

Getting started with ASP.NET Core MVC and Entity Framework Core using Visual Studio (1 of 10)

By [Tom Dykstra](#) and [Rick Anderson](#)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017.

The sample application is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample application from scratch.

[Download or view the completed application.](#)

EF Core 1.1 is the latest version of EF but does not yet have all the features of EF 6.x. For information about how to choose between EF 6.x and EF Core 1.0, see [EF Core vs. EF6.x](#). If you choose EF 6.x, see [the previous version of this tutorial series](#).

Note

For the Visual Studio 2015 version of this tutorial, see the [VS 2015 version of ASP.NET Core documentation in PDF format](#).

Prerequisites

[Visual Studio 2017](#) with the ASP.NET and web development and .NET Core cross-platform development workloads installed.

Troubleshooting

If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the [completed project](#). For a list of common errors and how to solve them, see [the Troubleshooting section of the last tutorial in the series](#). If you don't find what you need there, you can post a question to StackOverflow.com for [ASP.NET Core](#) or [EF Core](#).

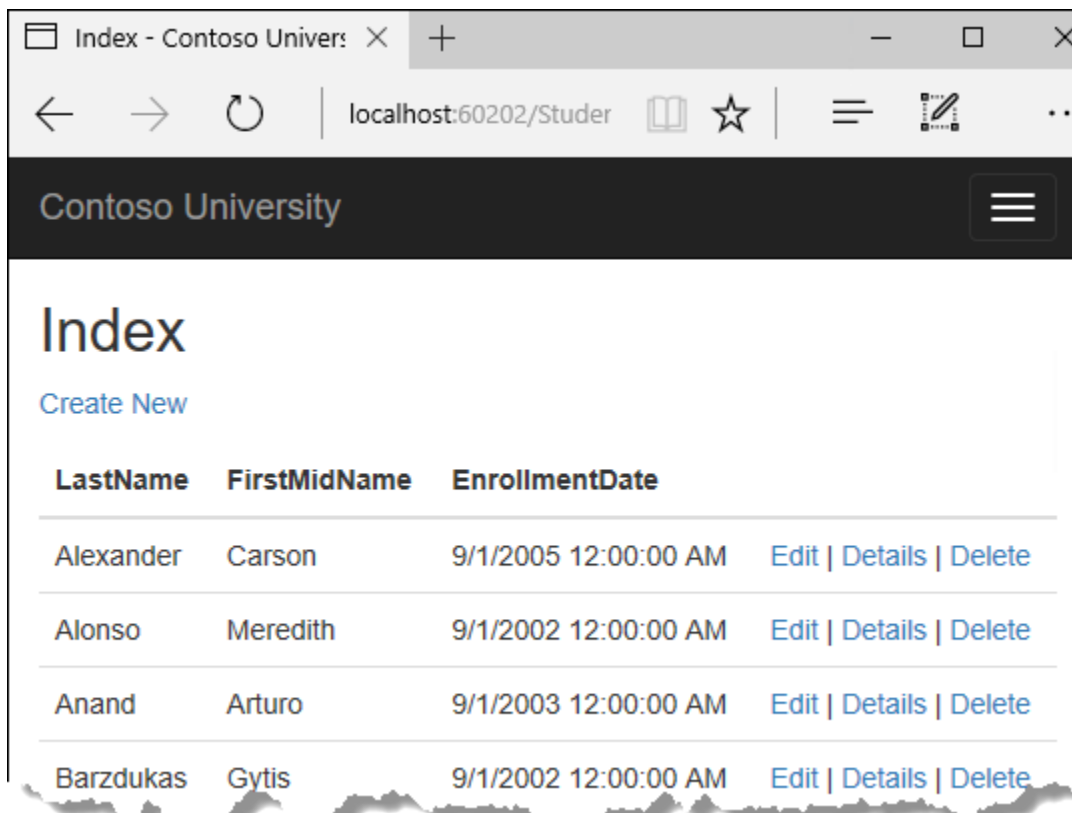
Tip

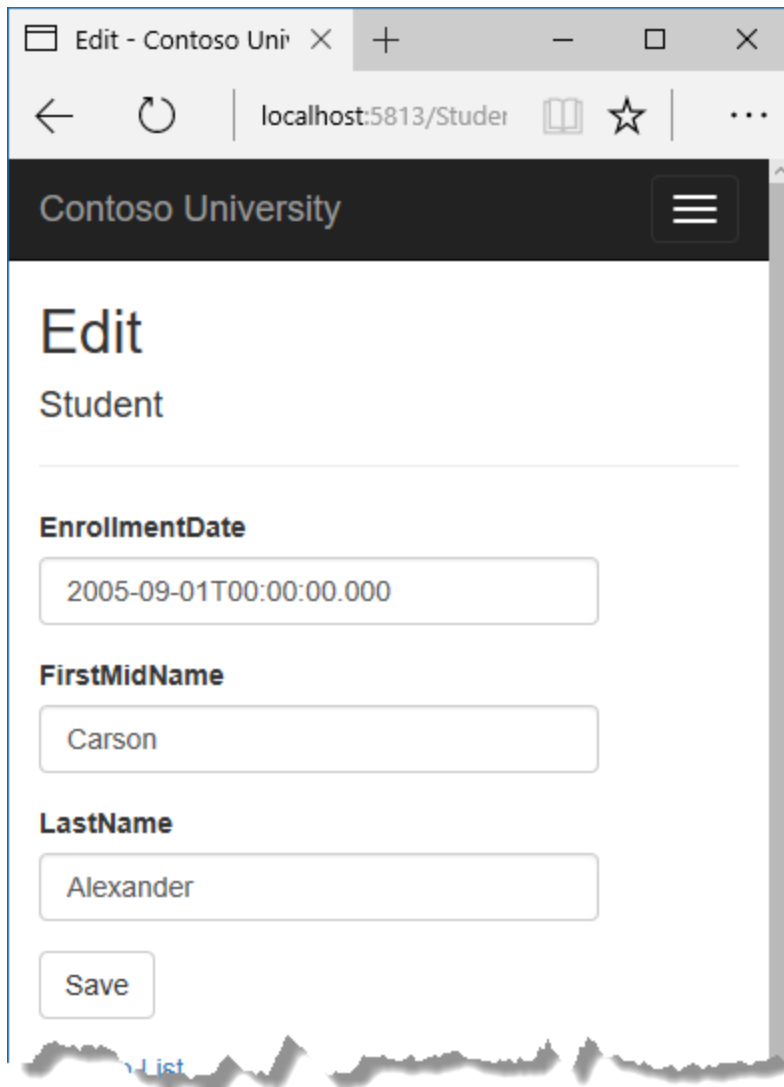
This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

The Contoso University web application

The application you'll be building in these tutorials is a simple university web site.

Users can view and update student, course, and instructor information. Here are a few of the screens you'll create.



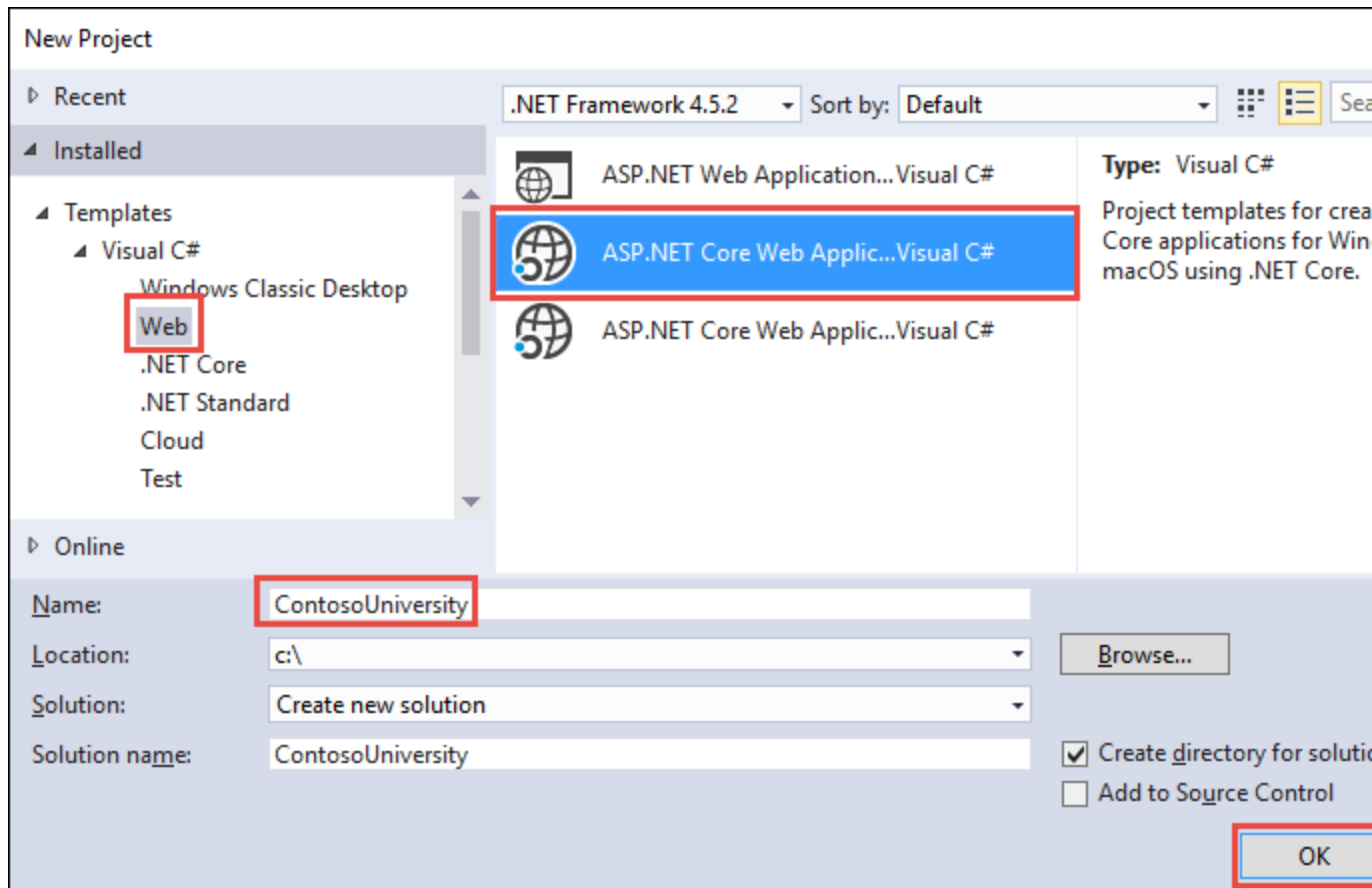


The UI style of this site has been kept close to what's generated by the built-in templates, so that the tutorial can focus mainly on how to use the Entity Framework.

Create an ASP.NET Core MVC web application

Open Visual Studio and create a new ASP.NET Core C# web project named "ContosoUniversity".

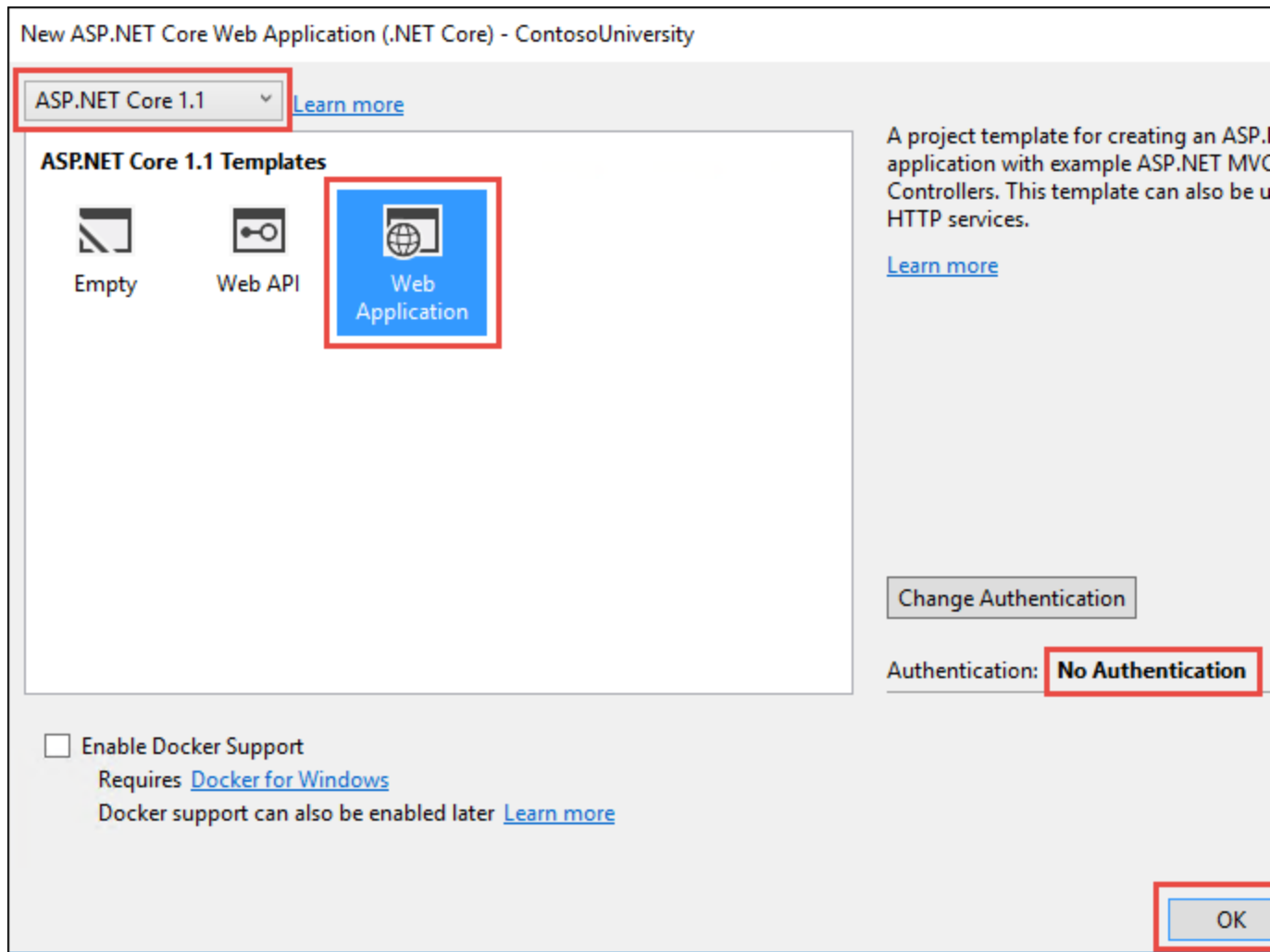
- From the File menu, select New > Project.
- From the left pane, select Templates > Visual C# > Web.
- Select the ASP.NET Core Web Application (.NET Core) project template.
- Enter ContosoUniversity as the name and click OK.



- Wait for the New ASP.NET Core Web Application (.NET Core) dialog to appear
- Select ASP.NET Core 1.1 and the Web Application template.

Note: This tutorial requires ASP.NET Core 1.1 and EF Core 1.1 or later -- make sure that ASP.NET Core 1.0 is not selected.

- Make sure Authentication is set to No Authentication.
- Click OK



Set up the site style

A few simple changes will set up the site menu, layout, and home page.

Open *Views/Shared/_Layout.cshtml* and make the following changes:

- Change each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- Add menu entries for Students, Courses, Instructors, and Departments, and delete the Contact menu entry.

The changes are highlighted.

htmlCopy

```

@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>

    <environment names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
    @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Home" asp-action="Index"
class="navbar-brand">Contoso University</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-
action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-
action="About">About</a></li>

```

```

        <li><a asp-area="" asp-controller="Students" asp-
action="Index">Students</a></li>
        <li><a asp-area="" asp-controller="Courses" asp-
action="Index">Courses</a></li>
        <li><a asp-area="" asp-controller="Instructors" asp-
action="Index">Instructors</a></li>

```

```

        <li><a asp-area="" asp-controller="Departments" asp-
action="Index">Departments</a></li>
    </ul>
</div>
</div>
</nav>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2017 - Contoso University</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
asp-fallback-test="window.jQuery"
crossorigin="anonymous"
integrity="sha384-
K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfHIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
crossorigin="anonymous"
integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNICPD7Txa">
    </script>

```

```

        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>

```

In *Views/Home/Index.cshtml*, replace the contents of the file with the following code to replace the text about ASP.NET and MVC with text about this application:

htmlCopy

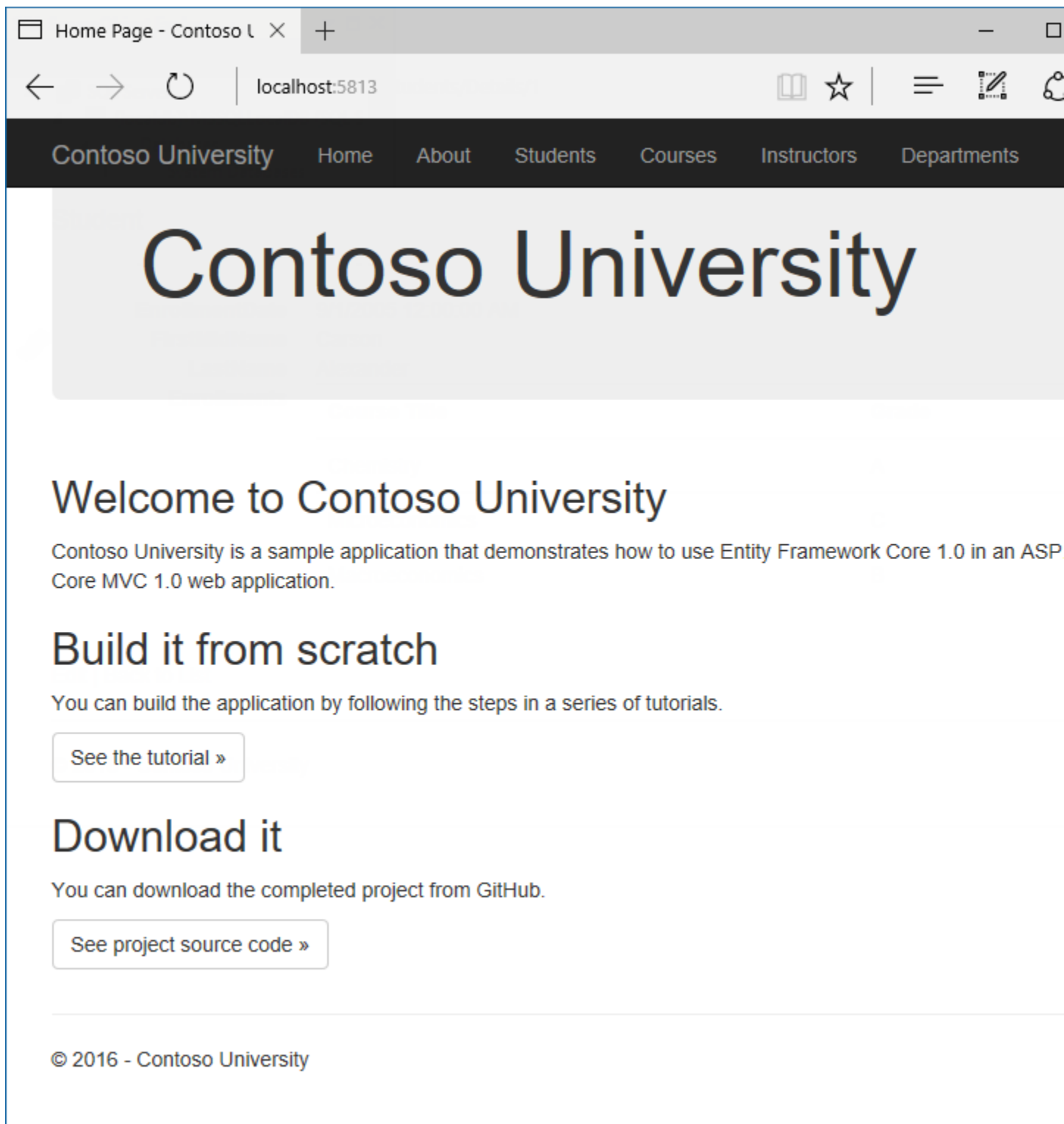
```

@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series of
tutorials.</p>
        <p><a class="btn btn-default" href="https://docs.asp.net/en/latest/data/ef-
mvc/intro.html">See the tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
        <p><a class="btn btn-default"
href="https://github.com/aspnet/Docs/tree/master/aspnetcore/data/ef-
mvc/intro/samples/cu-final">See project source code &raquo;</a></p>
    </div>
</div>

```


Press CTRL+F5 to run the project or choose Debug > Start Without Debugging from the menu. You see the home page with tabs for the pages you'll create in these tutorials.



Entity Framework Core NuGet packages

To add EF Core support to a project, install the database provider that you want to target. For this tutorial, install the SQL Server provider: [Microsoft.EntityFrameworkCore.SqlServer](#).

To install the package, enter the following command in Package Manager Console (PMC). (From the Tools menu, select NuGet Package Manager > Package Manager Console.)

Copy

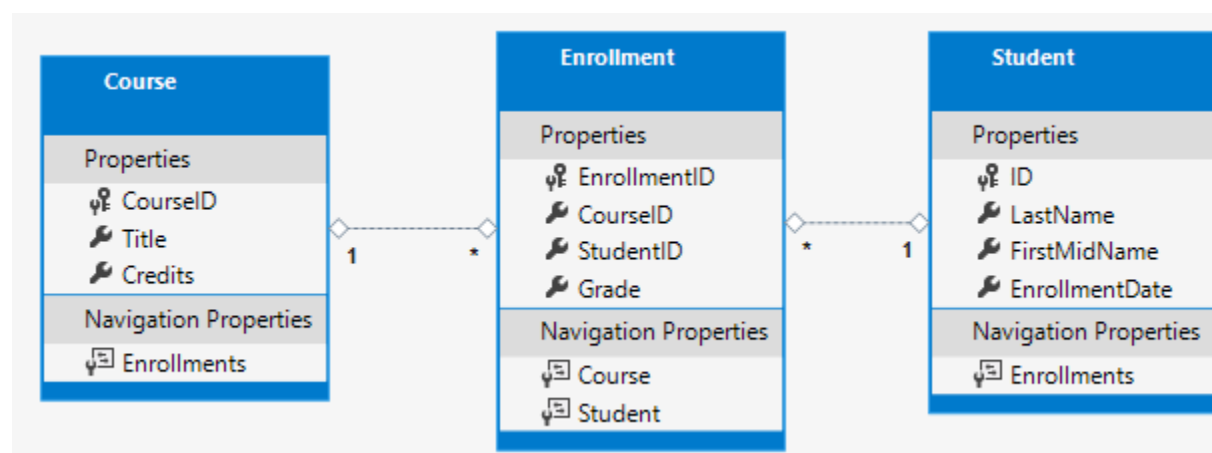
```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

This package and its dependencies (`Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`) provide run-time support for EF. You'll add a tooling package later, in the [Migrations](#) tutorial.

For information about other database providers that are available for Entity Framework Core, see [Database providers](#).

Create the data model

Next you'll create entity classes for the Contoso University application. You'll start with the following three entities.¹



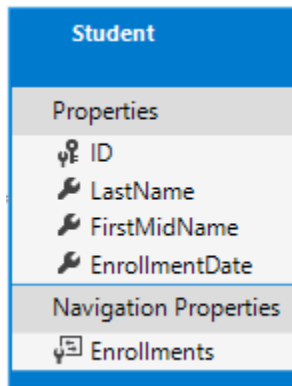
4

There's a one-to-many relationship between `Student` and `Enrollment` entities, and there's a one-to-many relationship between `Course` and `Enrollment` entities. In other

words, a student can be enrolled in any number of courses, and a course can have any number of students enrolled in it.

In the following sections you'll create a class for each one of these entities.

The Student entity



In the project folder, create a folder named *Models*.

In the *Models* folder, create a class file named *Student.cs* and replace the template code with the following code.

C#Copy

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

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }







        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property will become the primary key column of the database table that corresponds to this class. By default, the Entity Framework interprets a property that's named `ID` or `classnameID` as the primary key.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity will hold all of the `Enrollment` entities that are related to that `Student` entity. In other words, if a given `Student` row in the database has two related `Enrollment` rows (rows that contain that student's primary key value in their `StudentID` foreign key column), that `Student` entity's `Enrollments` navigation property will contain those two `Enrollment` entities.

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection<T>`. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

The Enrollment entity

Enrollment	
Properties	
	EnrollmentID
	CourseID
	StudentID
	Grade
Navigation Properties	
	Course
	Student

In the *Models* folder, create *Enrollment.cs* and replace the existing code with the following code:

C#Copy

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
```

```

    A, B, C, D, F
}

public class Enrollment
{
    public int EnrollmentID { get; set; }
    public int CourseID { get; set; }
    public int StudentID { get; set; }
    public Grade? Grade { get; set; }

    public Course Course { get; set; }
    public Student Student { get; set; }
}
}

```

The `EnrollmentID` property will be the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself as you saw in the `Student` entity. Ordinarily you would choose one pattern and use it throughout your data model. Here, the variation illustrates that you can use either pattern. In a [later tutorial](#), you'll see how using `ID` without classname makes it easier to implement inheritance in the data model.

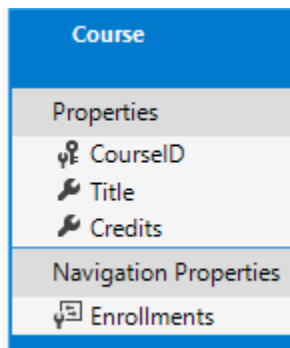
The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade -- null means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity (unlike the `Student.Enrollments` navigation property you saw earlier, which can hold multiple `Enrollment` entities).⁴

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a foreign key property if it's named `<navigation property name><primary key property name>` (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply `<primary key property name>` (for example, `CourseID` since the `Course` entity's primary key is `CourseID`).

The Course entity



In the *Models* folder, create *Course.cs* and replace the existing code with the following code:

C#Copy

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

We'll say more about the `DatabaseGenerated` attribute in a [later tutorial](#) in this series. Basically, this attribute lets you enter the primary key for the course rather than having the database generate it.

Create the Database Context

The main class that coordinates Entity Framework functionality for a given data model is the database context class. You create this class by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. In your code you specify which entities are included in the data model. You can also customize certain Entity Framework behavior. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named *Data*.

In the *Data* folder create a new class file named *SchoolContext.cs*, and replace the template code with the following code:²

C#Copy

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

This code creates a `DbSet` property for each entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.²

You could have omitted the `DbSet<Enrollment>` and `DbSet<Course>` statements and it would work the same. The Entity Framework would include them implicitly because the `Student` entity references the `Enrollment` entity and the `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural (Students rather than Student), but developers disagree about whether table names should be

pluralized or not. For these tutorials you'll override the default behavior by specifying singular table names in the DbContext. To do that, add the following highlighted code after the last DbSet property.

C#Copy

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

```
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the context with dependency injection

ASP.NET Core implements [dependency injection](#) by default. Services (such as the EF database context) are registered with dependency injection during application startup. Components that require these services (such as MVC controllers) are provided these services via constructor parameters. You'll see the controller constructor code that gets a context instance later in this tutorial.³

To register `SchoolContext` as a service, open *Startup.cs*, and add the highlighted lines to the `ConfigureServices` method.

C#Copy

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
```

```
    services.AddDbContext<SchoolContext>(options =>
```

```
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

```
    services.AddMvc();
```

```
}
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the [ASP.NET Core configuration system](#) reads the connection string from the *appsettings.json* file.

Add `using` statements

for `ContosoUniversity.Data` and `Microsoft.EntityFrameworkCore` namespaces, and then build the project.

C#Copy

```
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
```

Open the *appsettings.json* file and add a connection string as shown in the following example.

JSONCopy

```
{
```

```

    "ConnectionStrings": {
        "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;MultipleActiveResultSets=true"
    },
    "Logging": {
        "IncludeScopes": false,
        "LogLevel": {
            "Default": "Warning"
        }
    }
}

```

SQL Server Express LocalDB

The connection string specifies a SQL Server LocalDB database. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for application development, not production use. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB creates *.mdf* database files in the `C:/Users/<user>` directory.

Add code to initialize the database with test data

The Entity Framework will create an empty database for you. In this section, you write a method that is called after the database is created in order to populate it with test data.

Here you'll use the `EnsureCreated` method to automatically create the database. In a [later tutorial](#) you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class file named *DbInitializer.cs* and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

C#Copy

```

using ContosoUniversity.Models;
using System;

```

```

using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;    // DB has been seeded
            }

            var students = new Student[]
            {
                new
                Student{FirstMidName="Carson", LastName="Alexander", EnrollmentDate=DateTime.Parse("2005-09-01")},
                new
                Student{FirstMidName="Meredith", LastName="Alonso", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new
                Student{FirstMidName="Arturo", LastName="Anand", EnrollmentDate=DateTime.Parse("2003-09-01")},
                new
                Student{FirstMidName="Gytis", LastName="Barzdukas", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new
                Student{FirstMidName="Yan", LastName="Li", EnrollmentDate=DateTime.Parse("2002-09-01")},
                new
                Student{FirstMidName="Peggy", LastName="Justice", EnrollmentDate=DateTime.Parse("2001-09-01")},
                new
                Student{FirstMidName="Laura", LastName="Norman", EnrollmentDate=DateTime.Parse("2003-09-01")},
                new
                Student{FirstMidName="Nino", LastName="Olivetto", EnrollmentDate=DateTime.Parse("2005-09-01")}
            };
            foreach (Student s in students)
            {

```

```

        context.Students.Add(s);
    }
    context.SaveChanges();

    var courses = new Course[]
    {
        new Course{CourseID=1050,Title="Chemistry",Credits=3},
        new Course{CourseID=4022,Title="Microeconomics",Credits=3},
        new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
        new Course{CourseID=1045,Title="Calculus",Credits=4},
        new Course{CourseID=3141,Title="Trigonometry",Credits=4},
        new Course{CourseID=2021,Title="Composition",Credits=3},
        new Course{CourseID=2042,Title="Literature",Credits=4}
    };
    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }
    context.SaveChanges();

    var enrollments = new Enrollment[]
    {
        new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
        new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
        new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
        new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
        new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
        new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
        new Enrollment{StudentID=3,CourseID=1050},
        new Enrollment{StudentID=4,CourseID=1050},
        new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
        new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
        new Enrollment{StudentID=6,CourseID=1045},
        new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
    };
    foreach (Enrollment e in enrollments)
    {
        context.Enrollments.Add(e);
    }
    context.SaveChanges();
}
}
}

```

The code checks if there are any students in the database, and if not, it assumes the database is new and needs to be seeded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.

