Caution

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

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

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

In this article:

Tip

You can view this article's sample on GitHub.

Prerequisites

The following prerequisites are needed to complete this walkthrough:

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

Blogging database

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

Note

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

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

```sql
CREATE DATABASE [Blogging]
GO

USE [Blogging]
GO

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

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

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

Create a new project

- Open Visual Studio 2015
- File ‣ New ‣ Project...
- From the left menu select Templates ‣ Visual C# ‣ Web
- Select the **ASP.NET Core Web Application (.NET Core)** project template
- Enter **EFGetStarted.AspNetCore.ExistingDb** as the name and click **OK**
- Wait for the **New ASP.NET Core Web Application** dialog to appear
- Select the **Web Application** template and ensure that **Authentication** is set to **No Authentication**
- Click **OK**

Install Entity Framework

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

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

Note

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

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

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

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

Reverse engineer your model

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

- Tools –> NuGet Package Manager –> Package Manager Console
- Run the following command to create a model from the existing database. If you receive an error stating the term 'Scaffold-DbContext' is not recognized as the name of a cmdlet, then close and reopen Visual Studio.

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

The reverse engineer process created entity classes and a derived context based on the schema of the existing database. The entity classes are simple C# objects that represent the data you will be querying and saving.

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

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

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

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

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

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

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

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

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

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

Register your context with dependency injection

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

Remove inline context configuration

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

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

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

- Add the lines of code highlighted below

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

Register and configure your context in Startup.cs

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

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

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

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

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

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

Create a controller

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

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

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

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

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

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

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

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

            return View(blog);
        }

    }
}
```

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

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

Create views

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

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

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

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

@{
    ViewBag.Title = "Blogs";
}

<h2>Blogs</h2>

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

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

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

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

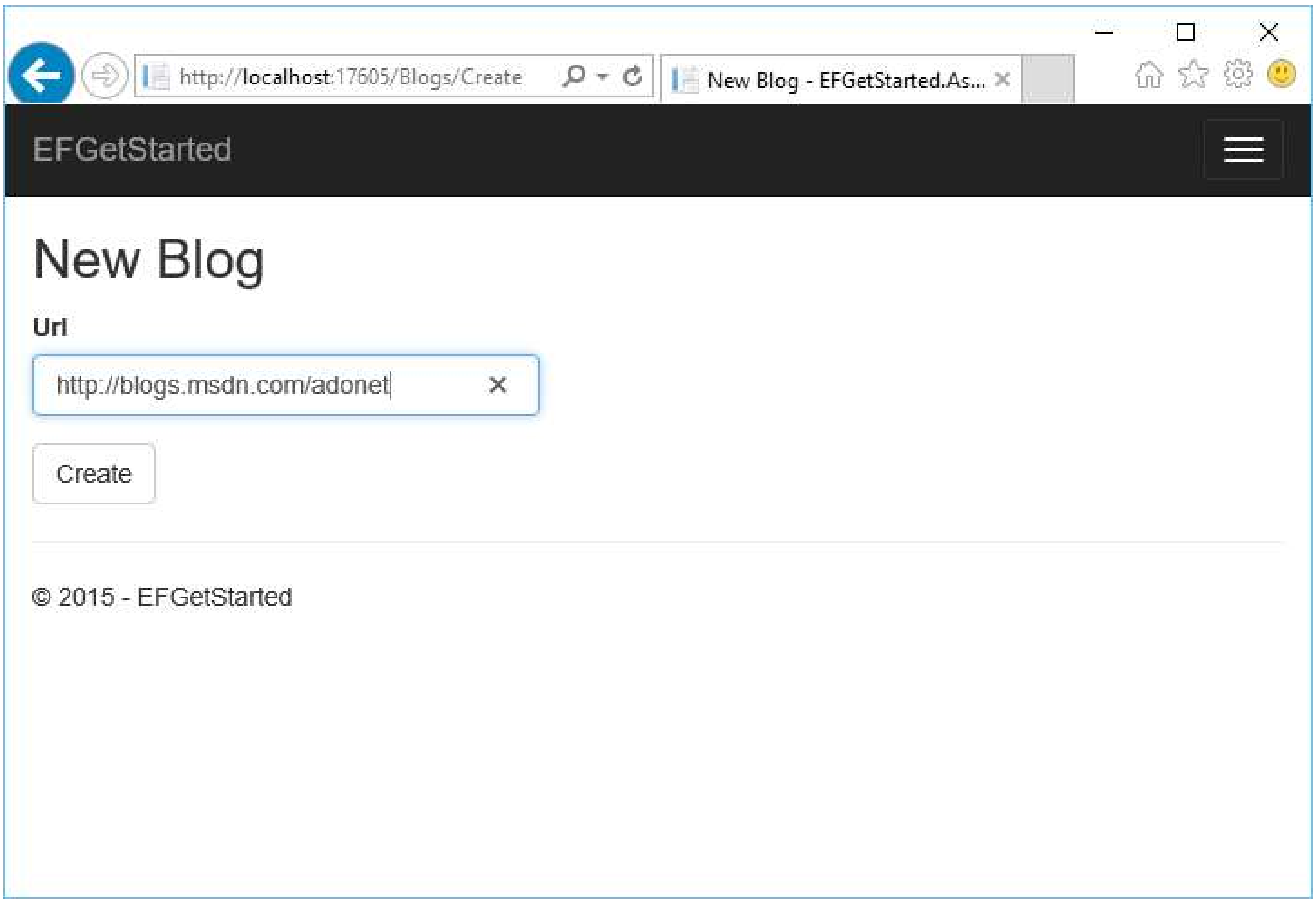
- Right-click on the **Blogs** folder and select Add ‣ New Item...
- From the left menu select Installed ‣ ASP.NET
- Select the **MVC View Page** item template
- Enter **Create.cshtml** as the name and click **Add**
- Replace the contents of the file with the following code

