Url { get; set; }
5
6      public BlogMetadata Metadata { get; set; }
7  }
8
9  [NotMapped]
10 public class BlogMetadata
11 {
12     public DateTime LoadedFromDatabase { get; set; }
13 }

```

Fluent API

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

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Ignore<BlogMetadata>();
8      }
9  }
10
11 public class Blog
12 {
13     public int BlogId { get; set; }
14     public string Url { get; set; }
15
16     public BlogMetadata Metadata { get; set; }
17 }
18
19 public class BlogMetadata
20 {
21     public DateTime LoadedFromDatabase { get; set; }
22 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Including & Excluding Properties*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

Conventions

By convention, every property on each entity type will be included in the model. This includes private properties and those without a getter or setter.

Data Annotations

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

```
1 public class Blog
2 {
3     public int BlogId { get; set; }
4     public string Url { get; set; }
5
6     [NotMapped]
7     public DateTime LoadedFromDatabase { get; set; }
8 }
```

Fluent API

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

```
1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Ignore(b => b.LoadedFromDatabase);
9     }
10 }
11
12 public class Blog
13 {
14     public int BlogId { get; set; }
15     public string Url { get; set; }
16 }
```

```

17     public DateTime LoadedFromDatabase { get; set; }
18 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Keys (primary)*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

Conventions

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

```

1     class Car
2     {
3         public string Id { get; set; }
4
5         public string Make { get; set; }
6         public string Model { get; set; }
7     }

```

```

1     class Car
2     {
3         public string CarId { get; set; }
4
5         public string Make { get; set; }
6         public string Model { get; set; }
7     }

```

Data Annotations

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

```

1  class Car
2  {
3      [Key]
4      public string LicensePlate { get; set; }
5
6      public string Make { get; set; }
7      public string Model { get; set; }
8  }
```

Fluent API

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

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Car>()
8              .HasKey(c => c.LicensePlate);
9      }
10 }
11
12 class Car
13 {
14     public string LicensePlate { get; set; }
15
16     public string Make { get; set; }
17     public string Model { get; set; }
18 }
```

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Car>()
8              .HasKey(c => new { c.State, c.LicensePlate });
9      }
10 }
11
12 class Car
13 {
14     public string State { get; set; }
15     public string LicensePlate { get; set; }
16
17     public string Make { get; set; }
18 }
```

```
18     public string Model { get; set; }  
19 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

Generated Properties

In this article:

- *Generated Properties*
 - *Value Generation Patterns*
 - * *No value generation*
 - * *Value generated on add*
 - * *Value generated on add or update*
 - *Conventions*
 - *Data Annotations*
 - * *No value generation (Data Annotations)*
 - * *Value generated on add (Data Annotations)*
 - * *Value generated on add or update (Data Annotations)*
 - *Fluent API*
 - * *No value generation (Fluent API)*
 - * *Value generated on add (Fluent API)*
 - * *Value generated on add or update (Fluent API)*

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.

: 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).

: 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.

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.

Data Annotations

No value generation (Data Annotations)

```
1 public class Blog
2 {
3     [DatabaseGenerated(DatabaseGeneratedOption.None)]
```

```

4     public int BlogId { get; set; }
5     public string Url { get; set; }
6 }

```

Value generated on add (Data Annotations)

```

1     public class Blog
2     {
3         public int BlogId { get; set; }
4         public string Url { get; set; }
5         [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
6         public DateTime Inserted { get; set; }
7     }

```

: 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)

```

1     public class Blog
2     {
3         public int BlogId { get; set; }
4         public string Url { get; set; }
5         [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
6         public DateTime LastUpdated { get; set; }
7     }

```

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

Fluent API

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

No value generation (Fluent API)

```

1     class MyContext : DbContext
2     {
3         public DbSet<Blog> Blogs { get; set; }
4
5         protected override void OnModelCreating(ModelBuilder modelBuilder)
6         {
7             modelBuilder.Entity<Blog>()
8                 .Property(b => b.BlogId)
9                 .ValueGeneratedNever();
10        }
11    }

```

```
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

Value generated on add (Fluent API)

```
1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.Inserted)
9             .ValueGeneratedOnAdd();
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17     public DateTime Inserted { get; set; }
18 }
```

: 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)

```
1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.LastUpdated)
9             .ValueGeneratedOnAddOrUpdate();
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17     public DateTime LastUpdated { get; set; }
18 }
```

: 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.