In *Startup.cs*, modify the `Configure` method to call this seed method on application startup. First, add the context to the method signature so that ASP.NET dependency injection can provide it to your `DbInitializer` class.

C#Copy

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory, SchoolContext context)
```

```
{  
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));  
    loggerFactory.AddDebug();  
}
```

Then call your `DbInitializer.Initialize` method at the end of the `Configure` method.

C#Copy

```
app.UseMvc(routes =>  
{  
    routes.MapRoute(  
        name: "default",  
        template: "{controller=Home}/{action=Index}/{id?}");  
});  
  
DbInitializer.Initialize(context);
```

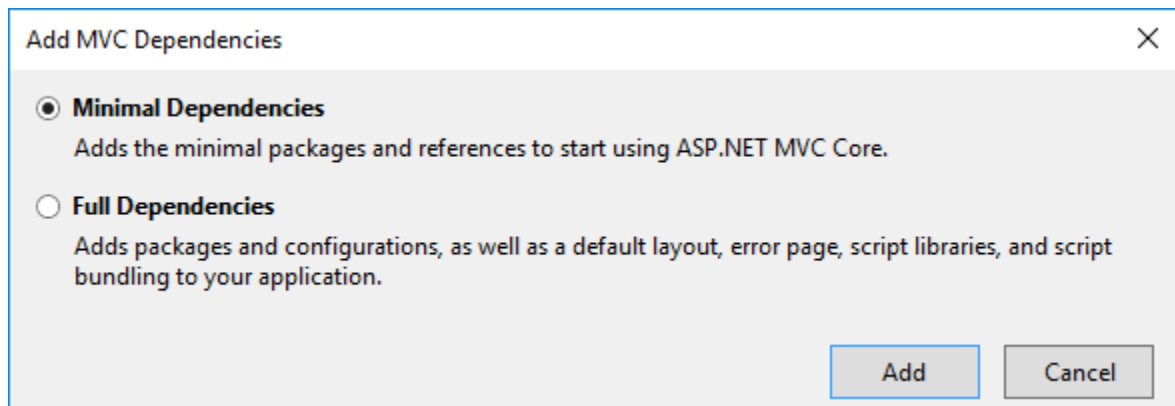
Now the first time you run the application the database will be created and seeded with test data. Whenever you change your data model, you can delete the database, update your seed method, and start afresh with a new database the same way. In later tutorials you'll see how to modify the database when the data model changes, without deleting and re-creating it.

Create a controller and views

Next, you'll use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding. Scaffolding differs from code generation in that the scaffolded code is a starting point that you can modify to suit your own requirements, whereas you typically don't modify generated code. When you need to customize generated code, you use partial classes or you regenerate the code when things change.

- Right-click the **Controllers folder** in Solution Explorer and select **Add > New Scaffolded Item**.
- In the **Add MVC Dependencies dialog**, select **Minimal Dependencies**, and select **Add**.



Visual Studio adds the dependencies needed to scaffold a controller, including a package with design-time EF functionality (`Microsoft.EntityFrameworkCore.Design`). A package that is needed only for scaffolding a DbContext from an existing database is also included (`Microsoft.EntityFrameworkCore.SqlServer.Design`). A *ScaffoldingReadMe.txt* file is created which you can delete.

- Once again, right-click the **Controllers folder** in Solution Explorer and select **Add > New Scaffolded Item**.
- In the **Add Scaffold dialog box**:
 - Select MVC controller with views, using Entity Framework.
 - Click **Add**.
- In the **Add Controller dialog box**:
 - In Model class select **Student**.
 - In Data context class select **SchoolContext**.
 - Accept the default **StudentsController** as the name.
 - Click **Add**.

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Student (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. There are three checked options under 'Views': 'Generate views', 'Reference script libraries', and 'Use a layout page:'. The 'Controller name' text box contains 'StudentsController'. The 'Add' button is highlighted with a red box.

When you click Add, the Visual Studio scaffolding engine creates a *StudentsController.cs* file and a set of views (.cshtml files) that work with the controller.

(The scaffolding engine can also create the database context for you if you don't create it manually first as you did earlier for this tutorial. You can specify a new context class in the Add Controller box by clicking the plus sign to the right of Data context class. Visual Studio will then create your `DbContext` class as well as the controller and views.)

You'll notice that the controller takes a `SchoolContext` as a constructor parameter.⁷

C#Copy

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)

        {
```

```
        _context = context;
    }
```

ASP.NET dependency injection will take care of passing an instance of `SchoolContext` into the controller. You configured that in the *Startup.cs* file earlier.¹

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

C#Copy

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

You'll learn about the asynchronous programming elements in this code later in the tutorial.

The *Views/Students/Index.cshtml* view displays this list in a table:¹

htmlCopy

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.LastName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var student in Model)
        {
            <tr>
                <td>@Html.DisplayFor(model => model.LastName)</td>
                <td>@Html.DisplayFor(model => model.FirstMidName)</td>
            </tr>
        }
    </tbody>
</table>
```



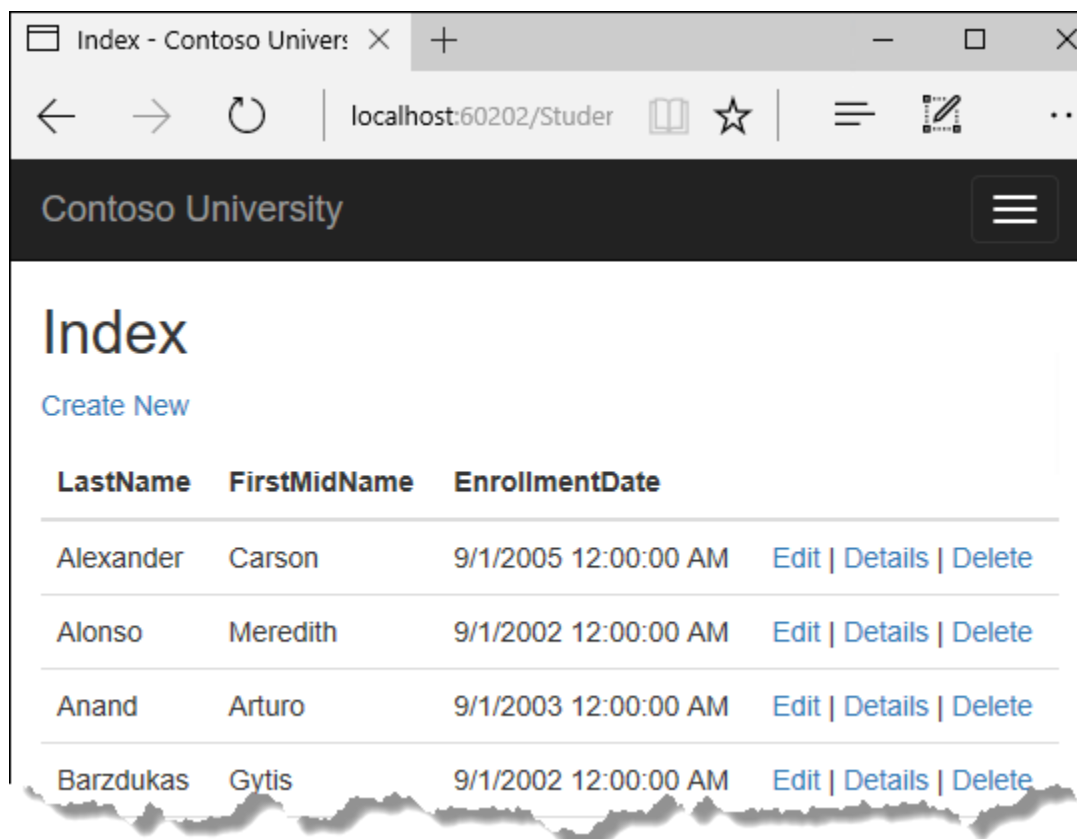
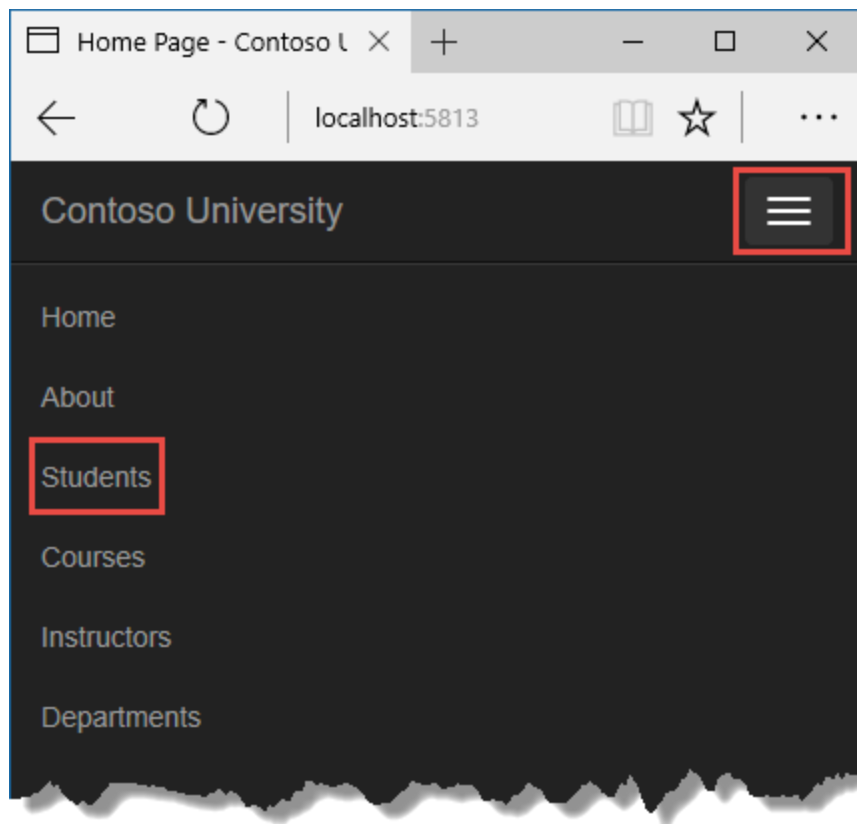
```

        <th>
            @Html.DisplayNameFor(model => model.EnrollmentDate)
        </th>
    </th></th>
</tr>
</thead>
<tbody>
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

Press CTRL+F5 to run the project or choose Debug > Start Without Debugging from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Student` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.



View the Database

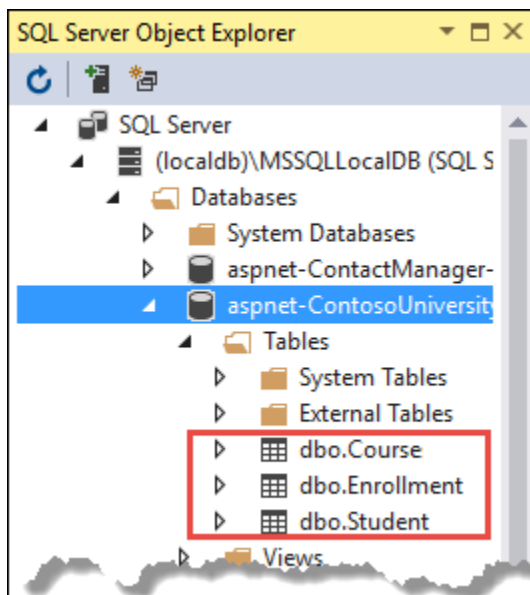
When you started the application, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database and so it created one, then the remainder of the `Initialize` method code populated the database with data. You can use SQL Server Object Explorer (SSOX) to view the database in Visual Studio.

Close the browser.

If the SSOX window isn't already open, select it from the View menu in Visual Studio.

In SSOX, click `(localdb)\MSSQLLocalDB > Databases`, and then click the entry for the database name that is in the connection string in your *appsettings.json* file.

Expand the Tables node to see the tables in your database.



Right-click the Student table and click View Data to see the columns that were created and the rows that were inserted into the table.

ID	EnrollmentDate	FirstMidName	LastName
1	9/1/2005 12:00:...	Carson	Alexander
2	9/1/2002 12:00:...	Meredith	Alonso
3	9/1/2003 12:00:...	Arturo	Anand
4	9/1/2002 12:00:...	Gytis	Barzdukas
5	9/1/2002 12:00:...	Yan	Li

The `.mdf` and `.ldf` database files are in the `C:\Users` folder.

Because you're calling `EnsureCreated` in the initializer method that runs on app start, you could now make a change to the `Student` class, delete the database, run the application again, and the database would automatically be re-created to match your change. For example, if you add an `EmailAddress` property to the `Student` class, you'll see a new `EmailAddress` column in the re-created table.

Conventions

The amount of code you had to write in order for the Entity Framework to be able to create a complete database for you is minimal because of the use of conventions, or assumptions that the Entity Framework makes.

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classnameID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named (for example, `StudentID` for the `Student` navigation property since the `Student` entity's primary key is `ID`). Foreign key properties can also be named simply (for example, `EnrollmentID` since the `Enrollment` entity's primary key is `EnrollmentID`).

Conventional behavior can be overridden. For example, you can explicitly specify table names, as you saw earlier in this tutorial. And you can set column names and set any property as primary key or foreign key, as you'll see in a [later tutorial](#) in this series.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the `async` keyword, `Task<T>` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

C#Copy

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that is returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that is started asynchronously. The second part is put into a callback method that is called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when you are writing asynchronous code that uses the Entity Framework:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It does not include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF context is not thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- If you want to take advantage of the performance benefits of async code, make sure that any library packages that you're using (such as for paging), also use async if they call any Entity Framework methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see [Async Overview](#).

Summary

You've now created a simple application that uses the Entity Framework Core and SQL Server Express LocalDB to store and display data. In the following tutorial, you'll learn how to perform basic CRUD (create, read, update, delete) operations.

Create, Read, Update, and Delete - EF Core with ASP.NET Core MVC tutorial (2 of 10)

2017-3-15 19 min to read Contributors

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

Note

It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see [the last tutorial in this series](#).

In this tutorial, you'll work with the following web pages:

Details - Contoso Unive

localhost:5813/Students/Details/1

Contoso University

Home

About

Students

Courses

Instructors

Departments

Register

Lo

Student

LastName

FirstMidName

EnrollmentDate

Enrollments

Alexander
Carson
9/1/2005 12:00:00 AM

Course Title	Grade
Chemistry	A
Microeconomics	C
Macroeconomics	B

Create ×

+

−


□

×

←

↺

| s/Create

| 

| ☆

| ...

Contoso University

≡

Create

Student

LastName

FirstMidName

EnrollmentDate

Create

Edit - C ×

+



—

□


×

←

↺

localhost:  

...

Contoso University 

Edit

Student

LastName

Alexander

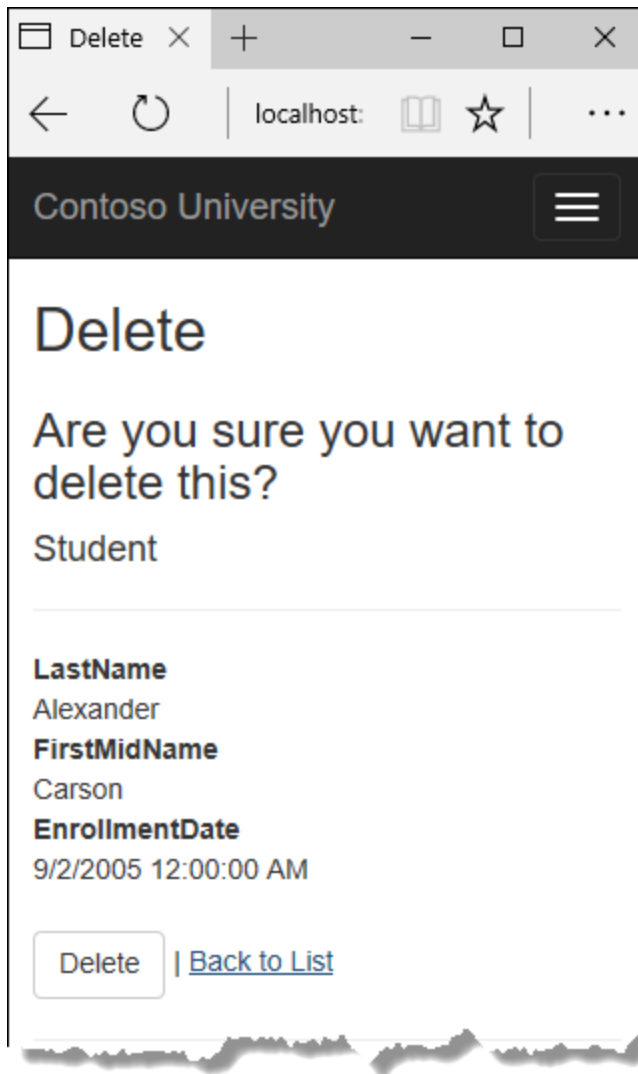
FirstMidName

Carson

EnrollmentDate

2005-09-01T00:00:00.000

Save



Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the Details page you'll display the contents of the collection in an HTML table.

In *Controllers/StudentsController.cs*, the action method for the Details view uses the `SingleOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls `Include`, `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

C#Copy

```
public async Task<IActionResult> Details(int? id)
```

```
{
    if (id == null)
    {
        return NotFound();
    }
}
```

```
var student = await _context.Students
    .Include(s => s.Enrollments)
    .ThenInclude(e => e.Course)
    .AsNoTracking()
```

```
    .SingleOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the [reading related data](#) tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned will not be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that is passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:1

C#Copy

```
app.UseMvc(routes =>
{
```

```
routes.MapRoute(  
    name: "default",  
    template: "{controller=Home}/{action=Index}/{id?}");  
});  
  
DbInitializer.Initialize(context);
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

Copy

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Details` method `id` parameter if you pass it as a query string value:

Copy

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

htmlCopy

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

htmlCopy

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

htmlCopy

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

htmlCopy

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see [Tag helpers in ASP.NET Core](#).

Add enrollments to the Details view

Open *Views/Students/Details.cshtml*. Each field is displayed using `DisplayNameFor` and `DisplayFor` helper, as shown in the following example:

htmlCopy

```
<dt>
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd>
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

htmlCopy

```
<dt>
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd>
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

```
                </td>
            </tr>
        }
    </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the application, select the `Students` tab, and click the `Details` link for a student. You see the list of courses and grades for the selected student:4

Details - Contoso Unive X +

localhost:5813/Students/Details/1

Contoso University Home About Students Courses Instructors Departments Register

Student

LastName	Alexander								
FirstMidName	Carson								
EnrollmentDate	9/1/2005 12:00:00 AM								
Enrollments									
	<table><thead><tr><th>Course Title</th><th>Grade</th></tr></thead><tbody><tr><td>Chemistry</td><td>A</td></tr><tr><td>Microeconomics</td><td>C</td></tr><tr><td>Macroeconomics</td><td>B</td></tr></tbody></table>	Course Title	Grade	Chemistry	A	Microeconomics	C	Macroeconomics	B
Course Title	Grade								
Chemistry	A								
Microeconomics	C								
Macroeconomics	B								

1

Update the Create page

In *StudentsController.cs*, modify the `HttpPost Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

C#Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
```

```
    try
```



```

    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction("Index");
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}

```

This code adds the Student entity created by the ASP.NET MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the ID value.²

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the Log for insight section in [Monitoring and Telemetry \(Building Real-World Cloud Apps with Azure\)](#).

The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the [FormTagHelper](#) and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information about CSRF, see [Anti-Request Forgery](#).

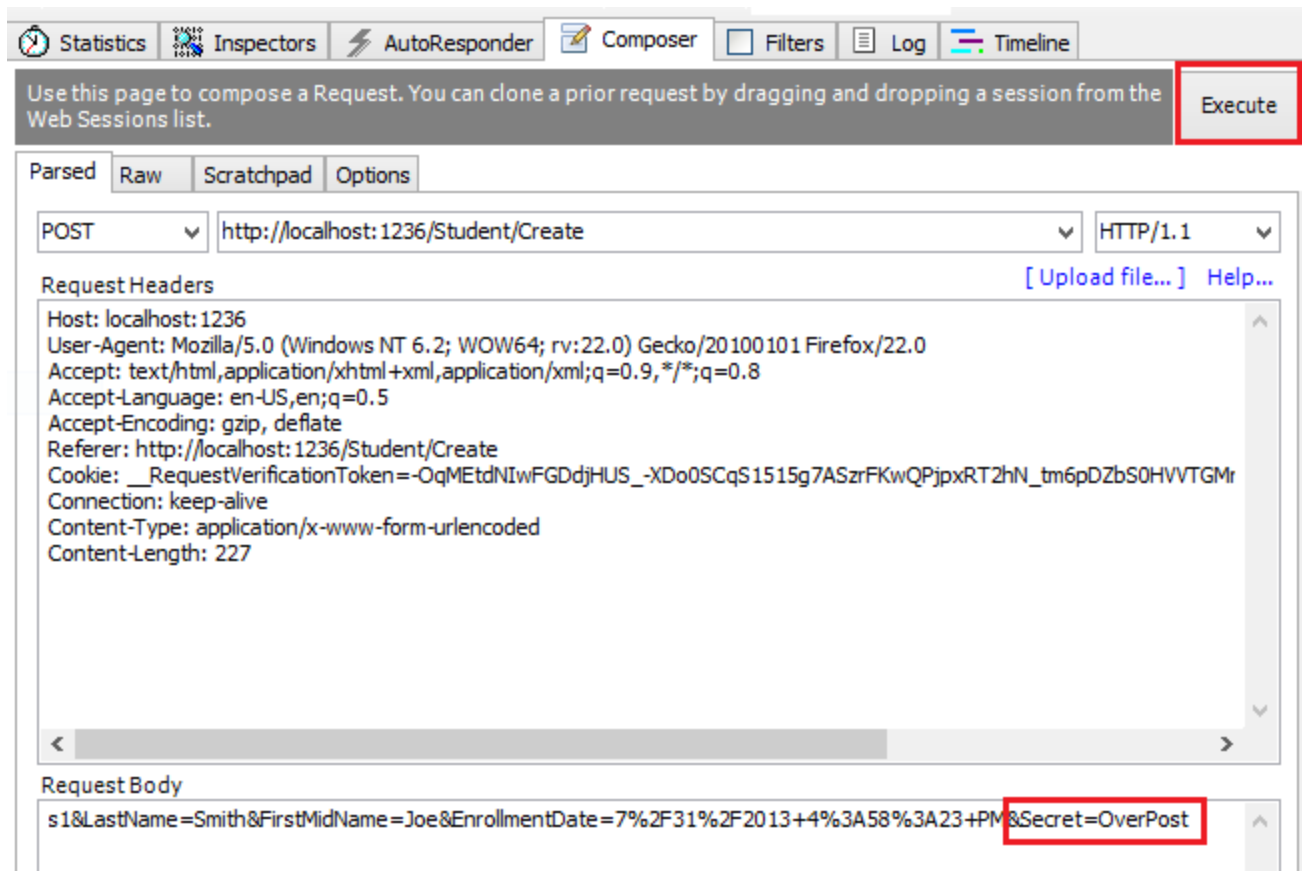
Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the `Student` entity includes a `Secret` property that you don't want this web page to set.

C#Copy

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a `Student` instance, the model binder would pick up that `Secret` form value and use it to create the `Student` entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That is the method used in these tutorials.

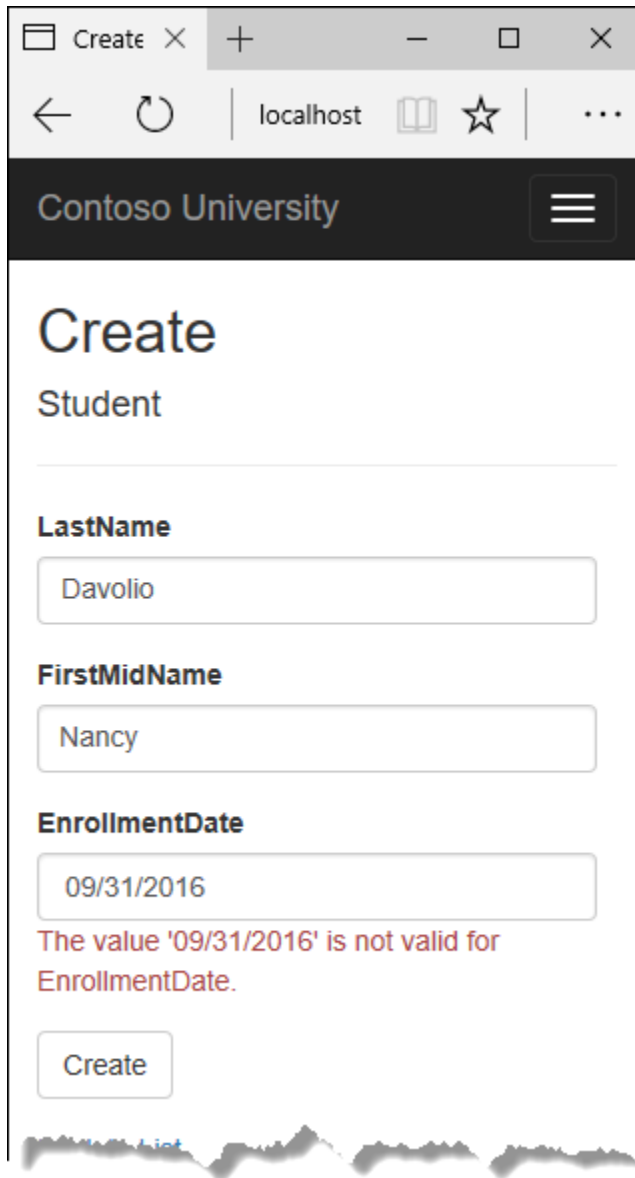
An alternative way to prevent overposting that is preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that is included in the view model. This method works in both edit and create scenarios.³

Test the Create page

The code in *Views/Students/Create.cshtml* uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the page by selecting the **Students** tab and clicking **Create New**.

Enter names and an invalid date and click **Create** to see the error message.



The screenshot shows a web browser window with the address bar at 'localhost'. The page title is 'Contoso University'. The main heading is 'Create Student'. Below this, there are three input fields: 'LastName' with the value 'Davolio', 'FirstMidName' with the value 'Nancy', and 'EnrollmentDate' with the value '09/31/2016'. Below the 'EnrollmentDate' field, a red error message is displayed: 'The value '09/31/2016' is not valid for EnrollmentDate.' At the bottom of the form is a 'Create' button. A 'Cancel' link is visible at the bottom left of the page.