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

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

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

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

Run the application

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

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


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

EFGetStarted

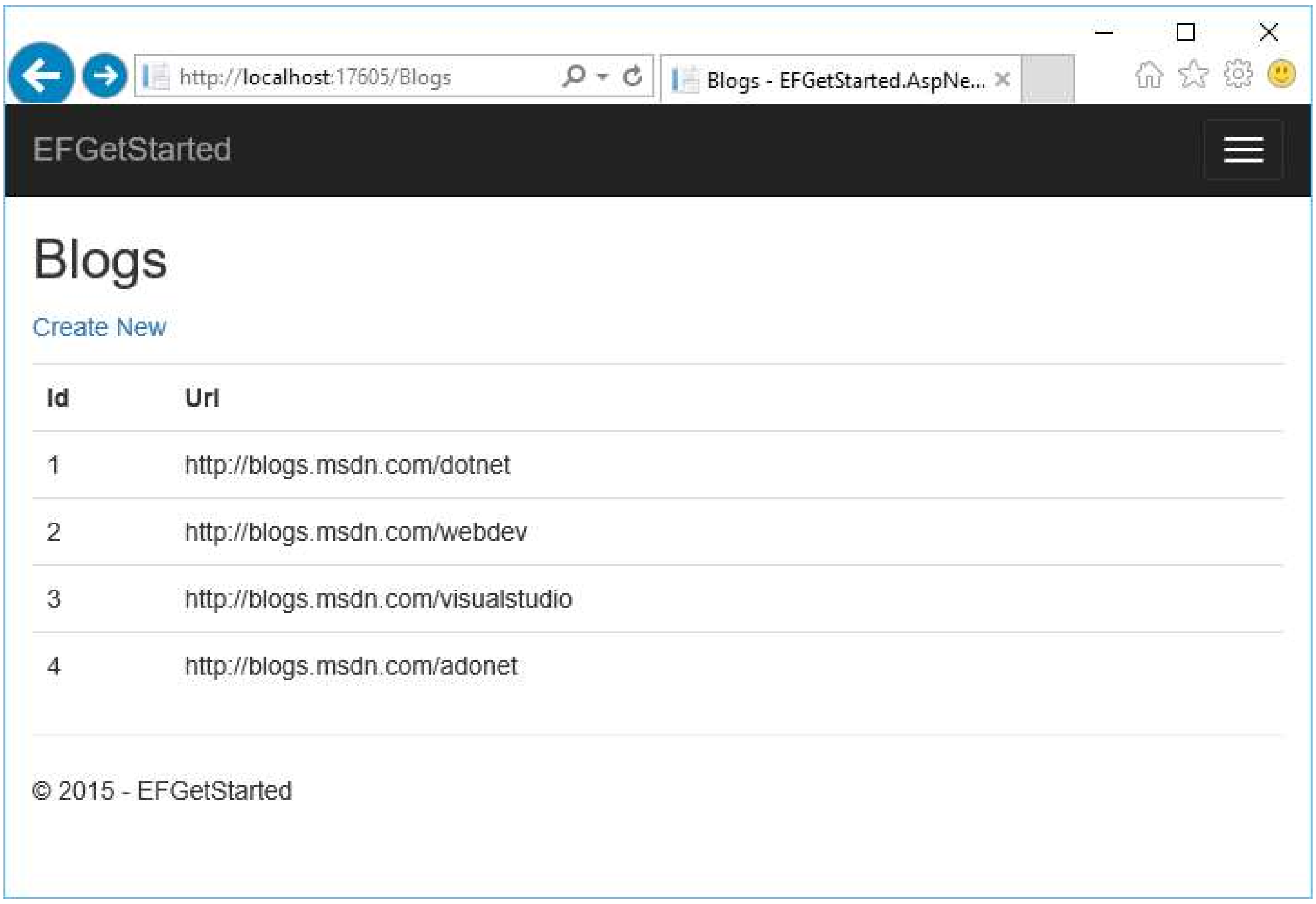
New Blog

Url

http://blogs.msdn.com/adonet

Create

© 2015 - EFGetStarted



EFGetStarted

Blogs

Create New

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

© 2015 - EFGetStarted