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Required/optional properties*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.).

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

Data Annotations

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

```
1 public class Blog
2 {
3     public int BlogId { get; set; }
4     [Required]
5     public string Url { get; set; }
6 }
```

Fluent API

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

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .Property(b => b.Url)
9              .IsRequired();
10     }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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[]`.

: 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:

- *Maximum Length*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

Conventions

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

Data Annotations

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

```

1 public class Blog
2 {
3     public int BlogId { get; set; }
4     [MaxLength(500)]
5     public string Url { get; set; }
6 }

```

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.

```

1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.Url)
9             .HasMaxLength(500);
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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 `SocialSecurityNumber` 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 `SocialSecurityNumber` 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:

- *Concurrency Tokens*
 - *How concurrency tokens work in EF*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*
 - *Timestamp/row version*
 - * *Conventions*
 - * *Data Annotations*
 - * *Fluent API*

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 [Name] = @p1
WHERE [PersonId] = @p0 AND [SocialSecurityNumber] = @p2;
```

Conventions

By convention, properties are never configured as concurrency tokens.

Data Annotations

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

```
1 public class Person
2 {
3     public int PersonId { get; set; }
4     [ConcurrencyCheck]
```

```

5     public string SocialSecurityNumber { get; set; }
6     public string Name { get; set; }
7 }

```

Fluent API

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

```

1     class MyContext : DbContext
2     {
3         public DbSet<Person> People { get; set; }
4
5         protected override void OnModelCreating(ModelBuilder modelBuilder)
6         {
7             modelBuilder.Entity<Person>()
8                 .Property(p => p.SocialSecurityNumber)
9                 .IsConcurrencyToken();
10        }
11    }
12
13    public class Person
14    {
15        public int PersonId { get; set; }
16        public string SocialSecurityNumber { get; set; }
17        public string Name { get; set; }
18    }

```

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.

```

1     public class Blog
2     {
3         public int BlogId { get; set; }
4         public string Url { get; set; }
5
6         [Timestamp]
7         public byte[] Timestamp { get; set; }
8     }

```

Fluent API

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

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .Property(p => p.Timestamp)
9              .ValueGeneratedOnAddOrUpdate()
10             .IsConcurrencyToken();
11     }
12 }
13
14 public class Blog
15 {
16     public int BlogId { get; set; }
17     public string Url { get; set; }
18     public byte[] Timestamp { get; set; }
19 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Shadow Properties*
 - *Conventions*

- *Data Annotations*
- *Fluent API*

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 with the name <principal type name><principal key property name>.

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

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5  }
6
7  public class Blog
8  {
9      public int BlogId { get; set; }
10     public string Url { get; set; }
11
12     public List<Post> Posts { get; set; }
13 }
14
15 public class Post
16 {
17     public int PostId { get; set; }
18     public string Title { get; set; }
19     public string Content { get; set; }
20
21     public Blog Blog { get; set; }
22 }
```

Data Annotations

Shadow properties can not be created with data annotations.

Fluent API

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

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

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
6      {
7          modelBuilder.Entity<Blog> ()
8              .Property<DateTime> ("LastUpdated");
9      }
10 }
11
12 public class Blog
13 {
14     public int BlogId { get; set; }
15     public string Url { get; set; }
16 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

Relationships

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

: 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:

- *Relationships*
 - *Definition of Terms*
 - *Conventions*
 - * *Fully Defined Relationships*
 - * *No Foreign Key Property*
 - * *Single Navigation Property*
 - * *Cascade Delete*
 - *Data Annotations*
 - * *[ForeignKey]*
 - * *[InverseProperty]*
 - *Fluent API*

- * *Single Navigation Property*
- * *Foreign Key*
- * *Principal Key*
- * *Required*
- * *Cascade Delete*
- *Other Relationship Patterns*
 - * *One-to-one*
 - * *Many-to-many*

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)

```

1  public class Blog
2  {
3      public int BlogId { get; set; }
4      public string Url { get; set; }
5
6      public List<Post> Posts { get; set; }

```

```
7     }
8
9     public class Post
10    {
11        public int PostId { get; set; }
12        public string Title { get; set; }
13        public string Content { get; set; }
14
15        public int BlogId { get; set; }
16        public Blog Blog { get; set; }
17    }
```

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.

: Relationships that are discovered by convention will always target the primary key of the principal entity. To target an alternate key, additional configuration must be performed using the Fluent API.

Fully Defined Relationships

The most common pattern for relationships is to have navigation properties defined on both ends of the relationship and a foreign

- 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.