This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.¹

C#Copy

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction("Index");
        }
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}

```

Change the date to a valid value and click Create to see the new student appear in the Index page.

Update the Edit page

In *StudentController.cs*, the `HttpGet Edit` method (the one without the `HttpPost` attribute) uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.¹²

C#Copy

```

[HttpPost, ActionName("Edit")]

```

```

[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.SingleOrDefaultAsync(s => s.ID ==
id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction("Index");
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}

```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code is not recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.¹

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity [based on user input in the posted form data](#). The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the

table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the Edit page are whitelisted in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the `HttpPost Edit` method is the same as the `HttpGet Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended `HttpPost` edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)⁸

C#Copy

```
public async Task<IActionResult> Edit(int id,
[Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student student)
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction("Index");
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
```

```

        "Try again, and if the problem persists, " +
        "see your system administrator.");
    }
}
return View(student);
}

```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity does not yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.
- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

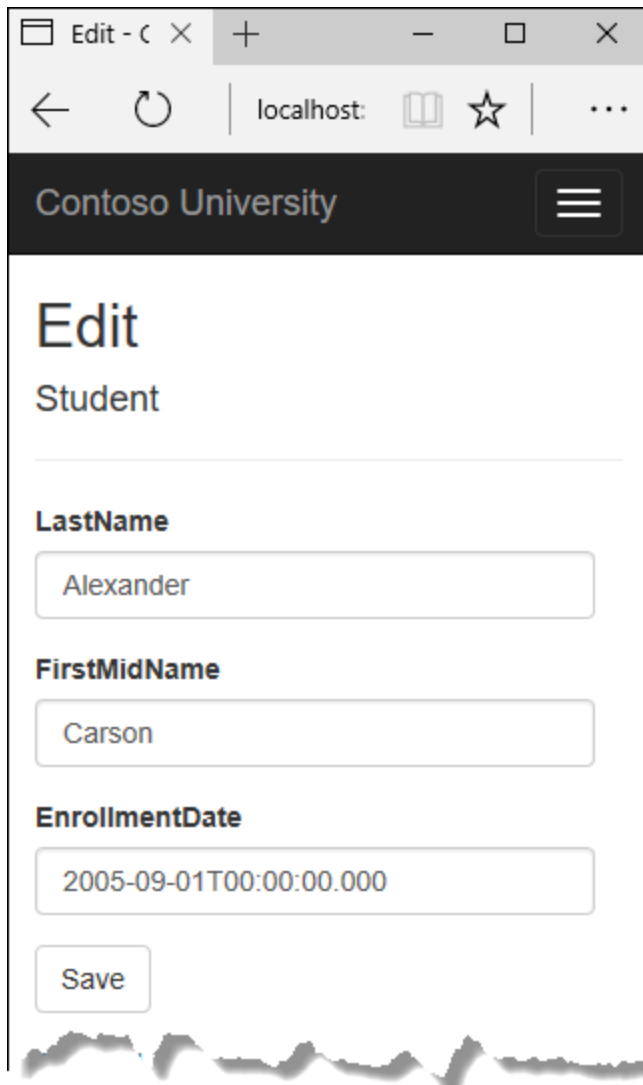
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the `HttpPost Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to `Modified` as is done in the alternative `HttpPost Edit` code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.²

If you want to avoid the read-first approach, but you also want the SQL `UPDATE` statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they are available when the `HttpPost Edit` method is called. Then you can create a `Student` entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the application and select the `Students` tab, then click an `Edit` hyperlink.



Browser window showing the 'Edit Student' page. The page displays the following information:

- Page Title: Edit Student
- Form Fields:
 - LastName: Alexander
 - FirstMidName: Carson
 - EnrollmentDate: 2005-09-01T00:00:00.000
- Buttons: Save

Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In *StudentController.cs*, the template code for the `HttpGet Delete` method uses the `SingleOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that

gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the `HttpPost Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the `HttpPost Delete` method calls the `HttpGet Delete` method, passing it a parameter that indicates that an error has occurred. The `HttpGet Delete` method then redisplay the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the `HttpGet Delete` action method with the following code, which manages error reporting.

C#Copy

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError = false)
```

```
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }
}
```

```
    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }
}
```

```
return View(student);
```

```
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the `HttpGet Delete` method is called without a previous failure. When it is called by the `HttpPost Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to `HttpPost Delete`

Replace the `HttpPost Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.⁴

C#Copy

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
```

```
    if (student == null)
    {
        return RedirectToAction("Index");
    }
```

```
    try
```

```
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
```

```

        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            return RedirectToAction("Delete", new { id = id, saveChangesError = true });
        }
    }
}

```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a `Student` entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

C#Copy

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {

```

```

        Student studentToDelete = new Student() { ID = id };

```

```

        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction("Delete", new { id = id, saveChangesError = true });
    }
}

```

If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

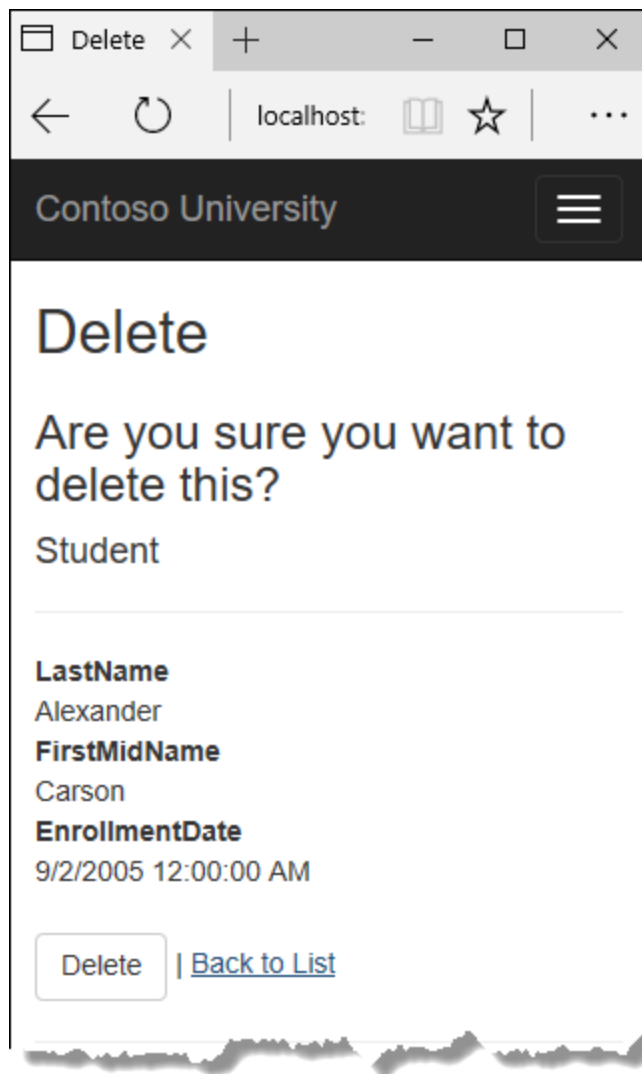
Update the Delete view

In *Views/Student/Delete.cshtml*, add an error message between the h2 heading and the h3 heading, as shown in the following example:

htmlCopy

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the page by selecting the Students tab and clicking a Delete hyperlink:



Click Delete. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Closing database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in [dependency injection](#) takes care of that task for you.

In *Startup.cs* you call the [AddDbContext extension method](#) to provision the `DbContext` class in the ASP.NET DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web

request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handling Transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Transactions](#).

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to [automatically load navigation properties with entities retrieved by separate queries](#). Frequently these conditions are met in a controller's `HttpGet` action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see [Tracking vs. No-Tracking](#).

Summary

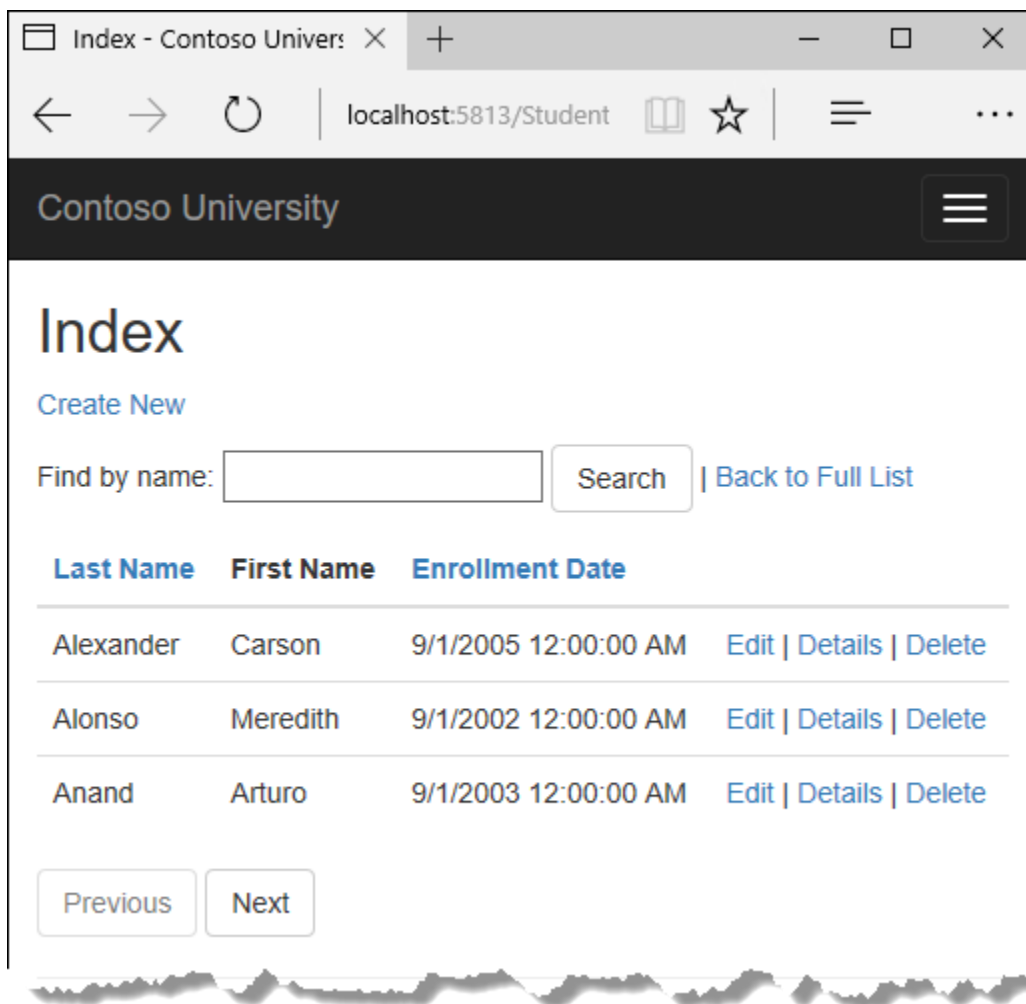
You now have a complete set of pages that perform simple CRUD operations for Student entities. In the next tutorial you'll expand the functionality of the `Index` page by adding sorting, filtering, and paging.

Sorting, filtering, paging, and grouping - EF Core with ASP.NET Core MVC tutorial (3 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.11 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



Add Column Sort Links to the Students Index Page

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code to the Student Index view.

Add sorting Functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code:

C#Copy

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (`NameSortParm` and `DateSortParm`) are used by the view to configure the column heading hyperlinks with the appropriate query string values.³

C#Copy

```
public async Task<IActionResult> Index(string sortOrder)
{
```

```
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
```

```
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                    select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, `NameSortParm` should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:¹

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query is not executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that is not executed until the `return View` statement.

This code could get verbose with a large number of columns. [The last tutorial in this series](#) shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in *Views/Students/Index.cshtml*, with the following code to add column heading hyperlinks. The changed lines are highlighted.

htmlCopy

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>
```

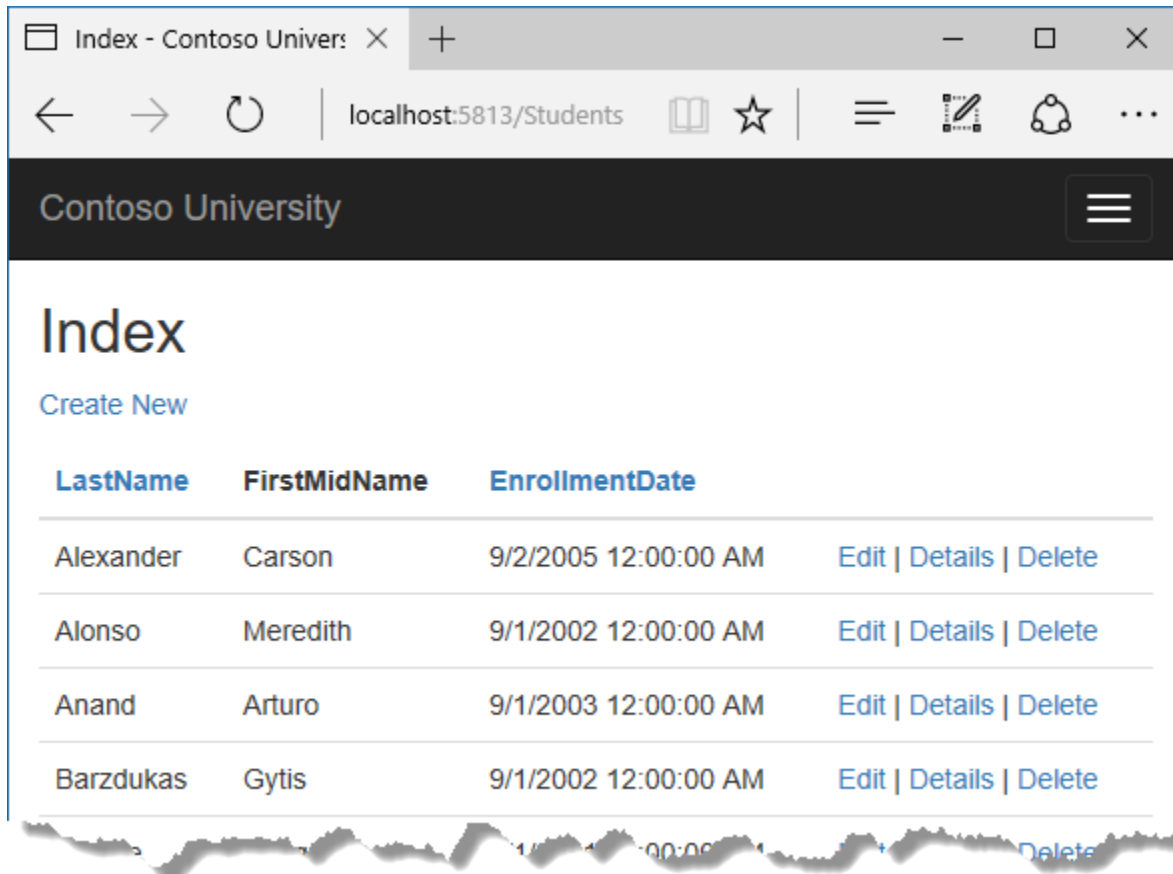
```

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model =>
model.LastName)</a>
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FirstMidName)
            </th>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model =>
model.EnrollmentDate)</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the page and click the Last Name and Enrollment Date column headings to verify that sorting works.



Contoso University

Index

[Create New](#)

LastName	FirstMidName	EnrollmentDate	
Alexander	Carson	9/2/2005 12:00:00 AM	Edit Details Delete
Alonso	Meredith	9/1/2002 12:00:00 AM	Edit Details Delete
Anand	Arturo	9/1/2003 12:00:00 AM	Edit Details Delete
Barzdukas	Gytis	9/1/2002 12:00:00 AM	Edit Details Delete

Add a Search Box to the Students Index page

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code (the changes are highlighted).

C#Copy

```

public async Task<IActionResult> Index(string sortOrder, string searchString)

{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                    select s;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                   || s.FirstMidName.Contains(searchString));
    }

    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}

```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

Note

Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper()))*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the .NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there is a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

In *Views/Student/Index.cshtml*, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a Searchbutton.4
htmlCopy

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

```
<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
```

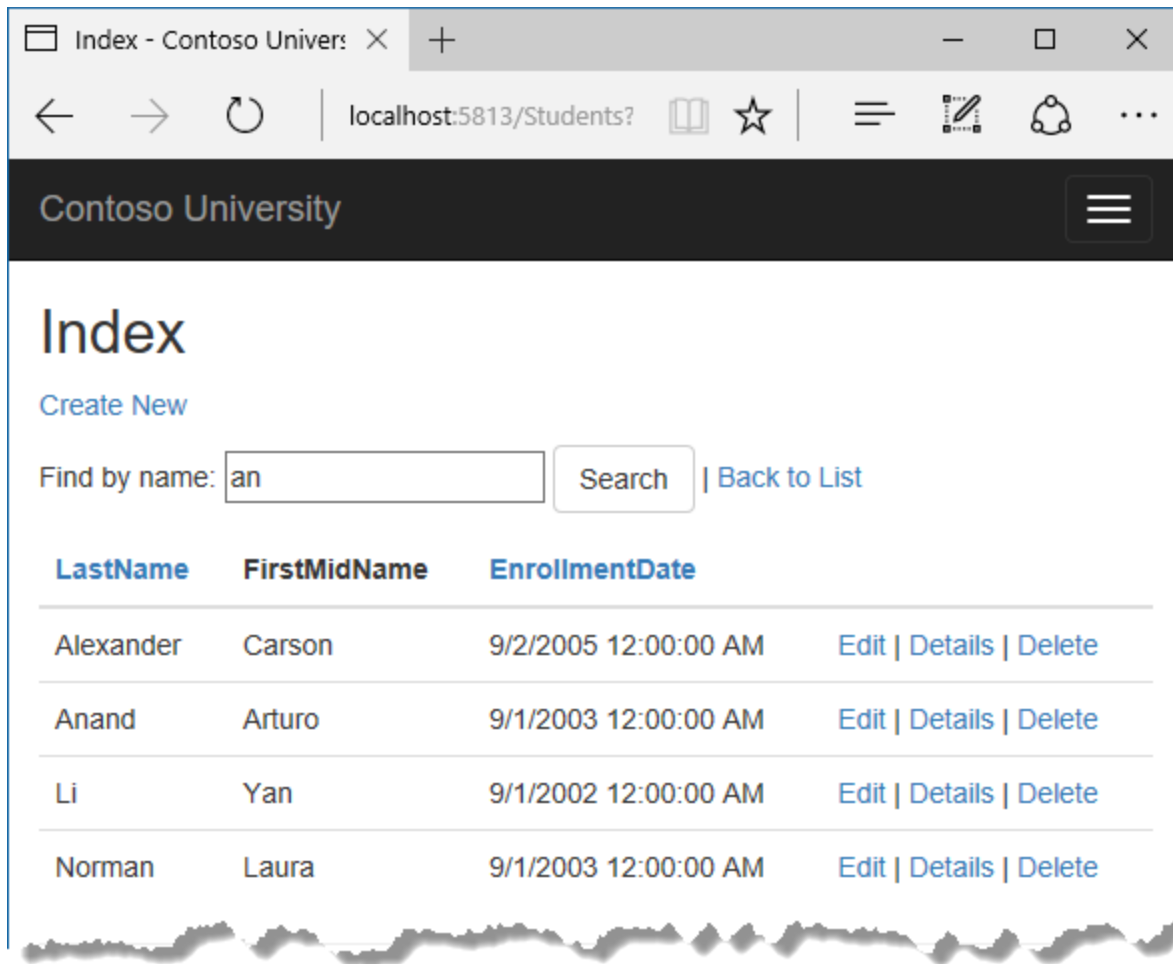
```
Find by name: <input type="text" name="SearchString"
value="@ViewData["currentFilter"]" />
<input type="submit" value="Search" class="btn btn-default" /> |
<a asp-action="Index">Back to Full List</a>
</p>
</div>
```

```
</form>
```

```
<table class="table">
```

This code uses the `<form>` [tag helper](#) to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action does not result in an update.

Run the page, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

htmlCopy

```
http://localhost:5813/Students?SearchString=an
```

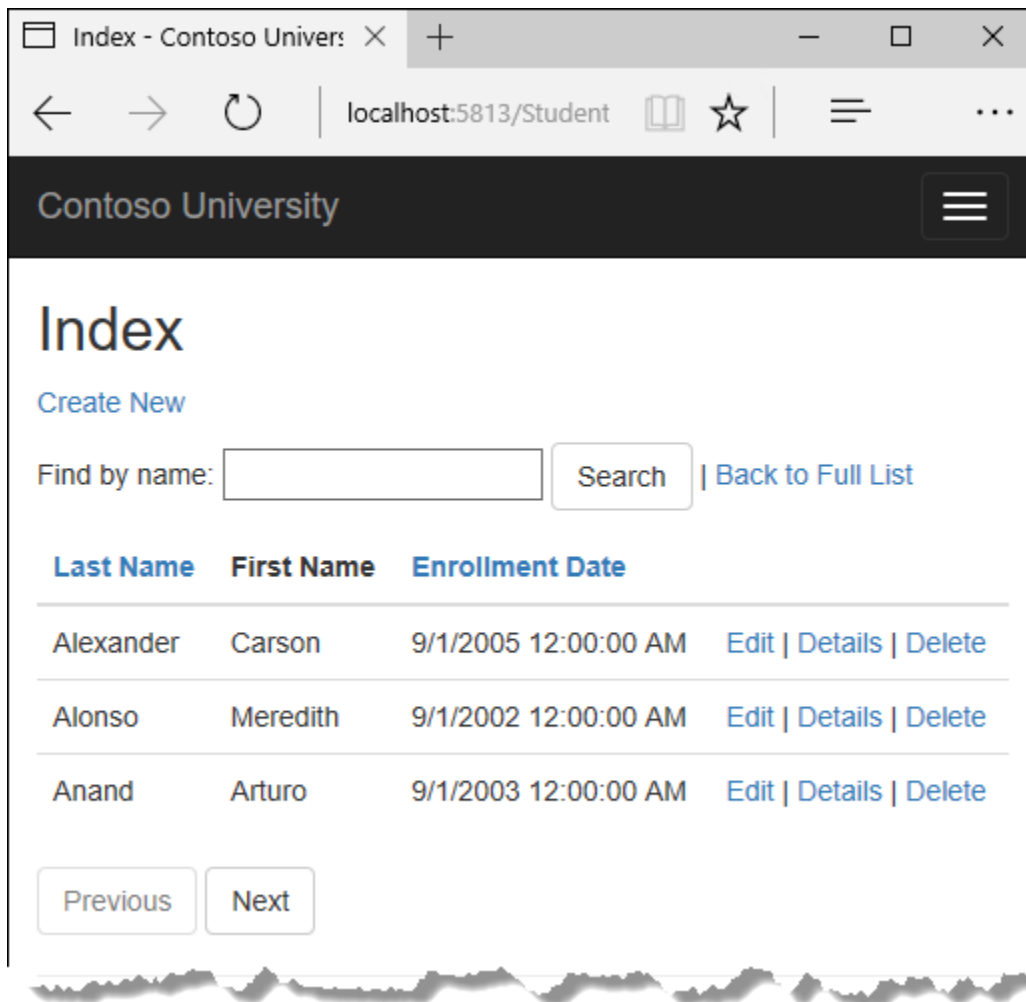
If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the Search box. You'll fix that in the next section.

Add paging functionality to the Students Index page

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving

all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder create `PaginatedList.cs`, and then replace the template code with the following code.

C#Copy

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

public class PaginatedList<T> : List<T>
{
    public int PageIndex { get; private set; }
}
```

```

public int TotalPages { get; private set; }

public PaginatedList(List<T> items, int count, int pageIndex, int pageSize)
{
    PageIndex = pageIndex;
    TotalPages = (int)Math.Ceiling(count / (double)pageSize);

    this.AddRange(items);
}

public bool HasPreviousPage
{
    get
    {
        return (PageIndex > 1);
    }
}

public bool HasNextPage
{
    get
    {
        return (PageIndex < TotalPages);
    }
}

public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T> source, int
pageIndex, int pageSize)
{
    var count = await source.CountAsync();
    var items = await source.Skip((pageIndex - 1) *
pageSize).Take(pageSize).ToListAsync();
    return new PaginatedList<T>(items, count, pageIndex, pageSize);
}
}

```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable Previous and Next paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging functionality to the Index method

In *StudentsController.cs*, replace the `Index` method with the following code.

C#Copy

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";

    if (searchString != null)
    {
        page = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                                     || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
```

```

{
    case "name_desc":
        students = students.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        students = students.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        students = students.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        students = students.OrderBy(s => s.LastName);
        break;
}

```

```
int pageSize = 3;
```

```

return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
page ?? 1, pageSize));
}

```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

C#Copy

```

public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? page)

```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named `CurrentSort` provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named `CurrentFilter` provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.¹

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter is not null.

C#Copy

```
if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PagedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

C#Copy

```
return View(await PagedList<Student>.CreateAsync(students.AsNoTracking(), page ?? 1, pageSize));
```

The `PagedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(page ?? 1)` means return the value of `page` if it has a value, or return 1 if `page` is null.

Add paging links to the Student Index view

In `Views/Students/Index.cshtml`, replace the existing code with the following code. The changes are highlighted.⁷

htmlCopy


```

@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

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

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString"
value="@ViewData["currentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["NameSortParm"]"
asp-route-currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]"
asp-route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>

```

```

        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.FirstMidName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EnrollmentDate)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

```

@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex + 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @nextDisabled">
    Next

```

```

</a>

```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

htmlCopy

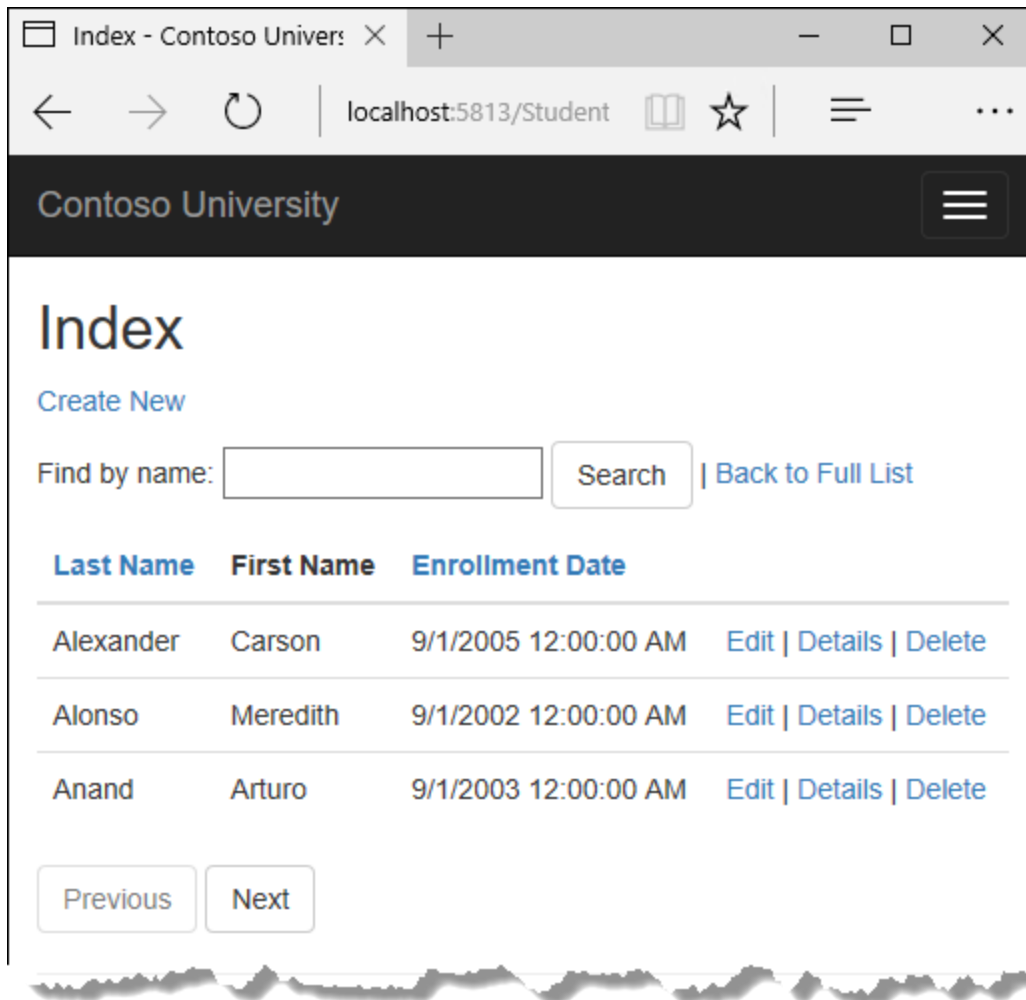
```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-route-currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

htmlCopy

```
<a asp-action="Index"
    asp-route-sortOrder="@ViewData["CurrentSort"]"
    asp-route-page="@((Model.PageIndex - 1))"
    asp-route-currentFilter="@ViewData["CurrentFilter"]"
    class="btn btn-default @prevDisabled btn">
    Previous
</a>
```

Run the page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.³

Create an About page that shows Student statistics

For the Contoso University website's About page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Modify the About method in the Home controller.
- Modify the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file *EnrollmentDateGroup.cs* and replace the template code with the following code:

C#Copy

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In *HomeController.cs*, add the following using statements at the top of the file:

C#Copy

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

C#Copy

```
public class HomeController : Controller
{
    private readonly SchoolContext _context;
```

```
public HomeController(SchoolContext context)
```

```
{  
    _context = context;  
}
```

Replace the `About` method with the following code:

C#Copy

```
public async Task<ActionResult> About()  
{  
    IQueryable<EnrollmentDateGroup> data =  
        from student in _context.Students  
        group student by student.EnrollmentDate into dateGroup  
        select new EnrollmentDateGroup()  
        {  
            EnrollmentDate = dateGroup.Key,  
            StudentCount = dateGroup.Count()  
        };  
    return View(await data.AsNoTracking().ToListAsync());  
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Note

In the 1.0 version of Entity Framework Core, the entire result set is returned to the client, and grouping is done on the client. In some scenarios this could create performance problems. Be sure to test performance with production volumes of data, and if necessary use raw SQL to do the grouping on the server. For information about how to use raw SQL, see [the last tutorial in this series](#).²

Modify the About View

Replace the code in the `Views/Home/About.cshtml` file with the following code:

htmlCopy

```

@model IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

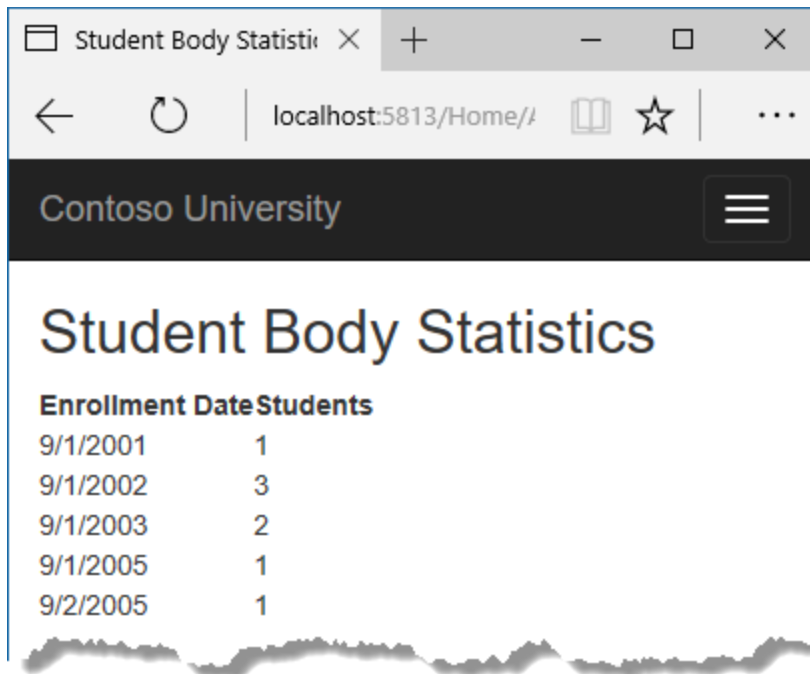
<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

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

```

Run the app and click the About link. The count of students for each enrollment date is displayed in a table.



Summary

In this tutorial you've seen how to perform sorting, filtering, paging, and grouping. In the next tutorial you'll learn how to handle data model changes by using migrations.

Migrations - EF Core with ASP.NET Core MVC tutorial (4 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

Introduction to migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

Entity Framework Core NuGet packages for migrations

To work with migrations, you can use the Package Manager Console (PMC) or the command-line interface (CLI). These tutorials show how to use CLI commands. Information about the PMC is at [the end of this tutorial](#).

The EF tools for the command-line interface (CLI) are provided in [Microsoft.EntityFrameworkCore.Tools.DotNet](#). To install this package, add it to the `DotNetCliToolReference` collection in the `.csproj` file, as shown. Note: You have to install this package by editing the `.csproj` file; you can't use the `install-`

`package` command or the package manager GUI. You can edit the `.csproj` file by right-clicking the project name in Solution Explorer and selecting Edit `ContosoUniversity.csproj`.

XMLCopy

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
Version="1.0.0" />
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
Version="1.0.0" />
</ItemGroup>
```

(The version numbers in this example were current when the tutorial was written.)2

Change the connection string

In the `appsettings.json` file, change the name of the database in the connection string to `ContosoUniversity2` or some other name that you haven't used on the computer you're using.

JSONCopy

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

This change sets up the project so that the first migration will create a new database. This isn't required for getting started with migrations, but you'll see later why it's a good idea.

Note

As an alternative to changing the database name, you can delete the database. Use SQL Server Object Explorer (SSOX) or the `database drop` CLI command:2

consoleCopy

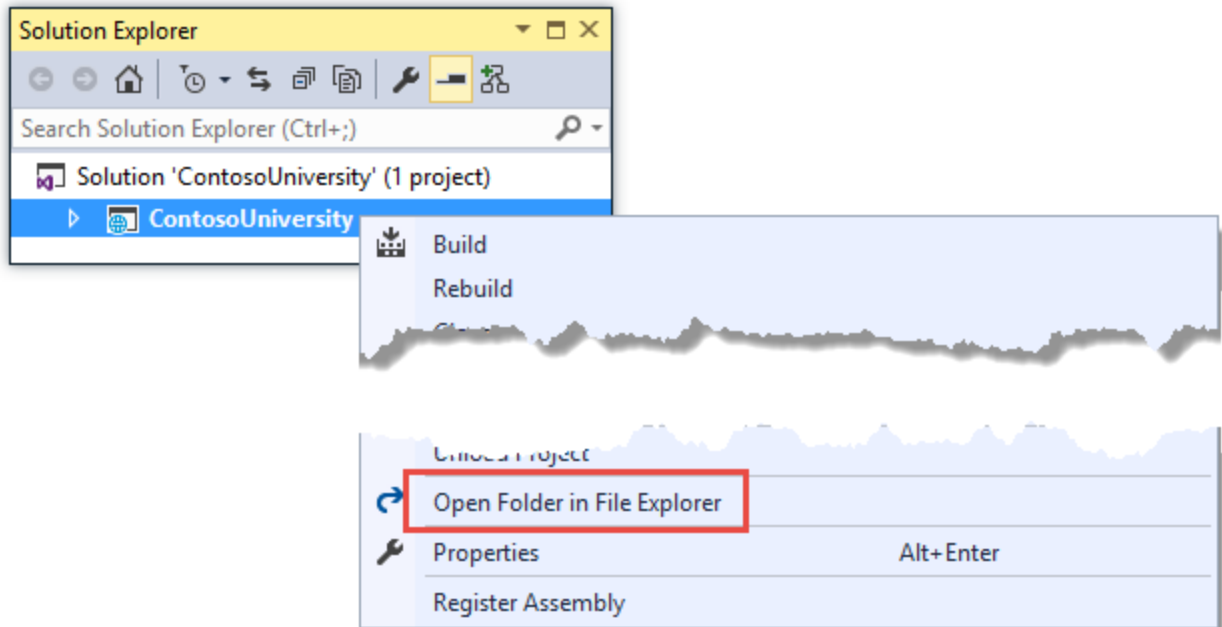
```
dotnet ef database drop
```

The following section explains how to run CLI commands.

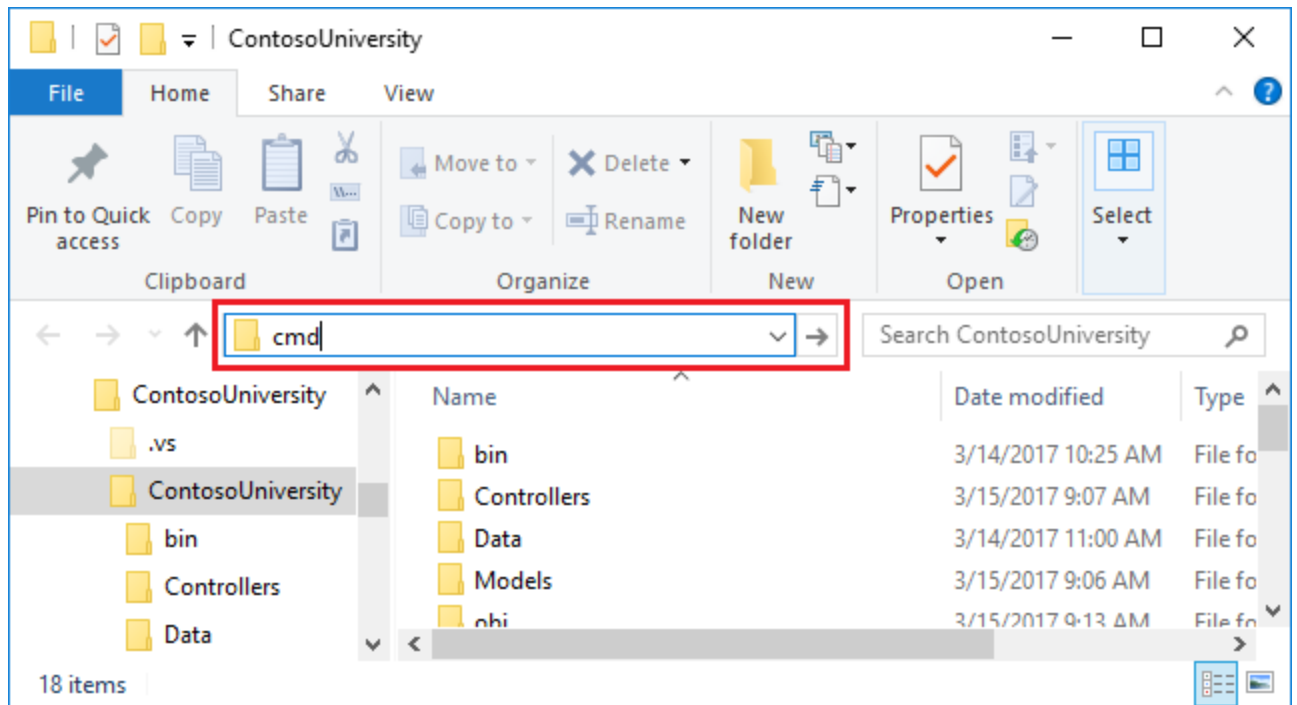
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In Solution Explorer, right-click the project and choose Open in File Explorer from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:2

```
consoleCopy
```

```
dotnet ef migrations add InitialCreate
```

You see output like the following in the command window:

```
consoleCopy
```

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

Time Elapsed 00:00:15.63
Done. To undo this action, use 'ef migrations remove'
```

Note

If you see an error message *No executable found matching command "dotnet-ef"*, see [this blog post](#) for help troubleshooting.

If you see an error message *"cannot access the file ... ContosoUniversity.dll because it is being used by another process."*, find the IIS Express icon in the Windows System Tray, and right-click it, then click ContosoUniversity > Stop Site.

Examine the Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

C#Copy

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Student",
            columns: table => new
            {
                ID = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                EnrollmentDate = table.Column<DateTime>(nullable: false),
                FirstMidName = table.Column<string>(nullable: true),
                LastName = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Student", x => x.ID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Course");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

Examine the data model snapshot

Migrations also creates a *snapshot* of the current database schema in `Migrations/SchoolContextModelSnapshot.cs`. Here's what that code looks like:

C#Copy

```
[DbContext(typeof(SchoolContext))]  
partial class SchoolContextModelSnapshot : ModelSnapshot  
{  
    protected override void BuildModel(ModelBuilder modelBuilder)  
    {  
        modelBuilder  
            .HasAnnotation("ProductVersion", "1.1.1")  
            .HasAnnotation("SqlServer:ValueGenerationStrategy",  
SqlServerValueGenerationStrategy.IdentityColumn);  
  
        modelBuilder.Entity("ContosoUniversity.Models.Course", b =>  
            {  
                b.Property<int>("CourseID");  
  
                b.Property<int>("Credits");  
  
                b.Property<string>("Title");  
  
                b.HasKey("CourseID");  
            });  
    }  
}
```

```

        b.ToTable("Course");
    });

    // Additional code for Enrollment and Student tables not shown

    modelBuilder.Entity("ContosoUniversity.Models.Enrollment", b =>
    {
        b.HasOne("ContosoUniversity.Models.Course", "Course")
            .WithMany("Enrollments")
            .HasForeignKey("CourseID")
            .OnDelete(DeleteBehavior.Cascade);

        b.HasOne("ContosoUniversity.Models.Student", "Student")
            .WithMany("Enrollments")
            .HasForeignKey("StudentID")
            .OnDelete(DeleteBehavior.Cascade);
    });
}
}

```

Because the current database schema is represented in code, EF Core doesn't have to interact with the database to create migrations. When you add a migration, EF determines what changed by comparing the data model to the snapshot file. EF interacts with the database only when it has to update the database.

The snapshot file has to be kept in sync with the migrations that create it, so you can't remove a migration just by deleting the file named `_cs`. If you delete that file, the remaining migrations will be out of sync with the database snapshot file. To delete the last migration that you added, use the [dotnet ef migrations remove](#) command.

Apply the migration to the database

In the command window, enter the following command to create the database and tables in it.

```
consoleCopy
```

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command.

textCopy

Build succeeded.

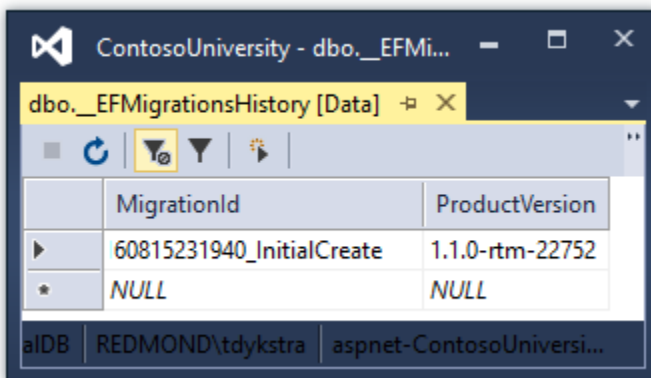
0 Warning(s)

0 Error(s)

Time Elapsed 00:00:17.34

Done.

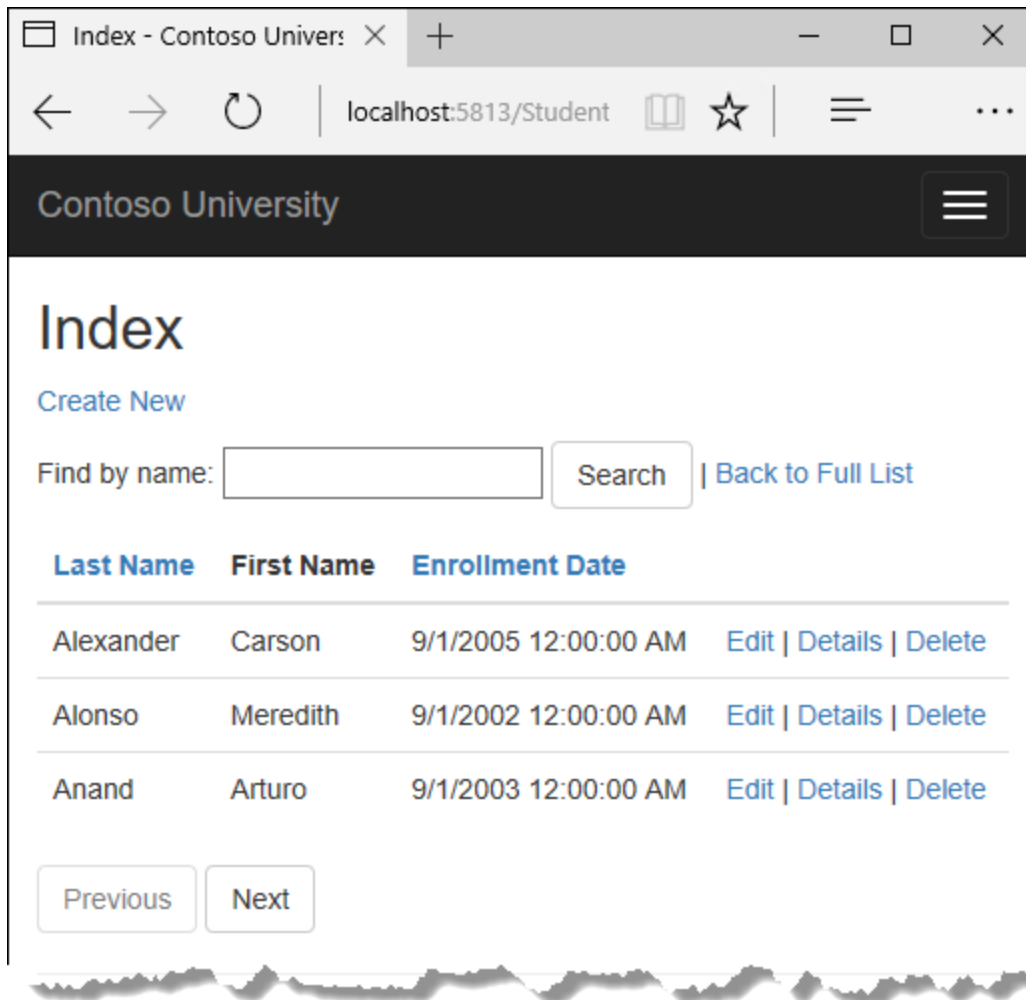
Use SQL Server Object Explorer to inspect the database as you did in the first tutorial. You'll notice the addition of an `__EFMigrationsHistory` table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one entry for the first migration.



The screenshot shows a SQL Server window titled 'ContosoUniversity - dbo.__EFMi...'. The active tab is 'dbo.__EFMigrationsHistory [Data]'. The table has two columns: 'MigrationId' and 'ProductVersion'. The first row contains the values '60815231940_InitialCreate' and '1.1.0-rtm-22752'. The second row contains 'NULL' and 'NULL'.

MigrationId	ProductVersion
60815231940_InitialCreate	1.1.0-rtm-22752
NULL	NULL

Run the application to verify that everything still works the same as before.



Command-line interface (CLI) vs. Package Manager Console (PMC)

The EF tooling for managing migrations is available from .NET Core CLI commands or from PowerShell cmdlets in the Visual Studio Package Manager Console (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

If you want to use the PMC commands, install the [Microsoft.EntityFrameworkCore.Tools](#) package. Unlike the CLI tools, you don't have to edit the `.csproj` file; you can install it by using the Package Manager Console or the NuGet Package Manager GUI. Note that this is not the same package as the one you install for the CLI: its name ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see [.NET Core CLI](#).

For more information about the PMC commands, see [Package Manager Console \(Visual Studio\)](#).

Summary

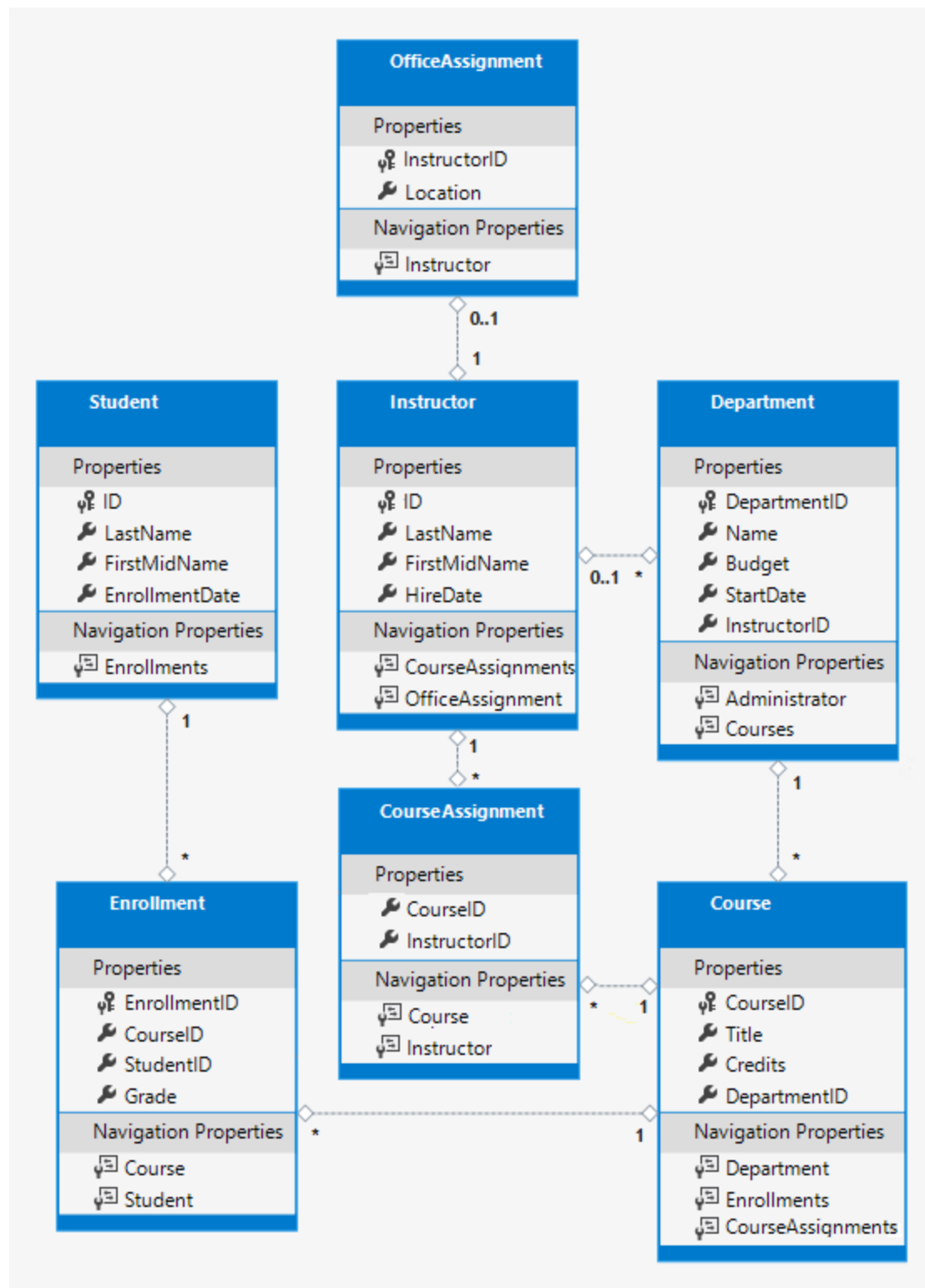
In this tutorial, you've seen how to create and apply your first migration. In the next tutorial, you'll begin looking at more advanced topics by expanding the data model. Along the way you'll create and apply additional migrations.

Creating a complex data model - EF Core with ASP.NET Core MVC tutorial (5 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorials you worked with a simple data model that was composed of three entities. In this tutorial you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



Customize the Data Model by Using Attributes

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In *Models/Student.cs*, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
```

```
[DataType(DataType.Date)]
```

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
    public DateTime EnrollmentDate { get; set; }

    public ICollection<Enrollment> Enrollments { get; set; }
}
```

```
}
```

The `DataType` attribute is used to specify a data type that is more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```
C#Copy
```

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
```

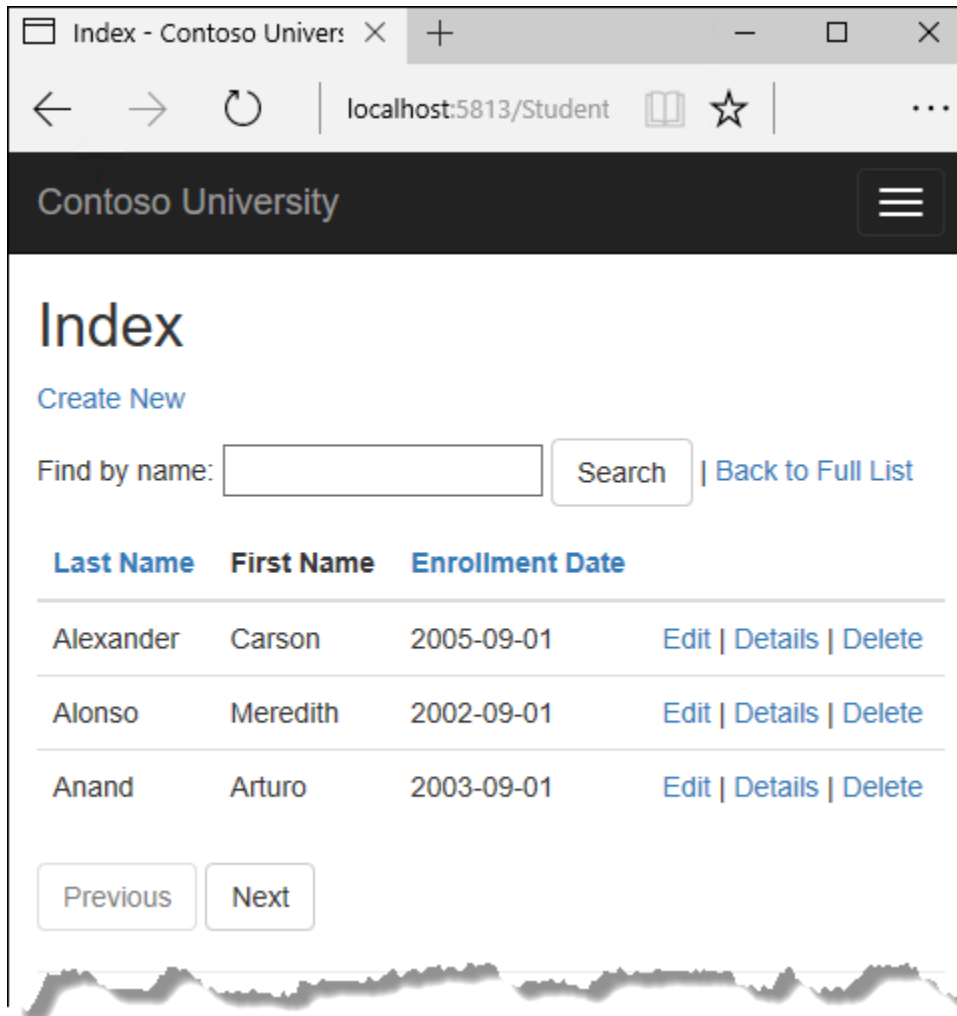
The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).
- By default, the browser will render data using the correct format based on your locale.

For more information, see the [<input> tag helper documentation](#).

Run the Students Index page again and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#Copy

```
[RegularExpression(@"^[A-Z]+[a-zA-Z' '-\s]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

consoleCopy

```
dotnet ef migrations add MaxLengthOnNames  
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named `_MaxLengthOnNames.cs`. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the Create page, and enter either name longer than 50 characters. When you click Create, client side validation shows an error message.

The screenshot shows a web browser window with the address bar displaying 'localhost:5813/Student'. The page title is 'Contoso University'. The main heading is 'Create Student'. There are three input fields: 'LastName' with the value 'Davolio very long last name longer than !', 'FirstMidName' with the value 'Nancy very long first name longer than 5', and 'EnrollmentDate' with the value '2/15/2017'. Below the 'LastName' field, a red error message states: 'The field LastName must be a string with a maximum length of 50.' Below the 'FirstMidName' field, a red error message states: 'First name cannot be longer than 50 characters.' At the bottom left, there is a 'Create' button. A small number '3' is visible in the bottom right corner of the browser window.

The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In

other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they are given the same name as the property name.

In the `Student.cs` file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

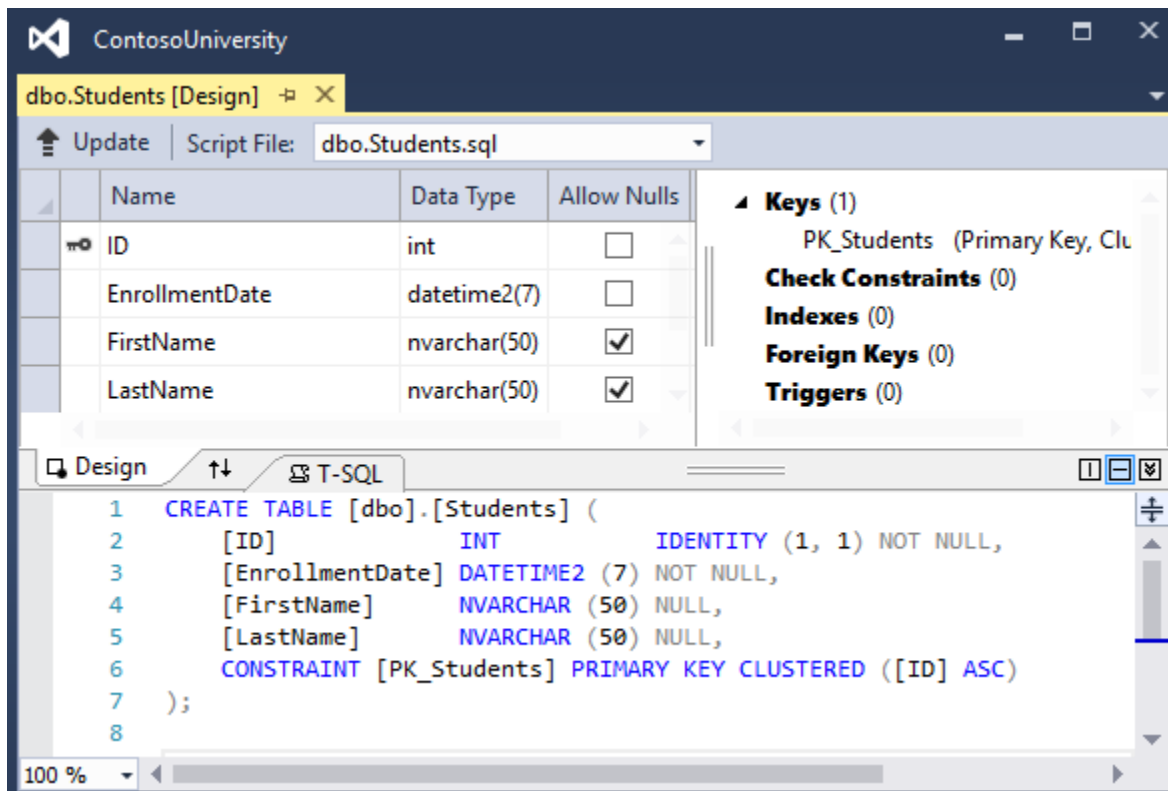
The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

consoleCopy

```
dotnet ef migrations add ColumnFirstName
dotnet ef database update
```

In SQL Server Object Explorer, open the Student table designer by double-clicking the Student table.








Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They are now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

Note

If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Final changes to the Student entity

Student	
Properties	
	ID
	LastName
	FirstMidName
	EnrollmentDate
Navigation Properties	
	Enrollments

In *Models/Student.cs*, replace the code you added earlier with the following code. The changes are highlighted.²

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
    }
}
```

```

    {
        get
        {
            return LastName + ", " + FirstMidName;
        }
    }

    public ICollection<Enrollment> Enrollments { get; set; }
}

```

The Required attribute

The `Required` attribute makes the name properties required fields.

The `Required` attribute is not needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

You could remove the `Required` attribute and replace it with a minimum length parameter for the `StringLength` attribute:

C#Copy

```

[Display(Name = "Last Name")]
[StringLength(50, MinimumLength=1)]
public string LastName { get; set; }

```

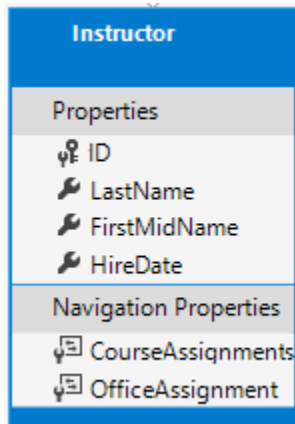
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create the Instructor Entity



Create *Models/Instructor.cs*, replacing the template code with the following code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }
    }
}
```

```

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

Notice that several properties are the same in the Student and Instructor entities. In the [Implementing Inheritance](#) tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

C#Copy

```

[DataType(DataType.Date),Display(Name = "Hire Date"),DisplayFormat(DataFormatString =
"{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

C#Copy

```

public ICollection<CourseAssignment> CourseAssignments { get; set; }

```


If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

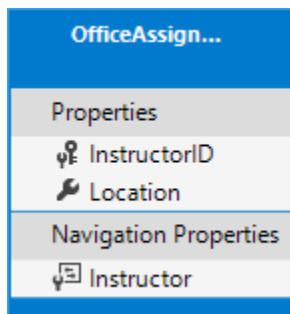
The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity (which may be null if no office is assigned).