```
1     public class Blog
2     {
3         public int BlogId { get; set; }
4         public string Url { get; set; }
5
6         public List<Post> Posts { get; set; }
7     }
8
9     public class Post
10    {
11        public int PostId { get; set; }
12        public string Title { get; set; }
13        public string Content { get; set; }
14
15        public int BlogId { get; set; }
16        public Blog Blog { get; set; }
17    }
```

: 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).

```

1  public class Blog
2  {
3      public int BlogId { get; set; }
4      public string Url { get; set; }
5
6      public List<Post> Posts { get; set; }
7  }
8
9  public class Post
10 {
11     public int PostId { get; set; }
12     public string Title { get; set; }
13     public string Content { get; set; }
14
15     public Blog Blog { get; set; }
16 }
```

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.

```

1  public class Blog
2  {
3      public int BlogId { get; set; }
4      public string Url { get; set; }
5
6      public List<Post> Posts { get; set; }
7  }
8
9  public class Post
10 {
11     public int PostId { get; set; }
12     public string Title { get; set; }
13     public string Content { get; set; }
14 }
```

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.

: 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.

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.

```
1 public class Blog
2 {
3     public int BlogId { get; set; }
4     public string Url { get; set; }
5
6     public List<Post> Posts { get; set; }
7 }
8
9 public class Post
10 {
11     public int PostId { get; set; }
12     public string Title { get; set; }
13     public string Content { get; set; }
14
15     public int BlogForeignKey { get; set; }
16
17     [ForeignKey("BlogForeignKey")]
18     public Blog Blog { get; set; }
19 }
```

: 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.

```
1 public class Post
2 {
3     public int PostId { get; set; }
4     public string Title { get; set; }
5     public string Content { get; set; }
6
7     public int AuthorUserId { get; set; }
8     public User Author { get; set; }
```

```

9
10     public int ContributorUserId { get; set; }
11     public User Contributor { get; set; }
12 }
13
14 public class User
15 {
16     public string UserId { get; set; }
17     public string FirstName { get; set; }
18     public string LastName { get; set; }
19
20     [InverseProperty("Author")]
21     public List<Post> AuthoredPosts { get; set; }
22
23     [InverseProperty("Contributor")]
24     public List<Post> ContributedToPosts { get; set; }
25 }

```

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.

```

1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4     public DbSet<Post> Posts { get; set; }
5
6     protected override void OnModelCreating(ModelBuilder modelBuilder)
7     {
8         modelBuilder.Entity<Post>()
9             .HasOne(p => p.Blog)
10            .WithMany(b => b.Posts);
11     }
12 }
13
14 public class Blog
15 {
16     public int BlogId { get; set; }
17     public string Url { get; set; }
18
19     public List<Post> Posts { get; set; }
20 }
21
22 public class Post
23 {
24     public int PostId { get; set; }
25     public string Title { get; set; }
26     public string Content { get; set; }
27
28     public Blog Blog { get; set; }
29 }

```

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Blog>()
9              .HasMany(b => b.Posts)
10             .WithOne();
11      }
12  }
13
14  public class Blog
15  {
16      public int BlogId { get; set; }
17      public string Url { get; set; }
18
19      public List<Post> Posts { get; set; }
20  }
21
22  public class Post
23  {
24      public int PostId { get; set; }
25      public string Title { get; set; }
26      public string Content { get; set; }
27  }
```

Foreign Key

You can use the Fluent API to configure which property should be used as the foreign key property for a given relationship.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Post>()
9              .HasOne(p => p.Blog)
10             .WithMany(b => b.Posts)
11             .HasForeignKey(p => p.BlogForeignKey);
12      }
13  }
14
15  public class Blog
16  {
17      public int BlogId { get; set; }
18      public string Url { get; set; }
```

```

19     public List<Post> Posts { get; set; }
20 }
21
22
23 public class Post
24 {
25     public int PostId { get; set; }
26     public string Title { get; set; }
27     public string Content { get; set; }
28
29     public int BlogForeignKey { get; set; }
30     public Blog Blog { get; set; }
31 }

```

The following code listing shows how to configure a composite foreign key.