C#Copy

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create the OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#Copy

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
```

```

        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}

```

The Key attribute

There's a one-to-zero-or-one relationship between the Instructor and the OfficeAssignment entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the Instructor entity. But the Entity Framework can't automatically recognize InstructorID as the primary key of this entity because its name doesn't follow the ID or classNameID naming convention. Therefore, the `Key` attribute is used to identify it as the key:

C#Copy

```

[Key]
public int InstructorID { get; set; }

```

You can also use the `key` attribute if the entity does have its own primary key but you want to name the property something other than classNameID or ID.

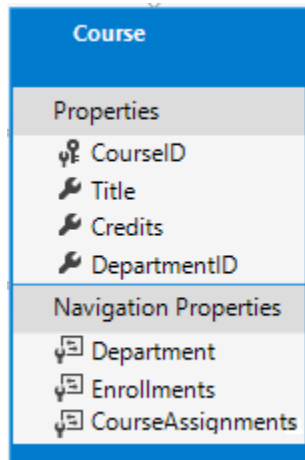
By default EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify the Course Entity



In *Models/Course.cs*, replace the code you added earlier with the following code. The changes are highlighted.

C#Copy

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

```
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they are needed and creates [shadow properties](#) for them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a course entity to edit, the Department entity is null if you don't load it, so when you update the course entity, you would have to first fetch the Department entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the Department entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

C#Copy

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]  
[Display(Name = "Number")]  
public int CourseID { get; set; }
```

By default, the Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for Course entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see [Generated Properties](#).

Foreign key and navigation properties

The foreign key properties and navigation properties in the Course entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

C#Copy

```
public int DepartmentID { get; set; }  
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

C#Copy

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained [later](#):2

C#Copy

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create the Department entity

Department
Properties
DepartmentID
Name
Budget
StartDate
InstructorID
Navigation Properties
Administrator
Courses

Create *Models/Department.cs* with the following code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
        [DisplayName = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server money type in the database:

C#Copy

```
[Column(TypeName="money")]  
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

C#Copy

```
public int? InstructorID { get; set; }  
public Instructor Administrator { get; set; }
```

A department may have many courses, so there's a Courses navigation property:

C#Copy

```
public ICollection<Course> Courses { get; set; }
```

Note

By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, EF would configure a cascade delete rule to delete the instructor when you delete the department, which is not what you want to have happen. If your business rules required

the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:

C#Copy

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

Modify the Enrollment entity

Enrollment
Properties
EnrollmentID
CourseID
StudentID
Grade
Navigation Properties
Course
Student

In *Models/Enrollment.cs*, replace the code you added earlier with the following code:

C#Copy

```
using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
```



```

    public int CourseID { get; set; }
    public int StudentID { get; set; }
    [DisplayFormat(NullDisplayText = "No grade")]
    public Grade? Grade { get; set; }

    public Course Course { get; set; }
    public Student Student { get; set; }
}
}

```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

C#Copy

```

public int CourseID { get; set; }
public Course Course { get; set; }

```

An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

C#Copy

```

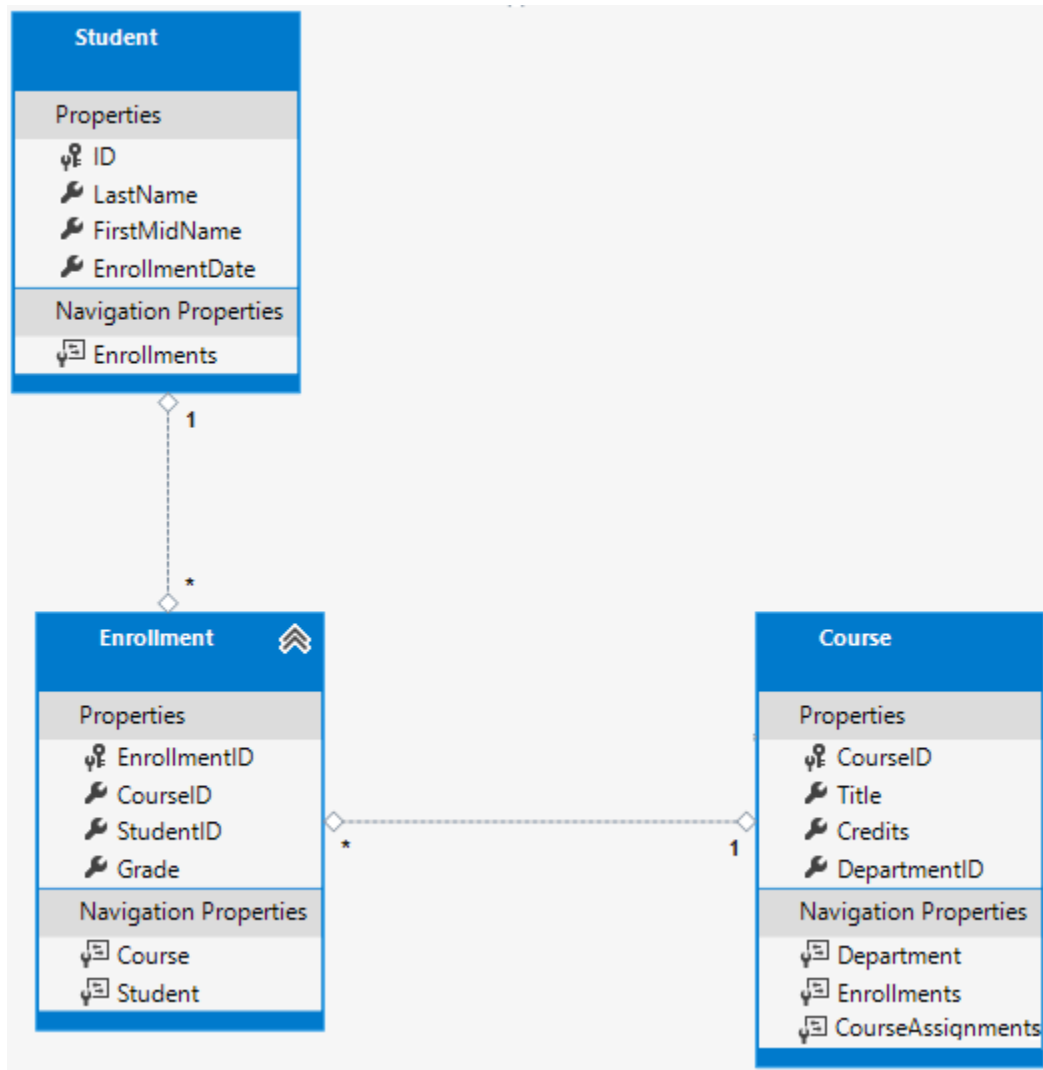
public int StudentID { get; set; }
public Student Student { get; set; }

```

Many-to-Many Relationships

There's a many-to-many relationship between the Student and Course entities, and the Enrollment entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the Enrollment table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a Grade property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)

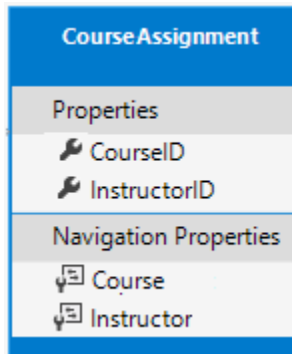


Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the Enrollment table didn't include grade information, it would only need to contain the two foreign keys CourseID and StudentID. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The Instructor and Course entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

(EF 6.x supports implicit join tables for many-to-many relationships, but EF Core does not. For more information, see the [discussion in the EF Core GitHub repository](#).)

The CourseAssignment entity



Create *Models/CourseAssignment.cs* with the following code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models

start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there is no need for a separate primary key.

The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see [Indexes](#).

Update the database context

Add the following highlighted code to the `Data/SchoolContext.cs` file:

C#Copy

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
    }
}
```

```
public DbSet<Enrollment> Enrollments { get; set; }
public DbSet<Student> Students { get; set; }
```

```
public DbSet<Department> Departments { get; set; }
public DbSet<Instructor> Instructors { get; set; }
public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
```

```
public DbSet<CourseAssignment> CourseAssignments { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Course>().ToTable("Course");
    modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
    modelBuilder.Entity<Student>().ToTable("Student");
```

```
    modelBuilder.Entity<Department>().ToTable("Department");
    modelBuilder.Entity<Instructor>().ToTable("Instructor");
    modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
    modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
```

```
    modelBuilder.Entity<CourseAssignment>()
```

```
        .HasKey(c => new { c.CourseID, c.InstructorID });
    }
}
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

Fluent API alternative to attributes

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the [EF Core documentation](#):¹

C#Copy

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

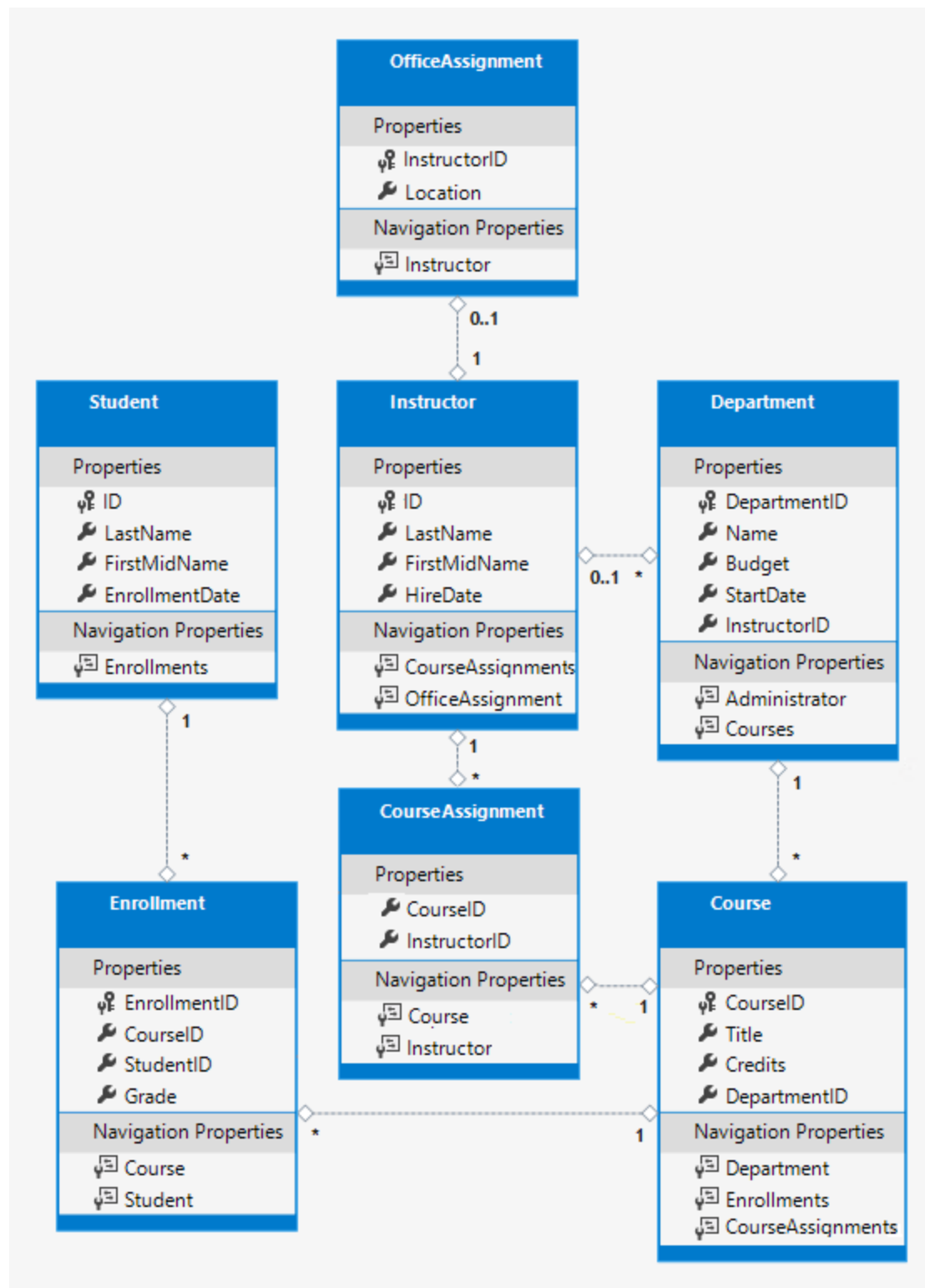
In this tutorial you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there is a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see [Methods of configuration](#).

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the Instructor and OfficeAssignment entities

and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed the Database with Test Data

Replace the code in the *Data/DbInitializer.cs* file with the following code in order to provide seed data for the new entities you've created.³

C#Copy

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return; // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson", LastName = "Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName = "Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo", LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis", LastName = "Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan", LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy", LastName = "Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
            };
        }
    }
}
```



```

        new Student { FirstMidName = "Laura",    LastName = "Norman",
                      EnrollmentDate = DateTime.Parse("2013-09-01") },
        new Student { FirstMidName = "Nino",     LastName = "Olivetto",
                      EnrollmentDate = DateTime.Parse("2005-09-01") }
    };

    foreach (Student s in students)
    {
        context.Students.Add(s);
    }
    context.SaveChanges();

    var instructors = new Instructor[]
    {
        new Instructor { FirstMidName = "Kim",    LastName = "Abercrombie",
                        HireDate = DateTime.Parse("1995-03-11") },
        new Instructor { FirstMidName = "Fadi",   LastName = "Fakhouri",
                        HireDate = DateTime.Parse("2002-07-06") },
        new Instructor { FirstMidName = "Roger",  LastName = "Harui",
                        HireDate = DateTime.Parse("1998-07-01") },
        new Instructor { FirstMidName = "Candace", LastName = "Kapoor",
                        HireDate = DateTime.Parse("2001-01-15") },
        new Instructor { FirstMidName = "Roger",  LastName = "Zheng",
                        HireDate = DateTime.Parse("2004-02-12") }
    };

    foreach (Instructor i in instructors)
    {
        context.Instructors.Add(i);
    }
    context.SaveChanges();

    var departments = new Department[]
    {
        new Department { Name = "English",    Budget = 350000,
                        StartDate = DateTime.Parse("2007-09-01"),
                        InstructorID = instructors.Single( i => i.LastName ==
"Abercrombie").ID },
        new Department { Name = "Mathematics", Budget = 100000,
                        StartDate = DateTime.Parse("2007-09-01"),
                        InstructorID = instructors.Single( i => i.LastName ==
"Fakhouri").ID },
        new Department { Name = "Engineering", Budget = 350000,
                        StartDate = DateTime.Parse("2007-09-01"),

```

```

        InstructorID = instructors.Single( i => i.LastName ==
"Harui").ID },
        new Department { Name = "Economics", Budget = 100000,
        StartDate = DateTime.Parse("2007-09-01"),
        InstructorID = instructors.Single( i => i.LastName ==
"Kapoor").ID }
    };

    foreach (Department d in departments)
    {
        context.Departments.Add(d);
    }
    context.SaveChanges();

    var courses = new Course[]
    {
        new Course {CourseID = 1050, Title = "Chemistry", Credits = 3,
        DepartmentID = departments.Single( s => s.Name ==
"Engineering").DepartmentID
        },
        new Course {CourseID = 4022, Title = "Microeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
        },
        new Course {CourseID = 4041, Title = "Macroeconomics", Credits = 3,
        DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
        },
        new Course {CourseID = 1045, Title = "Calculus", Credits = 4,
        DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
        },
        new Course {CourseID = 3141, Title = "Trigonometry", Credits = 4,
        DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
        },
        new Course {CourseID = 2021, Title = "Composition", Credits = 3,
        DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
        },
        new Course {CourseID = 2042, Title = "Literature", Credits = 4,
        DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
        },
    };

```

```

    foreach (Course c in courses)
    {
        context.Courses.Add(c);
    }
    context.SaveChanges();

    var officeAssignments = new OfficeAssignment[]
    {
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName ==
"Fakhouri").ID,
            Location = "Smith 17" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName ==
"Harui").ID,
            Location = "Gowan 27" },
        new OfficeAssignment {
            InstructorID = instructors.Single( i => i.LastName ==
"Kapoor").ID,
            Location = "Thompson 304" },
    };

    foreach (OfficeAssignment o in officeAssignments)
    {
        context.OfficeAssignments.Add(o);
    }
    context.SaveChanges();

    var courseInstructors = new CourseAssignment[]
    {
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Kapoor").ID
        },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Harui").ID
        },
        new CourseAssignment {
            CourseID = courses.Single(c => c.Title == "Microeconomics"
).CourseID,
            InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
        },
        new CourseAssignment {

```

```

        CourseID = courses.Single(c => c.Title == "Macroeconomics"
).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Zheng").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        InstructorID = instructors.Single(i => i.LastName ==
"Fakhouri").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
        InstructorID = instructors.Single(i => i.LastName == "Harui").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
        InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
    },
    new CourseAssignment {
        CourseID = courses.Single(c => c.Title == "Literature"
).CourseID,
        InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
    },
    },
};

foreach (CourseAssignment ci in courseInstructors)
{
    context.CourseAssignments.Add(ci);
}
context.SaveChanges();

var enrollments = new Enrollment[]
{
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID,
        Grade = Grade.A
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Microeconomics"
).CourseID,

```

```

        Grade = Grade.C
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alexander").ID,
        CourseID = courses.Single(c => c.Title == "Macroeconomics"
).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Calculus" ).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Alonso").ID,
        CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry" ).CourseID
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Anand").ID,
        CourseID = courses.Single(c => c.Title ==
"Microeconomics").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Barzdukas").ID,
        CourseID = courses.Single(c => c.Title == "Chemistry").CourseID,
        Grade = Grade.B
    },
    new Enrollment {
        StudentID = students.Single(s => s.LastName == "Li").ID,
        CourseID = courses.Single(c => c.Title ==
"Composition").CourseID,
        Grade = Grade.B
    }
}

```

```

        },
        new Enrollment {
            StudentID = students.Single(s => s.LastName == "Justice").ID,
            CourseID = courses.Single(c => c.Title == "Literature").CourseID,
            Grade = Grade.B
        }
    };

    foreach (Enrollment e in enrollments)
    {
        var enrollmentInDataBase = context.Enrollments.Where(
            s =>
                s.Student.ID == e.StudentID &&
                s.Course.CourseID == e.CourseID).SingleOrDefault();
        if (enrollmentInDataBase == null)
        {
            context.Enrollments.Add(e);
        }
    }
    context.SaveChanges();
}
}
}

```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

```
consoleCopy
```

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

```
textCopy
```

Build succeeded.

0 Warning(s)

0 Error(s)

Time Elapsed 00:00:11.58

An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.

Done. To undo this action, use 'ef migrations remove'

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable DepartmentID foreign key to the Course table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the DepartmentID column to the Course table.

C#Copy

```
migrationBuilder.AlterColumn<string>(  
    name: "Title",  
    table: "Course",
```

```
maxLength: 50,  
nullable: true,  
oldClrType: typeof(string),  
oldNullable: true);
```

```
//migrationBuilder.AddColumn<int>(  
//    name: "DepartmentID",  
//    table: "Course",  
//    nullable: false,
```

```
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the Department table:

C#Copy

```
migrationBuilder.CreateTable(  
    name: "Department",  
    columns: table => new  
    {  
        DepartmentID = table.Column<int>(nullable: false)  
            .Annotation("SqlServer:ValueGenerationStrategy",  
SqlServerValueGenerationStrategy.IdentityColumn),  
        Budget = table.Column<decimal>(type: "money", nullable: false),  
        InstructorID = table.Column<int>(nullable: true),  
        Name = table.Column<string>(maxLength: 50, nullable: true),  
        StartDate = table.Column<DateTime>(nullable: false)  
    },  
    constraints: table =>  
    {  
        table.PrimaryKey("PK_Department", x => x.DepartmentID);  
        table.ForeignKey(  
            name: "FK_Department_Instructor_InstructorID",  
            column: x => x.InstructorID,  
            principalTable: "Instructor",  
            principalColumn: "ID",  
            onDelete: ReferentialAction.Restrict);  
    });
```



```
migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget, StartDate) VALUES ('Temp', 0.00, GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(<
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the Course.DepartmentID column.

Save your changes and build the project.

Change the connection string and update the database

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in *appsettings.json* to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

JSONCopy

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
}
```

Save your change to *appsettings.json*.

Note

As an alternative to changing the database name, you can delete the database. Use SQL Server Object Explorer (SSOX) or the `database drop` CLI command:

```
consoleCopy
```

```
dotnet ef database drop
```

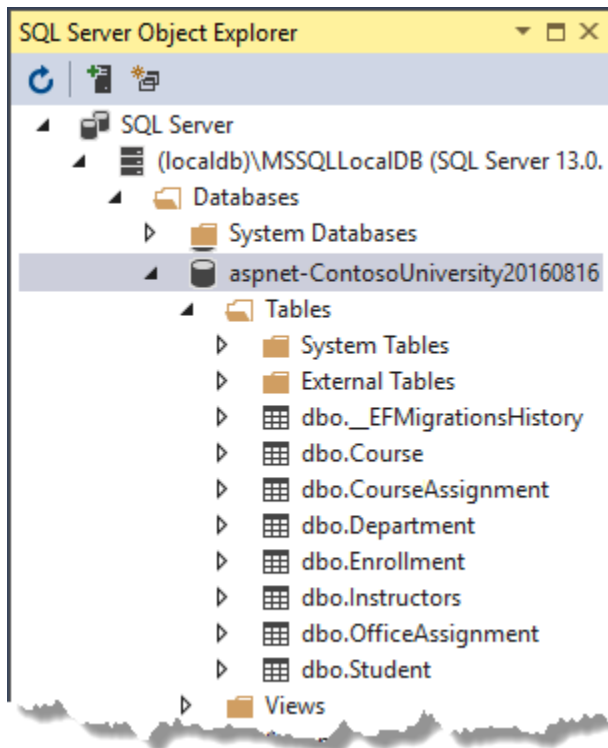
After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

```
consoleCopy
```

```
dotnet ef database update
```

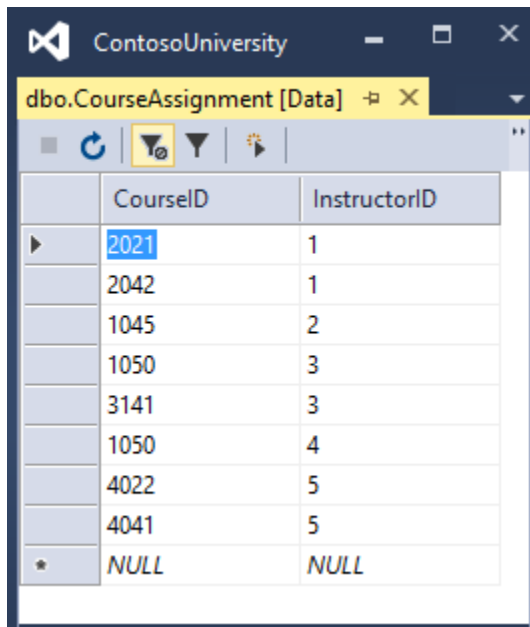
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the `Tables` node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the Refresh button.)



Run the application to trigger the initializer code that seeds the database.

Right-click the CourseAssignment table and select View Data to verify that it has data in it.²



	CourseID	InstructorID
▶	2021	1
	2042	1
	1045	2
	1050	3
	3141	3
	1050	4
	4022	5
	4041	5
★	NULL	NULL

Summary

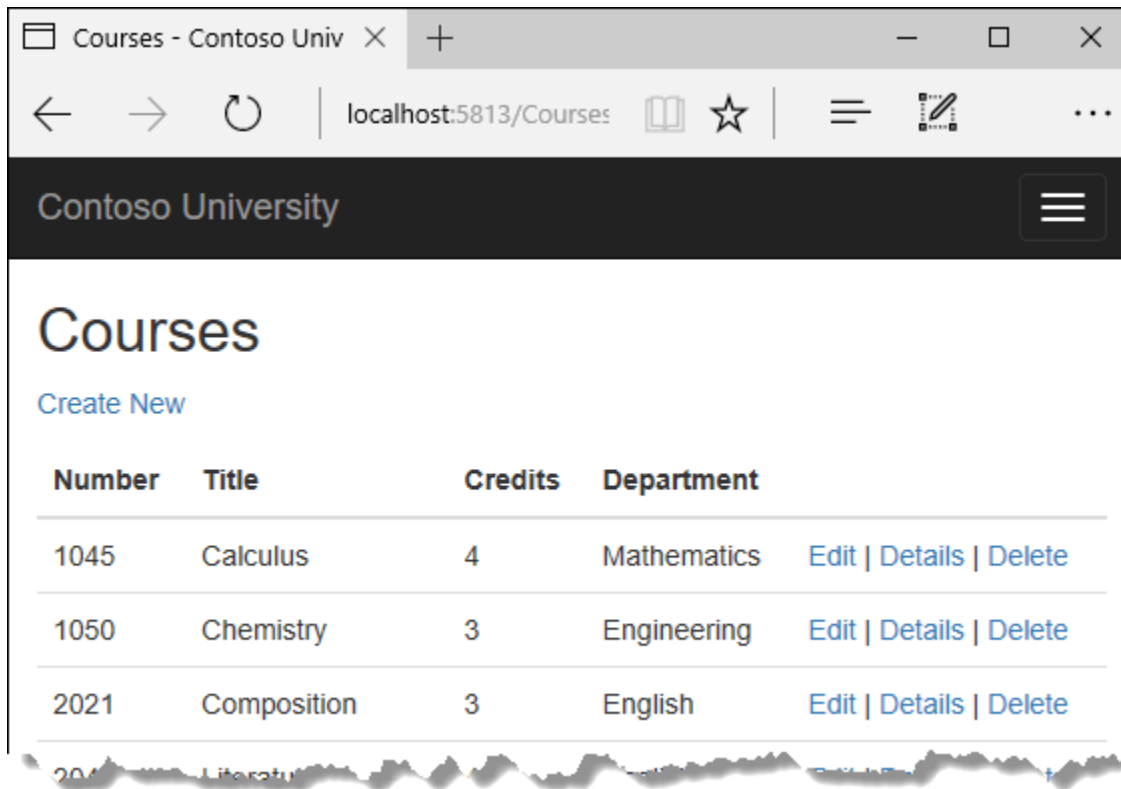
You now have a more complex data model and corresponding database. In the following tutorial, you'll learn more about how to access related data.

Reading related data - EF Core with ASP.NET Core MVC tutorial (6 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you completed the School data model. In this tutorial you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.



Instructors - Contoso University

localhost:5813/Instructors/Index/1?

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

Eager, explicit, and lazy Loading of related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

- Eager loading. When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Explicit loading. When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 does not support lazy loading.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page that displays Department name

The Course entity includes a navigation property that contains the Department entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the Name property from the Department entity that is in the `Course.Department` navigation property.

Create a controller named `CoursesController` for the `Course` entity type, using the same options for the MVC Controller with views, using Entity Framework scaffolder that you did earlier for the `Students` controller, as shown in the following illustration:

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Course (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below these, there is an empty text box and a button with three dots. At the bottom, the 'Controller name' text box contains 'CoursesController'. The 'Add' button is highlighted with a red dashed border.

Open `CoursesController.cs` and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns `Course` entities (`courses` instead of `schoolContext`):

C#Copy

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open `Views/Courses/Index.cshtml` and replace the template code with the following code. The changes are highlighted:

htmlCopy

```
@model IEnumerable<ContosoUniversity.Models.Course>
```

```
@{  
    ViewData["Title"] = "Courses";  
}
```

```
<h2>Courses</h2>
```

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

```
<table class="table">  
    <thead>  
        <tr>
```

```
            <th>
```

```
                @Html.DisplayNameFor(model => model.CourseID)
```

```
            </th>
```

```
            <th>
```

```
                @Html.DisplayNameFor(model => model.Title)
```

```
            </th>
```

```
            <th>
```

```
                @Html.DisplayNameFor(model => model.Credits)
```

```
            </th>
```

```
            <th>
```

```
                @Html.DisplayNameFor(model => model.Department)
```

```
            </th>
```

```
        <th></th>
```

```
    </tr>
```

```
</thead>
```

```
<tbody>
```

```
    @foreach (var item in Model)
```

```
    {
```

```
        <tr>
```

```
            <td>
```

```
                @Html.DisplayFor(modelItem => item.CourseID)
```

```
            </td>
```

```

        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Credits)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Department.Name)
        </td>
        <td>
            <a asp-action="Edit" asp-route-id="@item.CourseID">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.CourseID">Details</a>
|
            <a asp-action="Delete" asp-route-id="@item.CourseID">Delete</a>
        </td>
    </tr>
}
</tbody>
</table>

```

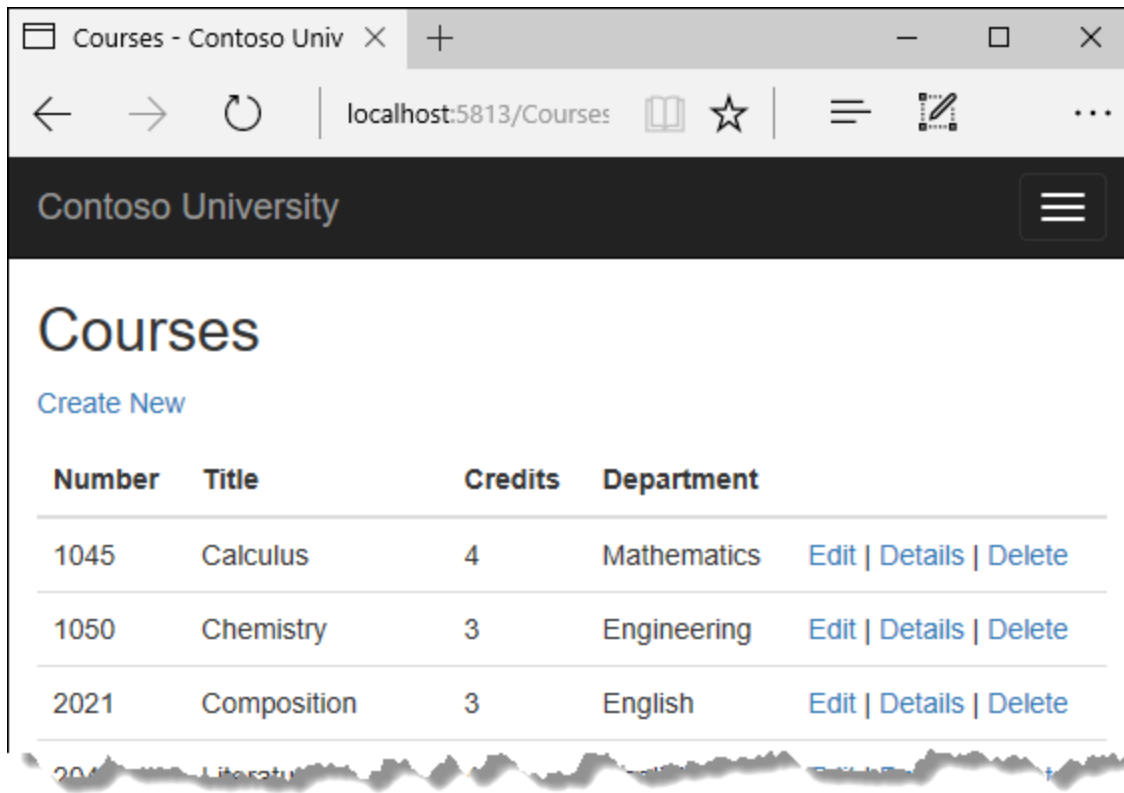
You've made the following changes to the scaffolded code:

- Changed the heading from Index to Courses.
- Added a Number column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they are meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the Department column to display the department name. The code displays the `Name` property of the Department entity that's loaded into the `Department` navigation property:

htmlCopy

```
@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the page (select the Courses tab on the Contoso University home page) to see the list with department names.



2

Create an Instructors page that shows Courses and Enrollments

In this section you'll create a controller and view for the Instructor entity in order to display the Instructors page:

Instructors - Contoso University

localhost:5813/Instructors/Index/1?

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

This page reads and displays related data in the following ways:

- The list of instructors displays related data from the OfficeAssignment entity. The Instructor and OfficeAssignment entities are in a one-to-zero-or-one relationship. You'll use eager loading for the OfficeAssignment entities. As explained earlier, eager loading is typically more efficient when you need the related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.
- When the user selects an instructor, related Course entities are displayed. The Instructor and Course entities are in a many-to-many relationship. You'll use eager loading for the Course entities and their related Department entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the Enrollments entity set is displayed. The Course and Enrollment entities are in a one-to-many relationship. You'll use separate queries for Enrollment entities and their related Student entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

In the *SchoolViewModels* folder, create *InstructorIndexData.cs* and replace the existing code with the following code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

```
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:

The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Instructor (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below these, there is an empty text box and a button with three dots. At the bottom, the 'Controller name' text box contains 'InstructorsController'. The 'Add' button is highlighted with a dashed border.

3

Open *InstructorsController.cs* and add a using statement for the ViewModels namespace:

C#Copy

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

C#Copy

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
```

```

        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
        .ThenInclude(i => i.Student)
    .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}

```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the [Select](#) hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

C#Copy

```

viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)

```

```

.Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)
            .ThenInclude(i => i.Student)
.Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
.AsNoTracking()
.OrderBy(i => i.LastName)
.ToListAsync();

```

Since the view always requires the OfficeAssignment entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.³

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

C#Copy

```

viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)

```

```

        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Enrollments)

```

```

            .ThenInclude(i => i.Student)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
.AsNoTracking()
.OrderBy(i => i.LastName)
.ToListAsync();

```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over

with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

C#Copy

```
viewModel.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
    .ThenInclude(i => i.Enrollments)
    .ThenInclude(i => i.Student)
```

```
.Include(i => i.CourseAssignments)
    .ThenInclude(i => i.Course)
```

```
.ThenInclude(i => i.Department)
    .AsNoTracking()
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

C#Copy

```
#endregion
```

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single `Instructor` entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it is empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the `Where` condition instead of calling the `Where` method separately:

C#Copy

```
.Single(i => i.ID == id.Value)
```

Instead of:

C#Copy

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

C#Copy

```
}

if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
```

Modify the Instructor Index view

In `Views/Instructors/Index.cshtml`, replace the template code with the following code. The changes are highlighted.

htmlCopy

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData
```

```
@{
    ViewData["Title"] = "Instructors";
}
```

```
<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
```

```
            <th>Office</th>
```

```
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
```

```

        <td>
            @if (item.OfficeAssignment != null)
            {
                @item.OfficeAssignment.Location
            }
        </td>
        <td>
            @{
                foreach (var course in item.CourseAssignments)
                {
                    @course.Course.CourseID @: @course.Course.Title <br />
                }
            }
        </td>
    </tr>
}
</tbody>
</table>

```

```

    </td>
    <td>
        <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
        <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
        <a asp-action="Details" asp-route-id="@item.ID">Details</a> |
        <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
    </td>
</tr>
}
</tbody>
</table>

```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from Index to Instructors.
- Added an Office column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` is not null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

htmlCopy

```

@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}

```

```
}
```

- Added a `Courses` column that displays courses taught by each instructor.
- Added code that dynamically adds `class="success"` to the `tr` element of the selected instructor. This sets a background color for the selected row using a Bootstrap class.

htmlCopy

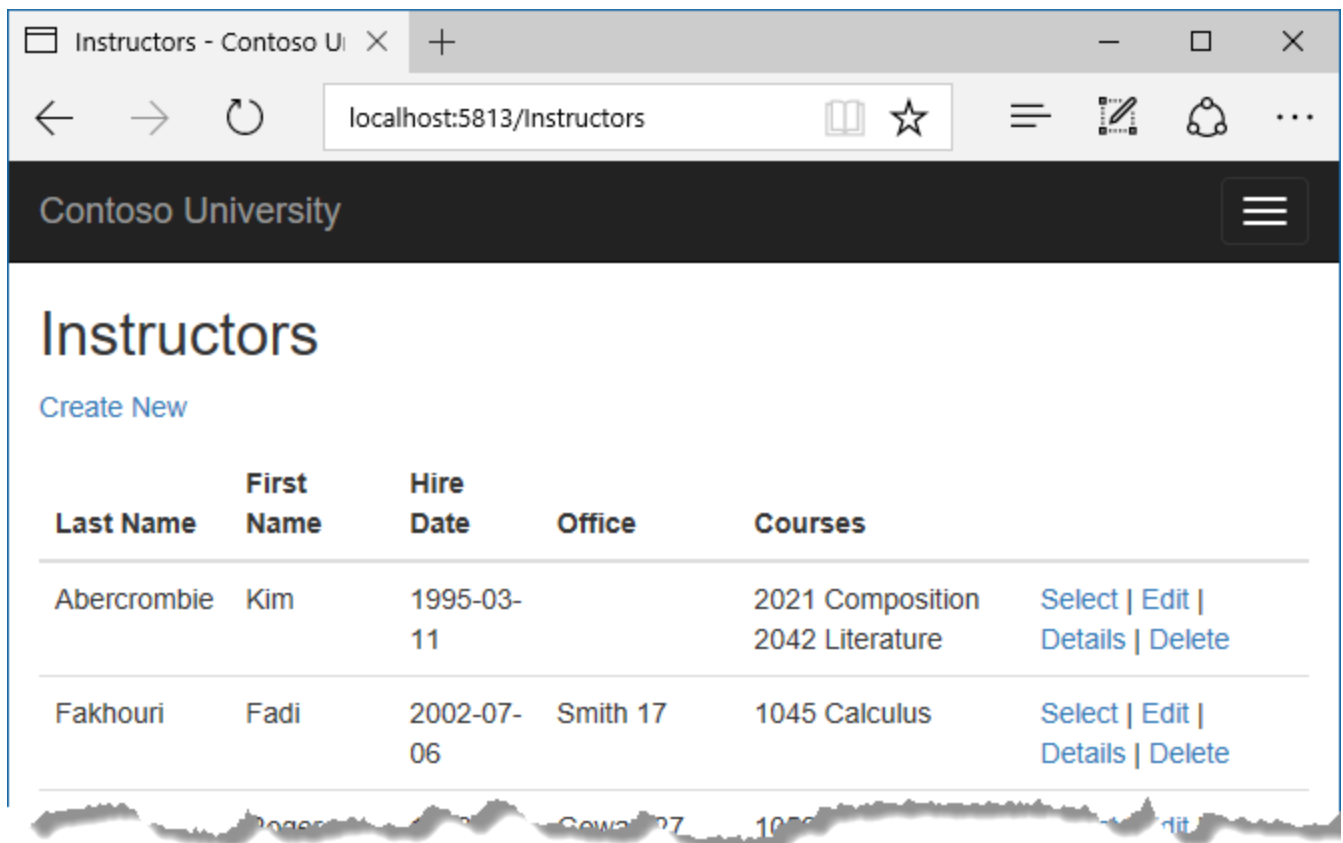
```
string selectedRow = "";
if (item.ID == (int?)ViewData["InstructorID"])
{
    selectedRow = "success";
}
```

- Added a new hyperlink labeled `Select` immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

htmlCopy

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the application and select the Instructors tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



In the *Views/Instructors/Index.cshtml* file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

htmlCopy

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
```

```

        selectedRow = "success";
    }
    <tr class="@selectedRow">
        <td>
            @Html.ActionLink("Select", "Index", new { courseID =
item.CourseID })
        </td>
        <td>
            @item.CourseID
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @item.Department.Name
        </td>
    </tr>
}

</table>
}

```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a `Select` hyperlink that sends the ID of the selected course to the `Index` action method.

Run the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.

Instructors - Contoso University

localhost:5813/Instructors/Index/1

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

htmlCopy

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
```



```
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the Enrollments property of the view model in order to display a list of students enrolled in the course.

Run the page and select an instructor. Then select a course to see the list of enrolled students and their grades.

Instructors - Contoso University

localhost:5813/Instructors/Index/1?

Contoso University

Instructors

[Create New](#)

Last Name	First Name	Hire Date	Office	Courses	
Abercrombie	Kim	1995-03-11		2021 Composition 2042 Literature	Select Edit Details Delete
Fakhouri	Fadi	2002-07-06	Smith 17	1045 Calculus	Select Edit Details Delete
Harui	Roger	1998-07-01	Gowan 27	1050 Chemistry 3141 Trigonometry	Select Edit Details Delete

Courses Taught by Selected Instructor

	Number	Title	Department
Select	2021	Composition	English
Select	2042	Literature	English

Students Enrolled in Selected Course

Name	Grade
Alonso, Meredith	B
Li, Yan	B

Explicit loading

When you retrieved the list of instructors in *InstructorsController.cs*, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for Enrollments and loads that property explicitly. The code changes are highlighted.

C#Copy

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .ThenInclude(i => i.Department)
        .OrderBy(i => i.LastName)
        .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID ==
courseID).Single();
        await _context.Entry(selectedCourse).Collection(x =>
x.Enrollments).LoadAsync();

        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x => x.Student).LoadAsync();
        }
    }
}
```

```
    }  
    viewModel.Enrollments = selectedCourse.Enrollments;  
}
```

```
    return View(viewModel);  
}
```

The new code drops the *ThenInclude* method calls for enrollment data from the code that retrieves instructor entities. If an instructor and course are selected, the highlighted code retrieves Enrollment entities for the selected course, and Student entities for each Enrollment.

Run the Instructor Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Summary

You've now used eager loading with one query and with multiple queries to read related data into navigation properties. In the next tutorial you'll learn how to update related data.

Updating related data - EF Core with ASP.NET Core MVC tutorial (7 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.

Edit - Coi ×

+

—

□

×

←

↺

localhost:58

📖

☆

⋮

Contoso University

☰

Edit

Course

Number

1000

Title

Algebra 2

×

Credits

5

Department

Mathematics

▼

Save

Edit - Contoso Universit X + - □ ×

← → ↻ | localhost:5813/Instruct 📖 ☆ | ≡ ...

Contoso University ≡

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

☐ 1000 Algebra 2 ☐ 1045 Calculus ☐ 1050 Chemistry
☒ 2021 Composition ☒ 2042 Literature ☐ 3141 Trigonometry
☐ 4022 Microeconomics ☐ 4041 Macroeconomics

Customize the Create and Edit Pages for Courses

When a new course entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The

drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate Department entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In *CoursesController.cs*, delete the four Create and Edit methods and replace them with the following code:

C#Copy

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

C#Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("CourseID,Credits,DepartmentID,Title")]
Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#Copy

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
```



```

    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

```

C#Copy

```

[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .SingleOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction("Index");
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}

```

After the `Edit` `HttpPost` method, create a new method that loads department info for the drop-down list.

C#Copy

```
private void PopulateDepartmentsDropDownList(object selectedDepartment = null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(),
    "DepartmentID", "Name", selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The `HttpGet Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department is not established yet:

C#Copy

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

The `HttpGet Edit` method sets the selected item, based on the ID of the department that is already assigned to the course being edited:

C#Copy

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
```

```

        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}

```

The `HttpPost` methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add `.AsNoTracking` to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and `HttpGet Delete` methods.

C#Copy

```

public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}

```

```
}
```

C#Copy

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In *Views/Courses/Create.cshtml*, add a "Select Department" option to the Department drop-down list, and change the caption for the field from DepartmentID to Department.1

htmlCopy

```
<div class="form-group">
    <label asp-for="Department" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <select asp-for="DepartmentID" class="form-control" asp-
items="ViewBag.DepartmentID">
            <option value="">-- Select Department --</option>
        </select>
        <span asp-validation-for="DepartmentID" class="text-danger" />
    </div>
</div>
```

In *Views/Courses/Edit.cshtml*, make the same change for the Department field that you just did in *Create.cshtml*.¹

Also in *Views/Courses/Edit.cshtml*, add a course number field before the Credits field. Because it's the primary key, it's displayed, but it can't be changed.

htmlCopy

```
<div class="form-group">
  <label asp-for="CourseID" class="col-md-2 control-label"></label>
  <div class="col-md-10">
    @Html.DisplayFor(model => model.CourseID)
  </div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks Save on the Edit page.⁵

In *Views/Courses/Delete.cshtml*, add a course number field at the top and change department ID to department name.

htmlCopy

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="dl-horizontal">
```

```
        <dt>
```

```
            @Html.DisplayNameFor(model => model.CourseID)
```

```
        </dt>
```

```
        <dd>
```

```
@Html.DisplayFor(model => model.CourseID)
```

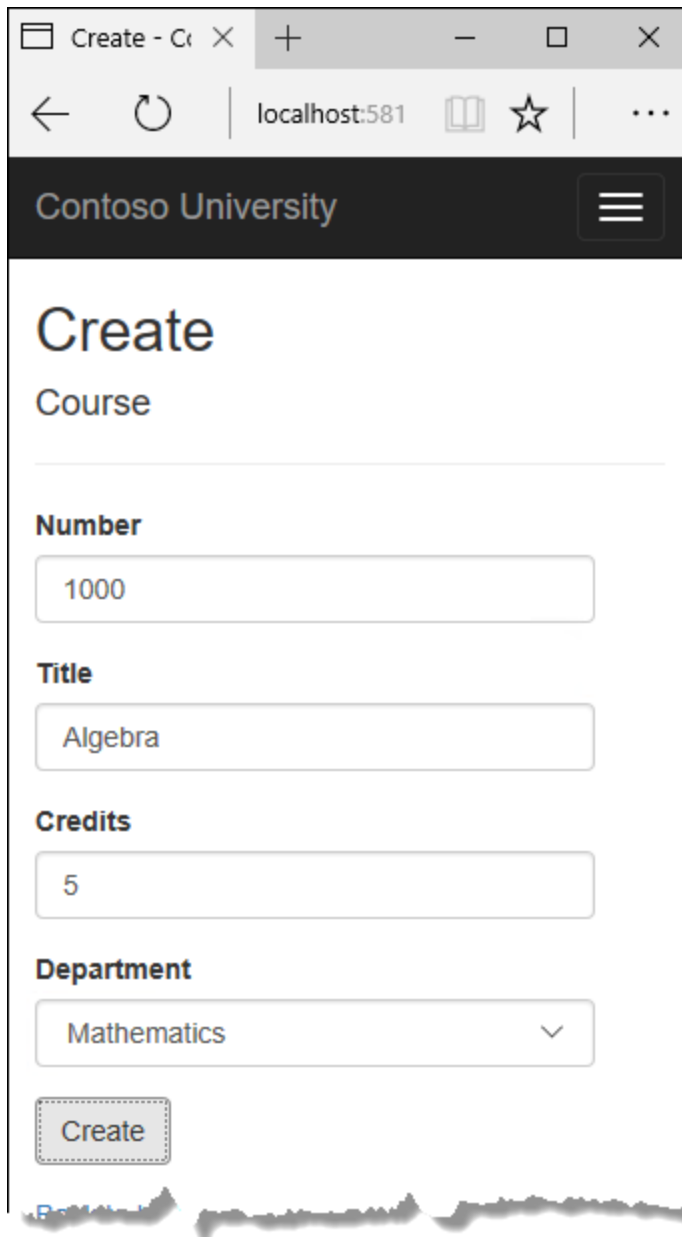
```
</dd>
<dt>
    @Html.DisplayNameFor(model => model.Title)
</dt>
<dd>
    @Html.DisplayFor(model => model.Title)
</dd>
<dt>
    @Html.DisplayNameFor(model => model.Credits)
</dt>
<dd>
    @Html.DisplayFor(model => model.Credits)
</dd>
<dt>
    @Html.DisplayNameFor(model => model.Department)
</dt>
<dd>
    @Html.DisplayFor(model => model.Department.Name)
</dd>
</dl>

<form asp-action="Delete">
    <div class="form-actions no-color">
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-action="Index">Back to List</a>
    </div>
</form>
</div>
```

In *Views/Course/Details.cshtml*, make the same change that you just did for *Delete.cshtml*.

Test the Course pages

Run the Create page (display the Course Index page and click Create New) and enter data for a new course:1



The screenshot shows a web browser window with the title 'Create - C...'. The address bar displays 'localhost:581'. The page header for 'Contoso University' includes a hamburger menu icon. The main content area is titled 'Create Course' and contains the following form fields:

- Number:** A text input field containing the value '1000'.
- Title:** A text input field containing the value 'Algebra'.
- Credits:** A text input field containing the value '5'.
- Department:** A dropdown menu with 'Mathematics' selected and a downward arrow.

At the bottom of the form is a 'Create' button with a dashed border. A small portion of the 'Edit' button is visible at the very bottom of the page.

Click Create. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Run the Edit page (click Edit on a course in the Course Index page).

The screenshot shows a web browser window with the address bar displaying 'localhost:58'. The page title is 'Edit - Course'. The header of the application is 'Contoso University' with a hamburger menu icon. The main content area is titled 'Edit Course'. Below the title, there is a horizontal line. The form contains the following fields:

- Number:** 1000
- Title:** Algebra 2
- Credits:** 5
- Department:** Mathematics

A 'Save' button is located at the bottom of the form.

Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add an Edit Page for Instructors

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the OfficeAssignment entity.
- If the user enters an office assignment value and it originally was empty, create a new OfficeAssignment entity.
- If the user changes the value of an office assignment, change the value in an existing OfficeAssignment entity.

Update the Instructors controller

In *InstructorsController.cs*, change the code in the `HttpGet Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

C#Copy

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the `HttpPost Edit` method with the following code to handle office assignment updates:

C#Copy

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
}
```

```

    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .SingleOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction("Index");
    }
    return View(instructorToUpdate);
}

```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the `HttpGet Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current Instructor entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the `HttpGet Edit` method.

- Updates the retrieved Instructor entity with values from the model binder. The `TryUpdateModel` overload enables you to whitelist the properties you want to include. This prevents over-posting, as explained in the [second tutorial](#).

C#Copy

```
if (await TryUpdateModelAsync<Instructor>(
    instructorToUpdate,
    "",
    i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
    i.OfficeAssignment))
```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

C#Copy

```
if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

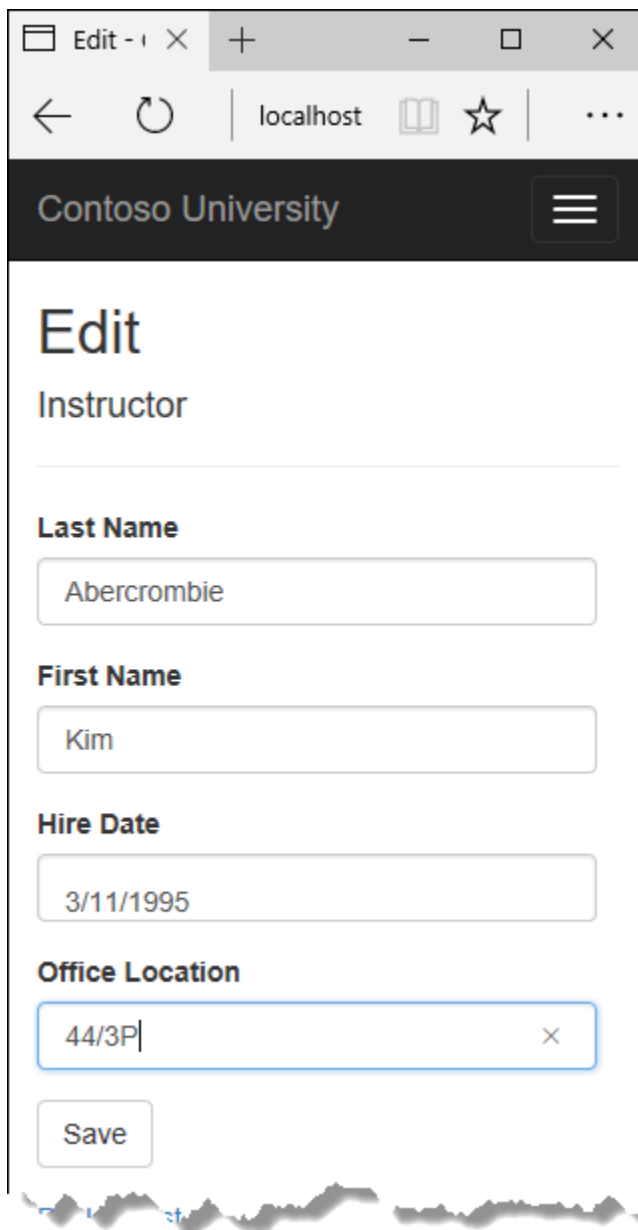
Update the Instructor Edit view

In `Views/Instructors/Edit.cshtml`, add a new field for editing the office location, at the end before the Save button :

htmlCopy

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="OfficeAssignment.Location" class="form-control" />
        <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
    </div>
</div>
```

Run the page (select the Instructors tab and then click Edit on an instructor). Change the Office Location and click Save.¹



The screenshot shows a web browser window with the address bar displaying 'localhost'. The page title is 'Contoso University'. The main heading is 'Edit Instructor'. Below the heading, there are four input fields: 'Last Name' with the value 'Abercrombie', 'First Name' with the value 'Kim', 'Hire Date' with the value '3/11/1995', and 'Office Location' with the value '44/3P'. A 'Save' button is located at the bottom of the form.

Add Course assignments to the Instructor Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of check boxes, as shown in the following screen shot:

Edit - Contoso Universit X + - □ ×

← → ↻ | localhost:5813/Instruct 📖 ☆ | ≡ ...

Contoso University ≡

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

☐ 1000 Algebra 2 ☐ 1045 Calculus ☐ 1050 Chemistry
☒ 2021 Composition ☒ 2042 Literature ☐ 3141 Trigonometry
☐ 4022 Microeconomics ☐ 4041 Macroeconomics

The relationship between the Course and Instructor entities is many-to-many. To add and remove relationships, you add and remove entities to and from the CourseAssignments join entity set.²

The UI that enables you to change which courses an instructor is assigned to is a group of check boxes. A check box for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear

check boxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of check boxes, you'll use a view model class.

Create *AssignedCourseData.cs* in the *SchoolViewModels* folder and replace the existing code with the following code:

C#Copy

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In *InstructorsController.cs*, replace the `HttpGet Edit` method with the following code. The changes are highlighted.

C#Copy

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
```

```

        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

```

```

private void PopulateAssignedCourseData(Instructor instructor)

```

```

{
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>(instructor.CourseAssignments.Select(c =>
c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}

```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the check box array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection.

The `Assigned` property is set to true for courses the instructor is assigned to. The view

will use this property to determine which check boxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks `Save`. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the `Instructor` entity.

C#Copy

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)

{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
        .SingleOrDefaultAsync(m => m.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
    {
        if (String.IsNullOrEmpty(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {

```



```

        //Log the error (uncomment ex variable name and write a log.)
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists, " +
            "see your system administrator.");
    }
    return RedirectToAction("Index");
}

```

```
UpdateInstructorCourses(selectedCourses, instructorToUpdate);
```

```

    PopulateAssignedCourseData(instructorToUpdate);
    return View(instructorToUpdate);
}

```

C#Copy

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new CourseAssignment {
InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });
            }
        }
        else
        {
            if (instructorCourses.Contains(course.CourseID))
            {

```



```

foreach (var course in _context.Courses)
{
    if (selectedCoursesHS.Contains(course.CourseID.ToString()))
    {
        if (!instructorCourses.Contains(course.CourseID))
        {
            instructorToUpdate.CourseAssignments.Add(new CourseAssignment {
InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });
        }
    }
    else
    {

        if (instructorCourses.Contains(course.CourseID))
        {
            CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.SingleOrDefault(i => i.CourseID ==
course.CourseID);
            _context.Remove(courseToRemove);
        }
    }
}
}

```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the check box for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection in the navigation property.

C#Copy

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }
}

```

```

var selectedCoursesHS = new HashSet<string>(selectedCourses);
var instructorCourses = new HashSet<int>
    (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
foreach (var course in _context.Courses)
{

```

```

    if (selectedCoursesHS.Contains(course.CourseID.ToString()))
    {
        if (!instructorCourses.Contains(course.CourseID))
        {
            instructorToUpdate.CourseAssignments.Add(new CourseAssignment {
InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });
        }
    }

```

```

    }
    else
    {
        if (instructorCourses.Contains(course.CourseID))
        {
            CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.SingleOrDefault(i => i.CourseID ==
course.CourseID);
            _context.Remove(courseToRemove);
        }
    }
}

```

If the check box for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

C#Copy

```

private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }
}

```

```

var selectedCoursesHS = new HashSet<string>(selectedCourses);
var instructorCourses = new HashSet<int>
    (instructorToUpdate.CourseAssignments.Select(c => c.Course.CourseID));
foreach (var course in _context.Courses)
{
    if (selectedCoursesHS.Contains(course.CourseID.ToString()))
    {
        if (!instructorCourses.Contains(course.CourseID))
        {
            instructorToUpdate.CourseAssignments.Add(new CourseAssignment {
InstructorID = instructorToUpdate.ID, CourseID = course.CourseID });
        }
    }
}

```

```

    else
    {

        if (instructorCourses.Contains(course.CourseID))
        {
            CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.SingleOrDefault(i => i.CourseID ==
course.CourseID);
            _context.Remove(courseToRemove);
        }
    }
}
}

```

Update the Instructor views

In *Views/Instructors/Edit.cshtml*, add a Courses field with an array of check boxes by adding the following code immediately after the `div` elements for the Office field and before the `div` element for the Save button.