```

1 class MyContext : DbContext
2 {
3     public DbSet<Car> Cars { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Car>()
8             .HasKey(c => new { c.State, c.LicensePlate });
9
10        modelBuilder.Entity<RecordOfSale>()
11            .HasOne(s => s.Car)
12            .WithMany(c => c.SaleHistory)
13            .HasForeignKey(s => new { s.CarState, s.CarLicensePlate });
14    }
15 }
16
17 public class Car
18 {
19     public string State { get; set; }
20     public string LicensePlate { get; set; }
21     public string Make { get; set; }
22     public string Model { get; set; }
23
24     public List<RecordOfSale> SaleHistory { get; set; }
25 }
26
27 public class RecordOfSale
28 {
29     public int RecordOfSaleId { get; set; }
30     public DateTime DateSold { get; set; }
31     public decimal Price { get; set; }
32
33     public string CarState { get; set; }
34     public string CarLicensePlate { get; set; }
35     public Car Car { get; set; }
36 }

```

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).

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<RecordOfSale>()
8              .HasOne(s => s.Car)
9              .WithMany(c => c.SaleHistory)
10             .HasForeignKey(s => s.CarLicensePlate)
11             .HasPrincipalKey(c => c.LicensePlate);
12     }
13 }
14
15 public class Car
16 {
17     public int CarId { get; set; }
18     public string LicensePlate { get; set; }
19     public string Make { get; set; }
20     public string Model { get; set; }
21
22     public List<RecordOfSale> SaleHistory { get; set; }
23 }
24
25 public class RecordOfSale
26 {
27     public int RecordOfSaleId { get; set; }
28     public DateTime DateSold { get; set; }
29     public decimal Price { get; set; }
30
31     public string CarLicensePlate { get; set; }
32     public Car Car { get; set; }
33 }

```

The following code listing shows how to configure a composite principal key.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<RecordOfSale>()
8              .HasOne(s => s.Car)
9              .WithMany(c => c.SaleHistory)
10             .HasForeignKey(s => new { s.CarState, s.CarLicensePlate })
11             .HasPrincipalKey(c => new { c.State, c.LicensePlate });
12     }
13 }
14
15 public class Car
16 {
17     public int CarId { get; set; }
18     public string State { get; set; }
19     public string LicensePlate { get; set; }
20     public string Make { get; set; }
21     public string Model { get; set; }

```

```

22     public List<RecordOfSale> SaleHistory { get; set; }
23 }
24
25
26 public class RecordOfSale
27 {
28     public int RecordOfSaleId { get; set; }
29     public DateTime DateSold { get; set; }
30     public decimal Price { get; set; }
31
32     public string CarState { get; set; }
33     public string CarLicensePlate { get; set; }
34     public Car Car { get; set; }
35 }

```

: 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).

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Post>()
9              .HasOne(p => p.Blog)
10             .WithMany(b => b.Posts)
11             .IsRequired();
12     }
13 }
14
15 public class Blog
16 {
17     public int BlogId { get; set; }
18     public string Url { get; set; }
19
20     public List<Post> Posts { get; set; }
21 }
22
23 public class Post
24 {
25     public int PostId { get; set; }
26     public string Title { get; set; }
27     public string Content { get; set; }
28
29     public Blog Blog { get; set; }
30 }

```

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 principal is deleted:

- **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.

: 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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Post>()
9              .HasOne(p => p.Blog)
10             .WithMany(b => b.Posts)
11             .OnDelete(DeleteBehavior.Cascade);
12     }
13 }
14
15 public class Blog
16 {
17     public int BlogId { get; set; }
18     public string Url { get; set; }
19
20     public List<Post> Posts { get; set; }
21 }
22
23 public class Post
24 {
25     public int PostId { get; set; }
26     public string Title { get; set; }
27     public string Content { get; set; }
28
29     public int? BlogId { get; set; }
30     public Blog Blog { get; set; }
31 }
```

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.

```

1  public class Blog
2  {
3      public int BlogId { get; set; }
4      public string Url { get; set; }
5
6      public BlogImage BlogImage { get; set; }
7  }
8
9  public class BlogImage
10 {
11     public int BlogImageId { get; set; }
12     public byte[] Image { get; set; }
13     public string Caption { get; set; }
14
15     public int BlogId { get; set; }
16     public Blog Blog { get; set; }
17 }

```

: EF will chose 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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<BlogImage> BlogImages { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Blog>()
9              .HasOne(p => p.BlogImage)
10             .WithOne(i => i.Blog)
11             .HasForeignKey<BlogImage>(b => b.BlogForeignKey);
12     }
13 }
14
15 public class Blog
16 {
17     public int BlogId { get; set; }
18     public string Url { get; set; }
19
20     public BlogImage BlogImage { get; set; }
21 }
22
23 public class BlogImage
24 {
25     public int BlogImageId { get; set; }
26     public byte[] Image { get; set; }
27     public string Caption { get; set; }

```

```

28     public int BlogForeignKey { get; set; }
29     public Blog Blog { get; set; }
30 }
31

```

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Post> Posts { get; set; }
4      public DbSet<Tag> Tags { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<PostTag>()
9              .HasKey(t => new { t.PostId, t.TagId });
10
11         modelBuilder.Entity<PostTag>()
12             .HasOne(pt => pt.Post)
13             .WithMany(p => p.PostTags)
14             .HasForeignKey(pt => pt.PostId);
15
16         modelBuilder.Entity<PostTag>()
17             .HasOne(pt => pt.Tag)
18             .WithMany(t => t.PostTags)
19             .HasForeignKey(pt => pt.TagId);
20     }
21 }
22
23 public class Post
24 {
25     public int PostId { get; set; }
26     public string Title { get; set; }
27     public string Content { get; set; }
28
29     public List<PostTag> PostTags { get; set; }
30 }
31
32 public class Tag
33 {
34     public string TagId { get; set; }
35
36     public List<PostTag> PostTags { get; set; }
37 }
38
39 public class PostTag
40 {
41     public int PostId { get; set; }
42     public Post Post { get; set; }
43
44     public string TagId { get; set; }
45     public Tag Tag { get; set; }

```

46

}

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

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:

- *Indexes*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

Conventions

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

Data Annotations

Indexes can not be created using data annotations.

Fluent API

You can use the Fluent API specify an index on a single property. By default, indexes are non-unique.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .HasIndex(b => b.Url);
9      }
10 }
```

```
11 public class Blog
12 {
13     public int BlogId { get; set; }
14     public string Url { get; set; }
15 }
16
```

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).

```
1 modelBuilder.Entity<Blog>()
2     .HasIndex(b => b.Url)
3     .IsUnique();
```

You can also specify an index over more than one column.

```
1 class MyContext : DbContext
2 {
3     public DbSet<Person> People { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Person>()
8             .HasIndex(p => new { p.FirstName, p.LastName });
9     }
10 }
11
12 public class Person
13 {
14     public int PersonId { get; set; }
15     public string FirstName { get; set; }
16     public string LastName { get; set; }
17 }
```

: 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.

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

Alternate Keys

An alternate key serves as a alternate unique identifier for each entity instance in addition to the primary key. When using a relational database this maps to the concept of a unique index/constraint. 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:

- *Alternate Keys*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Post>()
9              .HasOne(p => p.Blog)
10             .WithMany(b => b.Posts)
11             .HasForeignKey(p => p.BlogUrl)
12             .HasPrincipalKey(b => b.Url);
13     }
14 }
15
16 public class Blog
17 {
18     public int BlogId { get; set; }
19     public string Url { get; set; }
20
21     public List<Post> Posts { get; set; }
22 }
23
24 public class Post
25 {
26     public int PostId { get; set; }
27     public string Title { get; set; }
28     public string Content { get; set; }
29
30     public string BlogUrl { get; set; }
31     public Blog Blog { get; set; }
32 }
```

Data Annotations

Alternate keys can not be configured using Data Annotations.

Fluent API

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

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Car>()
8              .HasAlternateKey(c => c.LicensePlate);
9      }
10 }
11
12 class Car
13 {
14     public int CarId { get; set; }
15     public string LicensePlate { get; set; }
16     public string Make { get; set; }
17     public string Model { get; set; }
18 }

```

You can also use the Fluent API to configure multiple properties to be an alternate key (known as a composite alternate key).