Note

When you paste the code in Visual Studio, line breaks will be changed in a way that breaks the code. Press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to

be perfect, but the `@</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab twice to line up the new code with the existing code.³

htmlCopy

```
<div class="form-group">
  <div class="col-md-offset-2 col-md-10">
    <table>
      <tr>
        @{
          int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

          foreach (var course in courses)
          {
            if (cnt++ % 3 == 0)
            {
              @:</tr><tr>
            }
            @:<td>
              <input type="checkbox"
                name="selectedCourses"
                value="@course.CourseID"
                @(Html.Raw(course.Assigned ? "checked=\"checked\""
: "")) />

              @course.CourseID @: @course.Title
            @:</td>
          }
        @:</tr>
      }
    </table>
  </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a check box followed by a caption that consists of the course number and title. The check boxes all have the same name ("selectedCourses"), which informs the model binder that they are to be treated as a group. The value attribute of each check box is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the

controller that consists of the `CourseID` values for only the check boxes which are selected.²

When the check boxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the Instructor Index page, and click Edit on an instructor to see the Editpage.

Edit - Contoso Universit × + − □ ×

← → ↻ | localhost:5813/Instruct | ☆ | ≡ ...

Contoso University ≡

Edit

Instructor

Last Name

First Name

Hire Date

Office Location

☐ 1000 Algebra 2 ☐ 1045 Calculus ☐ 1050 Chemistry
☒ 2021 Composition ☒ 2042 Literature ☐ 3141 Trigonometry
☐ 4022 Microeconomics ☐ 4041 Macroeconomics

Change some course assignments and click Save. The changes you make are reflected on the Index page.

Note

The approach taken here to edit instructor course data works well when there is a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update the Delete page

In *InstructorsController.cs*, delete the `DeleteConfirmed` method and insert the following code in its place.

C#Copy

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

```
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
```

```
        .SingleOrDefault(i => i.ID == id);
```

```
    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
```

```
    departments.ForEach(d => d.InstructorID = null);
```

```
    _context.Instructors.Remove(instructor);
```

```
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to the Create page

In *InstructorsController.cs*, delete the `HttpGet` and `HttpPost Create` methods, and then add the following code in their place:

C#Copy

```
public IActionResult Create()
{
```

```
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
```

```
    PopulateAssignedCourseData(instructor);
    return View();
}
```

```
// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public async Task<IActionResult>
Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor instructor,
string[] selectedCourses)
```

```
{
```

```
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
```

```

    {
        var courseToAdd = new CourseAssignment { InstructorID = instructor.ID,
CourseID = int.Parse(course) };
        instructor.CourseAssignments.Add(courseToAdd);
    }

```

```

    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The `HttpGet Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The `HttpPost Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

C#Copy

```

instructor.CourseAssignments = new List<CourseAssignment>();

```

As an alternative to doing this in controller code, you could do it in the Instructor model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:4

C#Copy

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new
List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In *Views/Instructor/Create.cshtml*, add an office location text box and check boxes for courses before the Submit button. As in the case of the Edit page, [fix the formatting if Visual Studio reformats the code when you paste it](#).²

htmlCopy

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="OfficeAssignment.Location" class="form-control" />
        <span asp-validation-for="OfficeAssignment.Location" class="text-danger" />
    </div>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                <td>
                    @{
                        int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;
```

```

        foreach (var course in courses)
        {
            if (cnt++ % 3 == 0)
            {
                @:</tr><tr>
            }
            @:<td>
                <input type="checkbox"
                    name="selectedCourses"
                    value="@course.CourseID"
                    @(Html.Raw(course.Assigned ? "checked=\"checked\""
: "")) />
                @course.CourseID @: @course.Title
            @:</td>
        }
        @:</tr>
    }
</table>
</div>
</div>

```

Test by running the Create page and adding an instructor.

Handling Transactions

As explained in the [CRUD tutorial](#), the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see [Transactions](#).

Summary

You have now completed the introduction to working with related data. In the next tutorial you'll see how to handle concurrency conflicts.

Handling concurrency conflicts - EF Core with ASP.NET Core MVC tutorial (8 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In earlier tutorials you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the Department entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.

Department × Edit - Cont × + - □ ×

← → ↻ | localhost:5813/Depart | ☆ | ...

Contoso University

Edit

Department

- The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.

Name

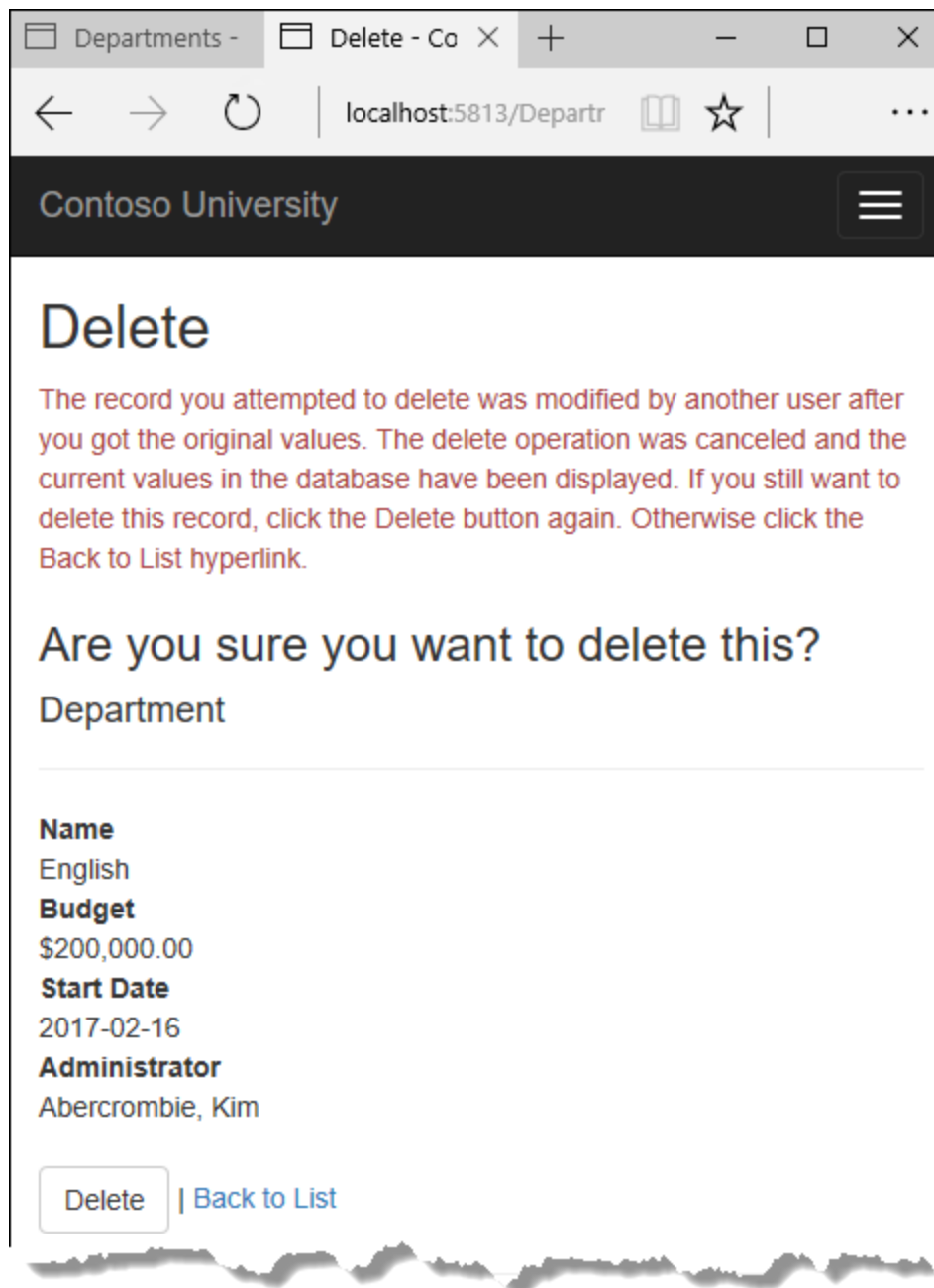
Budget

Current value: \$50,000.00

Start Date

InstructorID

Save



Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if it isn't really critical if some

changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from \$350,000.00 to \$0.00.

Edit - Cont × + − □ ×

← ↻ | localhost:581 | ☆ | ...

Contoso University ☰

Edit

Department

Budget

0 ×

Administrator

Abercrombie, Kim ▾

Name

English

Start Date

9/1/2007

Save

Before Jane clicks *Save*, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.²

Edit - Cont ✕ + - □ ✕

← ↻ | localhost:581 📖 ☆ | ...

Contoso University ☰

Edit

Department

Budget

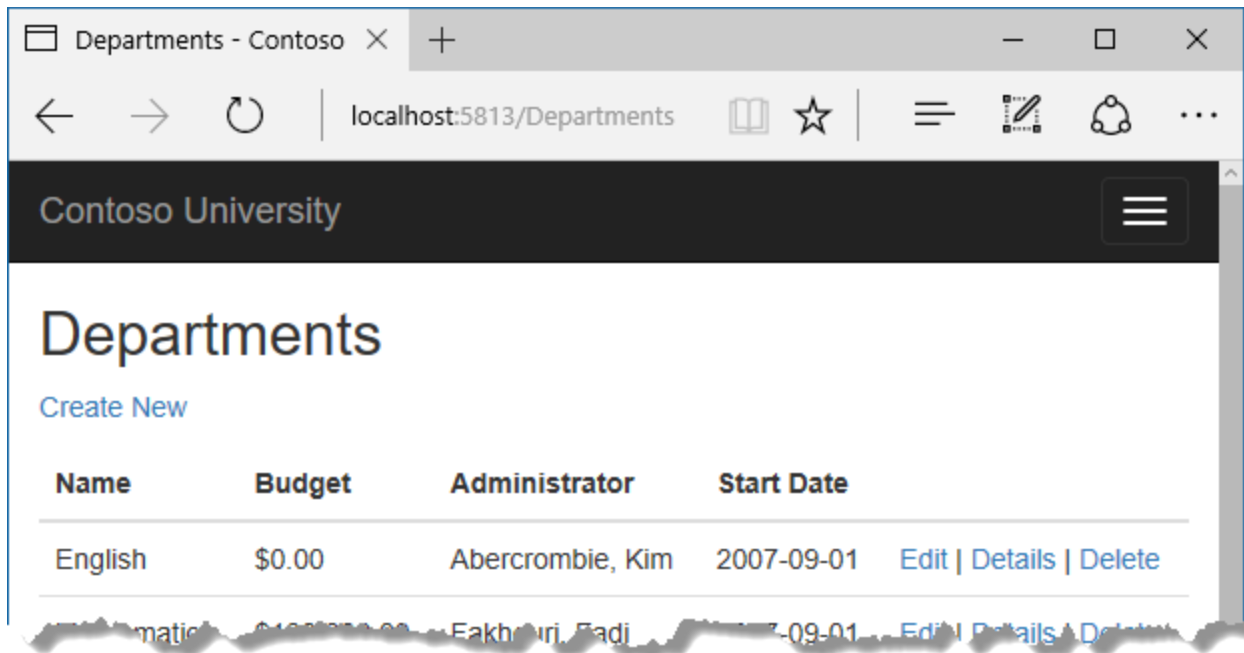
Administrator

Name

Start Date

Save

Jane clicks Save first and sees her change when the browser returns to the Index page.



Then John clicks [Save](#) on an [Edit](#) page that still shows a budget of \$350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they'll see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

The next time someone browses the English department, they'll see 9/1/2013 and the restored \$350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version). If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property to the Department entity

In *Models/Department.cs*, add a tracking property named RowVersion:

C#Copy

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }
    }
}
```

```

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

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

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}

```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsConcurrencyToken` method (in `Data/SchoolContext.cs`) to specify the tracking property, as shown in the following example:2

C#Copy

```

modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsConcurrencyToken();

```

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:1

consoleCopy

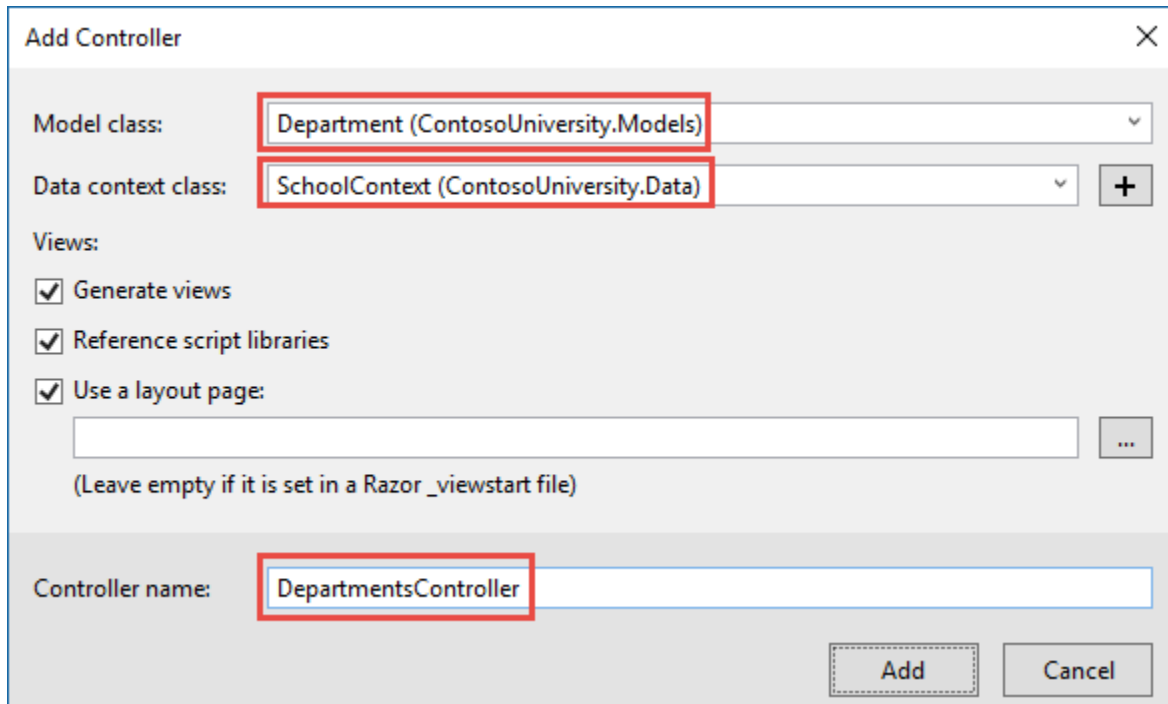
```

dotnet ef migrations add RowVersion
dotnet ef database update

```

Create a Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



The screenshot shows the 'Add Controller' dialog box. The 'Model class' dropdown is set to 'Department (ContosoUniversity.Models)'. The 'Data context class' dropdown is set to 'SchoolContext (ContosoUniversity.Data)'. Under the 'Views' section, three checkboxes are checked: 'Generate views', 'Reference script libraries', and 'Use a layout page:'. Below these checkboxes is an empty text box with a three-dot menu icon. At the bottom, the 'Controller name' text box contains 'DepartmentsController'. The 'Add' button is highlighted with a red dashed border.

In the *DepartmentsController.cs* file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

C#Copy

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",  
department.InstructorID);
```

Update the Departments Index view

The scaffolding engine created a RowVersion column in the Index view, but that field shouldn't be displayed.

Replace the code in *Views/Departments/Index.cshtml* with the following code.

htmlCopy

```
@model IEnumerable<ContosoUniversity.Models.Department>
```

```

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Administrator.FullName)
                </td>
            </tr>
        }
    </tbody>
</table>

```



```

                <a asp-action="Edit" asp-route-id="@item.DepartmentID">Edit</a> |
                <a asp-action="Details" asp-route-
id="@item.DepartmentID">Details</a> |
                <a asp-action="Delete" asp-route-
id="@item.DepartmentID">Delete</a>
            </td>
        </tr>
    }
</tbody>
</table>

```

This changes the heading to "Departments" deletes the RowVersion column, and shows full name instead of first name for the administrator.

Update the Edit methods in the Departments controller

In both the `HttpGet Edit` method and the `Details` method, add `AsNoTracking`. In the `HttpGet Edit` method, add eager loading for the Administrator.2

C#Copy

```

var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .SingleOrDefaultAsync(m => m.DepartmentID == id);

```

Replace the existing code for the `HttpPost Edit` method with the following code:

C#Copy

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i =>
i.Administrator).SingleOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)

```

```

{
    Department deletedDepartment = new Department();
    await TryUpdateModelAsync(deletedDepartment);
    ModelState.AddModelError(string.Empty,
        "Unable to save changes. The department was deleted by another user.");
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", deletedDepartment.InstructorID);
    return View(deletedDepartment);
}

_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue =
rowVersion;

if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty,
                "Unable to save changes. The department was deleted by another
user.");
        }
        else
        {
            var databaseValues = (Department)databaseEntry.ToObject();

            if (databaseValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
            }
            if (databaseValues.Budget != clientValues.Budget)
            {

```

```

        ModelState.AddModelError("Budget", $"Current value:
{databaseValues.Budget:c}");
    }
    if (databaseValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("StartDate", $"Current value:
{databaseValues.StartDate:d}");
    }
    if (databaseValues.InstructorID != clientValues.InstructorID)
    {
        Instructor databaseInstructor = await
_context.Instructors.SingleOrDefaultAsync(i => i.ID == databaseValues.InstructorID);
        ModelState.AddModelError("InstructorID", $"Current value:
{databaseInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty, "The record you attempted to
edit "
                                + "was modified by another user after you got the original
value. The "
                                + "edit operation was canceled and the current values in the
database "
                                + "have been displayed. If you still want to edit this
record, click "
                                + "the Save button again. Otherwise click the Back to List
hyperlink.");
    departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
    ModelState.Remove("RowVersion");
}
}
}
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID", "FullName",
departmentToUpdate.InstructorID);
return View(departmentToUpdate);
}

```

The code begins by trying to read the department to be updated. If the `SingleOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a department entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the department entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

C#Copy

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue = rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

C#Copy

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.³

C#Copy

```
var clientValues = (Department)exceptionEntry.Entity;  
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

C#Copy

```
var databaseValues = (Department)databaseEntry.ToObject();  
  
if (databaseValues.Name != clientValues.Name)  
{  
    ModelState.AddModelError("Name", $"Current value: {databaseValues.Name}");  
}
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks `Save`, only concurrency errors that happen since the redisplay of the Edit page will be caught.

C#Copy

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;  
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update the Department Edit view

In *Views/Departments/Edit.cshtml*, make the following changes:

- Remove the `<div>` element that was scaffolded for the `RowVersion` field.
- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

htmlCopy

```
@model ContosoUniversity.Models.Department  
  
@{  
    ViewData["Title"] = "Edit";  
}  
  
<h2>Edit</h2>  
  
<form asp-action="Edit">  
    <div class="form-horizontal">  
        <h4>Department</h4>  
        <hr />  
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>  
        <input type="hidden" asp-for="DepartmentID" />  
        <input type="hidden" asp-for="RowVersion" />  
  
        <div class="form-group">
```

```

        <label asp-for="Name" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Name" class="form-control" />
            <span asp-validation-for="Name" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Budget" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Budget" class="form-control" />
            <span asp-validation-for="Budget" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="StartDate" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="StartDate" class="form-control" />
            <span asp-validation-for="StartDate" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="InstructorID" class="control-label col-md-2"></label>
        <div class="col-md-10">

```

```

            <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                <option value="">-- Select Administrator --</option>

```

```

            </select>
            <span asp-validation-for="InstructorID" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" value="Save" class="btn btn-default" />
        </div>
    </div>
</div>
</form>

<div>
    <a asp-action="Index">Back to List</a>
</div>

```

```
@section Scripts {  
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}  
}
```

Test concurrency conflicts in the Edit page

Run the site and click Departments to go to the Departments Index page.

Right click the Edit hyperlink for the English department and select Open in new tab, then click the Edit hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click Save.

Contoso University

Edit Department

Name

English

Budget

50000.00

Start Date

9/1/2007

InstructorID

Abercrombie, Kim

Save

The browser shows the Index page with the changed value.

Change a field in the second browser tab.

Departments - Edit - Conto X + - □ X

localhost:5813/Departr ☆ ...

Contoso University

Edit

Department

Name

English

Budget

200000.00 x

Start Date

9/1/2007

InstructorID

Abercrombie, Kim v

Save

Click Save. You see an error message:

Department × Edit - Cont × + − □ ×

← → ↻ | localhost:5813/Departr | ☆ | ...

Contoso University

Edit

Department

- The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.

Name

English

Budget

200000.00

Current value: \$50,000.00

Start Date

9/1/2007

InstructorID

Abercrombie, Kim ▼

Save

Click Save again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the `HttpGet Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the `HttpPost Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the `HttpGet Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In *DepartmentController.cs*, replace the `HttpGet Delete` method with the following code:

C#Copy

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
```

```
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
```

```
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction("Index");
        }
    }
```

```

    }
    return NotFound();
}

```

```

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to delete "
            + "was modified by another user after you got the original values. "
            + "The delete operation was canceled and the current values in the "
            + "database have been displayed. If you still want to delete this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }

```

```

    }

    return View(department);
}

```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the Department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.²

Replace the code in the `HttpPost Delete` method (named `DeleteConfirmed`) with the following code:⁴

C#Copy

```

[HttpPost]
[ValidateAntiForgeryToken]

```

```

public async Task<IActionResult> Delete(Department department)

```

```

{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID ==
            department.DepartmentID))
    }
}

```

```

    {
        _context.Departments.Remove(department);
        await _context.SaveChangesAsync();
    }
    return RedirectToAction("Index");
}
catch (DbUpdateConcurrencyException /* ex */)
{
    //Log the error (uncomment ex variable name and write a log.)
    return RedirectToAction("Delete", new { concurrencyError = true, id =
department.DepartmentID });
}
}

```

In the scaffolded code that you just replaced, this method accepted only a record ID:

C#Copy

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a Department entity instance created by the model binder. This gives EF access to the RowVersion property value in addition to the record key.

C#Copy

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplay the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In *Views/Department/Delete.cshtml*, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

htmlCopy

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
```

```
<form asp-action="Delete">
```

```
<input type="hidden" asp-for="DepartmentID" />
```

```
<input type="hidden" asp-for="RowVersion" />
<div class="form-actions no-color">
  <input type="submit" value="Delete" class="btn btn-default" /> |
  <a asp-action="Index">Back to List</a>
</div>
</form>
</div>
```

This makes the following changes:

- Adds an error message between the `h2` and `h3` headings.
- Replaces LastName with FullName in the Administrator field.
- Removes the RowVersion field.
- Adds hidden fields for the `DepartmentID` and `RowVersion` properties.

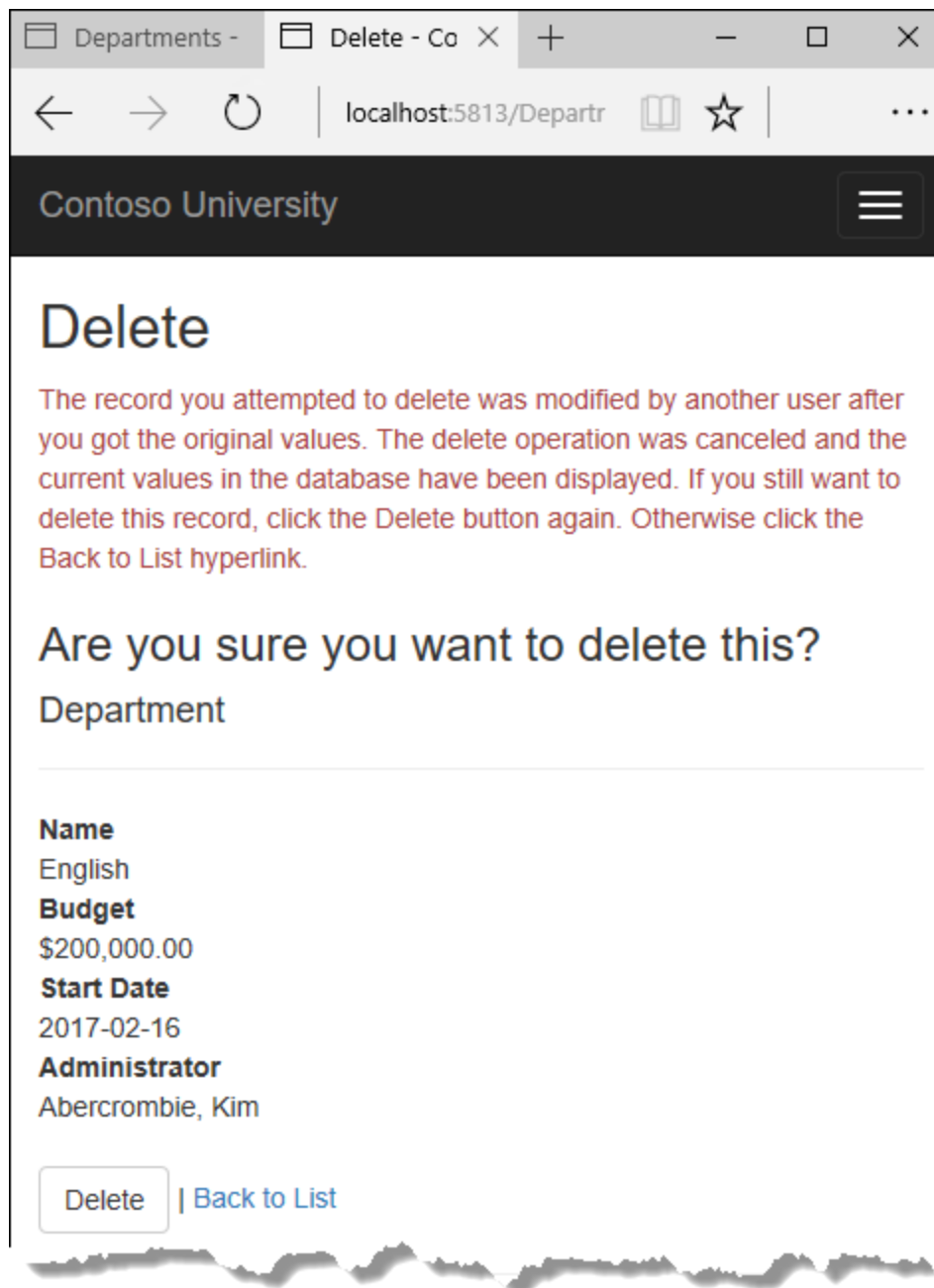
Run the Departments Index page. Right click the Delete hyperlink for the English department and select Open in new tab, then in the first tab click the Edit hyperlink for the English department.

In the first window, change one of the values, and click Save:

The screenshot shows a web browser with two tabs: 'Edit - Conto...' and 'Delete - Conto...'. The address bar shows 'localhost:5813/Departr'. The page header is 'Contoso University' with a hamburger menu icon. The main heading is 'Edit Department'. The form contains the following fields:

- Name**: Text input with value 'English'.
- Budget**: Text input with value '200000.00'.
- Start Date**: Text input with value '2/16/2017'. This field is highlighted with a red rectangular box.
- InstructorID**: Dropdown menu with value 'Abercrombie, Kim'.
- Save**: Button at the bottom of the form.

In the second tab, click Delete. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click Delete again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in *Views/Departments/Details.cshtml* to delete the RowVersion column and show the full name of the Administrator.

htmlCopy

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

```
</div>
```

Replace the code in *Views/Departments/Create.cshtml* to add a Select option to the drop-down list.

htmlCopy

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Department</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Name" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="Budget" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="StartDate" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger"></span>
            </div>
        </div>
        <div class="form-group">
            <label asp-for="InstructorID" class="col-md-2 control-label"></label>
            <div class="col-md-10">
```

```

        <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
        <option value="">-- Select Administrator --</option>

```

```

        </select>
    </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>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}

```

Summary

This completes the introduction to handling concurrency conflicts. For more information about how to handle concurrency in EF Core, see [Concurrency conflicts](#). The next tutorial shows how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Inheritance - EF Core with ASP.NET Core MVC tutorial (9 of 10)

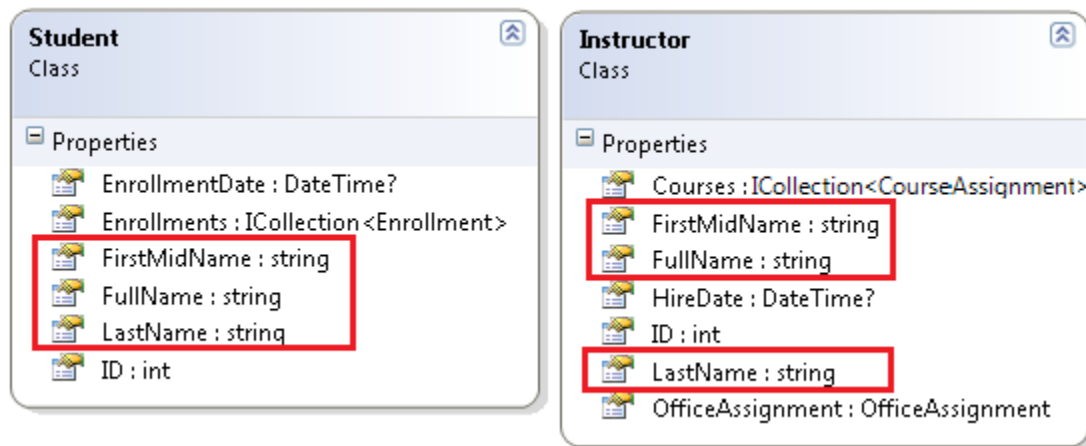
The Contoso University sample web application demonstrates how to create ASP.NET Core 1.1 MVC web applications using Entity Framework Core 1.1 and Visual Studio 2017. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

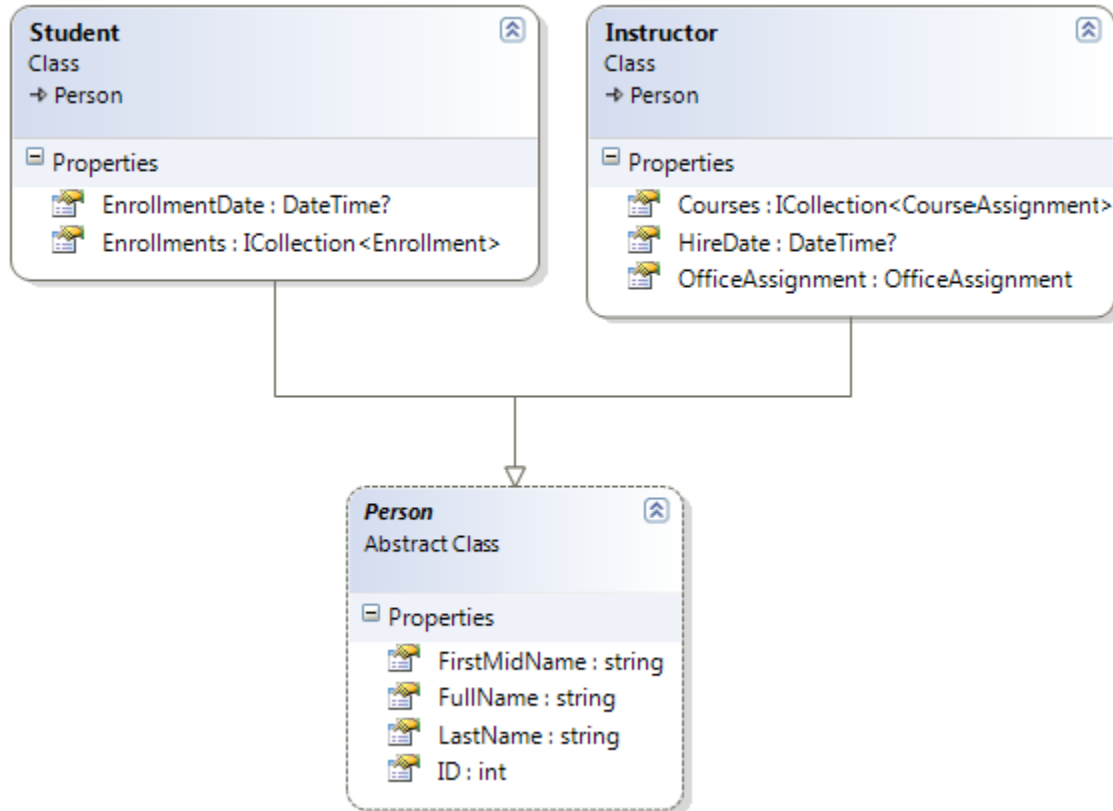
Options for mapping inheritance to database tables

The `Instructor` and `Student` classes in the School data model have several properties that are identical:

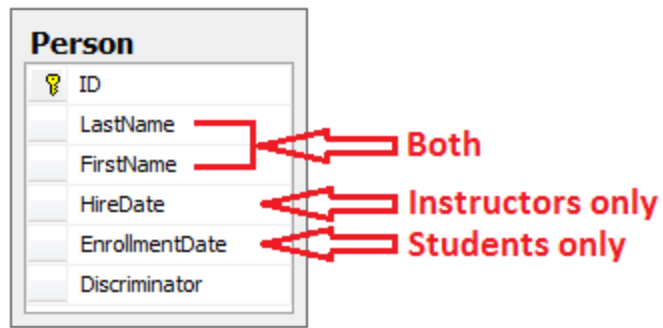


Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You

could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:

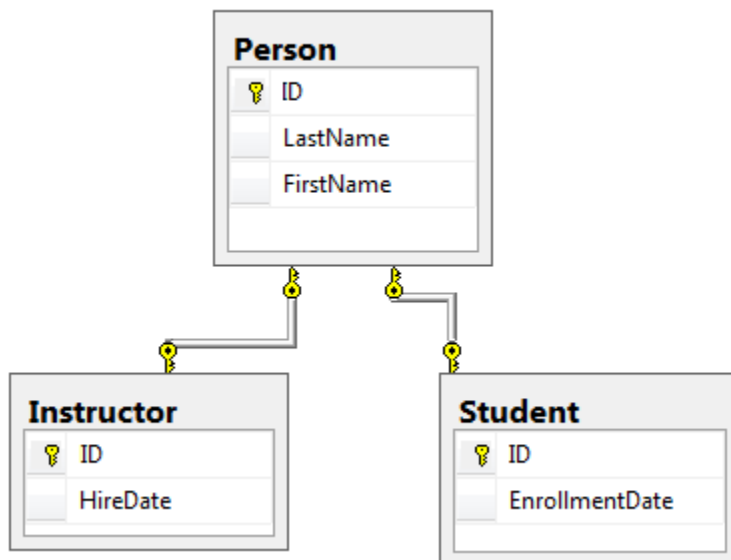


There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (`HireDate`), some only to students (`EnrollmentDate`), some to both (`LastName`, `FirstName`). Typically, you'd have a discriminator column to indicate which type each row represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called table-per-hierarchy (TPH) inheritance.¹

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the **Person** table and have separate **Instructor** and **Student** tables with the date fields.