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Car>()
8              .HasAlternateKey(c => new { c.State, c.LicensePlate });
9      }
10 }
11
12 class Car
13 {
14     public int CarId { get; set; }
15     public string State { get; set; }
16     public string LicensePlate { get; set; }
17     public string Make { get; set; }
18     public string Model { get; set; }
19 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of

the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

Inheritance

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

In this article:

- *Inheritance*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<RssBlog> RssBlogs { get; set; }
5  }
6
7  public class Blog
8  {
9      public int BlogId { get; set; }
10     public string Url { get; set; }
11 }
12
13 public class RssBlog : Blog
14 {
15     public string RssUrl { get; set; }
16 }

```

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<RssBlog>();

```

```
8     }  
9 }
```

Data Annotations

You cannot use Data Annotations to configure inheritance.

Fluent API

The Fluent API for inheritance depends on the database provider you are using. See *Inheritance (Relational Database)* for the configuration you can perform for a relational database provider.

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: 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 *EntityFramework.Relational* package).

Relational Database Modeling

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

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

Table Mapping

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

In this article:

- *Table Mapping*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

Conventions

By convention, each entity will be setup to map to a table with the same name as the entity.

Data Annotations

You can use Data Annotations to configure the table that a type maps to.

```

1  [Table("blogs")]
2  public class Blog
3  {
4      public int BlogId { get; set; }
5      public string Url { get; set; }
6  }
```

You can also specify a schema that the table belongs to.

```

1  [Table("blogs", Schema = "blogging")]
2  public class Blog
3  {
4      public int BlogId { get; set; }
5      public string Url { get; set; }
6  }
```

Fluent API

You can use the Fluent API to configure the table that a type maps to.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              ..ToTable("blogs");
9      }
10 }
11
12 public class Blog
13 {
14     public int BlogId { get; set; }
15     public string Url { get; set; }
16 }
```

You can also specify a schema that the table belongs to.

```
1     modelBuilder.Entity<Blog> ()
2         .ToTable("blogs", schema: "blogging");
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

Column Mapping

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

In this article:

- *Column Mapping*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.

```
1     public class Blog
2     {
3         [Column("blog_id")]
4         public int BlogId { get; set; }
5         public string Url { get; set; }
6     }
```

Fluent API

You can use the Fluent API to configure the column to which a property is mapped.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .Property(b => b.BlogId)
9              .HasColumnName("blog_id");
10     }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

Data Types

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

In this article:

- *Data Types*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.

```
1 public class Blog
2 {
3     public int BlogId { get; set; }
4     [Column(TypeName = "varchar(200)")]
5     public string Url { get; set; }
6 }
```

Fluent API

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

```
1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.Url)
9             .HasColumnType("varchar(200)");
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

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.

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

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of

the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .HasKey(b => b.BlogId)
9              .HasName("PrimaryKey_BlogId");
10     }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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 chose 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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.HasDefaultSchema("blogging");
8      }
9  }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Person> People { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Person>()
8              .Property(p => p.DisplayName)
9              .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
10     }
11 }
12
13 public class Person
14 {
15     public int PersonId { get; set; }
16     public string FirstName { get; set; }
17     public string LastName { get; set; }
18     public string DisplayName { get; set; }
19 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Order> Orders { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.HasSequence<int>("OrderNumbers");
8      }
9  }
10
11 public class Order
```

You can also configure additional aspect of the sequence, such as its schema, start value, and increment.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Order> Orders { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
8              .StartsAt(1000)
9              .IncrementsBy(5);
10     }
11 }
```

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Order> Orders { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
8              .StartsAt(1000)
9              .IncrementsBy(5);
10 }
```

```

11         modelBuilder.Entity<Order>()
12             .Property(o => o.OrderNo)
13             .HasDefaultValueSql("NEXT VALUE FOR shared.OrderNumbers");
14     }
15 }
16
17 public class Order
18 {
19     public int OrderId { get; set; }
20     public int OrderNo { get; set; }
21     public string Url { get; set; }
22 }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```

1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {

```

```
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.Rating)
9             .HasDefaultValue(3);
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17     public int Rating { get; set; }
18 }
```