This pattern of making a database table for each entity class is called table per type (TPT) inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called Table-per-Concrete Class (TPC) inheritance. If you implemented TPC inheritance for the **Person**, **Student**, and **Instructor** classes as shown earlier, the **Student** and **Instructor** tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

Tip

Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.²

Create the Person class

In the Models folder, create `Person.cs` and replace the template code with the following code:

C#Copy

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }

        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than 50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }

        [Display(Name = "Full Name")]
    }
}
```



```

        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}

```

Make Student and Instructor classes inherit from Person

In *Instructor.cs*, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

C#Copy

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}

```

Make the same changes in *Student.cs*.

C#Copy

```

using System;

```

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}

```

Add the Person entity type to the data model

Add the Person entity type to *SchoolContext.cs*. The new lines are highlighted.

C#Copy

```

using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) : base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
    }
}

```

```

public DbSet<Person> People { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Course>().ToTable("Course");
    modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
    modelBuilder.Entity<Student>().ToTable("Student");
    modelBuilder.Entity<Department>().ToTable("Department");
    modelBuilder.Entity<Instructor>().ToTable("Instructor");
    modelBuilder.Entity<OfficeAssignment>().ToTable("OfficeAssignment");
    modelBuilder.Entity<CourseAssignment>().ToTable("CourseAssignment");
    modelBuilder.Entity<Person>().ToTable("Person");

    modelBuilder.Entity<CourseAssignment>()
        .HasKey(c => new { c.CourseID, c.InstructorID });
}
}

```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and customize migration code

Save your changes and build the project. Then open the command window in the project folder and enter the following command:1

```
consoleCopy
```

```
dotnet ef migrations add Inheritance
```

Run the `database update` command:.

```
consoleCopy
```

```
dotnet ef database update
```

The command will fail at this point because you have existing data that migrations doesn't know how to handle. You get an error message like the following one:

The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_CourseAssignment_Person_InstructorID". The conflict occurred in database "ContosoUniversity09133", table "dbo.Person", column 'ID'.

Open *Migrations<timestamp>_Inheritance.cs* and replace the `Up` method with the following code:2

C#Copy

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table: "Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table: "Person",
nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table: "Person",
nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table: "Person",
nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person", nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName, HireDate,
EnrollmentDate, Discriminator, OldId) SELECT LastName, FirstName, null AS HireDate,
EnrollmentDate, 'Student' AS Discriminator, ID AS OldId FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID FROM
dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator = 'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
        name: "IX_Enrollment_StudentID",
        table: "Enrollment",
        column: "StudentID");
```

```
migrationBuilder.AddForeignKey(  
    name: "FK_Enrollment_Person_StudentID",  
    table: "Enrollment",  
    column: "StudentID",  
    principalTable: "Person",  
    principalColumn: "ID",  
    onDelete: ReferentialAction.Cascade);  
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.
- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they'll get new primary key values.
- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command again:

```
consoleCopy
```

```
dotnet ef database update
```

(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

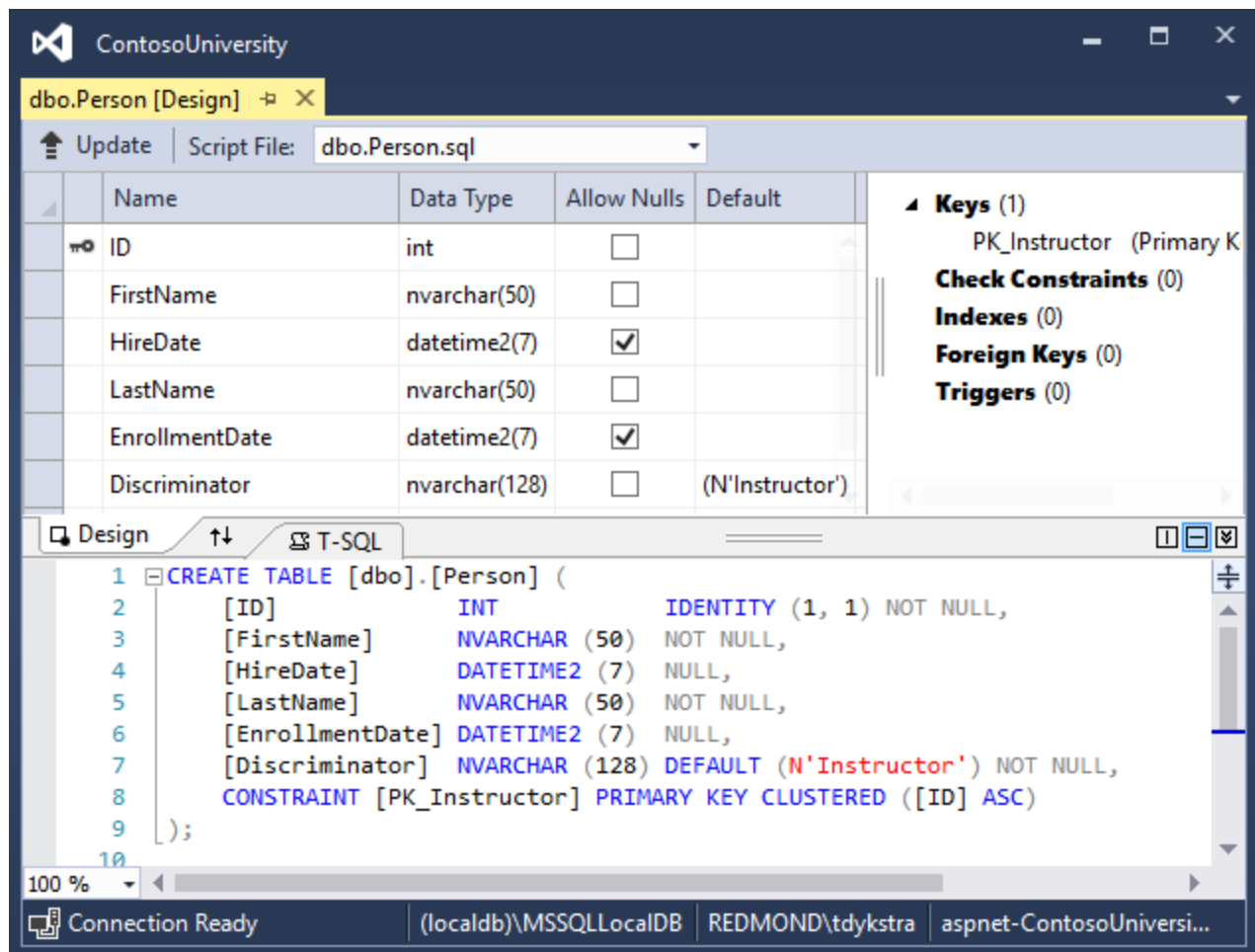
Note

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `drop database` CLI command.

Test with inheritance implemented

Run the site and try various pages. Everything works the same as it did before.

In SQL Server Object Explorer, **expand** Data Connections/SchoolContext and then Tables, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click Show Table Data to see the discriminator column.

ContosoUniversity

dbo.Person [Data]

Max Rows: 1000

	ID	FirstName	HireDate	LastName	Enrollme...	Discriminator
▶	1	Kim	3/11/1995...	Abercrombie	NULL	Instructor
	2	Fadi	7/6/2002 ...	Fakhouri	NULL	Instructor
	3	Roger	7/1/1998 ...	Harui	NULL	Instructor
	4	Candace	1/15/2001...	Kapoor	NULL	Instructor
	5	Roger	2/12/2004...	Zheng	NULL	Instructor
	7	Nancy	8/17/2016...	Davolio	NULL	Instructor
	8	Carson	NULL	Alexander	9/1/2010 ...	Student
	9	Meredith	NULL	Alonso	9/1/2012 ...	Student
	10	Arturo	NULL	Anand	9/1/2013 ...	Student
	11	Gytis	NULL	Barzdukas	9/1/2012 ...	Student
	12	Yan	NULL	Li	9/1/2012 ...	Student

Summary

You've implemented table-per-hierarchy inheritance for the `Person`, `Student`, and `Instructor` classes. For more information about inheritance in Entity Framework Core, see [Inheritance](#). In the next tutorial you'll see how to handle a variety of relatively advanced Entity Framework scenarios.

Advanced topics - EF Core with ASP.NET Core MVC tutorial (10 of 10)

The Contoso University sample web application demonstrates how to create ASP.NET Core 1.0 MVC web applications using Entity Framework Core 1.0 and Visual Studio 2015. For information about the tutorial series, see [the first tutorial in the series](#).

In the previous tutorial you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET web applications that use Entity Framework Core.

Raw SQL Queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they are automatically tracked by the database context unless you [turn tracking off](#).
- Use the `Database.ExecuteNonQueryCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query that returns entities

The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In *DepartmentsController.cs*, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

C#Copy

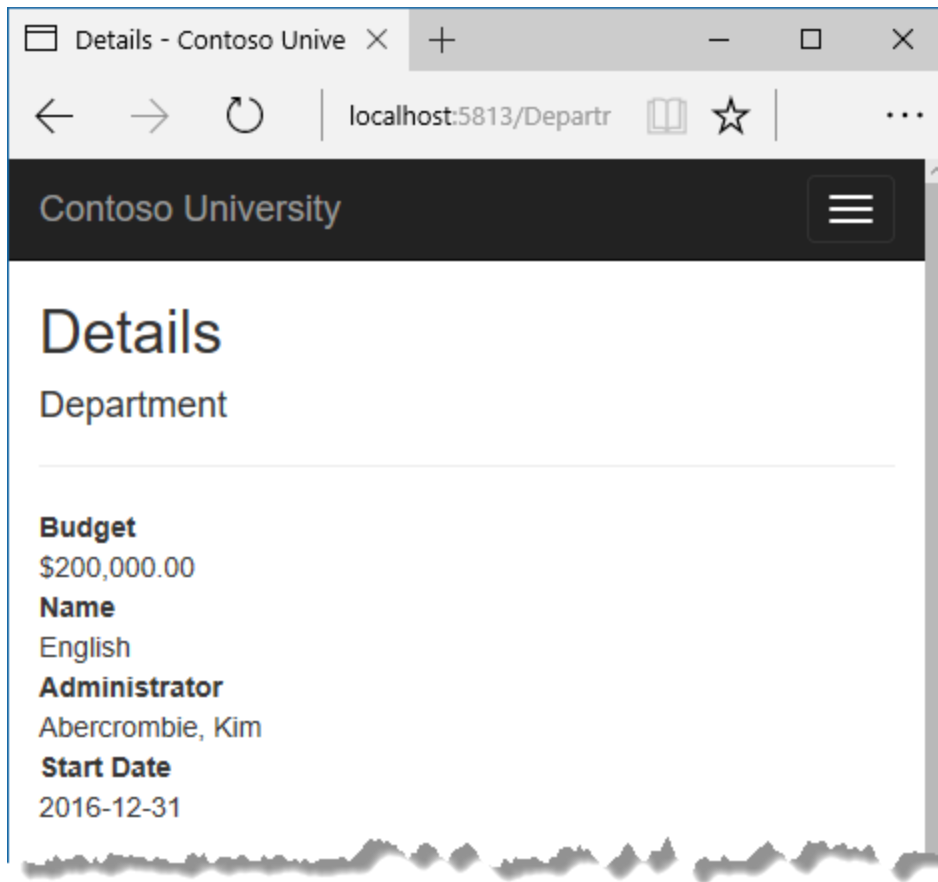
```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .SingleOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the Departments tab and then Details for one of the departments.



Call a query that returns other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is write ADO.NET code and get the database connection from EF.

In *HomeController.cs*, replace the `About` method with the following code:

C#Copy

```
public async Task<ActionResult> About()  
{
```

```
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
```

```

var conn = _context.Database.GetDbConnection();
try
{
    await conn.OpenAsync();
    using (var command = conn.CreateCommand())
    {
        string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount "
            + "FROM Person "
            + "WHERE Discriminator = 'Student' "
            + "GROUP BY EnrollmentDate";
        command.CommandText = query;
        DbDataReader reader = await command.ExecuteReaderAsync();

        if (reader.HasRows)
        {
            while (await reader.ReadAsync())
            {
                var row = new EnrollmentDateGroup { EnrollmentDate =
reader.GetDateTime(0), StudentCount = reader.GetInt32(1) };
                groups.Add(row);
            }
        }
        reader.Dispose();
    }
}
finally
{
    conn.Close();
}

```

```

return View(groups);
}

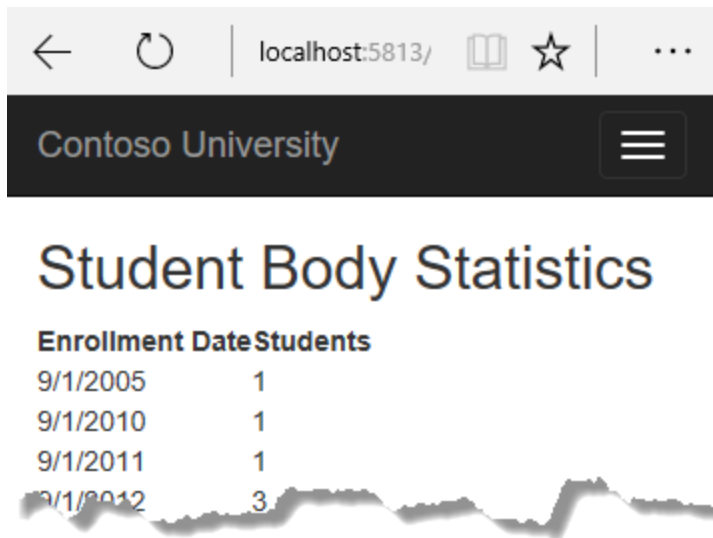
```

Add a using statement:

C#Copy

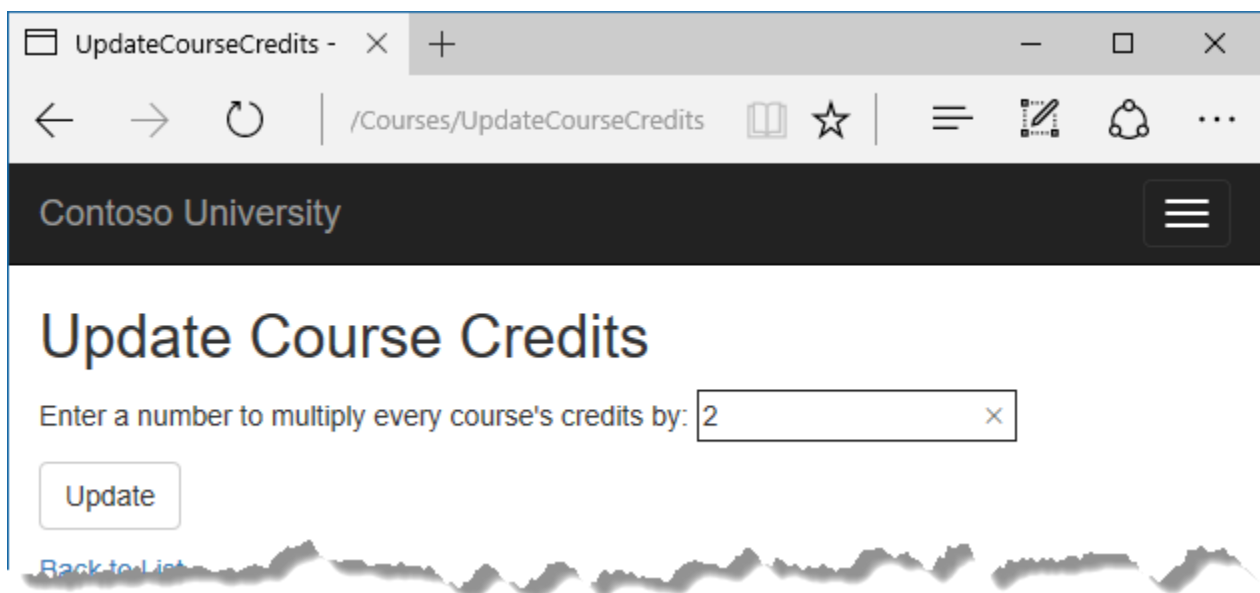
```
using System.Data.Common;
```

Run the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In *CoursesContoller.cs*, add *UpdateCourseCredits* methods for *HttpGet* and *HttpPost*:

C#Copy

```
public IActionResult UpdateCourseCredits()
{
    return View();
}
```

C#Copy

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an `HttpGet` request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the Update button is clicked, the `HttpPost` method is called, and `multiplier` has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In Solution Explorer, right-click the *Views/Courses* folder, and then click Add > New Item.

In the Add New Item dialog, click ASP.NET under Installed in the left pane, click MVC View Page, and name the new view *UpdateCourseCredits.cshtml*.

In *Views/Courses/UpdateCourseCredits.cshtml*, replace the template code with the following code:

htmlCopy

@{

```

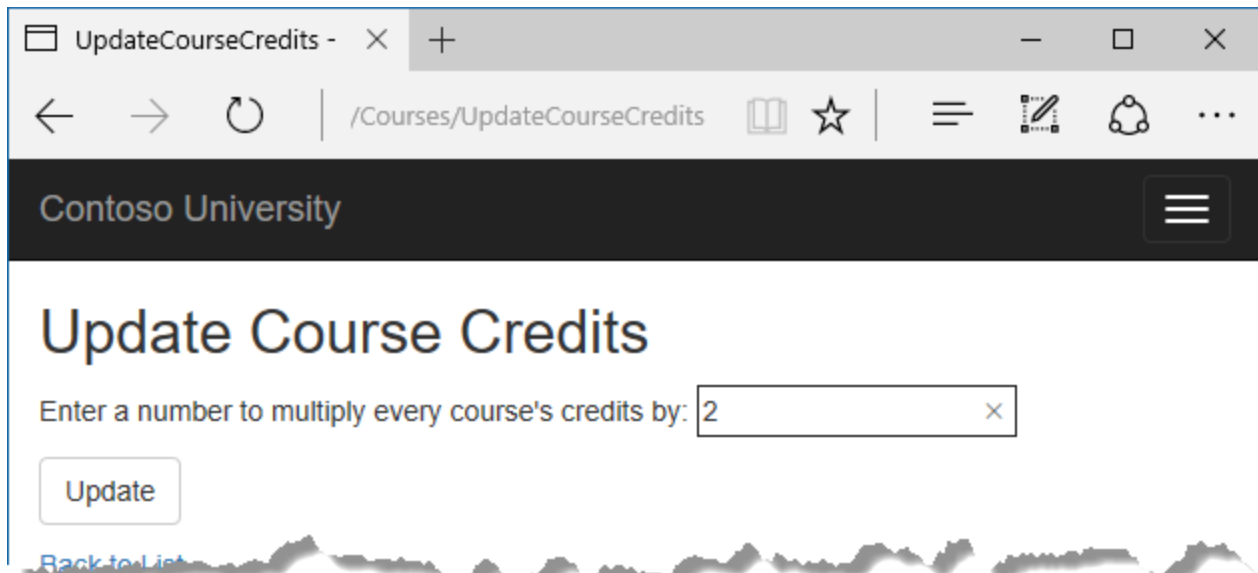
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
            <p>
                Enter a number to multiply every course's credits by:
                @Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default" />
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

Run the `UpdateCourseCredits` method by selecting the `Courses` tab, then adding `/UpdateCourseCredits` to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Course/UpdateCourseCredits`). Enter a number in the text box:



UpdateCourseCredits - X +

← → ↻ | /Courses/UpdateCourseCredits | ☆ ≡ ✎ 🔔 ⋮

Contoso University

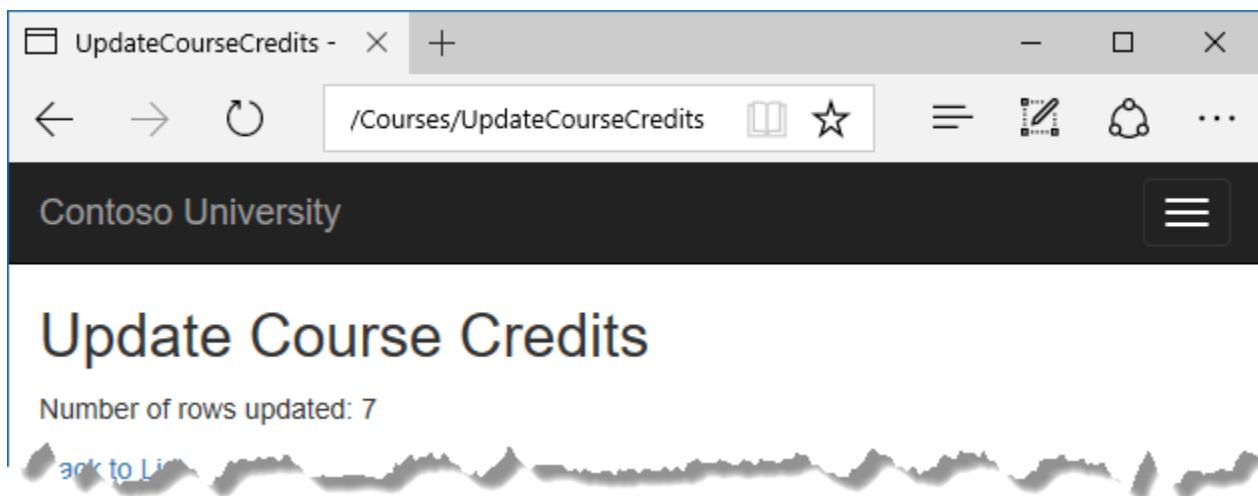
Update Course Credits

Enter a number to multiply every course's credits by:

Update

[Back to List](#)

Click Update. You see the number of rows affected:



UpdateCourseCredits - X +

← → ↻ | /Courses/UpdateCourseCredits | ☆ ≡ ✎ 🔔 ⋮

Contoso University

Update Course Credits

Number of rows updated: 7

[Back to List](#)

Click Back to List to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see [Raw SQL Queries](#).

Examine SQL sent to the database

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open *StudentsController.cs* and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the application in debug mode, and go to the Details page for a student.

Go to the Output window showing debug output, and you see the query:

Copy

```
Microsoft.EntityFrameworkCore.Storage.IRelationalCommandBuilderFactory:Information:
Executed DbCommand (225ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT [e].[EnrollmentID], [e].[CourseID], [e].[Grade], [e].[StudentID],
[c].[CourseID], [c].[Credits], [c].[DepartmentID], [c].[Title]
FROM [Enrollment] AS [e]
INNER JOIN (
    SELECT DISTINCT TOP(2) [s].[ID]
    FROM [Person] AS [s]
    WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
    ORDER BY [s].[ID]
) AS [s0] ON [e].[StudentID] = [s0].[ID]
INNER JOIN [Course] AS [c] ON [e].[CourseID] = [c].[CourseID]
ORDER BY [s0].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`). The `SingleOrDefaultAsync` method doesn't resolve to one row on the server. If the Where clause matches multiple rows, the method must return null, so EF only has to select a maximum of 2 rows, because if 3 or more match the Where clause, the result from the `SingleOrDefault` method is the same as if 2 rows match.³

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the Output window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Repository and unit of work patterns

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns is not always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see [the Entity Framework 5 version of this tutorial series](#).

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see [Testing with InMemory](#).

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- `DbContext.SaveChanges`
- `DbContext.Entry`
- `ChangeTracker.Entries`

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
C#Copy
```

```
_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

Entity Framework Core source code and development plans

The source code for Entity Framework Core is available at <https://github.com/aspnet/EntityFramework>. Besides source code, you can get nightly builds, issue tracking, feature specs, design meeting notes, [the roadmap for future development](#), and more. You can file bugs, and you can contribute your own enhancements to the EF source code.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the [scaffold-dbcontext](#) command. See the [getting-started tutorial](#).

Use dynamic LINQ to simplify sort selection code

The [third tutorial in this series](#) shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

C#Copy

```
public async Task<IActionResult> Index(  
    string sortOrder,  
    string currentFilter,  
    string searchString,  
    int? page)  
{
```

```

ViewData["CurrentSort"] = sortOrder;
ViewData["NameSortParm"] =
    String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
ViewData["DateSortParm"] =
    sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" : "EnrollmentDate";

if (searchString != null)
{
    page = 1;
}
else
{
    searchString = currentFilter;
}

ViewData["CurrentFilter"] = searchString;

var students = from s in _context.Students
                select s;

if (!String.IsNullOrEmpty(searchString))
{
    students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
}

if (string.IsNullOrEmpty(sortOrder))
{
    sortOrder = "LastName";
}

bool descending = false;
if (sortOrder.EndsWith("_desc"))
{
    sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
    descending = true;
}

if (descending)
{
    students = students.OrderByDescending(e => EF.Property<object>(e,
sortOrder));
}
else
{

```

```
        students = students.OrderBy(e => EF.Property<object>(e, sortOrder));
    }

    int pageSize = 3;
    return View(await PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
        page ?? 1, pageSize));
}
```

Next steps

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET MVC application.

For more information about EF Core, see the [Entity Framework Core documentation](#). A book is also available: [Entity Framework Core in Action](#).

For information about how to deploy your web application after you've built it, see [Publishing and deployment](#).

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see the [ASP.NET Core documentation](#).

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials.

Common errors

ContosoUniversity.dll used by another process

Error message:

Cannot open '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.'

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click Stop Site.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there is no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in *appsettings.json*. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click Delete, and then in the Delete Database dialog box select Close existing connections and click OK.

To delete a database by using the CLI, run the `database drop` CLI command:

```
consoleCopy
```

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections.
(provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.