You can also specify a SQL fragment that is used to calculate the default value.

```
1 class MyContext : DbContext
2 {
3     public DbSet<Blog> Blogs { get; set; }
4
5     protected override void OnModelCreating(ModelBuilder modelBuilder)
6     {
7         modelBuilder.Entity<Blog>()
8             .Property(b => b.Created)
9             .HasDefaultValueSql("getdate()");
10    }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17     public DateTime Created { get; set; }
18 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .HasIndex(b => b.Url)
9              .HasName("Index_Url");
10     }
11 }
12
13 public class Blog
14 {
15     public int BlogId { get; set; }
16     public string Url { get; set; }
17 }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```
1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<Post> Posts { get; set; }
5
6      protected override void OnModelCreating(ModelBuilder modelBuilder)
7      {
8          modelBuilder.Entity<Post>()
9              .HasOne(p => p.Blog)
10             .WithMany(b => b.Posts)
11             .HasForeignKey(p => p.BlogId)
12             .HasConstraintName("ForeignKey_Post_Blog");
13     }
14 }
15
16 public class Blog
17 {
18     public int BlogId { get; set; }
19     public string Url { get; set; }
20
21     public List<Post> Posts { get; set; }
22 }
23
24 public class Post
25 {
26     public int PostId { get; set; }
27     public string Title { get; set; }
28     public string Content { get; set; }
29
30     public int BlogId { get; set; }
31     public Blog Blog { get; set; }
```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Car> Cars { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Car>()
8              .HasAlternateKey(c => c.LicensePlate)
9              .HasName("AlteranteKey_LicensePlate");
10     }
11 }
12
13 class Car
14 {
15     public int CarId { get; set; }
16     public string LicensePlate { get; set; }
17     public string Make { get; set; }
18     public string Model { get; set; }

```

: This documentation is for EF7 onwards. For EF6.x and earlier release see <http://msdn.com/data/ef>. EF7.EF6.x <http://msdn.com/data/ef>.

: This article uses EF 7.0.0-rc1 which is the latest pre-release available on NuGet.org. You can find nightly builds of the EF7 code base hosted on <https://www.myget.org/F/aspnetvnext/> but we do not maintain up-to-date documentation for nightly builds.

: 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 *EntityFramework.Relational* package).

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:

- *Inheritance (Relational Database)*
 - *Conventions*
 - *Data Annotations*
 - *Fluent API*

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4      public DbSet<RssBlog> RssBlogs { get; set; }
5  }
6
7  public class Blog
8  {
9      public int BlogId { get; set; }
10     public string Url { get; set; }
11 }
12
13 public class RssBlog : Blog
14 {
15     public string RssUrl { get; set; }
16 }

```

Results				
	BlogId	Discriminator	Url	RssUrl
1	1	Blog	http://blogs.msdn.com/dotnet	NULL
2	2	RssBlog	http://blogs.msdn.com/adonet	http://blogs.msdn.com/b/adonet/atom.aspx

Data Annotations

You cannot use Data Annotations to configure inheritance.

Fluent API

You can use the Fluent API to configure the name and type of the discriminator column and the values that are used to identify each type in the hierarchy.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .HasDiscriminator<string>("blog_type")
9              .HasValue<Blog>("blog_base")
10             .HasValue<RssBlog>("blog_rss");
11     }
12 }
13
14 public class Blog
15 {
16     public int BlogId { get; set; }
17     public string Url { get; set; }
18 }
19
20 public class RssBlog : Blog
21 {
22     public string RssUrl { get; set; }
23 }

```

: You can view this article's [sample](#) on GitHub.

Methods of configuration

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.

```

1  class MyContext : DbContext
2  {
3      public DbSet<Blog> Blogs { get; set; }
4
5      protected override void OnModelCreating(ModelBuilder modelBuilder)
6      {
7          modelBuilder.Entity<Blog>()
8              .Property(b => b.Url)

```

```
9         .IsRequired();  
10     }  
11 }
```

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.

```
1 public class Blog  
2 {  
3     public int BlogId { get; set; }  
4     [Required]  
5     public string Url { get; set; }  
6 }
```

CHAPTER 4

Contribute

We accept pull requests! But you're more likely to have yours accepted if you follow the [guidelines for contributing](#).